**INTRODUCTION TO RUBY**

**Overview & Sneak Peek**

Ruby is a powerful, flexible programming language you can use in web/Internet development, to process text, to create games, and as part of the popular Ruby on Rails web framework. Ruby is:

* **High-level**, meaning reading and writing Ruby is really easy—it looks a lot like regular English!
* **Interpreted**, meaning you don’t need a compiler to write and run Ruby. You can write it here at Codecademy or even on your own computer (many are shipped with the Ruby interpreter built in—we’ll get to the interpreter later in this lesson).
* **Object-oriented**, meaning it allows users to manipulate data structures called objects in order to build and execute programs. We’ll learn more about objects later, but for now, all you need to know is *everything* in Ruby is an object.
* **Easy to use**. Ruby was designed by Yukihiro Matsumoto (often just called “Matz”) in 1995. Matz set out to design a language that emphasized human needs over those of the computer, which is why Ruby is so easy to pick up.

This course assumes no previous knowledge of Ruby in particular or programming/computer science in general.

**Instructions**

Ready to learn Ruby? Click the Next button to continue!

**script.rb**

# Welcome to Ruby!

**Variables**

One of the most basic concepts in computer programming is the **variable**. You can think of a variable as a word or name that grasps a single value. For example, let’s say you needed the number 25 from our last example, but you’re not going to use it right away. You can set a variable, say my\_num, to grasp the value 25 and hang onto it for later use, like this:

my\_num = 25

Declaring variables in Ruby is easy: you just write out a name like my\_num, use = to assign it a value, and you’re done! If you need to change a variable, no sweat: just type it again and hit = to assign it a new value.

**Instructions**

**1.**

Set the variable my\_num to the value 100, then click the Run button to run your code.

**script.rb**

my\_num = 100

# Write code above this line!

puts my\_num

**Math**

Ruby isn’t limited to simple expressions of assignment like my\_num = 100; it can also do all the math you learned about in school.

There are six arithmetic operators we’re going to focus on:

Addition (+)

Subtraction (-)

Multiplication (\*)

Division (/)

Exponentiation (\*\*)

Modulo (%)

The only ones that probably look weird to you are exponentiation and modulo. Exponentiation raises one number (the base) to the power of the other (the exponent). For example, 2\*\*3 is 8, since 2\*\*3 means “give me 2 \* 2 \* 2“ (2 multiplied together 3 times). 3\*\*2 is 9 (3 \* 3), and so on.

Modulo returns the remainder of division. For example, 25 % 7 would be 4, since 7 goes into 25 three times with 4 left over.

**Instructions**

**1.**

Do a little math practice in the editor. When you’re ready, click Next.

**script.rb**

first\_number = 2

second\_number = 3

result = first\_number + second\_number

puts result

**'puts' and 'print'**

The print command just takes whatever you give it and prints it to the screen. puts (for “put string”) is slightly different: it adds a new (blank) line after the thing you want it to print. You use them like this:

puts "What's up?"  
print "Oxnard Montalvo"

No parentheses or semicolons needed!

**Instructions**

**1.**

In the editor, use at least one print statement and at least one puts statement. You can print out any strings you like! (Make sure to put your strings between quotes, like this: "Hello!".)

**script.rb**

uts "Hello"

print "Andres "

print "Netanyahu"

## Everything in Ruby is an Object

Because everything in Ruby is an object (more on this later), everything in Ruby has certain built-in abilities called **methods**. You can think of methods as “skills” that certain objects have. For instance, strings (words or phrases) have built-in methods that can tell you the length of the string, reverse the string, and more.

We also promised to tell you more about the **interpreter**. The interpreter is the program that takes the code you write and runs it. You type code in the **editor**, the interpreter reads your code, and it shows you the result of running your code in the **console**.

### Instructions

Enough small talk. Click Next to start learning about string methods!

**script.rb**

name = 'Andres'

puts(name)

**The '.length' Method**

Methods are summoned using a .. If you have a string, "I love espresso", and take the .length of it, Ruby will return the length of the string (that is, the number of characters—letters, numbers, spaces, and symbols). Check it out:

"I love espresso".length  
# ==> 15

(That little line starting with the # isn’t part of what you need to write—it just shows you the output Ruby will provide. More on this in the next section!)

Taking the length of input can be useful if, for example, you want to make a website that takes credit card payments. Ruby can check to make sure the credit card number appears to be valid.

**Instructions**

**1.**

Call the .length method on your name (remember to use quotes around your name).

puts the answer to the console if you want to see the value.

**script.rb**

puts "Andres R. Bucheli".length

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**The '.reverse' Method**

The .reverse method is called the same way .length is, but instead of asking Ruby to tell you how long a string is, it spits out a backwards version of the string you gave it. For instance,

"Eric".reverse

will result in

"cirE"

Reversing input can be useful if you want to sort a list of values from highest to lowest instead of lowest to highest. (We’ll get to sorting in later lessons.)

**Instructions**

**1.**

Call the .reverse method on your name. Don’t forget those quotation marks around your name to make it a string!

puts the answer to the console if you want to see the value.

**script.js**

puts "Andres R. Bucheli".reverse

**'.upcase' & '.downcase'**

Let’s try one more method (er, *two* methods). As you might have guessed, the .upcase and .downcase methods convert a string to ALL UPPER CASE or all lower case, respectively.

**Instructions**

**1.**

Call .upcase on your name to make your name ALL CAPS and use puts to print it to the screen, like this:

puts "eric".upcase    # ==> ERIC

On the next line, call .downcase to make your name all lower case. Make sure to use puts so you can see it printed out!

**script.js**

puts "Jamie".upcase

puts "Jamie".downcase

**Single-Line Comments**

You probably saw us use the # sign a few times in earlier exercises. The # sign is for **comments** in Ruby. A comment is a bit of text that Ruby won’t try to run as code: it’s just for humans to read. Writing good comments not only clarifies your code for other people who may read it, but helps remind you of what you were doing when you wrote the code days, months, or even years earlier.

The # sign should come before your comment and affects anything you write after it, so long as you’re on a single line. (We’ll show you how to do multi-line comments in a second.) Check out these examples:

# I'm a full line comment!  
"Eric".length # I'm a comment, too!

The second example will return 4, since the comment comes after the code that Ruby will execute.

**Instructions**

**1.**

Write a comment in the editor. It can be anything you like!

**script.rb**

# This is a one-line comment

**Multi-Line Comments**

You can write a comment that spans multiple lines by starting each line with a #, but there’s an easier way. If you start with =begin and end with =end, *everything* between those two expressions will be a comment. Take a look at this example:

=begin  
I'm a comment!  
I don't need any # symbols.  
=end

Don’t put any space between the = sign and the words begin or end. You can do that with math (2 + 5 is the same as 2+5), but in this case, Ruby will get confused. =begin and =end also need to be on lines all by themselves, just as shown above.

**Instructions**

**1.**

Create a multi-line comment in the editor to the right. Make sure =begin and =end are on their own lines!

**script.rb**

=begin

This is a

multiline comment

=end

**Naming Conventions**

There are many different kinds of variables you’ll encounter as you progress through these courses, but right now we’re just concerned with regular old **local variables**. By convention, these variables should start with a lowercase letter and words should be separated by underscores, like counter and masterful\_method. Ruby won’t stop you from starting your local variables with other symbols, such as capital letters, $s, or @s, but by convention these mean different things, so it’s best to avoid confusion by doing what the Ruby community does.

**Instructions**

**1.**

Create a variable name in the editor and set it equal to your name as a string (between quotes, like this: "Eric"). Your string can be capitalized, but name should be all lower case!

Checkpoint 2 Passed

Hint

Remember how to set a variable to a string? You do it like this:

rb some\_variable = "Your string here!"

**script.rb**

name = "andres"

**Variables & Data Types**

Let’s quickly review how to declare and set variables. Remember, you declare a variable just by saying its name, and you set it using =. You can always check the **Hint** below if you need more help.

**Instructions**

**1.**

Create a variable called my\_name and set it equal to your name as a string. Create a second variable called my\_age and set it equal to your age as a number (don’t use any quotes around the number).

Checkpoint 2 Passed

Hint

Remember, variable declaration looks like this:

variable\_name

Setting a variable looks like this:

rb name = "Eric" age = 26 hungry = true

**script.rb**

my\_name = "Andres"

my\_age = 45

**Math**

Good! Now let’s do a little math.

**Instructions**

**1.**

In the editor, set the variables:

* sum equal to 13 + 379
* product equal to 923 \* 15
* quotient equal to 13209 / 17

Print each variable to the console using puts if you want to see the results.

**script.rb**

sum = 13 + 379

product = 923 \* 15

quotient = 13209 / 17

puts sum

puts product

puts quotient

**Strings and String Methods**

Well done! Let’s do a little review of string methods. Remember, you call a method by using the . operator, like this: "string".method.

**Instructions**

**1.**

Declare a variable name and set it equal to a string containing your name. Call .downcase on your name to make it all lower case, call .reverse on your lowercase name to make it backwards, then call .upcase on your backwards name to make it ALL CAPS.

In Ruby, you can do this two ways: each method call on a separate line, or you can *chain them together*, like this:

name.method1.method2.method3

puts the method call if you want to see the result.

Checkpoint 2 Passed

Hint

For instance, if your name is “Steve,” you might type

name = "Steve".downcase

to convert your name to "steve".

**script.rb**

name = "Andres".downcase.reverse.upcase

puts name

**Comments**

Great work! For our last review exercise for this lesson, let’s go over single- and multi-line comments.

**Instructions**

**1.**

Create a single-line comment and a multi-line comment in the editor to the right. (Check the Hint if you need a reminder on the syntax.) Hit Run to see your glorious comments and complete the Introduction to Ruby!

Checkpoint 2 Passed

Hint

Remember, a single-line comment is created using #, and a multi-line comment starts with =begin (no space between = and begin!) and ends with =end. =begin and =end need to be on their own lines.

**script.rb**

# This is a single line comment

=begin

This is a

multiline comment

=end