

Getting Started

Create a git repository on GitHub, called "ruby_fundamentals1" or something similar. Clone it onto your own computer. This assignment will walk you through creating several Ruby programs which you should add to your git repository. Don't forget to commit after each exercise!

Disclaimer

This assignment is about walking you through the fundamental features of the Ruby language using short, simplified code examples. If you find yourself wondering "why would I ever write this code?" or "ok, but what is this thing *for*?", try not to panic! Today's assignment is about discovering what is possible in Ruby, not why those features are useful. However, if you can start to envision how you might use these features in more complicated programming scenarios, that's great!

Moreover, if you finish a section but felt like you didn't quite understand, it's very helpful to experiment and explore on your own.

Programming Languages

Ruby is a programming language, and like every other programming language, you can use it to command your computer. A very wide range of programming languages exists and many are tailored to work best in specific domains.

Perhaps you've heard of some other popular programming languages, such as Java, C++, Python, Objective-C, or JavaScript?

Every programming language has its own unique syntax, rules, pros, and cons.

Running Ruby Programs

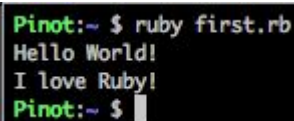
When writing Ruby programs, programmers work text editors and save their code in files.

The file extension used to indicate that a file is a Ruby program is `.rb`

Imagine we have a file called `first.rb`. To "run" or "execute" the program, run this in your terminal:

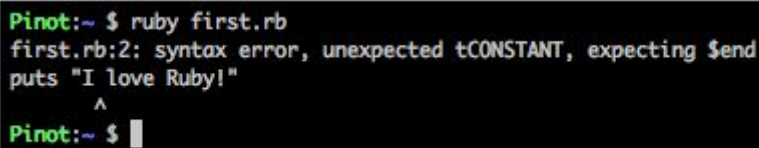
```
ruby first.rb
```

This will run the program, output any results and return to the command prompt when it's done.

A terminal window with a black background and green text. The prompt is 'Pinot:~ \$'. The command 'ruby first.rb' has been entered and executed. The output is 'Hello World!' followed by 'I love Ruby!' on the next line. The prompt is now 'Pinot:~ \$' with a cursor.

```
Pinot:~ $ ruby first.rb
Hello World!
I love Ruby!
Pinot:~ $
```

If there are any errors, the output will look something like this

A terminal window with a black background and green text. The prompt is 'Pinot:~ \$'. The command 'ruby first.rb' has been entered and executed. The output is an error message: 'first.rb:2: syntax error, unexpected tCONSTANT, expecting \$end' followed by 'puts "I love Ruby!"' on the next line. An arrow points to the end of the line. The prompt is now 'Pinot:~ \$' with a cursor.

```
Pinot:~ $ ruby first.rb
first.rb:2: syntax error, unexpected tCONSTANT, expecting $end
puts "I love Ruby!"
      ^
Pinot:~ $
```

Exercise 1

Create a file called `exercise1.rb` and open it in your text editor. Write this code:

```
2 + 3
```

and save the file. Now, let's run the file by typing in your terminal:

```
ruby exercise1.rb
```

Nothing happened, right?

Ruby won't print out anything unless we explicitly tell it to. Let's edit our file and change it to:

```
puts 2 + 3
```

and run `ruby exercise1.rb` again.

Did it print 5?

`puts` is for displaying messages. Lets add some more lines at the beginning of our program so it looks like the following, and then go ahead and run it:

```
puts 2
puts 3
puts 2 + 3
```

We have just created and run our first multi-line Ruby program. As you can see, we have written each Ruby statement on its own line. Try writing everything on the same line and running it again, like so:

```
puts 2 puts 3 puts 2 + 3
```

Oh no!

```
exercise1.rb:1: syntax error, unexpected tIDENTIFIER, expecting $end
puts(2) puts(3) puts(2 + 3)
           ^
```

We got an error, but there's no need to panic. Error messages are Ruby's way of telling us what's wrong. In this case Ruby couldn't understand our program with every statement on the same line. We can simply put it back the way it was: with one "statement" per line. Run it again to confirm that it's fixed.

Now that you have written a working program in Ruby, make sure to commit it to git. Remember to check the output of `git status` after every git command to verify that everything is as it should be. Feel free to review the [Github cheat sheet](#) at any time.

irb

Irb (which stands for "interactive Ruby") is a program that allows you to run Ruby statements within the terminal instead of writing them in a file.

Run the command `irb` to start it within your terminal:

```
Pinot:~ $ irb
irb(main):001:0> 
```

Now you can type in some code and irb will automatically output the result. As a simple example, type in `1 + 1` and press enter. irb will respond by printing 2.

You can type `exit` at any time to return back to your original terminal.

Try out irb

Start irb and try running each of these commands, one at a time. Make sure you type them out yourself rather than copying and pasting. You'll learn much more that way.

```
5 + 1
```

```
5+1
```

```
5+ 1
```

```
1
```

```
5
```

As you can see, Ruby can do math. Also, Ruby (generally) doesn't care that much about spaces. Whitespace refers to spaces, tabs, and blank lines, and in most cases they don't make a difference to Ruby. Try out some more commands:

```
3 + 7 + 1
```

```
5 * 3 # multiplication uses the asterisk (*) operator, not the letter x
```

What's happening in that last line of code? If you write a pound/number sign (#) Ruby ignores everything that comes after it until the end of the line. So you can (and should) use it to write useful comments throughout your code. Comments make it easier for you to understand your code when you come back to it in future and no longer remember how it works. They also serve the same purpose for other people who might be trying to read your code.

Using irb to experiment with bits of Ruby code as you work on future assignments and projects is an excellent habit to get into!

Basic Data Types

Data types allow us to represent different kinds of information. Let's look at some basic Ruby data types.

Numbers

Numbers without decimal points (eg. 1, 250, 99999) are called integers, and numbers with decimal points (eg. 1.5, 150.3985, 50.0) are usually called floating-point numbers or, more simply, floats.

Doing operations with numbers is simple. Fire up `irb` in your terminal and try out the following:

```
# Basic arithmetic -----
```

```
5 + 3
```

```
5 - 3
```

```
5 * 3
```

```
5 / 3
```

```
5 / 3.0
```

```
5 % 3 (modulo/remainder)
```

```
# Comparisons -----
```

```
5 > 3
```

```
5 < 3
```

```
5 > 5
```

```
5 >= 5
```

```
2 == 2 # note: two equals symbols, not one
```

```
2 == 3
```

```
2 != 3
```

Ruby has arithmetic operators such as `+`, `-`, `*`, `/`, `%`, and comparison operators such as `>`, `<`, `>=`, `<=`, `==` and `!=` (not equal). You must *always* use *two* equals signs when doing a comparison.

The [list of Ruby operators](#) is extensive. Feel free to try some other ones out in irb.

Strings

Strings are sequences of characters between quotation marks. This is the data type that allows you to incorporate words and sentences in your programs.

Remember `puts` from [exercise 1](#)? "Puts" is short for "put string".

To create a string, type some characters between single or double quotes. Below is an example of how to print strings using either single or double quotes and `puts`:

```
puts 'Hello world'
puts "Hello world"
```

So what difference is there between single quotes and double quotes in Ruby? In the above example, there's no difference. However, consider the following code:

```
puts "Betty's pie shop"
puts 'Betty\'s pie shop'
```

Because the word "Betty's" contains an apostrophe, which is the same character as the single quote, in the second line we need to use a backslash to escape the apostrophe so that Ruby understands that the apostrophe is part of the string and not marking the end of the string. The combination of the backslash followed by the single quote is called an escape sequence.

Using double quotes for this string allows us to avoid having to use an escape sequence.

Single Quotes

Single quotes only support two escape sequences.

`\'` single quote

`\\` single backslash

Except for these two escape sequences, all other characters between single quotes are treated literally.

Double Quotes

Double quotes allow for many more escape sequences than single quotes. They also allow you to embed Ruby code inside of a string – this is commonly referred to as interpolation.

String Interpolation

String interpolation is a means of embedding Ruby code into a string by wrapping it in special characters like so: `#{code goes here}`. Try out the example below:

```
puts "Ada Lovelace lived for #{1852 - 1815} years."
```

```
# Ada Lovelace lived for 37 years.
```

Escape Sequences

Below are some of the more common escape sequences that can appear inside of double quotes:

`\"` double quote

`\\` single backslash

`\a` bell/alert

`\b` backspace

`\r` carriage return

`\n` newline

`\s` space

`\t` tab

Try out this example code to better understand escape sequences:

```
puts "Hello\t\tworld"
puts "Hello\b\b\b\b\bGoodbye world"
puts "Hello\rStart over world"
puts "1. Hello\n2. World"
```

Strings can also work with some arithmetic operators (+, -, *, etc) and comparison operators (>, <=, ==, etc). Try a few in irb to see what works and what doesn't.

Symbols

Symbols are similar to strings, but come with more limited behaviour in exchange for performance benefits. They are represented by a word with a colon in front it, such as:

`:code`

If you're interested there are [many articles](#) online that go further into the topic. For now it's not important that you understand all the nuances of the difference between symbols and strings in Ruby, but you need to be aware of both data types.

Booleans

In Ruby, boolean data types allow us to represent the concepts of true and false. Boolean expressions are very common in programming and allow computers to evaluate statements as being either true or false.

Boolean expressions work with logical operators:

`&&` and

`||` or

`!` not

Exercise: Try the following examples for yourself in irb.

```
irb(main):001:0> true && true
=> true
irb(main):002:0> true && false
=> false
irb(main):003:0> true && !false
=> true
irb(main):004:0> true || false
=> true
irb(main):005:0> true || true
=> true
irb(main):006:0> false || true
=> true
irb(main):007:0> false || false
=> false
irb(main):008:0> !false || false
=> true
```

You can also try combining comparison and logical operators like so:

```
(1 < 3) && (3 < 5)
# true
```

```
(1 > 1) && (2 > 2)
# false
```

```
(1 == 2) || (2 < 3)
# true
```

```
(1 != 2) || (2 < 3)
# true
```

```
(1 == 2) || (2 == 3)
#false
```

The purpose of boolean logic will become clearer later on when we talk about [control structures](#).

Exercise 2

Create a file called `exercise2.rb` and in it enter the solution for the four problems below, then commit. Try annotating your code by leaving comments (using the `#` symbol) in the file before each of your answers to the following questions:

How would you calculate a good tip for a 55 dollar meal? Use `puts` to print the answer.

Try adding a string and an integer with the `+` operator. What happens? Find a way to convert the integer into a string first and use `puts` to print the result.

Try outputting the result of 45628 multiplied by 7839 in a sentence by using [string interpolation](#).

What's the value of the expression `(10 < 20 && 30 < 20) || !(10 == 11)`? Try figuring it out on your own before typing it in.

Variables

To store a number or a string in your computer's memory for use later in your program, you need to assign it to a variable, like so:

```
my_variable = 'my_variable now contains this string'
```

We can now refer to that variable whenever we want to access that string. Try the following in irb:

```
name = "Sandra"
greeting = "Hello #{name}! It's good to see you again."
mission = "Your mission, should you choose to accept it..."

puts "#{greeting} #{mission}"
```

Variables and Boolean Logic (together at last)

```
my_number = 12
my_number > 10
# true
```

```
my_number < 10
# false
```

```
your_number = 1
```

```
my_number == your_number
# false
```

```
my_number != your_number
# true
```

Exercise: Try out the following example in irb to get more familiar with how Ruby deals with assignment:

```
amount = 20
new_amount = amount
new_amount          # 20
```

```
amount = "twenty"
```

```
amount          # "twenty"
new_amount      # 20
```

In the above example, we set `amount` to the value `20`.

We then set `new_amount` to `20` (because `amount -> 20`).

We then decide to change `amount` to contain the value `"twenty"` instead, but we haven't changed `new_amount`.

Try some more examples:

```
animal = "cats"
number = 20
location = "the yard"
```

```
"There are #{number} #{animals} in #{location}!"
```

```
who = "Mrs. Peacock"
where = "the library"
what = "rope"
```

```
accusation = "It was #{who} in #{where} with the #{what}."
```

accusation

Operator and Assignment Shorthand

We can do calculations with variables without changing their values:

```
counter = 25
counter + 1
counter          # counter is still 25
```

We are not actually assigning a new value to `counter`. We're simply calculating the sum of 1 and the value in `counter`.

Alternatively we can reassign `counter` to the result of that calculation:

```
counter = counter + 1
counter          # counter is now 26
```

Programmers are obsessed with efficiency, even when it comes to typing, which means most programming languages contain a lot of typing short cuts. Combining operators and variable reassignment is a commonly used shortcut. For example:

```
counter += 1
```

is the same as `counter = counter + 1`.

but takes nine fewer characters(!!!) to type. Try this out in irb to see the value of `counter` change. Then try it with different variables and different amounts.

+= and -=

`+=` is the combination of the addition and assignment operator. It adds the value on the right-hand side to the current value of the variable on the left-hand side and assigns the result to that variable. For example:

```
amount = 1
amount += 10
amount          # 11
```

`--` is the combination of the subtraction and assignment operator. It subtracts the value on the right-hand side from the current value of the variable on the left-hand side and assigns the result to that variable. For example:

```
amount = 30
amount -= 5
amount      # 25
```

Exercise 3

Let's make a Ruby program that greets someone by name. Let's call it `exercise3.rb`.

Start with displaying a question:

```
puts "What is your name?"
```

Run your program to verify that it works so far. If it works, commit what you've got to git with a meaningful commit message.

Getting User Input

The next step is to get input from your hypothetical user (which for now is just you). We can do that with `gets` (which stands for "get string"). `gets` will pause the execution of your program and give your user the chance to type something into their terminal. When the user finishes typing and hits "enter", the value that they typed in is returned by `gets` and your program resumes normal execution. Try assigning `gets` to a variable in order to save your user's input.

```
puts "What is your name?"
user_name = gets
puts "Hello, #{user_name}"
```

Having that string in a variable allows us to display it back to the user later on.

```
gets VS gets.chomp
```

You may have also seen `gets.chomp` in other Ruby tutorials. `chomp` removes the unwanted line break (AKA new line AKA enter) character from the end of your user's input. In `irb`, try using both `gets` on its own as well as `gets.chomp` and look for the difference between the values that they return.

Don't forget to commit your work again!

Now try asking your user how old they are and have your program output what year they were born in.

PROTIP: Keep an eye out to ensure you have the proper data type. You might need to convert the string returned by `gets` to ensure you have a number you can perform math operations on by using [`to_i`](#).

Control Structures

if

`if` statements can be used to manage a program's "control flow", allowing you to either execute or skip a block of code based on a condition that evaluates to `true` or `false` (remember boolean values?). The syntax looks like this:

```
if my_number > 1
  puts "The number is greater than 1"
end
```

if/else

If you want to provide two different blocks of code for your `if` statement to choose between — ie. "do this thing or else do this other thing" — you can tack an `else` statement on to the end of your `if` statement, like so:

```
number = gets.to_i # the user types in a number

if number > 0
  puts "#{number} is positive" # this line executes if the user enters a
  positive number
else
  puts "#{number} is negative" # this line executes if the user enters a
  negative number
end
```

elsif

You can add additional options to your `if/else` statement using `elsif`:

```
x = gets.to_i
y = gets.to_i

if x > y
  puts "x is greater than y!"
elsif x < y
  puts "x is less than y!"
else
  puts "x equals y!"
end
```

unless

You may find yourself expecting a boolean expression to be false rather than true. Instead of writing:

```
if x != 10
  puts "I get printed!"
end
```

You can instead use `unless`, which is equivalent to "if not":

```
unless x == 10
  puts "I get printed!"
end
```

It accomplishes the same thing, but now it reads more like English!

User Input and Conditionals (together at last)

Exercise 4

Create new `.rb` files for each of the following challenges:

Ask the user to enter a number. Use an `if` statement to print "that's a big number!" if the number is 100 or more, or "why not dream a little bigger?" otherwise.

Ask the user to enter their age, and then display a message telling them how many years apart in age you are from them. If they enter a number larger than 105, print "I'm not sure I believe you".

Save your name as a string into a variable, then ask the user to enter their name. If the two names match, print "We have the same name!".

Ask the user to enter their name. If their name is longer than 10 letters, print "hi, " and then their name. If their name is less than 10 letters, print "hello, " and then their name. If their name is exactly 10 letters, print "hey, " and then their name.

Pick a number and save it in a variable called `secret_number`. Ask the user to enter a number. If they enter the secret number, print "You win!". If they are off by 1, print "So close!". Otherwise, print "Try again".

Loops

Ruby includes a `while` loop that will execute a block of code over and over until its condition is no longer true.

while

```
while true
  puts "I'm an infinite loop!"
end
```

```
# this program will never finish running because the condition given to the
while loop will never stop being true
```

```
counter = 1
```

```
while counter < 4
  puts "counter currently at #{counter}."
  counter += 1 # increase the value of counter by 1
end
```

```
#counter currently at 1.
#counter currently at 2.
#counter currently at 3.
```

until

You may also want to have a loop execute as long as the given condition is false. In this case you can use an `until` loop, which is equivalent to "while not":

```
until false
  puts "I'm an infinite loop!"
end
```

```
# this program will never finish running because the condition given to the
```

```
while loop will never stop being false

counter = 3

until counter == 0
  puts "counter currently at #{counter}."
  counter -= 1
end

#counter currently at 3.
#counter currently at 2.
#counter currently at 1.
```

Exercise 5

You decide to get some exercise and fresh air, but you want to keep track of how far from home you are.

Ask the user for input on what action to take - walk or run. If they walk, the total distance should go up by one, and you should update the user on their total distance traveled as follows:

"Distance from home is 6 km."

If they run, their total distance should go up by 5. Your program should keep asking for input - you don't know where you're going until you get there! Each time, you should print the total distance traveled.

```
Would you like to walk or run?
$ walk
Distance from home is 1km.
Would you like to walk or run?
$ walk
Distance from home is 2km.
Would you like to walk or run?
$ run
Distance from home is 7km.
Would you like to walk or run?
$ run
Distance from home is 12km.
```

Suggestions:

- Break this problem down into smaller pieces. What do you know how to do? Start with that!
- Read the problem very carefully. If the question uses the word 'if', you almost certainly need an `if` statement!
- You can press CTRL-C to end your program if it keeps asking you for input.

Exercise 6.1

Allow the user to go home when they are done exercising. The program should stop asking for input if the user enters 'go home'.

See if you can also make the program tell the user when they have entered a command that does not exist.

Exercise 6.2

You started the day with energy, but you are going to get tired as you travel! Keep track of your energy.

If you walk, your energy should increase. If you run, it should decrease. Moreover, you should not be able to run if your energy is zero.

...then, go crazy with it! Allow the user to rest and eat. Do whatever you think might be interesting.

Congrats for making it this far! You're done for today. :)

Submitting

When you're done the exercises and have pushed your work to Github