**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](http://ruby-doc.org/core/classes/String.html). 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 ratingshash. 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!

>