“Anjali”

"Anjali".r­everse

=> "ilajnA" //reverse string

**Success!**

**>**"Anjali".l­ength

=> 6 //tells length of string

**Success!**

**>**"Anjali"\*5­

=> "AnjaliAnjaliAnjaliAnjaliAnjali"

Methods are actions.

The message is telling you that there is no methodreverse for number values in Ruby!

But, hmm...maybe if you can turn it into a string. Try this: 40.to\_s.reverse gives 04

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.

Create an empty array by

[]

**>**[12,47,35]­

=> [12, 47, 35]

**Success!**

//created an array

**>**[12,47,35]­.max //gives maximum number

**> ticket=**[12,47,35]­ //save into variable

> ticket.sort!

//! Mark implies to make changes in the current array only and not create a new array

//sorts the array permanently

**>**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!**

**>**poem['toas­t']= 'Anja­li'

=> "Anjali"

**Success!**

**>**print poem

=> "My Anjali 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."

//only the first ‘toast’ changes to ‘Anjali’

**>**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!**

//reverse 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  
"]

**Success!**

//convert lines to array and then reverse the array

**>**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!**

//make array into a poem(string) again by ‘join’

* **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.

**>**poem.inclu­de? "my hand"­

=> true

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

**>**books={}

=> {} //create empty hash

**Success!**

**>**books["The­ Reven­ant"]= :spl­endid

=> :splendid

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

**>**ratings= Hash.­new(0)

=> {}

**Success!**

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.

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

*(That*|*in the code is called the pipe character.)*

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, typeratings 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.

**>**books.valu­es.each { |rate­| ratin­gs[rate] += 1}

=> [:splendid]

**>**ratings

=> {:splendid=>1}

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.

**>**5.times{pr­int "Odel­ay!"}

=> "Odelay!Odelay!Odelay!Odelay!Odelay!"

Blocks are always attached to methods. Eg- ‘times’ here.

* **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 }

**>**Dir.entrie­s "/"

=> [".", "..", "Home", "Libraries", "MouseHole", "Programs", "Tutorials", "comics.txt"]

**Success!**

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, andentries is being called *on* the Dir variable. The entriesmethod 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"]

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.

**>**Dir.entrie­s "/"

=> [".", "..", "Home", "Libraries", "MouseHole", "Programs", "Tutorials", "comics.txt"]

**Success!**

**>**Dir["/\*.tx­t"]

=> ["/comics.txt"]

**Success!**

**>**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!**

Now we can start to use files to store things. This is great because normally when we exit Ruby, all our variables will be gone. Ruby, by itself, forgets these things. But if we save things in files, we can read those files in future Ruby escapades.

First thing we'll do is make a copy of the comics file and put in new folder called 'Home'.

To do that, you'll want to use a copying method calledcp on a variable called FileUtils.

**>**FileUtils.­cp('/comic­s.txt', '/Hom­e/comics.t­xt')

=> nil

**Success!**

**>**Dir["/Home­/\*.txt"]

=> ["/Home/comics.txt"]

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.

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 ownend.

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

**..** f<< "Cat and Girl:­ http:­//catandgi­rl.com/"

**..** end

=> #<File:/Home/comics.txt (closed)>

**>**print File.­read("/Hom­e/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/  
Cat and Girl: http://catandgirl.com/”

**>**File.mtime­("/Home/co­mics.txt")­

=> 2016-06-07 11:32:03 UTC

**Success!**

//time of edit of file

**>**File.mtime­("/Home/co­mics.txt")­.hour

=> 11

**Success!**

**//just the hour**

A new method is born. And now, let us use it! Type

comics = load\_comics('/comics.txt').

**>**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/"}

You're passing in the pathvariable as an argument, and you're getting back the comicsvariable just before the end of the method. Ruby looks for something to return just before a method's end.

* **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 quick method removes extra spaces around the url. Just in case.

> def load\_­comics(pat­h)

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

=> nil

code for loading a library ‘popup’ : require 'popup'

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.

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 termlink is exactly what you'd think: a link to website.

Popup.make do

h1 "Things To Do"

list do

p "Try out Ruby"

p "Ride a tiger"

p "(down River Euphrates)"

end

end