



# 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"



# 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.



# Get started writing Ruby code

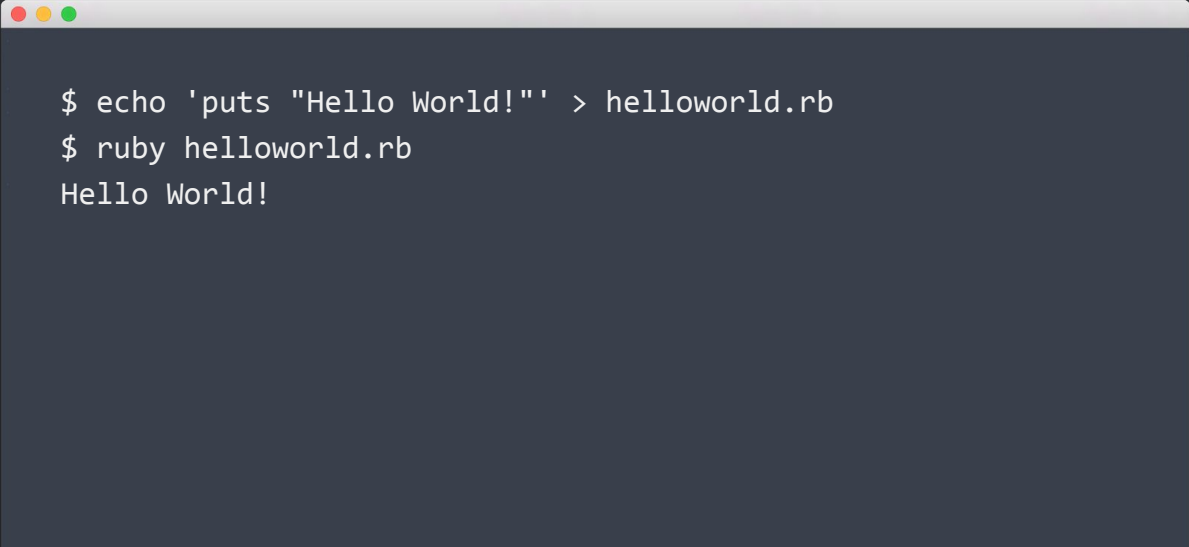
Once you've installed Ruby in your machine (<https://www.ruby-lang.org/en/documentation/installation/>) there are two main ways to run your Ruby code:

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



# The Ruby command

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

A terminal window with a dark background and a light gray title bar containing three colored window control buttons (red, yellow, green). The terminal displays a sequence of commands and their output.

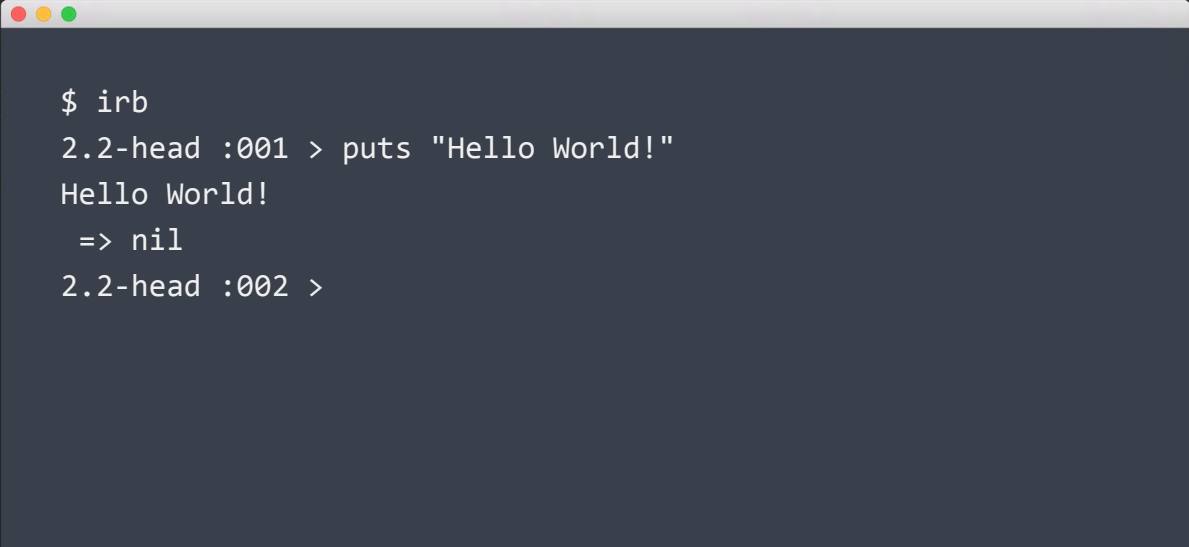
```
$ echo 'puts "Hello World!"' > helloworld.rb
$ ruby helloworld.rb
Hello World!
```



@xharekx33

# IRB

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

A terminal window with a dark blue background and a light gray title bar containing three colored window control buttons (red, yellow, green). The terminal shows the following text:

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

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



@xharekx33

# Printing stuff to the screen

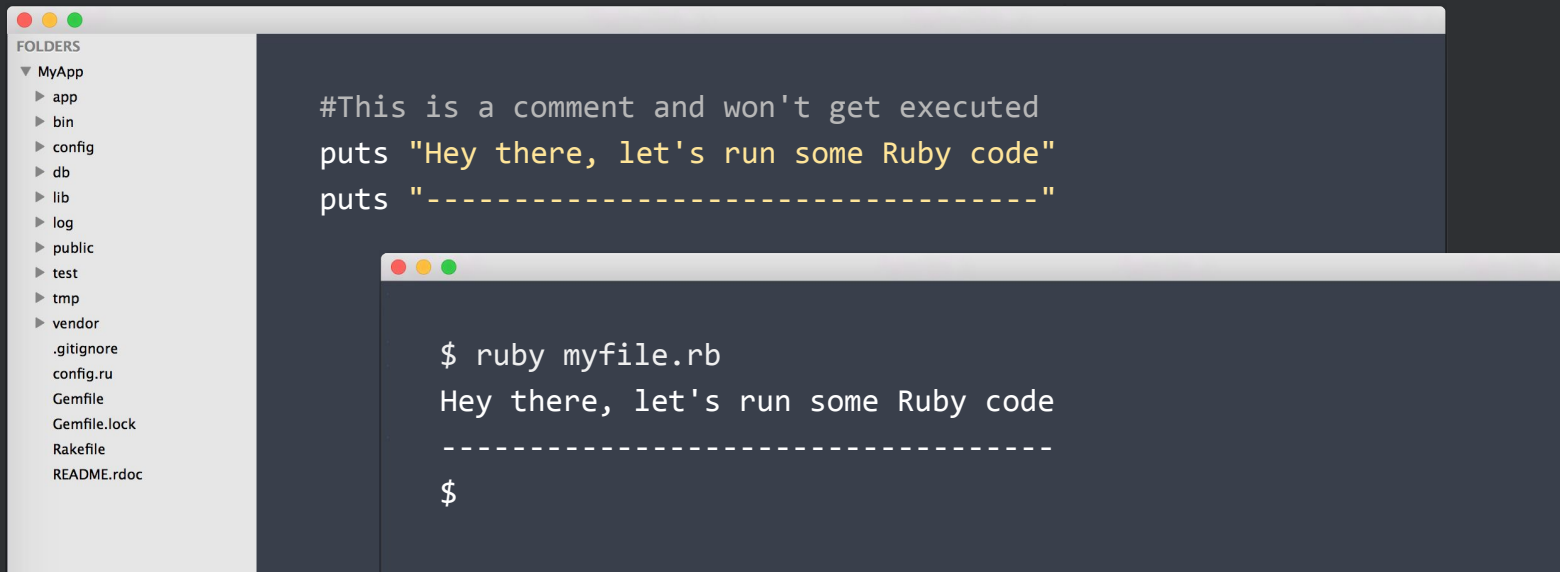
To make a program print a message to the screen we can use  
`puts "My awesome message"` or `print "My other awesome message"`

```
$ 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.



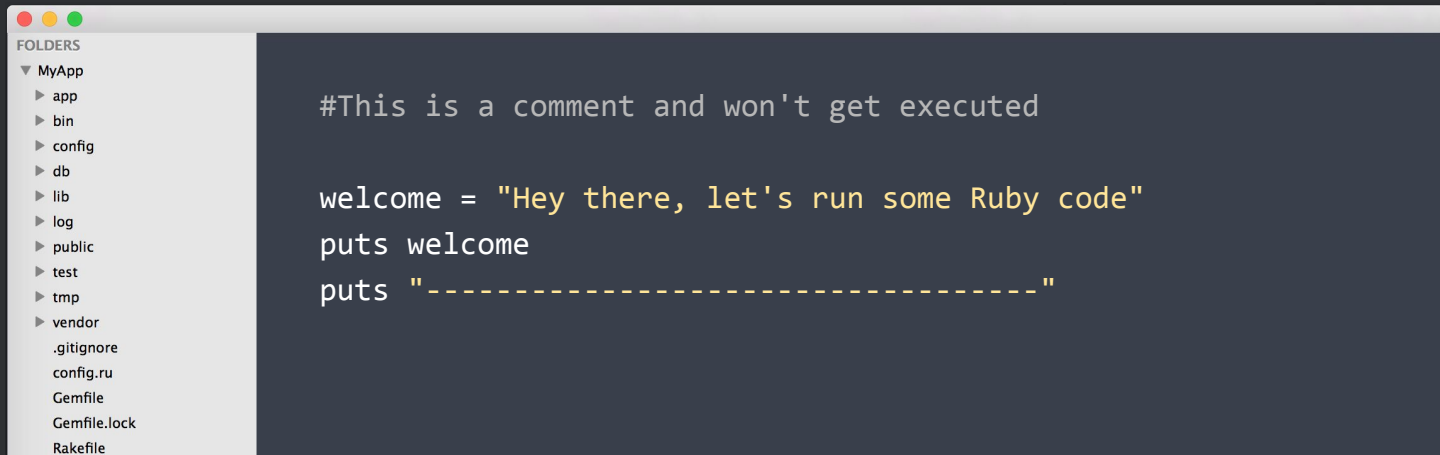
@xharekx33



# 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.



The image shows a code editor window with a sidebar on the left displaying a file tree under the heading 'FOLDERS'. The tree includes 'MyApp' with subfolders 'app', 'bin', 'config', 'db', 'lib', 'log', 'public', 'test', 'tmp', and 'vendor', as well as files '.gitignore', 'config.ru', 'Gemfile', 'Gemfile.lock', and 'Rakefile'. The main editor area contains the following Ruby code:

```
#This is a comment and won't get executed

welcome = "Hey there, let's run some Ruby code"
puts welcome
puts "-----"
```



@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
```



@xharekx33

# 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
```



@xharekx33

# 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
```



@xharekx33

# 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"
```



@xharekx33

# 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 > 1524157875019052152415787501905211.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

---

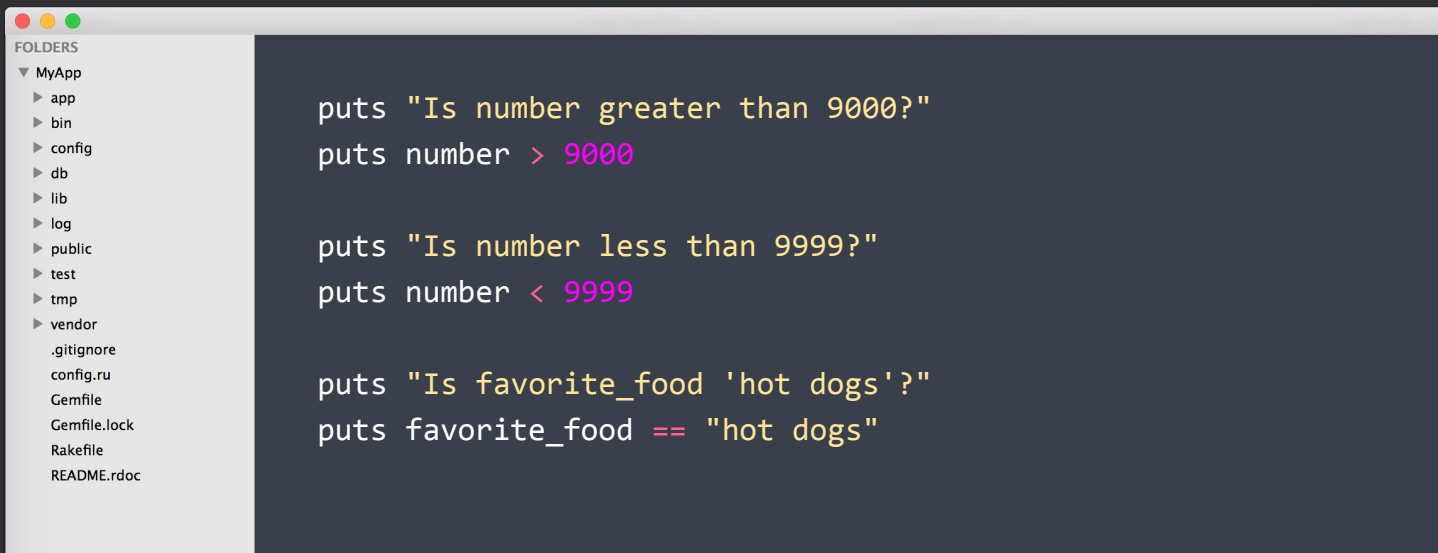
Decimal numbers are called floats in programming (floating point numbers). Create them writing a number with a fractional part.

```
$ 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



The image shows a screenshot of a code editor with a sidebar on the left and a main code area on the right. The sidebar, titled 'FOLDERS', shows a project structure for 'MyApp' with subfolders like 'app', 'bin', 'config', 'db', 'lib', 'log', 'public', 'test', 'tmp', and 'vendor'. It also lists files like '.gitignore', 'config.ru', 'Gemfile', 'Gemfile.lock', 'Rakefile', and 'README.rdoc'. The main code area contains three lines of Ruby code, each preceded by 'puts'.

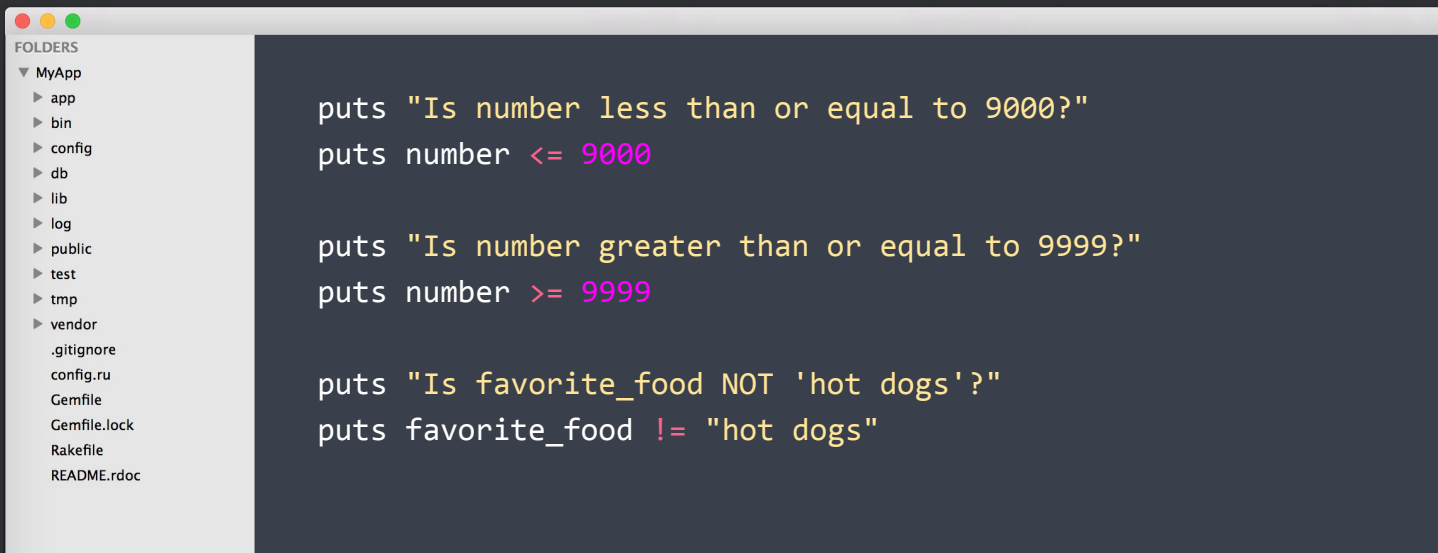
```
puts "Is number greater than 9000?"  
puts number > 9000  
  
puts "Is number less than 9999?"  
puts number < 9999  
  
puts "Is favorite_food 'hot dogs'?"  
puts favorite_food == "hot dogs"
```



@xharekx33

# Boolean expressions: $\leq$ , $\geq$ , $\neq$

We can also check the opposite: less than or equal to:  $\leq$ , greater than or equal to:  $\geq$ , not equal to:  $\neq$



The image shows a screenshot of a code editor window. On the left, there is a sidebar with a file explorer titled 'FOLDERS'. It shows a project named 'MyApp' with several subfolders: 'app', 'bin', 'config', 'db', 'lib', 'log', 'public', 'test', 'tmp', and 'vendor'. Below the folders, there are several files: '.gitignore', 'config.ru', 'Gemfile', 'Gemfile.lock', 'Rakefile', and 'README.rdoc'. The main area of the editor displays three lines of Ruby code. Each line starts with 'puts' followed by a string and a boolean expression. The first line is 'puts "Is number less than or equal to 9000?"' followed by 'puts number <= 9000'. The second line is 'puts "Is number greater than or equal to 9999?"' followed by 'puts number >= 9999'. The third line is 'puts "Is favorite\_food NOT \'hot dogs\'?"' followed by 'puts favorite\_food != "hot dogs"'. The code is color-coded: 'puts' is yellow, strings are orange, and the boolean expressions are purple.

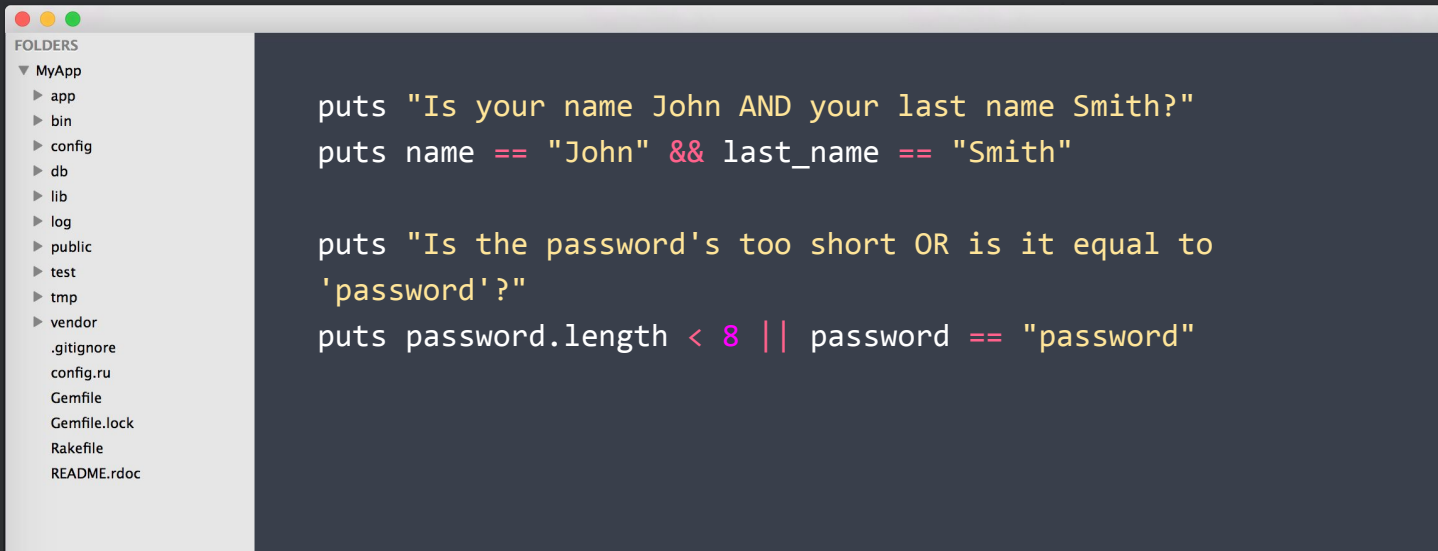
```
puts "Is number less than or equal to 9000?"  
puts number <= 9000  
  
puts "Is number greater than or equal to 9999?"  
puts number >= 9999  
  
puts "Is favorite_food NOT 'hot dogs'?"  
puts favorite_food != "hot dogs"
```



@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.



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

puts "Is your name John AND your last name Smith?"
puts name == "John" && last_name == "Smith"

puts "Is the password's too short OR is it equal to
'password'?"
puts password.length < 8 || password == "password"
```

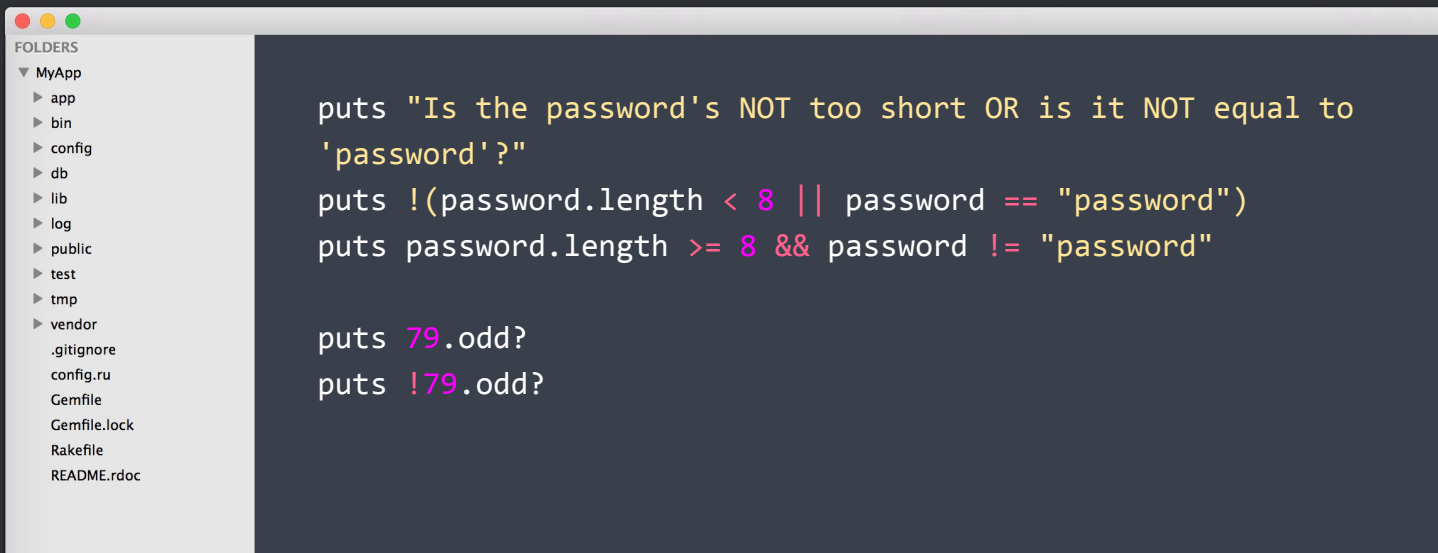


@xharekx33



# 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.



The image shows a screenshot of a code editor window. On the left, there is a sidebar titled 'FOLDERS' showing a directory structure for 'MyApp'. The main area of the editor contains Ruby code. The code uses the `puts` method to print strings and boolean expressions. It demonstrates the use of the negation operator `!` to negate boolean expressions like `password.length < 8` and `password == "password"`.

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

puts "Is the password's NOT too short OR is it NOT equal to 'password'?"
puts !(password.length < 8 || password == "password")
puts password.length >= 8 && password != "password"

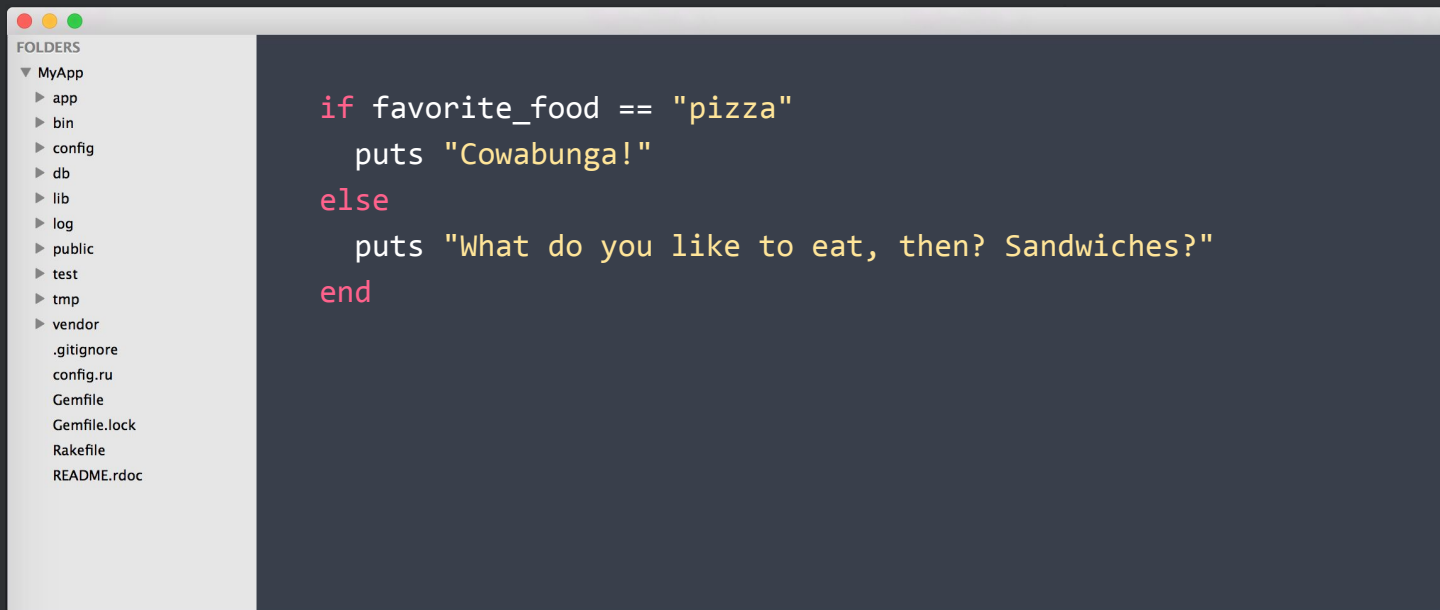
puts 79.odd?
puts !79.odd?
```



@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.

A screenshot of a code editor window. On the left is a sidebar with a 'FOLDERS' section containing a tree view of a project named 'MyApp'. The tree includes folders like 'app', 'bin', 'config', 'db', 'lib', 'log', 'public', 'test', 'tmp', and 'vendor', as well as files like '.gitignore', 'config.ru', 'Gemfile', 'Gemfile.lock', 'Rakefile', and 'README.rdoc'. The main area of the editor displays a Ruby code snippet using an if-else conditional block. The code checks if 'favorite\_food' is equal to 'pizza'. If true, it prints 'Cowabunga!'. If false, it prints 'What do you like to eat, then? Sandwiches?'.

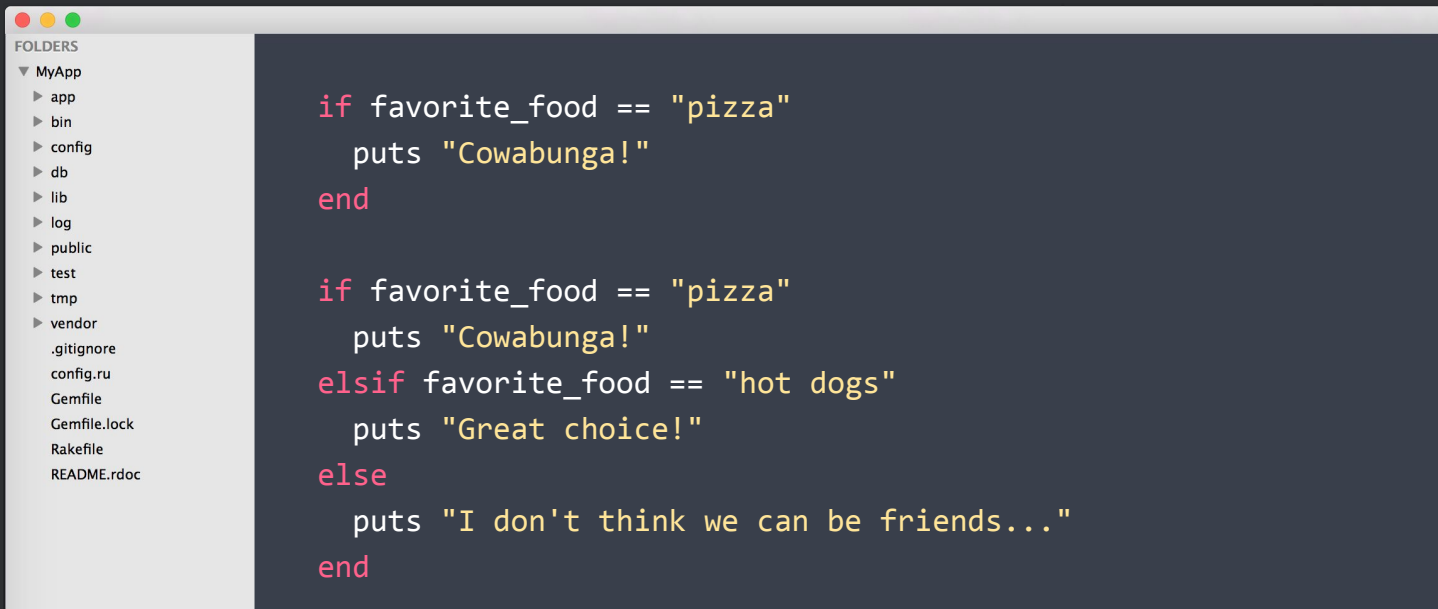
```
if favorite_food == "pizza"
  puts "Cowabunga!"
else
  puts "What do you like to eat, then? Sandwiches?"
end
```



@xharekx33

# Conditionals: if, elsif.

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

A screenshot of a code editor window. On the left is a sidebar titled 'FOLDERS' showing a project structure for 'MyApp' with subfolders like 'app', 'bin', 'config', 'db', 'lib', 'log', 'public', 'test', 'tmp', and 'vendor', along with files like '.gitignore', 'config.ru', 'Gemfile', 'Gemfile.lock', 'Rakefile', and 'README.rdoc'. The main editor area contains two Ruby code snippets. The first snippet shows an 'if' statement for 'favorite\_food == "pizza"' that prints 'Cowabunga!'. The second snippet shows an 'if' statement for 'favorite\_food == "pizza"' that prints 'Cowabunga!', followed by an 'elsif' statement for 'favorite\_food == "hot dogs"' that prints 'Great choice!', and an 'else' statement that prints 'I don't think we can be friends...'.

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

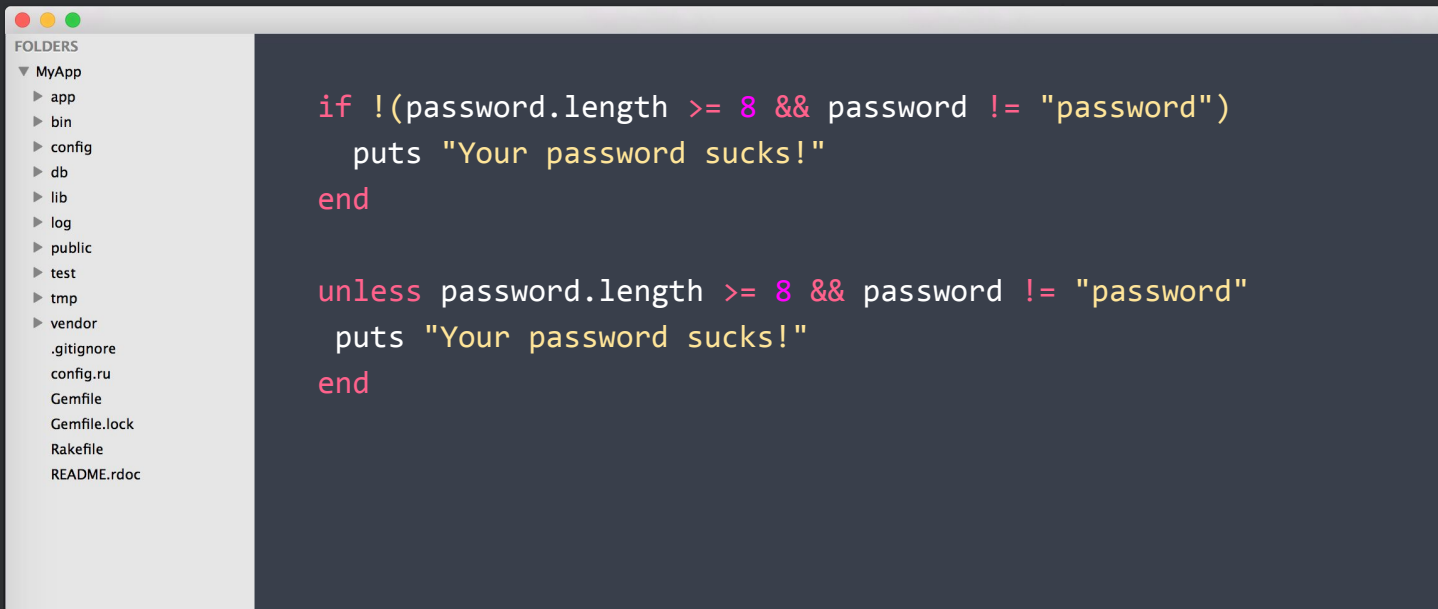
if favorite_food == "pizza"
  puts "Cowabunga!"
elsif favorite_food == "hot dogs"
  puts "Great choice!"
else
  puts "I don't think we can be friends..."
end
```



@xharekx33

# Conditionals: unless.

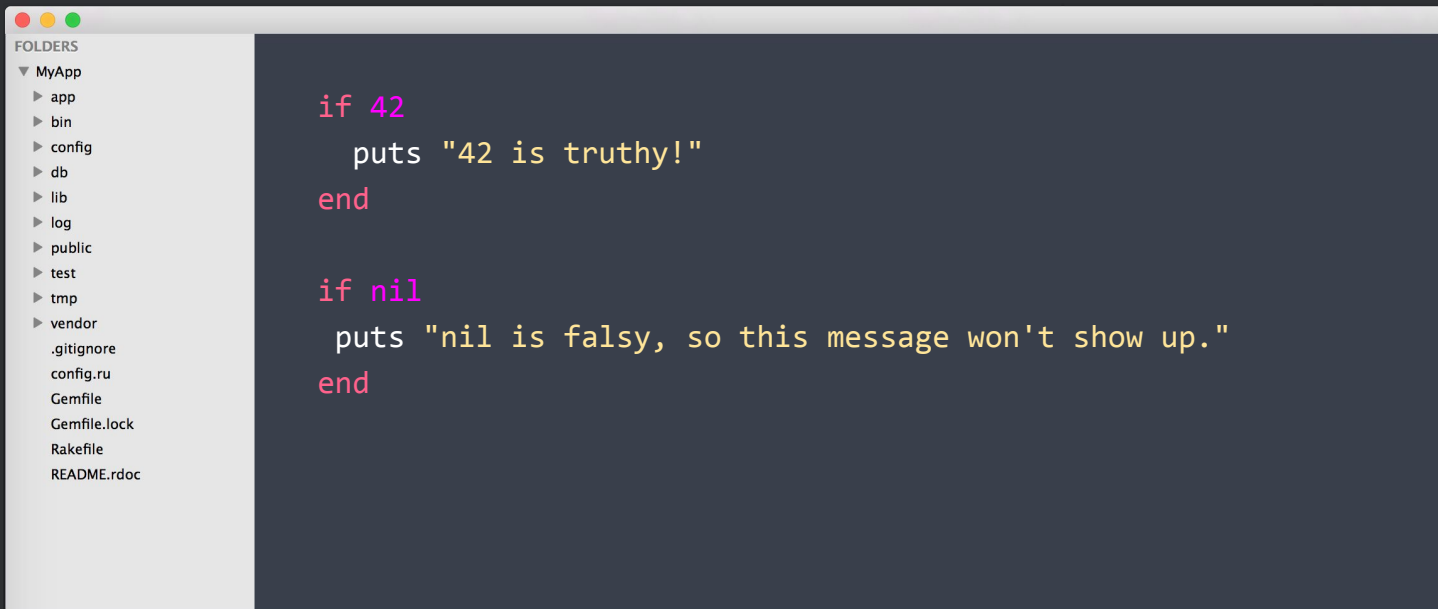
Many times writing something like `if !(condition)` feels... wrong. In those cases you can use `unless` and make everything feel more natural!



@xharekx33

# Truthiness and falsiness.

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



The image shows a screenshot of a code editor window. On the left, there is a sidebar titled 'FOLDERS' showing a directory structure for 'MyApp'. The main area of the editor contains two Ruby code snippets. The first snippet checks if the number 42 is truthy and prints a message. The second snippet checks if nil is falsy and prints a message that will not be shown.

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

if 42
  puts "42 is truthy!"
end

if nil
  puts "nil is falsy, so this message won't show up."
end
```

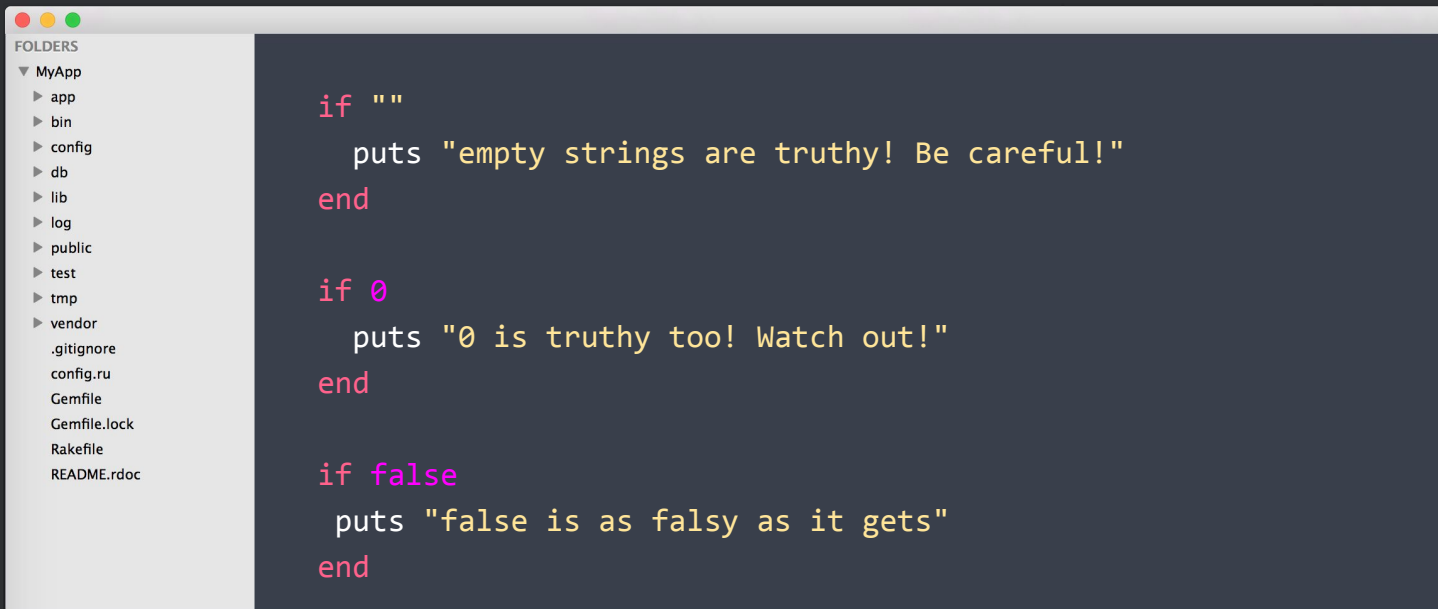


@xharekx33

# Falsiness: nil and false

---

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



```
if ""
  puts "empty strings are truthy! Be careful!"
end

if 0
  puts "0 is truthy too! Watch out!"
end

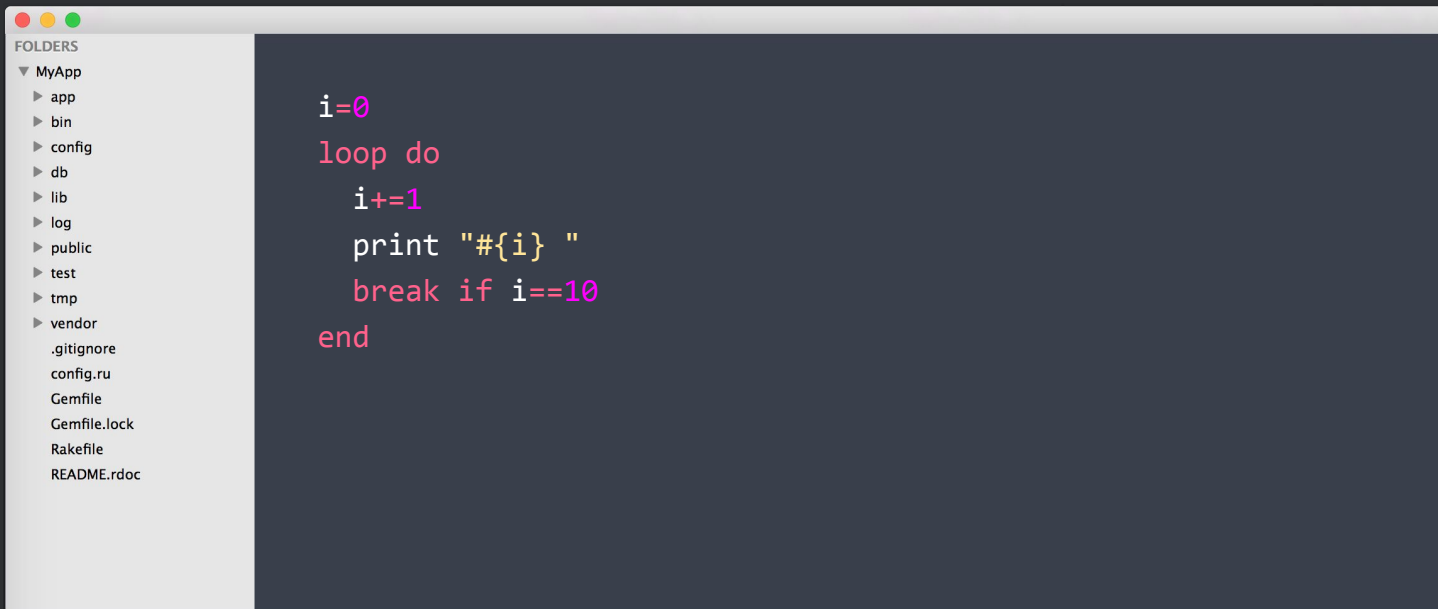
if false
  puts "false is as falsy as it gets"
end
```



@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.



The image shows a screenshot of a code editor window. On the left, there is a sidebar titled 'FOLDERS' showing a project structure for 'MyApp'. The main area of the editor displays a Ruby code snippet that demonstrates a loop with a break statement.

```
i=0
loop do
  i+=1
  print "#{i} "
  break if i==10
end
```

The sidebar 'FOLDERS' contains the following items:

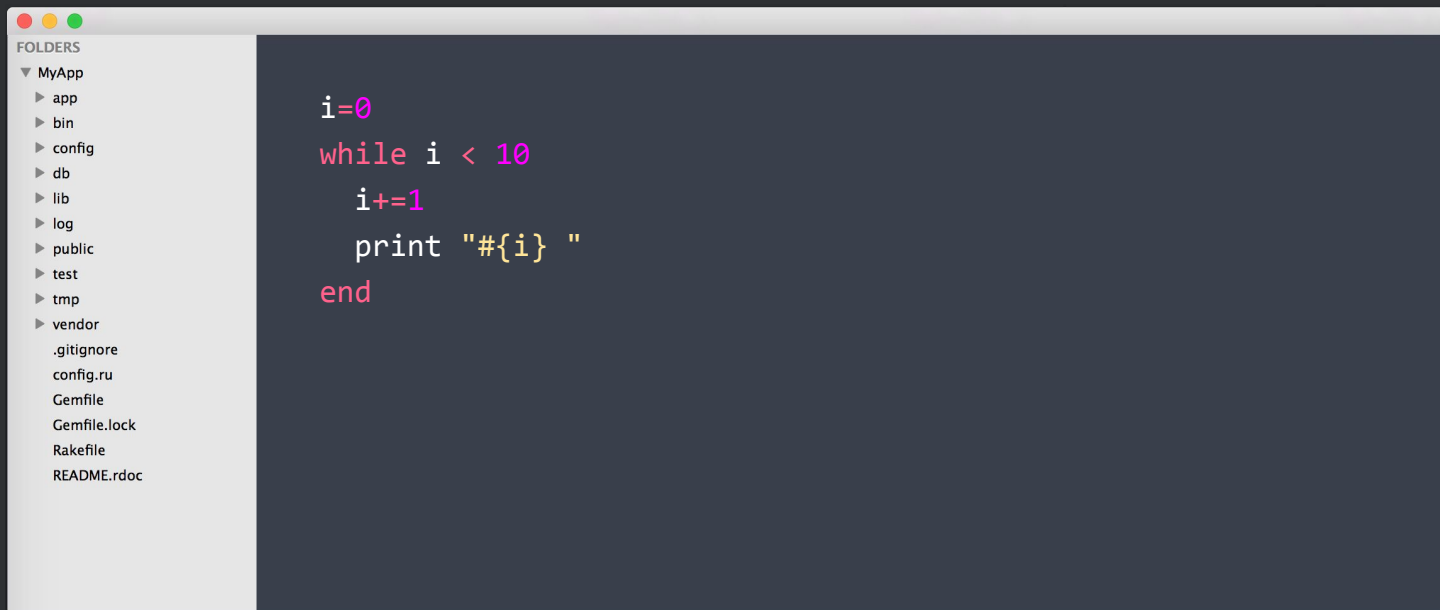
- ▼ MyApp
  - ▶ app
  - ▶ bin
  - ▶ config
  - ▶ db
  - ▶ lib
  - ▶ log
  - ▶ public
  - ▶ test
  - ▶ tmp
  - ▶ vendor
- .gitignore
- config.ru
- Gemfile
- Gemfile.lock
- Rakefile
- README.rdoc



@xharekx33

# Iteration: while

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



The image shows a screenshot of a code editor window. On the left side, there is a sidebar titled 'FOLDERS' showing a directory structure for 'MyApp'. The main area of the editor displays a Ruby code snippet using a while loop to print numbers from 0 to 9.

```
i=0
while i < 10
  i+=1
  print "#{i} "
end
```

**FOLDERS**

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

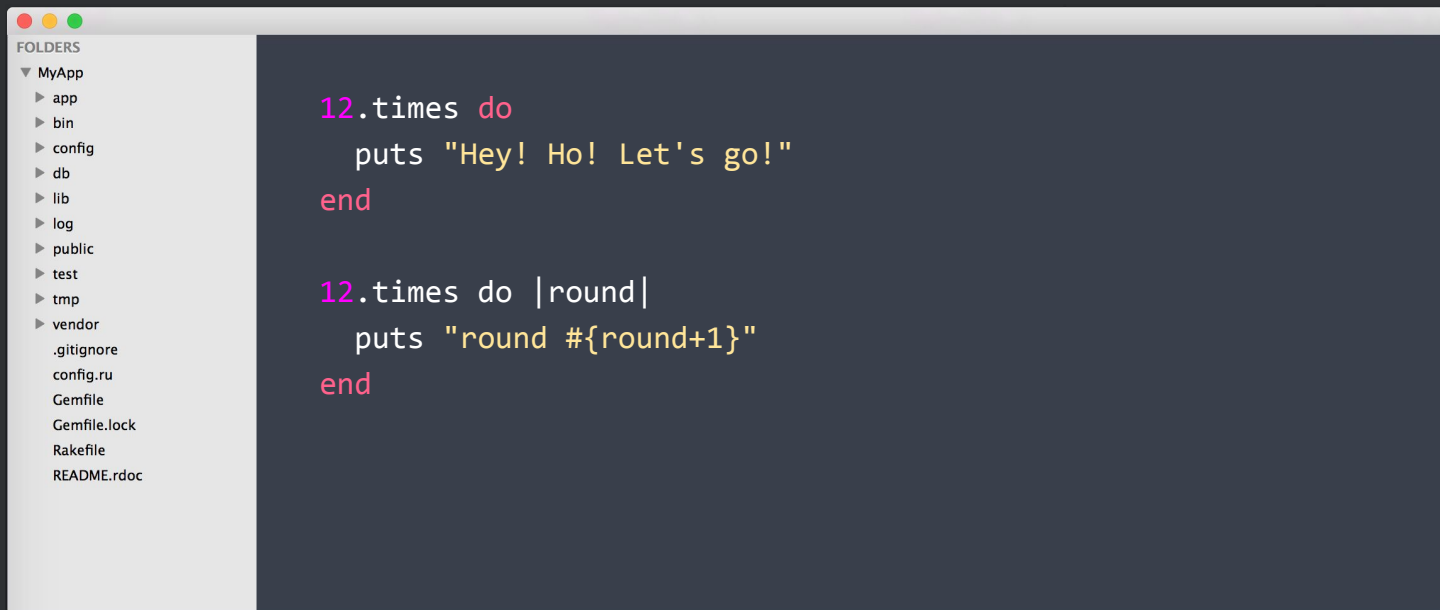


@xharekx33



# Iteration: times

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



The image shows a screenshot of a code editor window. On the left, there is a sidebar titled 'FOLDERS' showing a directory structure for 'MyApp'. The main area of the editor displays two Ruby code snippets. The first snippet uses the `times` method to print a message 12 times. The second snippet uses `times` with a block argument `|round|` to print the round number for each iteration.

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

12.times do
  puts "Hey! Ho! Let's go!"
end

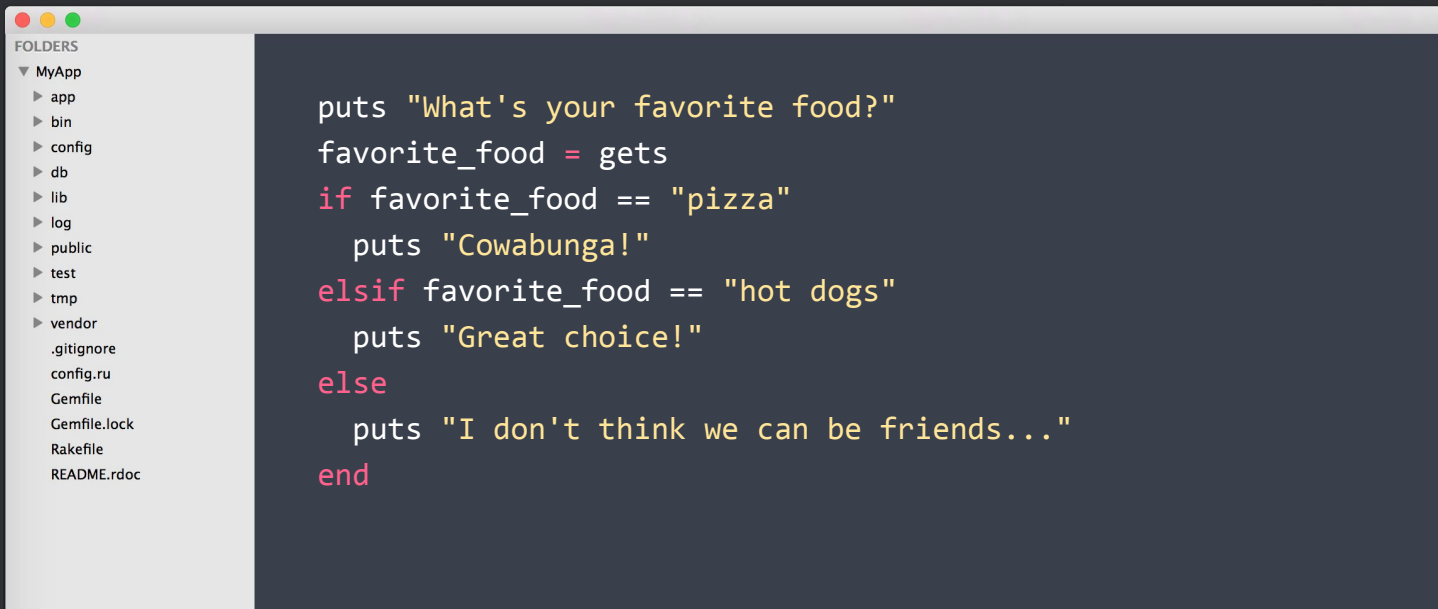
12.times do |round|
  puts "round #{round+1}"
end
```



@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:

A screenshot of a code editor window. On the left is a sidebar with a 'FOLDERS' section. Under 'MyApp', there is a list of folders: 'app', 'bin', 'config', 'db', 'lib', 'log', 'public', 'test', 'tmp', and 'vendor'. Below these are several files: '.gitignore', 'config.ru', 'Gemfile', 'Gemfile.lock', 'Rakefile', and 'README.rdoc'. The main area of the editor is dark blue and contains Ruby code. The code asks the user for their favorite food and uses conditional statements to respond. The code is as follows:

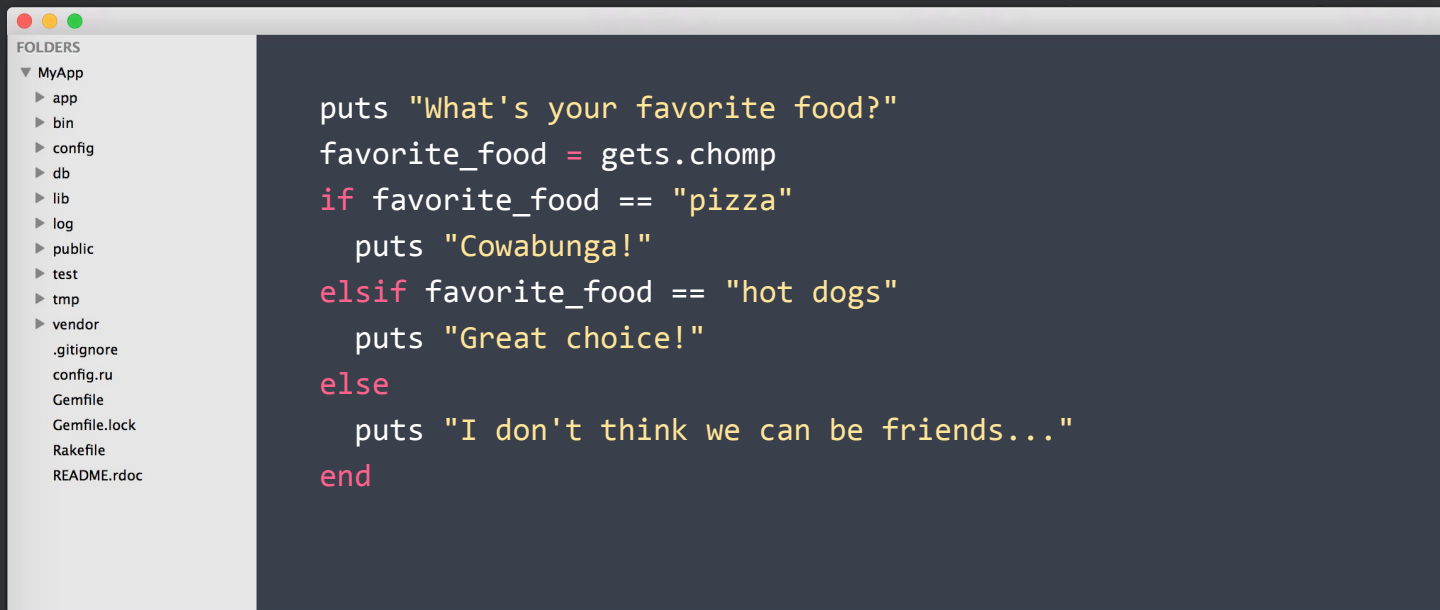
```
puts "What's your favorite food?"
favorite_food = gets
if favorite_food == "pizza"
  puts "Cowabunga!"
elsif favorite_food == "hot dogs"
  puts "Great choice!"
else
  puts "I don't think we can be friends..."
end
```



@xharekx33

# User input: gets + chomp

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:



```
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
```

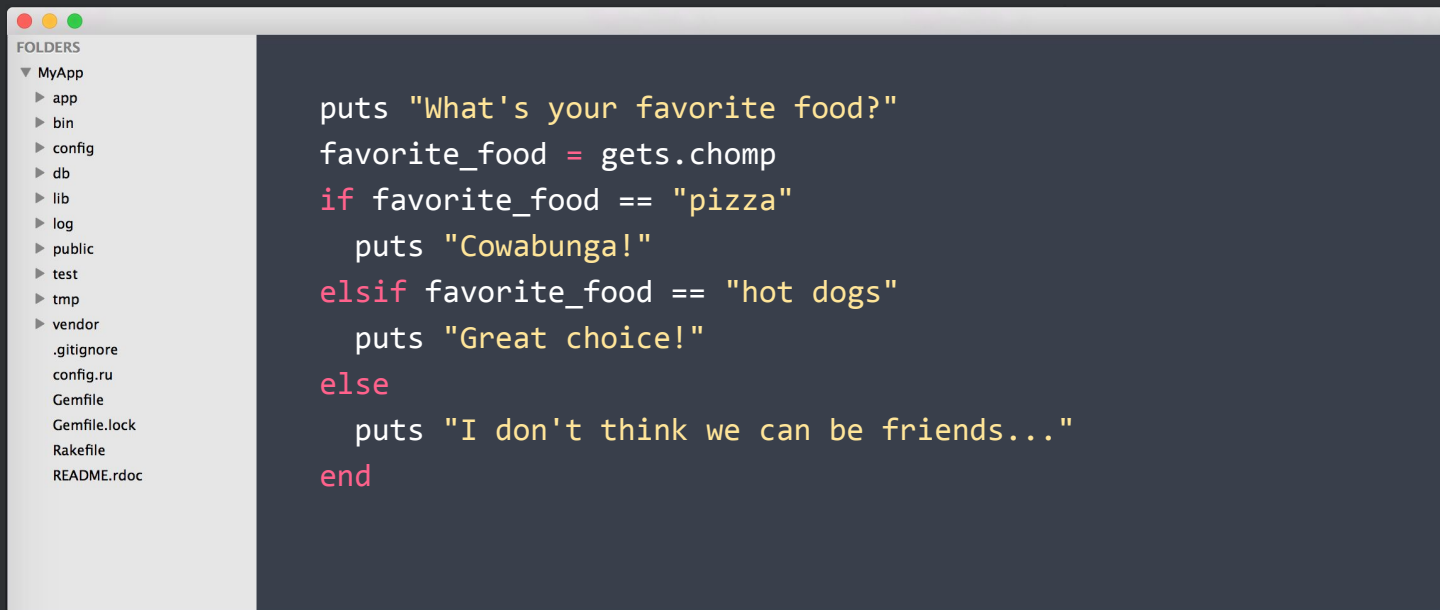
The image shows a code editor window with a sidebar on the left displaying a file tree for a project named 'MyApp'. The sidebar lists folders like 'app', 'bin', 'config', 'db', 'lib', 'log', 'public', 'test', 'tmp', and 'vendor', as well as files like '.gitignore', 'config.ru', 'Gemfile', 'Gemfile.lock', 'Rakefile', and 'README.rdoc'. The main editor area contains a Ruby script that prompts the user for their favorite food, reads the input using `gets.chomp`, and then uses a series of `if`, `elsif`, and `else` statements to print a response based on the input.



@xharekx33

# User input: gets + chomp

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:



```
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
```

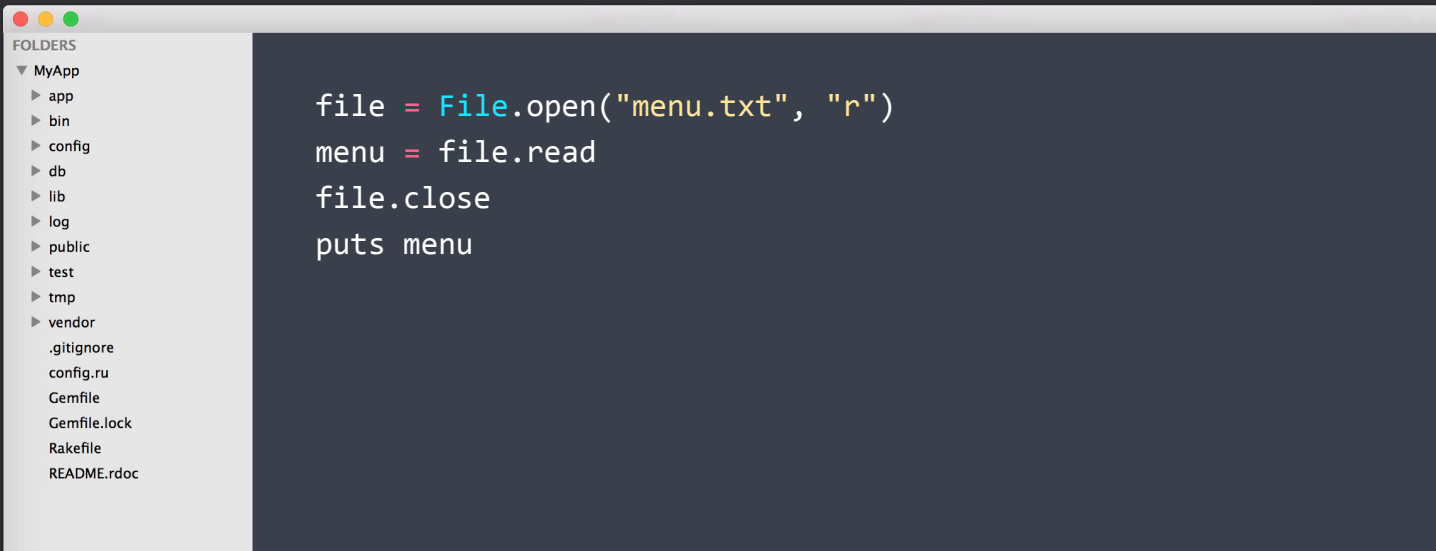
The image shows a code editor window with a sidebar on the left displaying a file tree for a project named 'MyApp'. The sidebar lists folders like 'app', 'bin', 'config', 'db', 'lib', 'log', 'public', 'test', 'tmp', and 'vendor', as well as files like '.gitignore', 'config.ru', 'Gemfile', 'Gemfile.lock', 'Rakefile', and 'README.rdoc'. The main editor area contains a Ruby script that prompts the user for their favorite food and uses `gets.chomp` to capture the input without the newline character. The script then uses `if`, `elsif`, and `else` statements to provide different responses based on the input.



@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.

A screenshot of a code editor window. On the left is a sidebar with a 'FOLDERS' section containing a tree view of a project named 'MyApp'. The tree includes folders like 'app', 'bin', 'config', 'db', 'lib', 'log', 'public', 'test', 'tmp', and 'vendor', as well as files like '.gitignore', 'config.ru', 'Gemfile', 'Gemfile.lock', 'Rakefile', and 'README.rdoc'. The main editor area displays Ruby code for opening and reading a file named 'menu.txt' in read mode ('r').

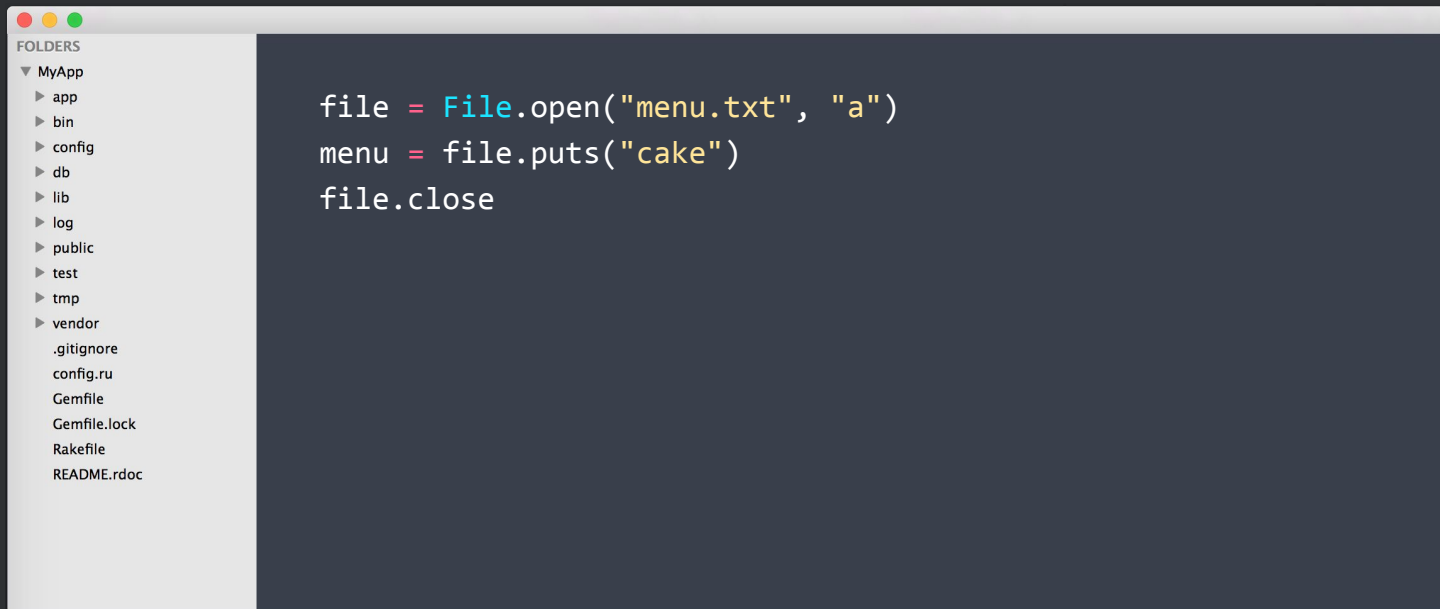
```
file = File.open("menu.txt", "r")
menu = file.read
file.close
puts menu
```



@xharekx33

# 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).

A screenshot of a code editor window. On the left is a sidebar with a file explorer. The main area shows Ruby code for opening a file, writing to it, and closing it. The code uses 'File.open' with mode 'a', 'file.puts' to write 'cake', and 'file.close' to close the file.

```
file = File.open("menu.txt", "a")
menu = file.puts("cake")
file.close
```

**FOLDERS**

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



@xharekx33