RubyGuides

The Definitive Ruby Tutorial

If you have decided to learn Ruby & become a Ruby developer then you're in the right place.

You have a lot of learning ahead of you, but don't let that stop you from getting started.

Learning is a beautiful thing.

Now:

Where do you begin?

You should start by learning the **core programming concepts**.

Things like variables, data structures, and conditional statements.

You'll find a lot of new words in your journey, **don't worry about that**, you'll learn along the way.

You also need to understand that a programming language is a formal language.

In English, if I make a grammar mistake there is a pretty good chance you can still understand me.

But if you make a grammar mistake in Ruby, or any other programming language, you are going to get an error.

The reason I'm telling you this is because I don't want you to give up early if you're seeing a lot of error messages & if things don't make a lot of sense.

These things are normal, **you are learning something new & it's going to take some time until it starts to sink in**.

One of the keys is repetition.

Work on every topic until you understand how it's useful in the big picture, how to use it, how to explain it to other people.

You can do this!

Let's get started with the first step...

How to Download & Install Ruby

If you are using Windows you want to go to this site to download Ruby:

https://rubyinstaller.org/downloads/

You want the recommended version, which at the time of this writing is `Ruby+Devkit 2.4.4-2 (x64)`.

Just download & install it like any other program.

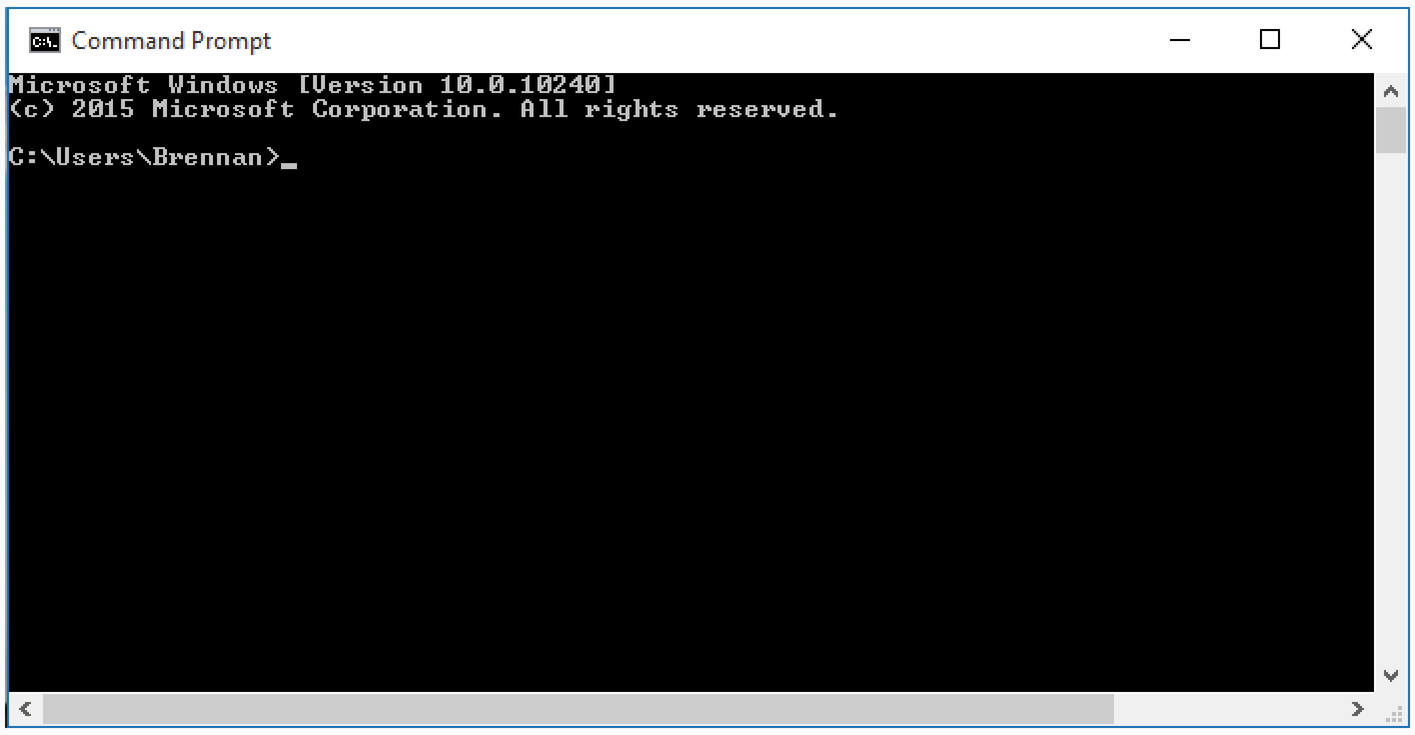
If no errors appear then you have Ruby installed on your computer!

Now **to start writing your Ruby programs you will need to open a terminal**.

To do that open the windows menu & type `cmd.exe`.

Then press enter.

It will look something like this:



```
Microsoft Windows [Version 10.0.10240]
(c) 2015 Microsoft Corporation. All rights reserved.

C:\Users\Brennan>_
```

At this point you should be able to type `ruby -v` inside this window & get the current version of Ruby printed in there.

Ruby Setup For Linux & Mac Users

If you are on Linux or MacOS then you probably already have Ruby installed.

You can confirm this by **opening a terminal** (search for "terminal" in your menu), then typing `ruby -v`.

This should print your Ruby version.

Like this:

```
ruby 2.4.1 (2017-03-22 revision 58053) [i686-linux]
```

If you don't get a Ruby version then refer to this site for more details on how to install Ruby for your particular Linux version or Mac.

Let's Write Some Code!

Now that you are set up, I want you to open `irb`.

This is a Ruby program that allows you to type Ruby code & see the results right away.

It's a great way to practice & learn about Ruby.

To open `irb` you have to **type the word irb inside that black terminal window** I had you open before.

Press enter.

Then **you should see something like this**:

```
irb(main):001:0>
```

Start by typing this into `irb`:

```
5 + 5
```

Then press enter.

You'll see the result of this operation.

> "But I can do that with a calculator program!"

Of course.

The point of this is to get you used to typing inside the terminal window.

And you're going to be using numbers a lot inside your Ruby programs:

- As data (price, age, year, etc.)
- To count things
- To access information

Now it's time to practice!

Try these math operations:

```
10 * 2
500 / 100
1 + 2 + 3 + 4 + 5
```

Also I want you to close your terminal window, open it again, open `irb` & type more math.

Do that a few times so you'll remember how to do it the next time.

Ruby Variables

You can save a value & give it a name by using a variable.

A variable is **just a label**...

...a name for something that you can use to reference this value in your Ruby programs.

Just like the name we give to real-world things.

When I say "apple" **you know what I'm talking about**.

I don't have to describe it to you.

That's what variables do!

Creating Variables

Here's how you can create a variable in Ruby:

```ruby
age = 32
```

Now when you type age Ruby will translate that into 32.

Try it!

There is nothing special about the word age.

You could use `bacon = 32` & the value would still be 32.

Using Variables

To use a variable you write its name:

```
age * 10
# 320
```

You can combine multiple variables together:

```
age = 32
multiplier = 10

age * multiplier
```

And save the result of calculations into a new variable:

```
total = age * multiplier
```

Variable Types

In Ruby, we have different variable types.

What I showed you is called a "local variable".

But there are other kinds too:

- global variable
- instance variable
- class variable
- constant

You don't need to worry too much about these right now, but **it's good to know they exist**.

The difference between them?

It's on their "scope".

A **variable scope answers this question**:

> "From where can I access this variable?"

This is only going to matter when you start learning about Object-Oriented Programming, we'll revisit this topic by then.

Practice Time!

Now:

This concept of variables is very useful, **if you don't use variables** you would have to repeat every single calculation every time that you want to refer to it.

And **you wouldn't have a way to name things** so you know what they are.

Open up `irb` & do this:

1. Create a variable named `orange` & give it a value of `300`.
2. Create another variable named `apple` & give it a value of `120`.
3. Multiply the value of `orange` & `apple`, then assign the result to a new variable `total`.

When you feel comfortable using variables then go to the next lesson.

You're doing great!

Strings, Arrays & Hashes

You have learned about variables & basic math operations in Ruby.

Next you're going to learn about strings!

A string is a sequence of characters inside quotes.

Example:

```
"bacon"
```

You need the quotes **so you can tell the difference between strings & variables**.

A string is data.

A variable is a name for something.

In fact, you can use a variable to name a string:

```
food = "bacon"
```

This helps you reuse the string as many times as you want.

What Can You Do With Strings?

Strings can do many things for you.

For example, you can ask a string for its size:

```
"bacon".size

# 5
```

Or you can tell the string to uppercase itself:

```
"bacon".upcase

# "BACON"
```

We call these "methods", and there are many of them.

There are methods that allow you to substitute all or part of the string.

Like `gsub`:

```
"bacon".gsub("acon", "inary")

# "binary"
```

Or to split the string into a list of characters:

```
"bacon".chars

# ["b", "a", "c", "o", "n"]
```

Check this article for a more complete list of string methods.

When To Use Strings

You use strings whenever you want to print a message to the screen.

Like this:

```
puts "Hello There!"
```

Also when you read files, read input from the user (with the `gets` method), or when you want to combine several pieces of information together.

Strings can contain numbers, but that doesn't mean you can treat them like numbers.

Here's what I mean:

```
"1" + "1"
```

Gives you:

```
"11"
```

If you want to convert a string with numbers into an actual integer value you have to use the `to_i` method.

> **Remember**: A method is a command you can give to any object in Ruby. Depending on the kind of object you're working with they will respond to different methods.

Converting a string to integer:

```
"1".to_i
```

Converting an integer to a string:

```ruby
1.to_s
```

Why is there a difference?

Because integers are numbers, they must behave like numbers & allow for mathematical operations.

Strings have a different purpose, and a different set of methods.

How to Combine Strings Using String Interpolation

To combine numbers & strings you need a technique named "string interpolation".

Here's an example:

```ruby
age = 20
name = "David"

puts "Hello #{name}, you're #{age} years old!"
```

This is like a template.

Ruby replaces these `#{name}` & `#{age}` by their values, producing the combined string.

How to Use Arrays

If you want to have **many of the same thing** then arrays are very useful.

An array is a collection of items in a single location.

Here's an example:

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

This is an array of integers.

You can access every element by its position. We call that position an index.

Example:

```
letters = ['a', 'b', 'c']

letters[0]
# 'a'

letters[1]
# 'b'

letters[2]
# 'c'
```

Important!

The **first element in an array is always on position 0**.

Don't ask me why, but you have to remember that.

If you ask for an index that is bigger than the array size you'll get a nil value.

It looks like this:

```
letters[4]

# nil
```

A nil value is Ruby telling you:

"I can't find what you want so here is a generic response"

And just like strings, arrays have a set of methods you can use to make them do things.

For example:

```
letters.size

# 3
```

You can add new elements to an array like this:

```
numbers = []

numbers << 1
numbers << 2
numbers << 3

numbers
# [1, 2, 3]
```

This is a very useful array method, so write it down.

Both strings & arrays are **very important building blocks for writing your Ruby programs**.

If you don't understand how these 2 work you won't be able to write even the most basic projects you can think of.

Don't just read this...

Go open `irb` now & **practice**.

If you don't know what `irb` is or how to open it you need to go back to chapter 1 of this guide.

How to Use a Ruby Hash

A hash is like a dictionary.

It helps you **match two values together**...

...like a domain name to an IP address, or a phone number to a person's name.

Here's how to create a hash in Ruby:

```ruby
ip_to_domain = { "rubyguides.com" => "185.14.187.159" }
```

You can get the value for a hash key like this:

```ruby
ip_to_domain["rubyguides.com"]

# "185.14.187.159"
```

And you can change the value like this:

```ruby
ip_to_domain["rubyguides.com"] = "8.8.8.8"
```

These are the 3 main operations you can do with a hash.

Do These Exercises Now

- Create an array with the name of `food` with these values: "bacon", "orange", "apple"
- Access the first element of the `food` array
- Access the last elements of the `food` array
- Add a new element into the array: "yogurt"
- Create a hash with 3 key/value pairs representing country codes (like ES) & their country names (like Spain).
- Answer this question: What is the difference between 1 and `"1"`?

Ruby Loops Explained

In this lesson you'll learn many different ways to write a **Ruby loop**.

A loop lets you repeat an action many times.

You can go over a list of things, like an array or a hash, and work with each individual element.

Let's start with the most important looping method...

The Each Loop

This loop requires you to have a collection of items, like an array, a range or a hash to be able to use it.

Example:

```
numbers = [1, 3, 5, 7]

numbers.each { |n| puts n }
```

In plain English this is saying:

> "For each element in `numbers` print its value."

The way you tell the each method what do with every item is by using a block...

...in this example the whole thing after each is a block: `{ |n| puts n }`.

What happens is that each will use the block once for every element in the array & pass every individual element into it, so this `n` is a variable that changes.

Each Method With a Hash

If you want to use each with a hash you will need two parameters (one for the key & another for the value).

Example:

```
hash = {bacon: 300, coconut: 200}

hash.each { |key,value| puts "#{key} price is #{value}" }
```

Give this a try!

How to Use Each With Index

There are cases where you want to use each but you need the index number.

You can use the each_with_index method:

```ruby
animals = ["cat", "dog", "tiger"]

animals.each_with_index do |animal, idx|
  puts "We have a #{animal} with index #{idx}"
end
```

The Times Loop

This is the easiest loop you can work with.

Look at this code:

```ruby
10.times { puts "hello" }
```

This will print the word `"hello"` 10 times.

There isn't much to it & it should be easy to remember.

But what if you want the number?

In the last example, with the each loop, we had access to this n variable so we could print it.

You can also do that with `times`.

Example:

```ruby
10.times { |i| puts "hello #{i}" }
```

This will print `hello 0`, `hello 1`, `hello 2`, etc.

Give it a try!

The key here is the little `|i|` thing, which by the way, can be any valid variable name. It doesn't have to be an `|i|`. It could be `|n|` or `|foo|`, or `|bacon|`...

It's just a name!

If you are familiar with methods, this `|n|` is like a method parameter.

In other words, it's just a variable that becomes the current value for each iteration of our `times` loop.

Range Looping

You may have noticed that when using the `times` method it starts counting from 0.

This can be a bit inconvenient **if you want to start with a different number**.

You can use a range & the each method to have more control over the starting & ending numbers.

Example:

```ruby
(1..10).each { |i| puts i }
```

This will print all the numbers from 1 to 10.

Ruby While Loop

The while loop is available in most programming languages so it's always useful to know. It's also the kind of loop that you can fall-back to when everything else fails.

And there are situations when only a `while` loop would make sense. For example, if you don't know how many times you need to loop in advance.

Here's a code example:

```ruby
n = 0

while n < 10
  puts n
  n += 1
end
```

This will print all the numbers from 0 to 9 (10 excluded).

Notice that there are some important components:

- The n variable
- The condition (`n < 10`)
- The `n += 1`

All of these components are critical for this to work.

The variable n holds the value we are using for counting, the condition (`n < 10`) tells Ruby when to stop this loop (when the value of n is greater or equal to `10`), and the `n += 1` advances the counter to make progress.

If you forget to increase the counter in your while loop you'll run into a program that never ends.

An infinite loop.

Skipping Iterations

In all of these loop types you can skip iterations.

Let's say that you are going over an array of numbers & you want to skip odd numbers.

You could do something like this:

```ruby
10.times do |i|
  next unless i.even?

  puts "hello #{i}"
end
```

The key here is the `next` keyword, which skips to the next loop iteration (the next number in this case).

A better way to do this is to use other methods like `step` & `select`.

Example:

```ruby
(0...10).step(2) { |i| puts i }
```

How to Stop A Loop Early

You can also break out of a loop early, before the condition is met, or before you go over all the elements of the collection.

The following example stops when it finds a number higher than 10:

```ruby
numbers = [1,2,4,9,12]

numbers.each do |n|
  break if n > 10

  puts n
end
```

The key here is the Ruby break keyword.

Summary

You have learned many different ways to loop in Ruby!

Including the `times` method, the each method & the `while` keyword.

You have also learned how to control the loops by skipping iterations with `next` & breaking out of loops with `break`.

Conditional Statements

Like these:

- "If the room is too cold turn on the heater"
- "If we don't have enough stock of this product then send an order to buy more"
- "If this customer has been with us for more than 3 years then send him a thank you gift"

Things like that are what I mean by making decisions.

If something is true (the condition) then you can do something.

In Ruby, you do this using **if statements**:

```ruby
stock = 10

if stock > 1
  puts "Sorry we are out of stock!"
end
```

Notice the syntax. It's important to get it right.

The `stock < 1` part is what we call a "condition".

This is what needs to be true for the code inside the condition to work.

In **plain English** this is saying:

"If the value of `stock` is less than 1 then print the 'out of stock' message, otherwise do nothing."

Types Of Conditions

In the last example I'm using the "less than" symbol <, but there are other symbols you can use for different meanings.

Here's a table:

Symbol	Meaning
<	Less than
>	Greater than
==	Equals
!=	Not equals
>=	Greater OR equal to
<=	Less OR equal to

Notice that we use two equal == symbols to mean equality!

One equals sign = in Ruby means "assignment", make sure to use == when you want to find out if two things are the same.

If you don't this right you won't get the expected results.

Ruby Unless Statement

With an `if` statement you can check if something is `true`.

But when you want to check for "not true" there is **two things you can do**.

You can reverse the value with `!`:

```
if !condition
  # ...
end
```

Or you can use `unless`, which is like `if`, but it checks for "not true":

```
unless condition
  # ...
end
```

The If Else Statement

You can also say "if this is NOT true then do this other thing":

```
if stock < 1
  puts "Sorry we are out of stock!"
else
  puts "Thanks for your order!"
end
```

The else part is always optional, but it can help you write more advanced logic.

You can take this one step further & use an elsif statement:

```
if stock < 1
  puts "Sorry we are out of stock!"
elsif stock == 10
  puts "You get a special discount!"
else
  puts "Thanks for your order!"
end
```

With **elsif** you can say:

> "If `stock` is less than 1 print this message, else if stock equals 10 print this special message, otherwise if none of these are true then print the thank you message."

How to Use Multiple Conditions

If you'd like to write compound conditions, where you are checking if **two things are true at the same time**, then this section is for you.

You can do this by using the && operator:

```ruby
if name == "David" && country == "UK"
  # ...
end
```

This is saying:

"If the name is equal to 'David' and `country` is equal to 'UK' then do something."

You can also use the `||` operator:

```ruby
if age == 10 || age == 20
end
```

This means:

"If the age is `10` or `20` then do something."

Notice how these two operators (&&, `||`) allow you to **combine conditions**, but they need to be proper conditions.

In other words, you CAN'T do this:

```ruby
if age == 10 || 20
end
```

This is not valid.

You need a full condition on each side (`age == 10 || age == 20`).

Things to Watch Out For

Just before we end this lesson & want to mention a few problems you may run into & what to do about them.

The first is about comparing strings.

When comparing two strings **they must look exactly the same**!

Including the "casing".

This means that "hello" & "Hello" are different words.

You can solve this by **making them as equal as possible**:

```ruby
name   = "David"
expected_name = "david"

if expected_name.downcase == name.downcase
  puts "Name is correct!"
end
```

The key here is the `downcase` method on `name`.

By making both strings downcase you can make sure they'll match if they have the same content.

For example:

"David" becomes "david", and "david" stays "david".

Now both are "david" so you can compare them.

Special Symbols in Strings

Another problem you may come across with related to arrays is "special symbols".

These symbols are for things like new lines n & the tab key t.

The problem is when you try to compare two strings that look the same, but they have one of these special symbols.

To see these **special symbols** you will need to use the p method:

```
name = gets
p name
```

Try this code, type something in, and you will notice that name contains **the newline character** (which is not normally visible with puts).

To remove this character you can use the chomp method.

```
name = gets.chomp
p name
```

If Construct in One Line

It's possible to write an **if statement** using just one line of code.

Like this:

```
puts 123 if 2.even?
```

Which is the same as:

```
if 2.even?
  puts 123
end
```

This is **a shorthand version** which can be useful if you have a simple condition.

Is There an Alternative?

If you have an if else expression there is also a shorthand for that.

It's called the **ternary operator**:

```ruby
40 > 100 ? "Greater than" : "Less than"
```

I have another article where you can learn more about how this works & learn about other useful Ruby operators.

Read that, then practice & review what you learned today.

Take notes. Practice. Repeat.

A very important component of learning is repetition, so do this & master Ruby if statements to make progress in your journey of becoming a great Ruby developer.

I know you can do it.

Summary

Conditions allow you to take decisions in your code, this is what makes your program "think".

You also learned how to use the `if` statement & the `else` statement to handle different situations where you want to make decisions.

Finally, you learned about a few things to watch out for, like string "casing" & special symbols.

Now it's your turn to take action!

How To Convert Human Logic Into Computer Logic

If you studied (not just read once) the previous chapters of this tutorial now you have all the components you need to build Ruby programs.

To review:

- Variables
- Strings & Arrays
- If statements
- Loops

With these 4 things, plus Ruby built-in commands (we call these methods) you have all the power of Ruby in your hands.

But how do you build your own programs?

Where do you start?

Start with a plan!

Planning Your Ruby Project

I want you to get an idea of what you want to build.

What will this program do?

Start with something simple.

When you know what you want to do **write a list of steps in plain English**.

Here's an example:

Let's say that I have a folder full of mp3 files & I want to print them sorted by file size.

These are the steps I'd write:

1. Read list of mp3 into array
2. Sort list by file size
3. Print sorted list

It's like a recipe.

Now:

To turn these steps into code **you'll need to get creative** & use everything you have learned about Ruby.

Go over every step & make it into a "how" question:

- "How do I get a list of files in Ruby?"
- "How do I sort an array?"
- "How do I find the file size?"

You can ask Mr. Google for some help.

Tips For Effective Problem-Solving

Read these a few times until they sink in:

- You always want to build towards a solution, **one step at a time**.
- You can store temporary data in an array, string or hash to build this solution.
- Think like a car factory, every step the cars gets closer to completion. Every step has one job.
- Break down **big steps into smaller steps**.
- Remember previous exercises & try to draw parallels from that experience.
- Use irb to your advantage & test things in isolation.
- Work "inside out". Figure out how to make something work for one case then generalize with a loop.
- If you are writing code directly in a file **run your code often** so you can find errors early.
- **Details matter**! That missing parenthesis, a typo, or not having the right data type can cause your code to not work as you expect.

How to Handle Error Messages

It's completely normal to get error messages.

Error messages are there to help you.

Most of the time **the error message has information to help you fix the problem**.

There are two things you want to look for in an error message:

1. Where the error is located
2. What kind of error you're dealing with

If you get an error like this:

```
exception.rb:7:in 'bar': undefined local variable or method 'bacon'
    from exception.rb:3:in 'foo'
    from exception.rb:10:in 'main'
```

This tells you that the error started at the `exception.rb` file, on line 7.

That's a good place to start your investigation.

In this message you'll also find the error type:

```
undefined local variable or method 'bacon'
```

This means you tried to use something with the name of bacon, but Ruby **can't find anything with that name**.

It's possible that you made a typo, or maybe you forgot to create that variable.

Another error that's very common looks like this:

```
undefined method `foo` for nil:NilClass (NoMethodError)
```

This happens because you are trying to call a method on `nil`.

Many Ruby commands (methods is the more technical term) can return this value to you **when they can't find what you're asking for.**

Let's say you have an array:

```ruby
letters = ['a', 'b', 'c']
```

If you try to access it with an invalid index:

```ruby
letters[5]
```

Ruby allows you to do that, but you get `nil`.

If you try to do something with that value:

```ruby
letters[5].foo
```

You get the undefined method error you saw above.

One solution is to check if you're working with a non-nil value:

```ruby
if letters[5]
  letters[5].foo
end
```

If you find another error that you don't understand you can drop it into Google & you'll find some hints on how to fix it.

Finding Ruby Methods

When converting your steps into code it's useful to know what you're working with.

If you are starting with a string then string methods will be useful.

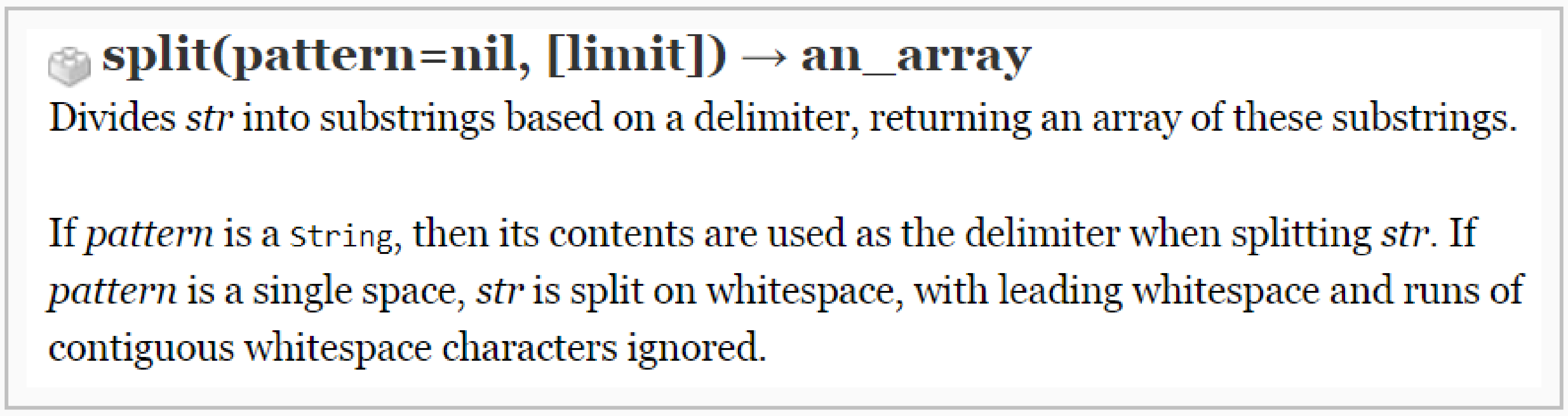
If you are starting with an array then look for an Array method that does what you want.

You can find all the available methods using the Ruby documentation.

For example:

If I have a string like "a,b,c,d" & I want to break it down into an array of characters without the commas.

I may notice the `split` method in the String documentation:



split(pattern=nil, [limit]) → an_array

Divides *str* into substrings based on a delimiter, returning an array of these substrings.

If *pattern* is a `String`, then its contents are used as the delimiter when splitting *str*. If *pattern* is a single space, *str* is split on whitespace, with leading whitespace and runs of contiguous whitespace characters ignored.

A quick test in irb reveals that this is what we were looking for!

If you don't know what kind of object you are working with you can use the `class` method.

Example:

```
"apple".class

# String
```

If you're working with a string, but you want to do something that only an array can do, or you're working with an integer but you want to work with the individual digits.

You can use these conversion methods:

Method	Conversion

Method	Conversion
to_i	String -> Integer
to_s	Integer -> String
chars	String -> Array (individual characters)
split	String -> Array (split by spaces by default)
join	Array -> String (join without spaces)

Both `split` & `join` take an optional parameter where you can specify the separator character.

Example:

```
"a-b-c".split("-")

# ["a", "b", "c"]
```

Ruby Gems

Sometimes what you want to do is more complicated than this.

You may want to pull down data from a website & find images to download.

In that case what you're looking for is a Ruby gem.

Ruby gems are small Ruby applications that you can add into your program & they'll help you do something.

For example:

- Nokogiri helps you read HTML code & extract information from it.
- Thor helps you write command-line applications.
- RSpec & Minitest help you write tests to check if your code is working correctly.

You don't need the gems to do these things, but they can save you a lot of work.

Object-Oriented Programming

You don't need OOP (Object Oriented Programming) to write code.

You could write any Ruby program without OOP.

But using OOP makes things easier if you're writing a non-trivial program.

Why?

Because OOP is all about how you design & organize your code.

You **create classes that represent concepts** in your program.

And each class is responsible for doing something.

For example, you may have a user class that knows the username & email address.

Or a Store class that knows how to manage its own inventory.

How to Create a Ruby Class

To create a class you'll need the `class` keyword.

Here's a Book class:

```ruby
class Book
end
```

This class does nothing, but **you can create objects from it**.

An object is an individual product of the class.

The class is the blueprint & the objects are the products.

Every object is different from each other.

Why?

Because this allows you to create many objects that are the same thing (like a Book) but have different data (title, author, etc.)

How to Create Ruby Objects

If you have a class you can create objects from it.

Here's how:

```ruby
book = Book.new
```

Notice the new method we are calling on Book.

That's how you create a Book object.

You could also do this with Ruby built-in classes.

Like `Array`:

```ruby
array = Array.new
```

But since these are built into Ruby we get special syntax to create them.

Example:

```ruby
array = []
hash  = {}
string = ""
```

For your own classes you have to use new to create objects.

Ruby Methods - How Classes Learn To Do Things

Let's make our Book class smarter & teach it how to do something.

You do this by writing methods.

A method is **a command that you can reuse multiple times** & it's associated with a specific class.

It's like variables but for code instead of data.

Here's an example:

```
class Book
  def what_am_i
    puts "I'm a book!"
  end
end
```

I've created a method named what*am*i.

When you call this method by its name it will print some text.

This is how you call a method:

```
book = Book.new

book.what_am_i
```

Important!

You call the method on an object of Book, not on Book itself.

These are "instance methods".

Another name for an object is "instance of a class".

Storing Data With Instance Variables

Objects can hold their own data.

Here's how:

```ruby
class Book
  def initialize(title, author)
    @title  = title
    @author = author
  end
end
```

There are a few things going on here:

First, we have this `initialize` method.

This is **a special Ruby method that is called when you create an object**.

Then we have these `@something` variables.

What are those?

Instance variables.

Why do we need them?

To store data that all the methods (in the same class) can use.

Without instance variables these variables wouldn't exist outside the `initialize` method.

Attribute Accessors

The last piece of the puzzle is **how to access this data** we prepared inside `initialize`.

There are two ways:

1. If you're inside an instance method you can reference it directly (`@variable_name`)
2. If you're outside the class you'll need an attribute accessor

Let's say we create two books:

```ruby
deep_dive = Book.new("Ruby Deep Dive", "Jesus Castello")
boba = Book.new("BOBA", "Grant Cardone")
```

If I want to access the title:

```ruby
deep_dive.title
```

I'm going to get this error:

```
NoMethodError: undefined method 'title'
```

Instance variables are private by default, that's why we can't get the title directly.

If you want access **you have to be explicit about it**.

You can do that using attribute accessors.

Like this:

```ruby
class Book
  attr_reader :title, :author

  def initialize(title, author)
    @title  = title
    @author = author
  end
end
```

This `attr_reader` is **like opening a window so people can peer into the object & get the data they want**.

Now if I do:

```ruby
deep_dive.author

# "Jesus Castello"
```

It works!

There is nothing special about this, an `attr_reader` is a shortcut for writing an accessor method like this one:

```ruby
def author
  @author
end
```

Because methods have access to **instance variables** they can return their value.

There are other kinds of attribute accessors:

- `attr_reader` (read-only)
- `attr_writer` (write-only)
- `attr_accessor` (read & write)

When you use `attr`*`writer or attr`*`accessor` you can change the value of instance variables outside the class.

Example:

```
deep_dive.title = "Ruby"
```

As a general rule you want to minimize outside access to instance variables, if you only need reading use `attr`*`reader`, not `attr`*`accessor`.

What is a Class Method?

Everything you have seen so far is about instances, objects created from the class (using new).

But it's also possible to create class-level methods.

Like this:

```ruby
class Food
  def self.cook
  end
end
```

The `self` part before the method name makes this a class method.

The `self` in this context refers to the class name (Food).

Now:

The difference between an instance method & a class method is that **instance methods are for objects**.

And class methods are for the class itself.

How to Use Class Methods

You call them like this:

```
Food.cook
```

Why would you want to create a class method?

Well, you don't have to.

Most of the time **you'll be working with instance methods**.

But sometimes it doesn't make sense to create an object.

Like when using the `Math` class in Ruby...

...this gives you the square root of 25:

```
Math.sqrt(25)
```

You don't need a `Math` object **because `Math` doesn't store any data**.

The methods in `Math` take one value & give you an answer.

That's the use-case for class methods.

Practice Time!

I want you to practice what you learned.

That's the only way you'll develop this skill & improve your understanding!

Try this:

- Create a `Cat` class
- Add a `meow` method to the `Cat` class that prints `"I'm a cat, gimme food & pet me now!"`
- Create a cat object & call the `meow` method 3 times

What's Next?

If you read the whole guide...

Congratulations!

You're on your way to becoming a professional Ruby developer.

There is a lot more to learn, but you put the initial work & planted the seeds.

Now:

You have to water these seeds every day.

Make sure to review everything & practice, practice, practice.

Thanks for reading!

- **Jesus Castello** (www.rubyguides.com)