Ruby: Building Blocks

Juan “Harek” Urrios
@xharekx33

What’s Ruby?

Ruby is a dynamic, object-oriented, general-purpose programming language designed and developed by Yukihiro "Matz" Matsumoto.

Matz designed ruby so we could "be productive, enjoy programming, be happy".

For Matz, code should be treated as an essay, meant to be easily read and understood by human beings. He wants us to write “Beautiful Code”

IRON HACK

What’s “Beautiful Code”?

Brevity: “Succinctness is power”, Programs should ideally contain no unnecessary information. Eliminate Redundancy …

DRY: Don’t Repeat Yourself.

Simplicity: If a program is hard to understand, it can’t be beautiful. Obscure code leads to bugs, mistakes & confusion.

Flexibility: “Freedom from enforcement of tools”. Computers should serve programmers to maximise their productivity & happiness but these tools often increase the burden instead of lightening it.

IRON HACK

Get started writing Ruby code

Once you've installed Ruby in your machine (https://www.ruby-lang.org/en/documentation/installation/) there are two main ways to run your Ruby code:

- Saving code to a file and executing it with the ruby command
- Using IRB (the Interactive Ruby Shell)

IRON HACK

The Ruby command

If you create a new file and put some rails code inside, you can run it using the `ruby` command

```
$ echo 'puts "Hello World!"' > helloworld.rb
$ ruby helloworld.rb
Hello World!
```

IRB

The Interactive Ruby Shell lets you run Ruby commands in real-time and see the response immediately.

```
$ irb
2.2-head :001 > puts "Hello World!"
Hello World!
 => nil
2.2-head :002 >
```

@xharekx33

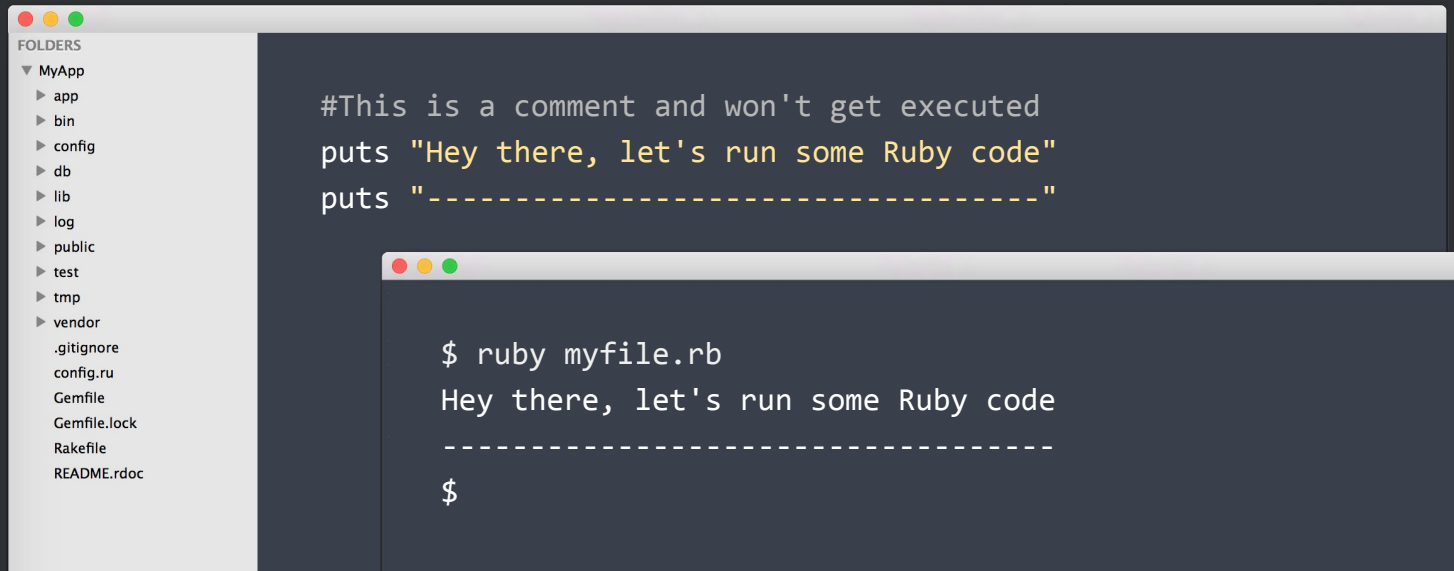
Printing stuff to the screen

To make a program print a message to the screen we can use `puts "My awesome message"` or `print "My other awesome message"`

```
$ irb
2.2-head :001 > puts "My awesome message"
My awesome message
 => nil
2.2-head :002 > print "My other awesome message"
My other awesome message => nil
2.2-head :003 >
```

Adding comments

You can use comments to take notes or temporarily remove code from your program's execution without actually deleting anything.



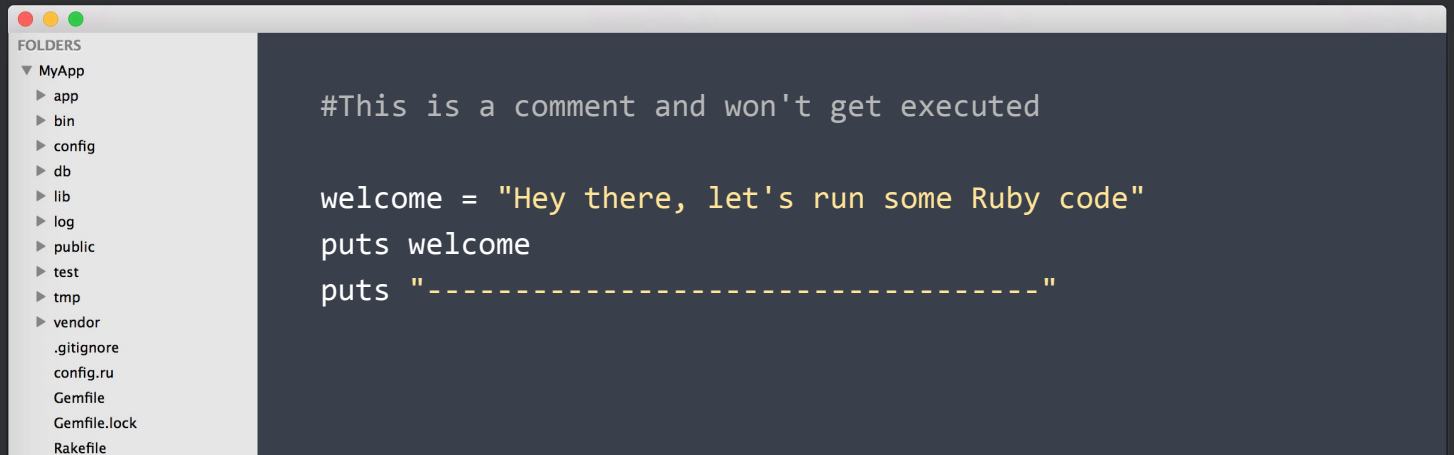
IRON HACK

@xharekx33

Variables

You can use variables to easily store and access values to use in your program. Variables can store any kind of values.

Assign them using the equal sign = and retrieve their value using their name.



IRON HACK

@xharekx33

Everything is an object

You will learn more about object oriented programming later. For now just keep in mind that every value has methods attached to it.

One of them is the `class` method that all objects have and this will let us know what type of object they are.

```
$ irb
2.2-head :001 > "Hey there".class
 => String
2.2-head :002 > 74.class
 => Fixnum
2.2-head :003 > 75.02.class
 => Float
```

Strings

To store text values we use `Strings`. We can easily create them using double or single quotes.

```
$ irb
2.2-head :001 > str = "foo"
 => "foo"
2.2-head :002 > str2 = 'bar'
 => "bar"
2.2-head :003 > str3 = String.new("foobar")
 => "foobar"
```

Strings: length

Now that we have created strings we can use their methods, such as `length` and check how long they are.

```
$ irb
2.2-head :001 > str = "foo"
 => "foo"
2.2-head :002 > str2 = 'foobar'
 => "foobar"
2.2-head :003 > str.length
 => 3
2.2-head :005 > str2.length
 => 6
```

Strings: concatenation

Strings can be stitched together using the plus sign +.

This is called String `concatenation`.

```
$ irb
2.2-head :001 > "foo" + "bar"
 => "foobar"
2.2-head :002 > str = "foo"
 => "foo"
2.2-head :003 > str2 = 'bar'
 => "bar"
2.2-head :004 > str + str2
 => "foobar"
```

Strings: concatenation

This is normally used when there are variables involved. Concatenating string literals is not really that useful.

```
$ irb
2.2-head :001 > name = "Jude"
 => "Jude"
2.2-head :02 > puts "Hey " + name + ", don't make it bad"
Hey Jude, don't make it bad
 => nil
```

Strings: interpolation

A better way to insert values into a string is to use `interpolation`. In Ruby that's done using `#{}` inside it.

```
$ irb
2.2-head :001 > name = "Jude"
 => "Jude"
2.2-head :02 > puts "Hey #{name} don't make it bad"
Hey Jude, don't make it bad
 => nil
```

Strings: single and double quoted

Ruby won't interpolate into single-quoted strings. Single quoted strings are literal strings and will print whatever is inside exactly as is:

```
$ irb
2.2-head :001 > name = "Jude"
 => "Jude"
2.2-head :02 > puts "Hey #{name} don't make it bad"
Hey Jude, don't make it bad
 => nil
2.2-head :03 > puts 'Hey #{name} dont make it bad'
Hey #{name} dont make it bad
 => nil
```

Watch out!

Strings: escaping special characters

When using double quoted strings you can escape special characters such as quote marks, newlines and tabs and format your output.

```
$ irb
2.2-head :001 > puts "Joe said \"Hey there\" with a smile"
Joe said "Hey there" with a smile
 => nil
2.2-head :02 > puts "This\sis\n\tgetting\n\t\tmore\sand more
tabbed\n\t\t\ton every\sline"
This is
     getting
          more and more tabbed
               on every line
 => nil
```

Strings: common methods

Here are some common methods you can use on Strings. Check the ruby doc (http://ruby-doc.org/core-2.2.3/String.html) for more.

```
$ irb
2.2-head :001 > "My string".downcase
 => "my string"
2.2-head :02 >  "My string".upcase
 => "MY STRING"
2.2-head :03 > "My string".include? "ring"
 => true
2.2-head :04 > "My string".include? "Ring"
 => false
2.2-head :05 >"My string".gsub("string", "replaced")
 => "My replaced"
```

Integers: (Fixnum & Bignum)

You can create Integers by simply writing a number. To store them we use different classes, such as `Fixnum` and `Bignum`, and use their methods and do math

```
$ irb
2.2-head :001 > 7.class
 => Fixnum
2.2-head :002 > 15241578750190521524157875019052 11.class
 => Bignum
2.2-head :003 > 9.even?
 => false
2.2-head :004 >  5*9
 => 45
2.2-head :005 > 5/9
 => 0
```

@xharekx33

Integers: division

As you can see 5/9 should have returned a decimal number, but since we are working with integers the result is omitting the fractional-part

```
$ irb
2.2-head :001 > 5/9
 =>  0
2.2-head :002 > 9/5
 => 1
```

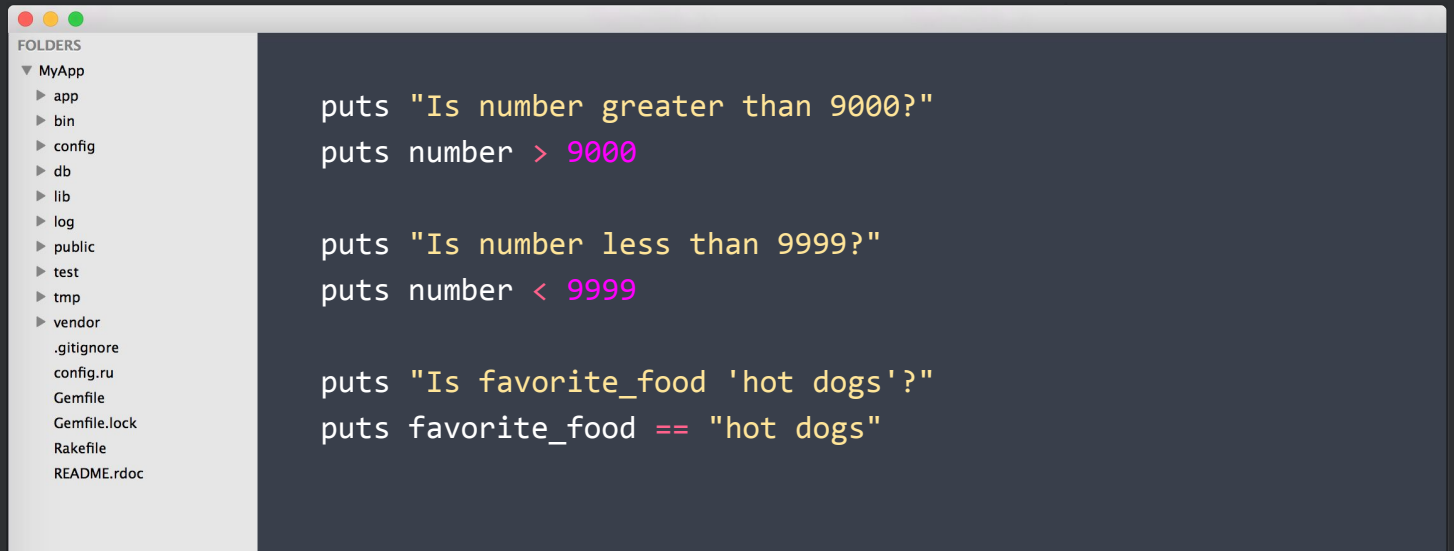
Floats

Decimal numbers are called floats in programming (floating point numbers). Create them writing a number with a fractional part.

```
$ irb
2.2-head :001 > 7.5.class
 =>  Float
2.2-head :002 > 9.0/5
 => 1.8
2.2-head :003 > a = 1.234567.round(2)
 => 1.23
2.2-head :003 > a + 5.2
6.43
```

Boolean expressions: >, <, ==

Expressions that can only return `true` or `false` are called Boolean expressions. We use this logical statements to test data to see if it is greater than: `>`, less than: `<` or equal to: `==` other data

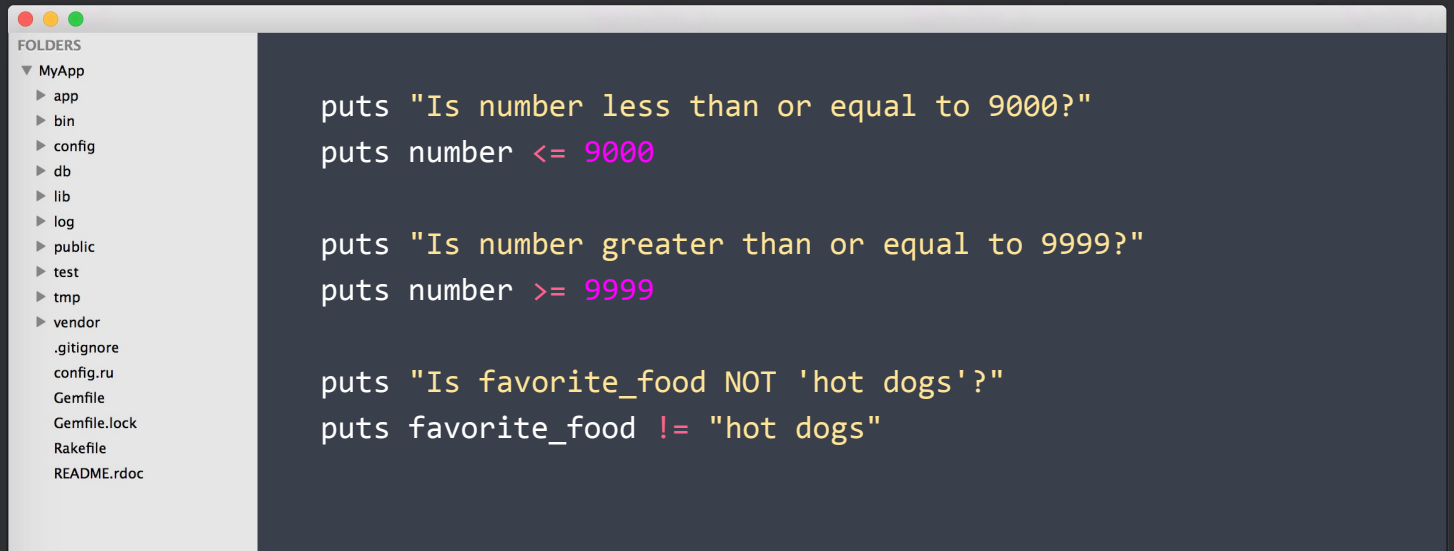


IRON HACK

@xharekx33

Boolean expressions: <=, >=, !=

We can also check the opposite: less than or equal to: <=, greater than or equal to: >=, not equal to: !=

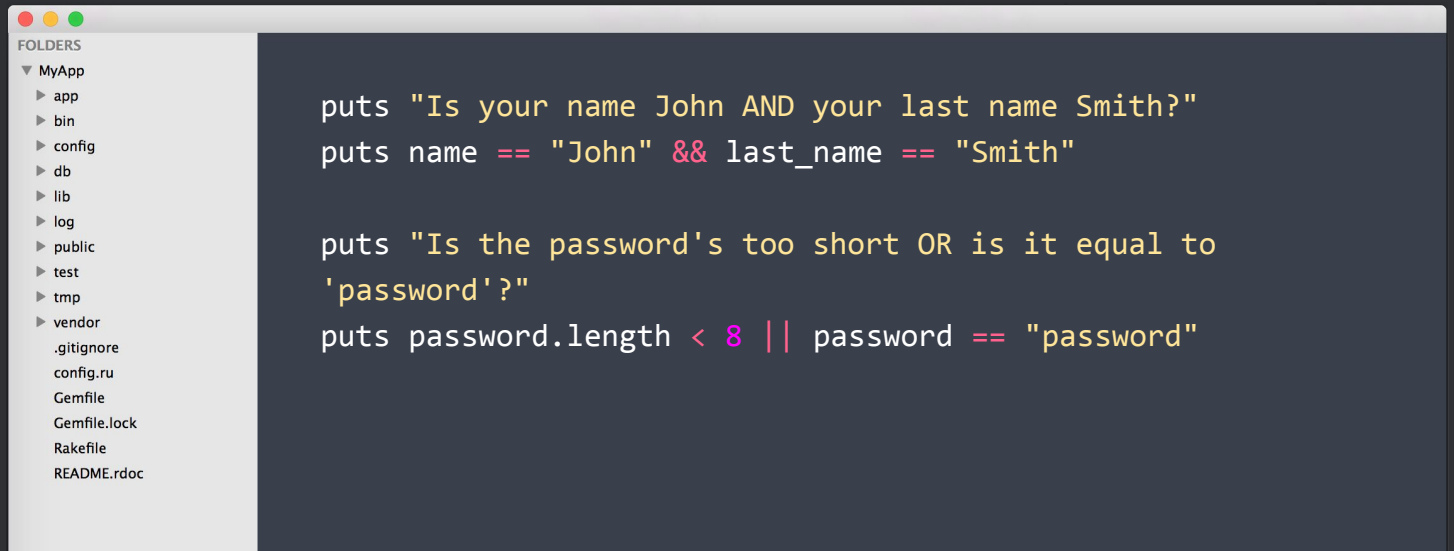


IRON HACK

@xharekx33

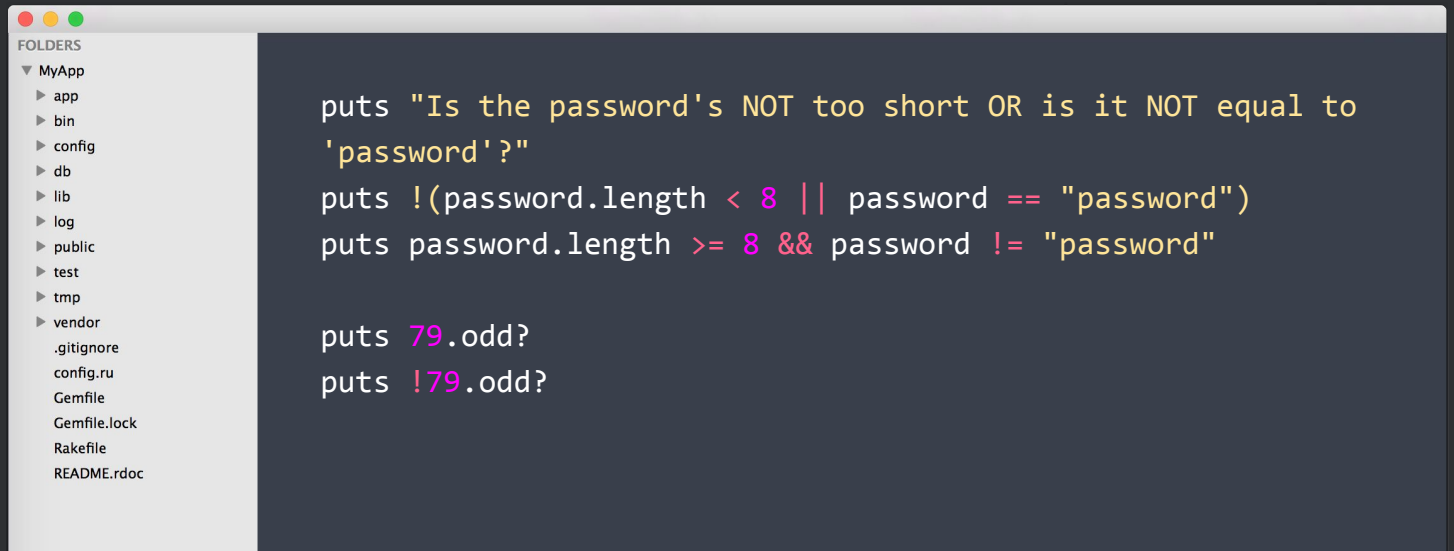
Combining boolean expressions: &&, ||

You can use && (and) to make a condition more specific by adding more constraints or use || (or) to make a condition more general by adding more options.



Negating boolean expressions: !

Remember the difference between == and != ?. You can use ! to negate complete boolean expressions. That makes thinking some complex expressions easier sometimes and can be used with methods that return booleans.

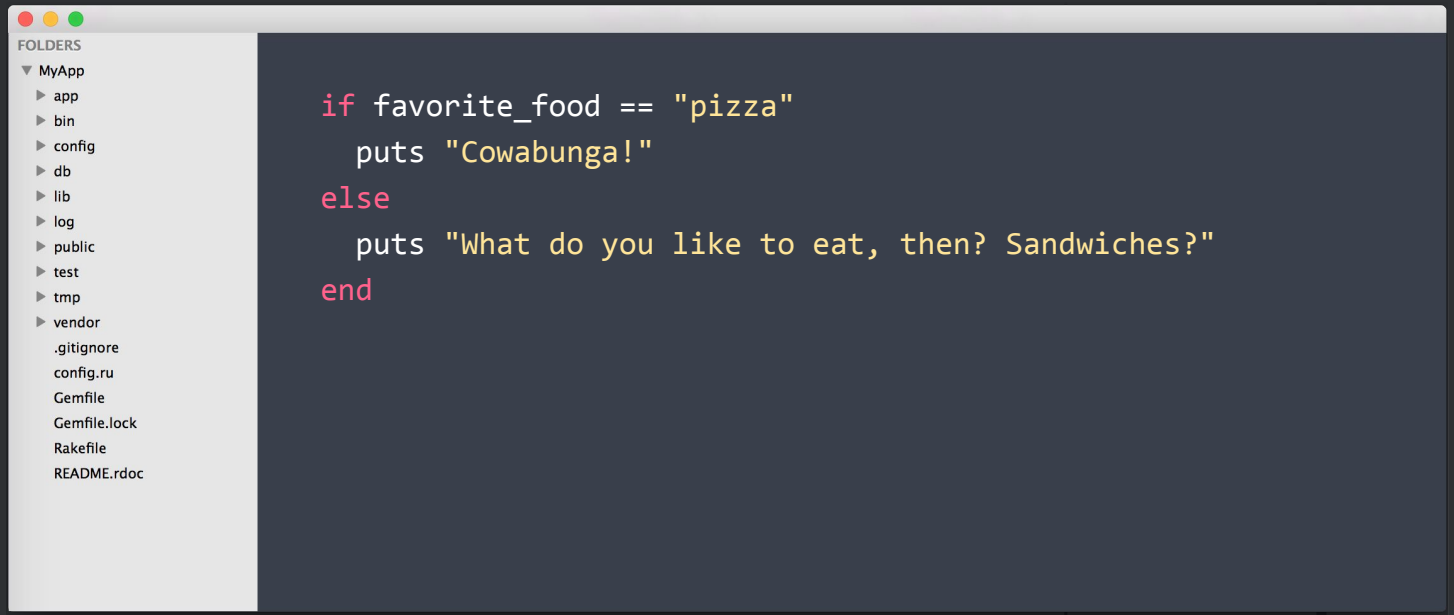


IRON HACK

@xharekx33

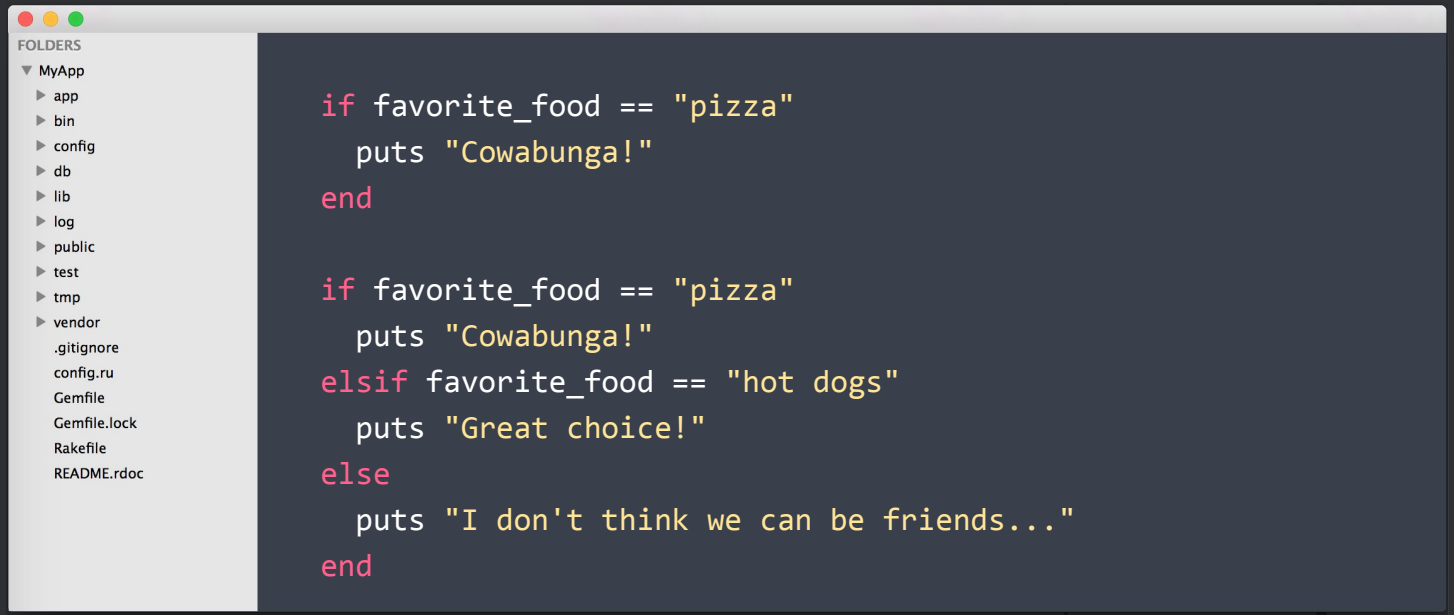
Conditionals: if...else

Boolean expressions are usually used with conditional blocks such as the `if...else` statement, to control the execution flow of the program.



Conditionals: if, elsif.

Sometimes you don't need an else block... or you may need more than one, so you use an `elsif` statement.



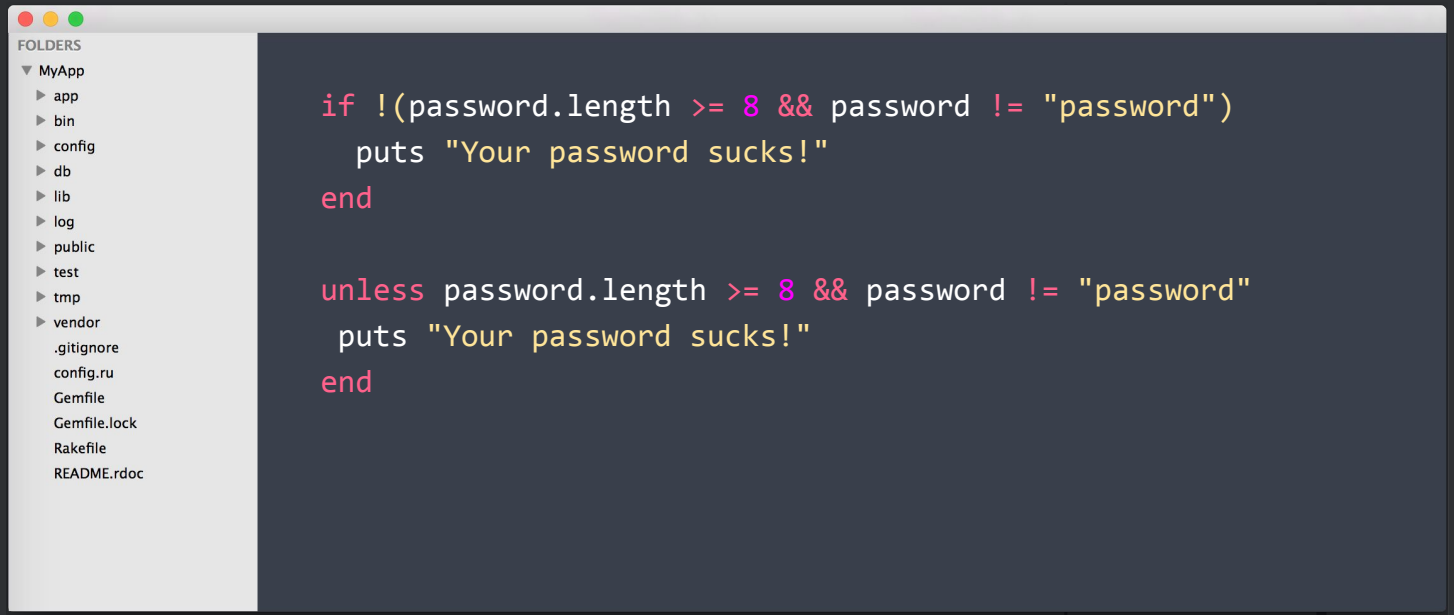

```ruby
if favorite_food == "pizza"
  puts "Cowabunga!"
end

if favorite_food == "pizza"
  puts "Cowabunga!"
elsif favorite_food == "hot dogs"
  puts "Great choice!"
else
  puts "I don't think we can be friends..."
end
```

IRON HACK

@xharekx33

Conditionals: unless.

Many times writing something like `if !(condition)` feels... wrong. In those cases you can use `unless` and make everything feel more natural!

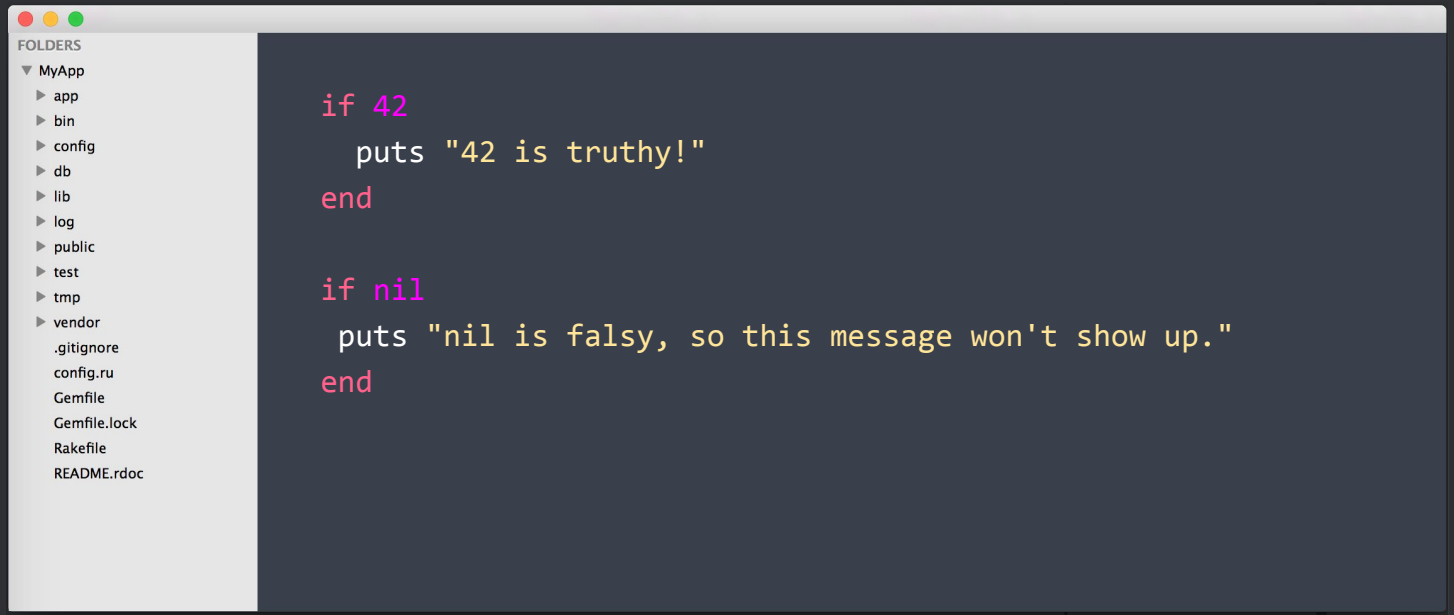


IRON HACK

@xharekx33

Truthiness and falsiness.

Every value in Ruby has an inherent truthiness or falsiness. That means you can use them as conditionals.

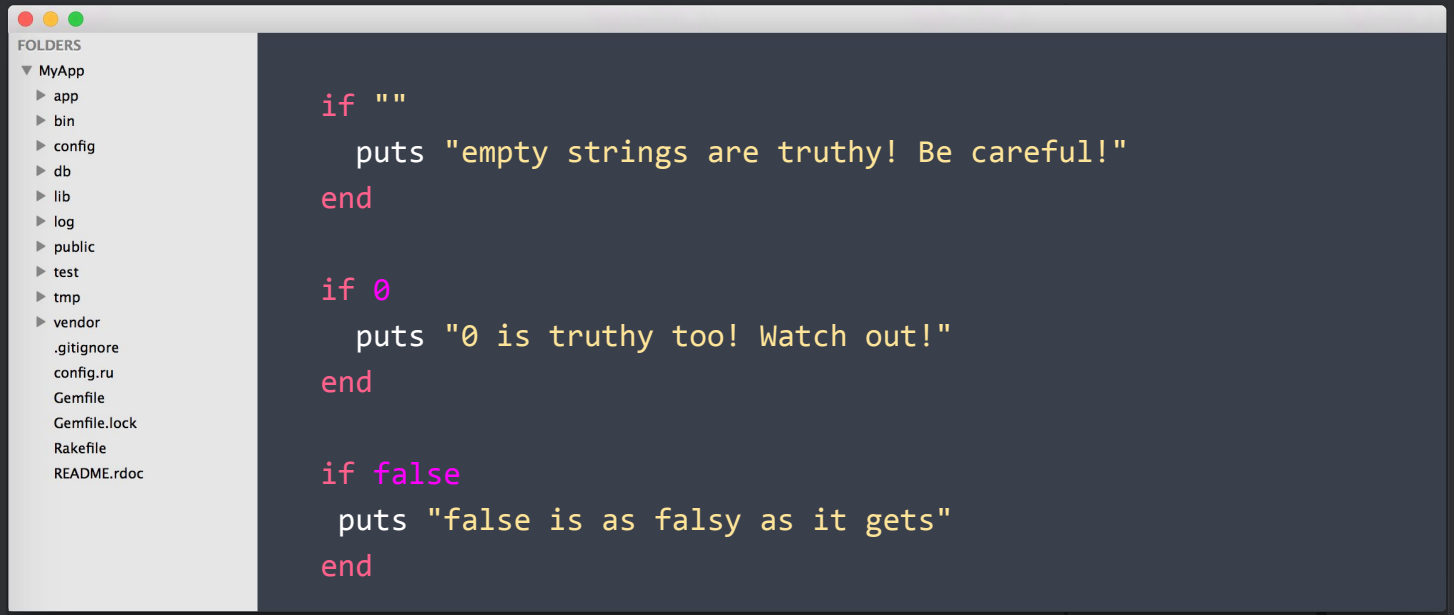


IRON HACK

@xharekx33

Falsiness: nil and false

The only values that are considered falsy are nil and false. Everything else is considered truthy. Even empty strings!

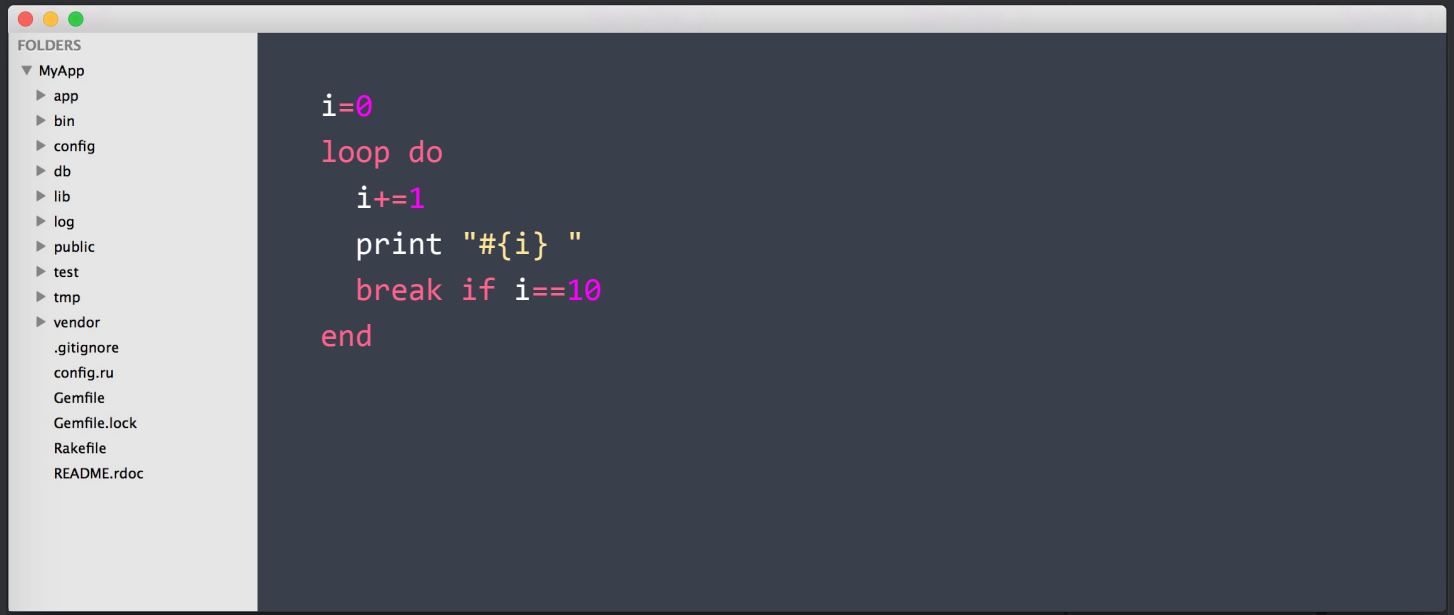


IRON HACK

@xharekx33

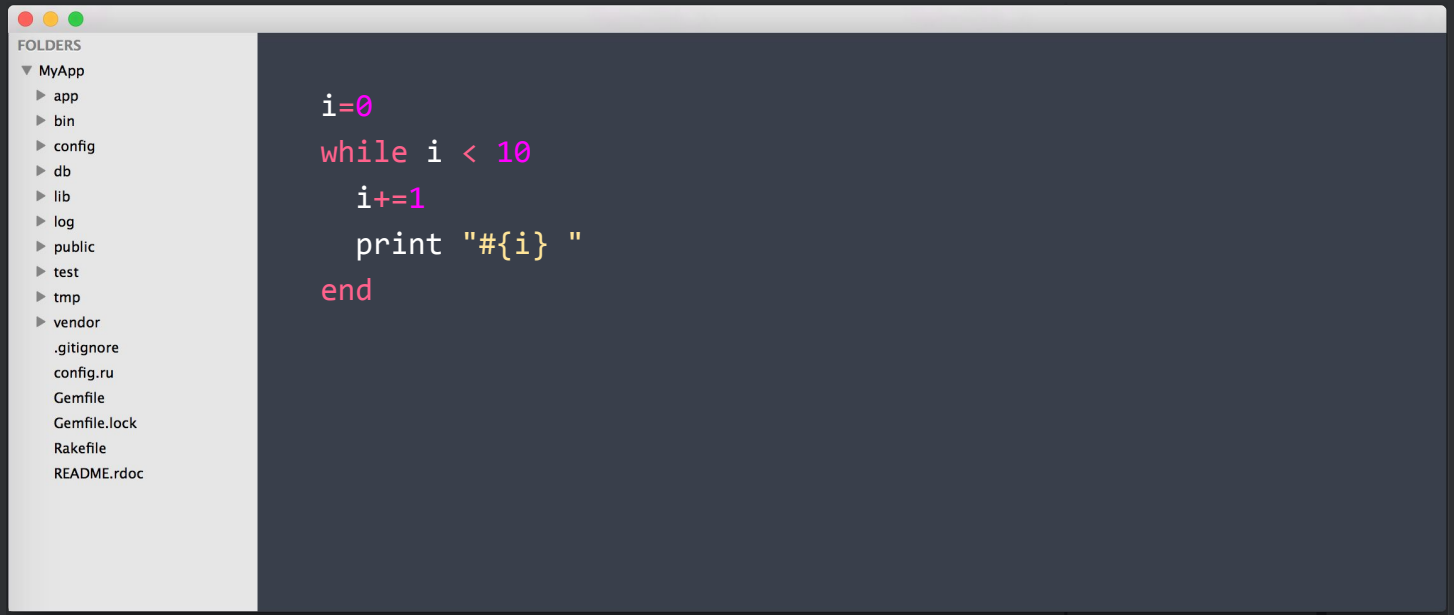
Iteration: loop

The basic way to do something more than once is to use `loop`. It will take a block of code and run it, looping until you tell it to stop using a break statement.



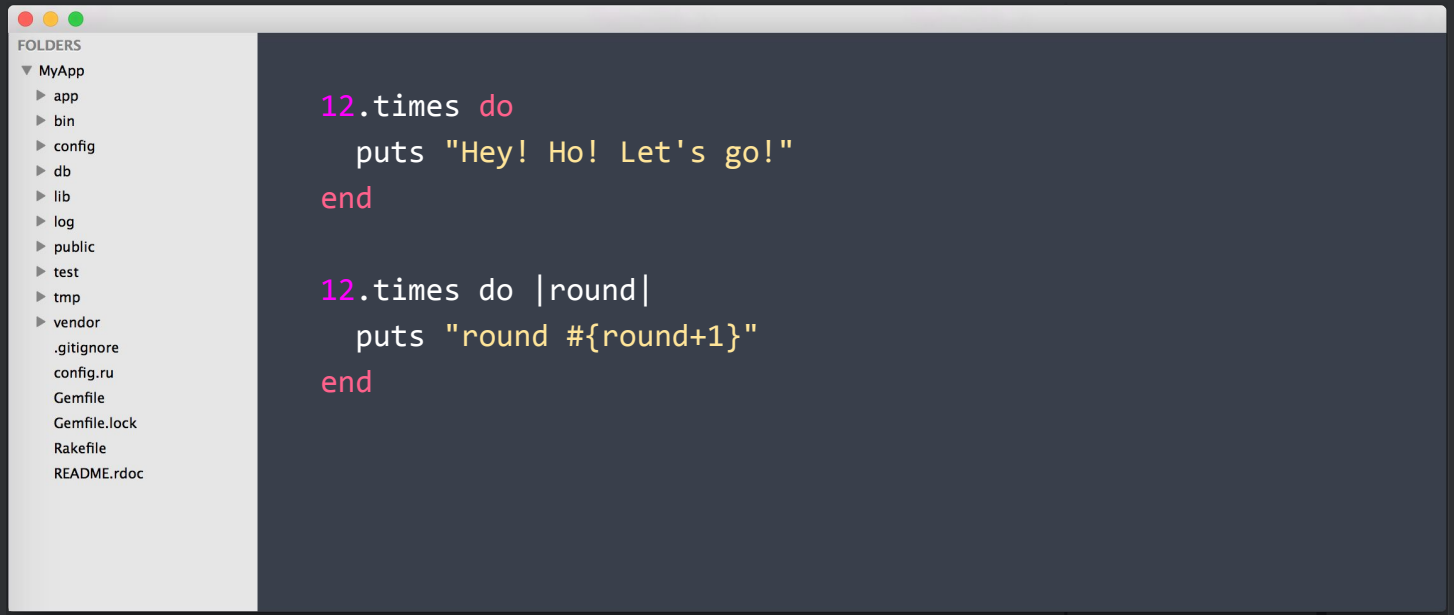
Iteration: while

A more elegant way would be to use a `while` loop to do the same thing:



Iteration: times

If you only want to do something a certain number of times you can simply use the times method.



@xharekx33

User input: gets

You can ask for user input using `gets` and save it to a variable. Now our previous `if` statements will make more sense:

FOLDERS
- MyApp
  - app
  - bin
  - config
  - db
  - lib
  - log
  - public
  - test
  - tmp
  - vendor
  - .gitignore
  - config.ru
  - Gemfile
  - Gemfile.lock
  - Rakefile
  - README.rdoc

```ruby
puts "What's your favorite food?"
favorite_food = gets
if favorite_food == "pizza"
  puts "Cowabunga!"
elsif favorite_food == "hot dogs"
  puts "Great choice!"
else
  puts "I don't think we can be friends..."
end
```

User input: gets + chomps

The problem is `gets` also captures the `\n` from the user hitting the return or enter key, so we need to use the `chomp` method too for our comparison to work:

FOLDERS
- MyApp
  - app
  - bin
  - config
  - db
  - lib
  - log
  - public
  - test
  - tmp
  - vendor
  - .gitignore
  - config.ru
  - Gemfile
  - Gemfile.lock
  - Rakefile
  - README.rdoc

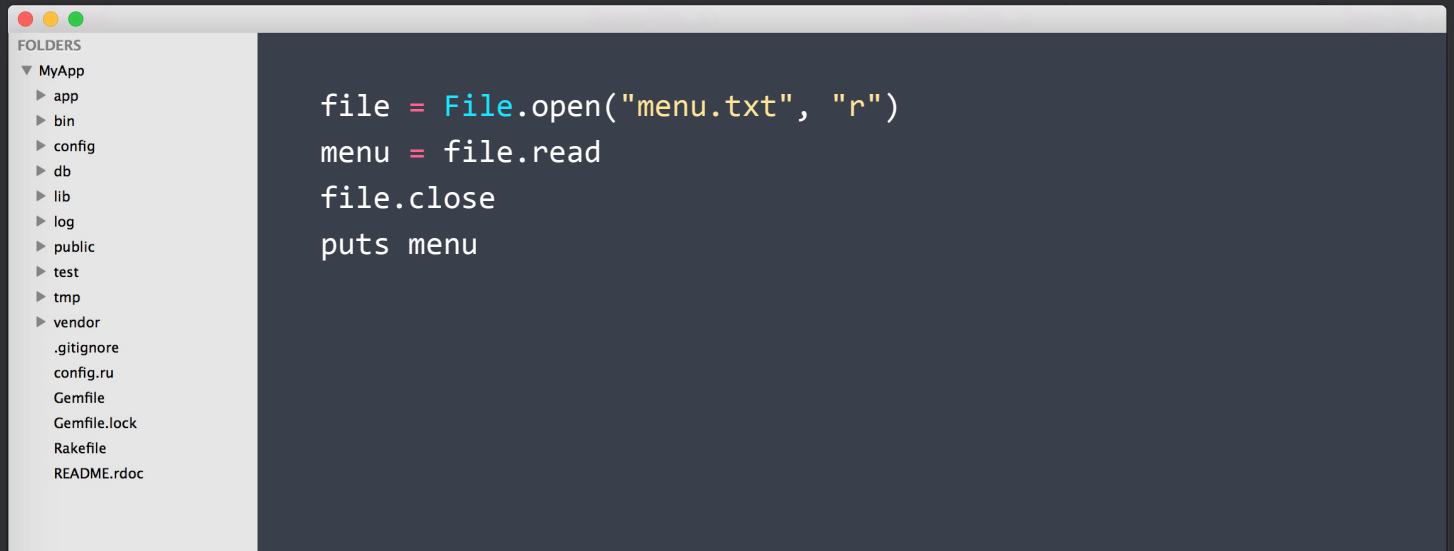
```ruby
puts "What's your favorite food?"
favorite_food = gets.chomp
if favorite_food == "pizza"
  puts "Cowabunga!"
elsif favorite_food == "hot dogs"
  puts "Great choice!"
else
  puts "I don't think we can be friends..."
end
```

IRON HACK

@xharekx33

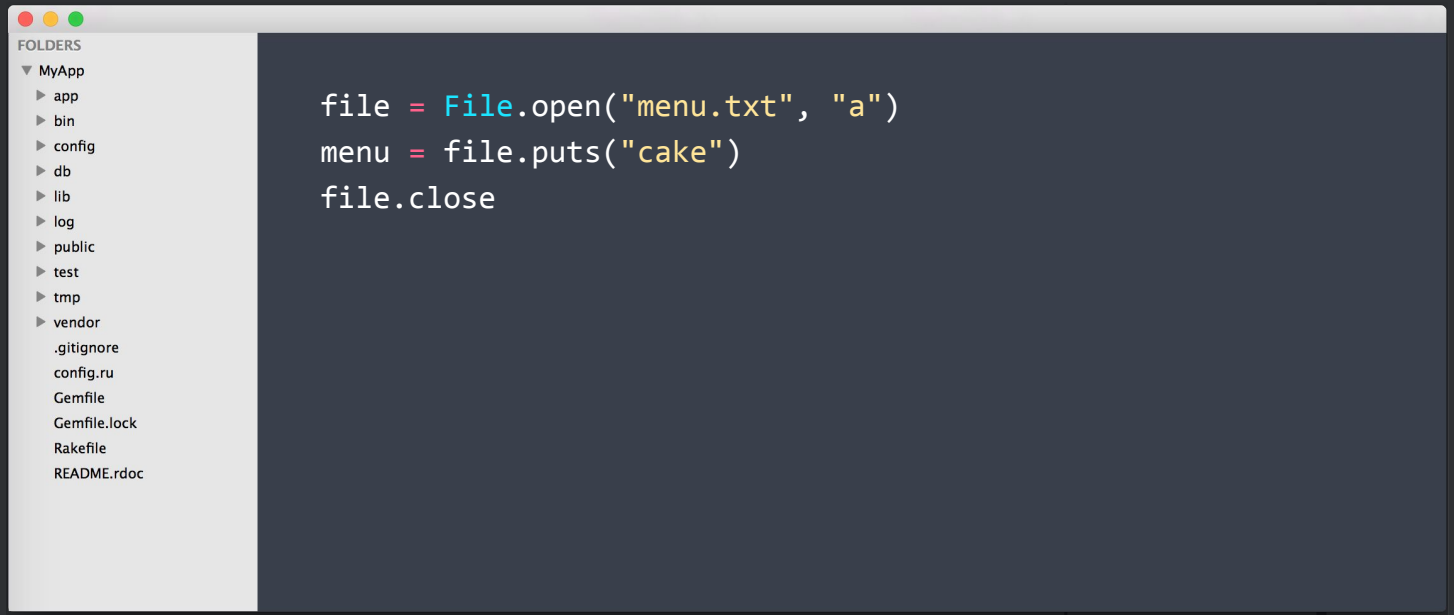
Reading files

You can also write and read files in Ruby. Open a file with `File.open`, choose the appropriate mode and use the `read` method to get its contents. Close it when you're done.



Writing files

Writing works just like reading. Simply use the appropriate mode and the `puts` method (or the `write` method if you don't want to add a newline at the end).



IRON HACK

@xharekx33