Got 15 minutes? Give Ruby a shot right now!

Objective:

Ruby is a programming language from Japan (available at ruby-lang.org) which is revolutionizing the web. The beauty of Ruby is found in its balance between simplicity and power.

Try out Ruby code in the prompt on the right. In addition to Ruby's built-in methods, the following commands are available:

- `help` → Starts the 15 minute interactive tutorial. Trust me, it's very basic!
- `clear` → Clears the screen. Useful if your browser starts slowing down. Your command history will be remembered.
- `next` → Allows you to skip to the next section of a lesson.
- `back` → Allows you to return to the previous section of a lesson.

If you want to save your progress, head on over to Code School and sign up for free. We'll wait for you here.

http://tryruby.org/levels/1/challenges/1

Using the Prompt

Objective:

The window to the right is a Ruby prompt.

Here you'll be able to type a line of Ruby code, hit *Enter*, and watch it run!

For your first bit of Ruby, try typing some math, like: `2 + 6`

Using the Prompt

Objective:

Great! You did a little bit of math.

See how the answer was returned? Ruby uses a fat arrow for responses to your entries.

Ruby recognizes numbers and mathematics operators. You could also try some other math like:

- `4 * 10`
- `5 - 12`
- `40 / 4`

Even though we've placed a space between the numbers and the operators above, it's not required. For now, stick with these basic operations; we'll try a few others later.

When you're finished experimenting, type `next` to move to the next lesson when you're finished.

Interactive ruby ready.

```
> back
> next
> 2 + 6
=> 8
Success!
> 4 * 10
=> 40
> 5 - 12
=> -7
> 40 / 4
=> 10
> back
> next
> next
```

Say Your Name

Objective:

Welp, we already know that computers are handy and fast for math.

But what about something *really* useful. Like, say, seeing the letters of your name reversed!

To do that super-cool task, let's first get you familiar with text in Ruby. Type your first name in quotes, like this: `"Jimmy"`

```
> "Rajani"
=> "Rajani"
Success!
```

```
> "Rajani".reverse
=> "inajaR"
Success!
```

```
> "Rajani".length
=> 6
Success!
```

On Repeat

Objective:

Now, maybe you're wondering what any of this is actually good for. Have you ever encountered a website that yelled at you for choosing a password that was too short? Turns out, the `length` property is often what that site uses to check for a correct length.

Let's get crazy now, and multiply your name by 5. Follow the following format: `"Jimmy" * 5`

```
> "Jimmy" * 5
=> "JimmyJimmyJimmyJimmyJimmy"
Success!
```

Hey, Whoa, Level #2 Already

Objective:

Let's look at what you've learned in the first minute.

- **The prompt.** Typing code into the prompt gives you an answer.
- **Numbers and strings** are Ruby's math and text objects.
- **Methods.** You've used English-language methods like `reverse` and symbolic methods like `*` (the multiplication method.) Methods are actions!

This is the essence of your learning. Taking simple things, toying with them and turning them into new things! Feeling comfortable yet? I promise that we'll get you there..

But now, let's do something a little uncomfortable. Try using that `reverse` method on a number: `40.reverse`

Stop, You're Barking Mad!

Objective:

You can't reverse the number `40`. I guess you can hold your monitor up to the mirror, but reversing a number just doesn't make sense! Ruby has tossed you a useful error message.

The message is telling you that there is no method `reverse` for number values in Ruby!

But, hmm...maybe if you can turn it into a string. Try this: `40.to_s.reverse`.

```
> 40.reverse
=> #<NoMethodError: undefined method `reverse' for 40:Fixnum>
Oh no!
> "40".reverse
=> "04"
That's better!
> back
> 40.to_s.reverse
=> "04"
That's better!
```

Boys are Different From Girls

Objective:

...just like numbers are different from strings. While you can use methods on any object in Ruby, some methods only work on certain types of values. A really cool thing about Ruby is that you can always convert between different types using Ruby's "to" methods.

- **to_s** converts values to **s**trings.
- **to_i** converts values to **i**ntegers (numbers.)
- **to_a** converts values to **a**rrays.

What in the world are arrays, you might ask?! They are simply lists. Let's make an empty one, by typing in a pair of brackets: `[]`.

```
> []
=> []
```

Standing in Line

Objective:

Great, you built an empty array. Now let's see what else we can do with it.

First off, a good thing to know is that arrays store their information in a **sequence**. Think of this like standing in line for popcorn. You are behind someone and you wouldn't dream of pushing them aside, right? And that guy behind you, you've got a close eye on him, too. First come, first serve.

Just like that line for popcorn, the order of an array's information will stay consistent for you after you build it...well, at least until you modify it.

To try building an array with some stuff in it, here's a list of lottery numbers for you: `[12, 47, 35]`. See those commas? They're important!

```
> [12, 47, 35]
=> [12, 47, 35]
```

Success!

One Raises Its Hand

Objective:

Sweet, you've got a short list of lottery numbers. Now, what if we wanted to know which one is the highest in the array?

Try this: `[12, 47, 35].max`.

```
> [12, 47, 35].min
=> 12
Success!
> [12, 47, 35].max
=> 47
```

Tucking a List Away

Objective:

Good, good. But it would be pretty annoying to have to retype that list every time, right?

Let's fix that by using a Ruby **variable**, which helps us **store** important data. Each variable has a unique name, so that it can be summoned up whenever we need the info it contains.

Call your new variable `ticket` and place your lottery numbers inside it, like so: `ticket = [12, 47, 35]`. That equal sign you see is what assigns your array to the new variable.

```
> ticket = [12, 47, 35]
=> [12, 47, 35]
Success!
> ticket
=> [12, 47, 35]
Success!
```

Saved, Tucked Away

Objective:

Fantastic! You've hung on to your lotto numbers, tucking them away inside a **variable** called `ticket`.

Now let's put your lotto numbers in order...sound good? Ruby has a great method for that. Use: `ticket.sort!`

You might notice that the method has an exclamation point at its end. This just signals that we intend for Ruby to directly modify the same array that we've built, rather than make a brand new copy that is sorted. You'll notice that if you try calling `ticket` again, it will be sorted permanently!

When you want to move on, just type `next`

```
> ticket.sort
=> [12, 35, 47]
```

Level #3 is Upon Us

Objective:

You built a list. Then you sorted the list. And as you've seen, the `ticket` variable is now changed.

Now, let's look at how your second level went down:

- **Errors.** If you try to reverse a number or do anything fishy, Ruby will skip the prompt and tell you to straighten up.
- **Arrays** are lists of stored information.
- **Variables** are a place to save stuff you might need again, as well as give that stuff a name. You used the equals sign to do this, in a process called assignment.
  Like: `ticket = [14, 37, 18].`

In all, there are just eight levels in this course. You are already two-eighths of the way to the end! This is simple stuff, don't you think? More good stuff up ahead.

Let's change directions for a moment. I've stuffed a bit of poetry for you in a certain variable. Take a look, by typing `print poem`

```
> print poem
=> "My toast has flown from my hand
And my toast has gone to the moon.
But when I saw it on television,
Planting our flag on Halley's comet,
More still did I want to eat it."
Success!
```

Sadly, You Hate Toast Poetry

Objective:

Look, it's okay. You don't have to like it. You may even want to hack it up. Welp, be my guest.

Instead of toast, maybe go for a melon or something. Try this one: `poem['toast'] = 'honeydew'`

```
> poem['toast'] = 'honeydew'
=> "honeydew"
Success!
```

Sadly, You Hate Toast Poetry

Objective:

Now type `print poem` once again to see the new poem.

See how you only changed the first toast? The joke's on you, bread hater.

When you want to move on, type `next`

```
> poem['toast'] = 'honeydew'
=> "honeydew"
Success!
> print poem
=> "My honeydew has flown from my hand
And my toast has gone to the moon.
But when I saw it on television,
Planting our flag on Halley's comet,
More still did I want to eat it."
> print poem['toast']
=> "toast"
> print poem["toast"]
=> "toast"
> poem["toast"]
=> "toast"
> poem['toast']
=> "toast"
> poem['toast'] = 'honeydew'
=> #<IndexError: string not matched>
> 'honeydew'
=> "hone?ydew"
> 'toast'
=> "toast"
> toast
=> #<NameError: undefined local variable or method `toast' for main:Object>
```

Sadly, You Hate Toast Poetry

Objective:

Now type `print poem` once again to see the new poem.

See how you only changed the first toast? The joke's on you, bread hater.

When you want to move on, type `next`

Ready, Aim

Objective:

The square brackets you just used are very common in Ruby. Remember, you typed: `poem['toast'] = 'honeydew'`. That box that holds the word *toast* uses a square bracket on each side. See 'em?

The two brackets are like a crosshairs used to line up precisely on a target. These brackets mean, "I am looking for **____** somewhere in here." Ready...aim...**data**. Here, you were looking specifically for toast in order to swap it out with a fruit.

Let's see if your new experience can help you produce the answer to this question: what happens when we reverse this whole poem? `poem.reverse`

```
> poem.reverse
=> ".ti tae ot tnaw I did llits eroM
,temoc s'yellaH no galf ruo gnitnalP
,noisivelet no ti was I nehw tuB
.noom eht ot enog sah tsaot ym dnA
dnah ym morf nwolf sah wedyenoh yM"
Success!
>
```

Too Much Reversal

Objective:

Okay, I suppose that was expected. The whole poem's been turned backwards, letter by letter. But say I really just wanted to reverse the lines only. In other words, move the last line up to first and the first line down to last. Backwards, yes, but not *that* backwards.

Ruby has a way. Try this: `poem.lines.to_a.reverse`

```
> poem.lines.to_a.reverse
=> ["More still did I want to eat it.
", "Planting our flag on Halley's comet,
", "But when I saw it on television,
", "And my toast has gone to the moon.
", "My honeydew has flown from my hand
"]
Success!
> poem.lines
#<Enumerator: "My honeydew has flown from my hand\
And my toast has gone to the moon.\
But when I saw it on television,\
Planting our flag on Halley's comet,\
More still did I want to eat it.\
":lines>
Make sure to print the reversed lines of the poem.
>
> poem.lines.to_s.reverse
">b3d43047x0:rotaremunE<#"
Make sure to print the reversed lines of the poem.
> poem.lines.to_a.reverse
["More still did I want to eat it.\
", "Planting our flag on Halley's comet,\
", "But when I saw it on television,\
", "And my toast has gone to the moon.\
", "My honeydew has flown from my hand\
"]
Make sure to print the reversed lines of the poem.
>
> poem.length
173
```

Ringlets of Chained Methods

Objective:

So...what actually happened there? You typed `poem.lines.to_a.reverse` and produced some Ruby magic.

First, you turned the `poem` into a list using `lines.to_a`. The `lines` component decided the way the string should be split up, and then one of our "to" methods, `to_a`, converted those splits into an Array. (**to_a**rray.)

Different methods, such as `bytes` and `chars` can be used in place of `lines`. By using `lines` here, Ruby split the poem up according to each new line.

After that, you `reverse`'d your Array. You had each line prepared in advance. And then you reversed them. That's it!

And now, let's tack one more method on the end there, if you don't mind. Try: `print poem.lines.to_a.reverse.join`.

```
> print poem.lines.to_a.reverse
["More still did I want to eat it.\
", "Planting our flag on Halley's comet,\
", "But when I saw it on television,\
", "And my toast has gone to the moon.\
", "My honeydew has flown from my hand\
"]
> print poem.lines.to_a.reverse.join
More still did I want to eat it.
Planting our flag on Halley's comet,
But when I saw it on television,
And my toast has gone to the moon.
My honeydew has flown from my hand
Success!
>
```

Brace Yourselves! Level #4 is Here Now

Objective:

Good show, my friend! The `join` method took that list of reversed lines and put them together into a single string. (Sure, you could have also just used `to_s`.)

Time for a quick review.

- **Exclamation Points.** Methods may have exclamation points in their name, which just means to impact the current data, rather than making a copy. No big deal.
- **Square Brackets.** With these, you can target and find things. You can even replace them if necessary.
- **Chaining** methods lets you get a lot more done in a single command. Break up a poem, reverse it, reassemble it: `poem.lines.to_a.reverse.join`.

Guess what? Methods can also have question marks. Try: `poem.include? "my hand"` to check it out.

At this point, you may want to tinker with the poem a bit more. A complete list of all the `String` methods is here. Go ahead and try a few (such as `poem.downcase` or `poem.delete`.)

And now on to something new. When you're ready to move on, type: `books = {}`

```
> poem.include? "my hand"
=> true
> poem.include
=>
> poem
=> "My honeydew has flown from my hand
And my toast has gone to the
     moon.
But when I saw it on television,
Planting our flag on Halley's
     comet,
More still did I want to eat it.
"
> poem.include? "my arms"
=> false
> poem.include? "gone to the moon"
=> false
> poem.include? "gone to the"
=> true
>
```

```
> poem.include? "my hand"
=> true
> poem.include
=>
> poem
=> "My honeydew has flown from my hand
And my toast has gone to the
        moon.
But when I saw it on television,
Planting our flag on Halley's
        comet,
More still did I want to eat it.
"
> poem.include? "my arms"
=> false
> poem.include? "gone to the moon"
=> false
> poem.include? "gone to the"
=> true
> poem.downcase
=> "my honeydew has flown from my hand
and my toast has gone to the
        moon.
but when i saw it on television,
planting our flag on halley's
        comet,
more still did i want to eat it.
"
> poem.upcase
=> "MY HONEYDEW HAS FLOWN FROM MY HAND
AND MY TOAST HAS GONE TO THE
        MOON.
BUT WHEN I SAW IT ON TELEVISION,
PLANTING OUR FLAG ON HALLEY'S
        COMET,
MORE STILL DID I WANT TO EAT IT.
"
> poem.sentencecase
=>
> poem.delete
=>
> books = {}
=> {}
Success!
> books
```

```
=> {}
> books.length
=> 0
> books.isempty
=>
> books.isemp
=>
> books.
.. books
=> {}
>
```

A Wee Blank Book

Objective:

You've made an empty **hash**, also known as: a *dictionary*. Hashes store related information by giving reusable labels to pieces of our data.

We're going to stuff some miniature book reviews in this hash. Here's our rating system:

- `:splendid` → a masterpiece.
- `:quite_good` → enjoyed, sure, yes.
- `:mediocre` → equal parts great and terrible.
- `:quite_not_good` → notably bad.
- `:abysmal` → steaming wreck.

To rate a book, put the title in square brackets and put the rating after the equals.

For example: `books["Gravity's Rainbow"] = :splendid`

```
> books["Gravity's Rainbow"] = :kewl
=> :kewl
> books["Gravity's Rainbow"] = :splendid
=> :splendid
Success!
>
```

More Bite-Size Reviews

Objective:

Keep going! Fill it up with some useful reviews. And if you want to see the whole list, just type: `books`

Again, the available ratings are: `:splendid`, `:quite_good`, `:mediocre`, `:quite_not_good`, and `:abysmal`.

Notice that these ratings are not strings. When you place a colon in front of a simple word, you get a Ruby **symbol**. Symbols are much cheaper than strings (in terms of computer memory.) If you need to use a word over and over in your program itself, use a symbol. Rather than having thousands of copies of that word in memory, the computer will store a symbol only *once*, and refer to it over and over.

Once you've got **three** or **four** books in there, type: `books.length`. You should see the right amount.

```
> books.length
=> 1
>
```

Wait, Did I Like Gravity's Rainbow?

Objective:

See, the `length` method works on strings, list and hashes. One great thing about Ruby is that method names are often reused, which means a lot less stuff for you to remember.

If you'd like to look up one of your old reviews, just put the title of the book in the square box again. Leave off the equal sign this time, though, since you're not assigning any information. You're just researching!

Do it like this: `books["Gravity's Rainbow"]`

```
> books["Gravity's Rainbow"]
=> :splendid
Success!
```

Hashes as Pairs

Objective:

Keep in mind that hashes won't keep things in order. That's not their job. It'll just pair up two things: a **key** and a **value**. In your reviews, the key is the book's title and the value is the rating, in this case a **symbol**.

If you want to see a nice list of the book titles you've reviewed: `books.keys`

When you want to move on, type `next`

```
> books.keys
=> ["Gravity's Rainbow"]
```

Are You Harsh?

Objective:

So are you giving out harsh, unfair reviews? Let's keep score with this hash: `ratings = Hash.new(0)`

```
> ratings = Hash.new(-1)
=> {}
Success!
> ratings = Hash.new(0)
=> {}
>
```

Are You Harsh?

Objective:

That command was another way to build an empty hash. The zero you passed in will set all of your initial rating counts to zero.

Okay, now let's count up your reviews. Stay with me on this one.

Type: `books.values.each { |rate| ratings[rate] += 1 }`

(That `|` in the code is called the pipe character. It's probably located right above the Enter key on your keyboard.)

This code will turn all your unique *values* in `books`...into *keys* within the new `ratings`hash. Crazy, right? Then, as it looks at each rating you originally gave in `books`, it will increase the count *value* for that rating in `ratings`

After you've built your new hash of count values, type `ratings` again to see the full tally. This new hash will show you a rating followed by the number of times you've given that rating.

When you want to move on, type `next`

```
books.values.each{|rate|ratings[rate]+=1}
=> [:splendid]
> books["Gravity's Rainbow"]
=> :splendid
Success!
> books.keys
=> ["Gravity's Rainbow"]
> next
> ratings = Hash.new(-1)
=> {}
Success!
> ratings = Hash.new(0)
=> {}
> books.values.each{|rate|ratings[rate]+=1}
```

```
=> [:splendid]
> ratings
=> {:splendid=>1}
> ratings
=> {:splendid=>1}
> books.values.each{|rate|ratings[rate]+=2}
=> [:splendid]
> ratings
=> {:splendid=>3}
> books.values.each{|rate|ratings[rate]*=2}
=> [:splendid]
> ratings
=> {:splendid=>6}
> books.values.each{|rate|ratings[rate] -=2}
=> [:splendid]
> ratings
=> {:splendid=>4}
> books.values.each{|rate|ratings[rate]/=2}
=> [:splendid]
> ratings
=> {:splendid=>2}
> books.values.each{|rate|ratings[rate]-=2}
=> [:splendid]
> ratings
=> {:splendid=>0}
> books.values.each{|rate|ratings[rate]-=2}
=> [:splendid]
> ratings
=> {:splendid=>-2}
> books.values.each{|rate|ratings[rate]-=2}
=> [:splendid]
> books.values.each{|rate|ratings[rate]-=1}
=> [:splendid]
> next
```

A Tally

Objective:

One of the amazing new things we've just used is called a **block**. Basically, a block is a chunk of Ruby code surrounded by curly braces. We'll take a closer look at them later.

But for now, let's try another block:

```
5.times { print "Odelay!" }
```

When you want to move on, type `next`. You want the badge, don't you?

```
> 5.times {print "Odelay!"}
=> "Odelay!Odelay!Odelay!Odelay!Odelay!"
Success!
> 5 * {print "Odelay!"}
=>
> 5 *  "Odelay!"
=>
> "Odelay!" * 5
=> "Odelay!Odelay!Odelay!Odelay!Odelay!"
Success!
>
```

Now Arriving at Level #5

Objective:

Blocks are always attached to methods. You saw this with the `times` method, which took the block and ran its code over and over. (In this case: **five** times.)

This last lesson was a bit longer. You've probably used up three minutes learning about:

- **Hashes.** The little 'dictionary' with the curly braces: `{}`.
- **Symbols.** Tiny, efficiently reusable code words with a colon: `:splendid`.
- **Blocks.** Chunks of code which can be tacked on to many of Ruby's methods. Here's the code you used to build a scorecard: `books.values.each { |rate| ratings[rate] += 1 }`.

On to the next thing, okay? On your computer, you probably have a lot of different files. Some files have pictures in them, some have programs in them. And files are often organized into folders, also called: **directories**.

I've prepared a few directories for you. Take a look, using the following command:
`Dir.entries "/"`

```
> Dir.entries "/"
=> [".", "..", "Home", "Libraries", "MouseHole", "Programs", "Tutorials", "comics.txt"]
Success!
>
```

The Private Collection of Dr. Dir

Objective:

You've just listed out everything in the top directory, which is called the *root*. It's indicated by the single slash in your string parameter. It contains some programs, as well as other tutorials and such.

So, what exactly is that `Dir.entries` method? Well, it's just a method, like the others you've seen. `Dir` has a collection of methods for checking out file directories, and `entries` is being called *on* the `Dir` variable. The `entries` method just lists everything in the directory you've indicated!

One other little thing we haven't really talked about quite yet: method arguments. A few are highlighted below.

- `Dir.entries` **`"/"`** -- Anything listed after a method is considered an 'attachment'.
- `print` **`poem`** -- See, `print` is just an ordinary method, while the poem is what got attached for printing.
- `print` **`"pre", "event", "ual", "ism"`** -- This bit has several arguments! Ruby makes us use commas to distinguish between them.

Next up, we'll list just the text files in our root directory using a bracket notation. Remember how it searches?

Try: `Dir["/*.txt"]`

```
> Dir["/*.txt"]
=> ["/comics.txt"]
Success!
```

Come, Read Comics With Me

Objective:

The `Dir[]` syntax is kind of like `entries`, but instead searches for files with wildcard characters.

Here, we see those square brackets again! Notice how they still mean, "I am looking for _____?"

`Dir["/*.txt"]` says to Ruby: "I am looking for any files which end with `.txt`." The asterisk indicates the "any file" part. Ruby then hands us every file that matches our request.

Alright, let's crack open this comics file, then! We'll use a new method to do it.

Here's the way: `print File.read("/comics.txt")`

```
> print File.read("/comics.txt")
=> "Achewood: http://achewood.com/
Dinosaur Comics: http://qwantz.com/
Perry Bible Fellowship: http://cheston.com/pbf/archive.html
Get Your War On: http://mnftiu.cc/
"
Success!
> print File.read("/comics.txt");
=> nil
can't find '/Home/comics.txt', where did you put it?
> FileUtils.cp('/comics.txt', '/Home/comics.txt')
=> nil
Success!
>
```

Mi Comicas, Tu Comicas

Objective:

Okay, you've got a copy, and it's located in the right directory. Check it out!

Use `Dir["/Home/*.txt"]`

Type `next` to move to the next lesson when you're finished.

```
> Dir["/Home/*.txt"]
=> ["/Home/comics.txt"]
> next
```

Your Own Turf

Objective:

To add your own comic to the list, let's open the file in **append** mode, which we indicate with the "a" parameter. This will allow us to put new stuff at the *end* of the file.

Start by entering this code: `File.open("/Home/comics.txt", "a") do |f|`

```
> File.open("/Home/comics.txt", "a") do |f|
..


.. f << "Cat and Girl: http://catandgirl.com" end
=> #<File:/Home/comics.txt (closed)>
>
```

Ruby Sits Still

Objective:

The last line will add the *Cat and Girl* comic to the list, but Ruby's going to wait until you're totally finished to take action.

This means we'll also need to wrap up the code block you've started. Turns out, you actually opened a new block when you typed that `do` keyword.

So far the blocks we've seen have used curly braces, but this time we'll be using `do` and `end` instead. A lot of Rubyists will use a `do...end` setup when the block goes on for many lines.

Let's get that block finished now, with your very own `end`.

```
File.open("/Home/comics.txt", "a") do |f|
  f << "Cat and Girl: http://catandgirl.com/"
end
```

```
> File.open("/Home/comics.txt", "a") do |f| f << "Cat and Girl: http://catandgirl.com/" end
=> #<File:/Home/comics.txt (closed)>
>
```

And Now For the Startling Conclusion

Objective:

So your prompt has *changed*, see that? Your prompt is a **double** dot now.

In this tutorial, this prompt means that Ruby is expecting you to type a little bit more. As you write further lines of Ruby code, the double-dots will continue until the tutorial sees you are completely finished.

Alright, so here's more code. You've already typed the first line, so just enter the second line.

```
File.open("/Home/comics.txt", "a") do |f|
  f << "Cat and Girl: http://catandgirl.com/"
end
```

```
> File.open("/Home/comics.txt", "a") do |f|
.. f << "Cat and Girl: http://catandgirl.com/"
.. end
=> #<File:/Home/comics.txt (closed)>
>
```

Ruby Sits Still

Objective:

Sweet! You've added that brand new comic to the end of the file. You can see for yourself, using the `read` method you saw earlier: `print File.read("/Home/comics.txt")`.

When you want to move on to the next lesson, type `next`.

```
> next
```

The Clock Nailed To the File

Objective:

I wonder, what time was it when you changed your file? We can check that out.

Type: `File.mtime("/Home/comics.txt")`.

```
> File.mtime("/Home/comics.txt")
=> 2015-05-13 18:26:01 UTC
```

```
Success!
>
```

Just the Hour Hand

Objective:

Excellent, there's the exact time, precisely when you added to the file. The `mtime` method gives you a nice Ruby Time object.

If you want to check just what hour it was, hit the up arrow key to put your previous entry in the console. Then modify the line to say: `File.mtime("/Home/comics.txt").hour`.

```
> File.mtime("/Home/comics.txt").hour
=> 18
Success!
>
```

Level 6 Descends. The End is Near

Objective:

Well done, well done, well done, well done! Truly, truly, truly, truly, truuuuuuuuly!

Here's the last few minutes of your life in review:

- **Files.** -- Lots of methods exist for editing files and looking around in directories.
- **Arguments.** -- Arguments are a list of things sent into a method, separated by commas.
- **Block Changes.** -- You can use **do** and **end** as another way to make a code block.

Welp, you totally know how to use Ruby now. I mean, you've got down the essentials, right? You just need to keep learning more methods and try out more complex blocks on your way to mastery.

But, there's still one side of Ruby we haven't settled: building your own methods!

Ahem! Let's get it over with, then.

Start building our new method with: `def load_comics( path )`

```
> def load_comics(path)
..
```

In Ruby, Def Leppard Means Define Leppard (a Method)!

Objective:

Great, you've started it up. You're making your own method! First, you used `def`, followed by the name of the method. Next came a list of arguments which the method will need in order to work when you need it. Don't worry, this isn't too scary and dangerous!

All we have to do is fill it up with Ruby and finish it up with `end`. Feel free to quickly copy and paste.

Here's the code:

```
def load_comics( path )
  comics = {}
  File.foreach(path) do |line|
    name, url = line.split(': ')
    comics[name] = url.strip
  end
  comics
end
```

Though the indentation isn't strictly necessary, it's a good habit to develop so that people can read your code easily.

```
> def load_comics(path)
.. comics = {}
.. File.foreach(path) do |line|
.... name, url = line.split(': ')
.... comics[name] = url.strip
.... end
.. comics
.. end
=> nil
>
```

The Ripened Fruit of Your Own Creation

Objective:

A new method is born. And now, let us use it! Type `comics = load_comics('/comics.txt')`.

Then, type `next` to start using your new `comics`.

```
> comics = load_comics('/comics.txt')
=> {"Achewood"=>"http://achewood.com/", "Dinosaur Comics"=>"http://qwantz.com/", "Perry Bible Fellowship"=>"http://cheston.com/pbf/archive.html", "Get Your War On"=>"http://mnftiu.cc/"}
> next
>
```

Hey, Cool, a Comics Thing

Objective:

In your Ruby console, look at the earlier code you wrote for the `load_comics` method.

What is happening here? You're passing in the `path` variable as an argument, and you're getting back the `comics` variable just before the end of the method. Ruby looks for something to return just before a method's end.

A number of methods were used to get the entire job done. See if you can spot them.

- **File.foreach** -- This method opens a file and hands each line of the file to the block. The `line` variable inside the `do...end` block took turns with each line in the file.
- **split** -- A method for strings which breaks the string up into an array, removing the piece you pass in. An axe is laid on the colon and the line is chopped in half, giving us the data to be stored in `url` and `name` for each comic.
- **strip** -- This quickie removes extra spaces around the url. Just in case.

Right on! Bravo. You've got the comics in a Ruby hash. But, uh, what now? What good is this work, really?

Welp, we can use it to make a page of links. Links are useful and cool, right? First, we'll need to load a little library I've made for you.

Here's the code for loading it: `require 'popup'`.

```
> require 'popup'
=> true
Success!
>
```

Browser Puppetry

Objective:

Excellent, you've loaded the *popup* library. It's saved in a file in the *Libraries* folder.

The popup library contains a bunch of methods I've written which will let you control a popup window here on the Try Ruby site.

For example, try this one: `Popup.goto "http://bing.com"`

When you've got that entered, checkout the new "Popup" tab that appeared near the top of your console! Click it to see the website you just loaded with the goto method. You can even try another one if you feel like it.

Once you're done investigating, type `next` to go to the next challenge.

> Popup.goto http://bing.com
> next

Making Links and Spinning Webs

Objective:

Well now, we've got our own lovely, little popup to manipulate. You can also fill it with your own goodies. We'll start small:

```
Popup.make {
  h1 "My Links"
  link "Go to Bing", "http://bing.com"
}
```

The term `h1` (*h-one*) means a level-one header. In HTML, this is the largest header size available. The term `link` is exactly what you'd think: a link to website.

When you've built the code (by entering it one line at a time), check the Popup tab to see your results.

And when the road finally calls again, type in `next` to move forward. The end is close.

```
> Popup.make {
.. h1 "My Links"
.. link "Go to Bing", "http://bing.com"
.. }
>
```

Popups Are So Easy, It's Crazy

Objective:

Looks great! You did it perfectly. So now, let's make a list.

Here's how you make a list with the popup library:

```
Popup.make do
  h1 "Things To Do"
  list do
    p "Try out Ruby"
    p "Ride a tiger"
    p "(down River Euphrates)"
  end
end
```

The `p` method is short for "paragraph". It will ensure each of your list items gets its own paragraph spot.

When you want to move on, `next` will bring you to the last moments of the course...

```
> Popup.make do
.. h1 "Things To Do"
.. list do
.... p "Try out Ruby"
.... p "Ride a tiger"
.... p "(down River Euphrates)"
.... end
.. end
>

> next
```

Spread the Comics on the Table

Objective:

Your journey has progressed wonderfully. You may think that this may be simple stuff, but keep in mind that you didn't know *any Ruby whatsoever* just a little while ago!

It's time for the last step. Let's tie it all together, you know? Time to make it all sing together like a very nice set of glistening chimes on the beach in the maginificent sunlight! Yeah, imagery!

Remember: you have already loaded your comics with `comics = load_comics( '/comics.txt' )`.

Now, let's make a list of the links to each comic:

```
Popup.make do
  h1 "Comics on the Web"
  list do
    comics.each do |name, url|
      link name, url
    end
  end
end
```

This ties each name of a comic to a url, creating a link! You can even click on the links and read the comics in the little window! Smashing! Dashing!

Congrats on completing Try Ruby! You can play through this course again by typing `next`.

If you're interested in Ruby on Rails, then check out Code School's Rails for Zombies course! Or if you want to continue your quest towards Ruby mastery, you can play the first level of Ruby Bits for free!

```
> Popup.make do
.. h1 "Comics on the Web"
.. list do
.... comics.each do |name, url|
...... link name, url
...... end
.... end
.. end
>
```