Ruby Programming/Data types

Contents

Ruby Data Types

As mentioned in the previous chapter, everything in Ruby is an object. Ruby has 8 primary data types and 3 more data types derives from the Numeric superclass. Everything has a class. Don't believe me? Try running this bit of code:

```
h = {"hash?" => "yep, it\'s a hash!", "the answer to everything" => 42, :linux => "fun for coders."}
puts "Stringy string McString!".class
puts 1.class
puts 1.class.superclass
puts 1.class.superclass.superclass
puts 4.3.class
puts 4.3.class.superclass
puts nil.class
puts h.class
puts :symbol.class
puts [].class
puts (1..8).class
```

displays

```
String
Fixnum
Integer
Numeric
Float
Numeric
NilClass
Hash
Symbol
Array
Range
```

See? Everything is an object. Every object has a method called `class` that returns that object's class. You can call methods on pretty much anything. Earlier you saw an example of this in the form of `3.times`. (Technically when

you call a method you're sending a message to the object, but I'll leave the significance of that for later.)
Something that makes this extreme object oriented-ness very fun for me is the fact that all classes are open, meaning you can add variables and methods to a class at any time during the execution of your code. This, however, is a discussion of datatypes.

Constants

We'll start off with constants because they're simple. Two things to remember about constants:
1. Constants start with capital letters. `Constant` is a constant. `constant` is not a constant.
2. You can change the values of constants, but Ruby will give you a warning. (Silly, I know... but what can you do?)
Congrats. Now you're an expert on Ruby constants.

Symbols

So did you notice something weird about that first code listing? "What the heck was that colon thingy about?" Well, it just so happens that Ruby's object oriented ways have a cost: lots of objects make for slow code. Every time you type a string, Ruby makes a new object. Regardless of whether two strings are identical, Ruby treats every instance as a new object. You could have "live long and prosper" in your code once and then again later on and Ruby wouldn't even realize that they're pretty much the same thing. Here is a sample irb session which demonstrates this fact:

```
irb> "live long and prosper".object_id
=> -507772268
irb> "live long and prosper".object_id
=> -507776538
```

Notice that the object ID returned by irb Ruby is different even for the same two strings.

To get around this memory hoggishness, Ruby has provided "symbols." `Symbol`s are lightweight objects best used for comparisons and internal logic. If the user doesn't ever see it, why not use a symbol rather than a string? Your code will thank you for it. Let us try running the above code using symbols instead of strings:

```
irb> :my_symbol.object_id
=> 150808
irb> :my_symbol.object_id
=> 150808
```

Symbols are denoted by the colon sitting out in front of them, like so: `:symbol_name`

Hashes

Hashes are like dictionaries, in a sense. You have a key, a reference, and you look it up to find the associated object, the definition.

The best way to illustrate this, I think, is with a quick demonstration:

```
1 hash = { :leia => "Princess from Alderaan", :han => "Rebel without a cause", :luke => "Farmboy turned Jedi"}
2 puts hash[:leia]
3 puts hash[:han]
4 puts hash[:luke]
```

displays

```
Princess from Alderaan
Rebel without a cause
Farmboy turned Jedi
```

I could have also written this like so:

```
1 hash = { :leia => "Princess from Alderaan", :han => "Rebel without a cause", :luke => "Farmboy turned Jedi"}
2 hash.each do |key, value|
3       puts value
4 end
```

This code cycles through each element in the hash, putting the key in the `key` variable and the value in the `value` variable, which is then displayed

```
Princess of Alderaan
Rebel without a cause
Farmboy turned Jedi
```

I could have been more verbose about defining my hash; I could have written it like this:

```
1 hash = Hash.[](:leia => "Princess from Alderaan", :han => "Rebel without a cause", :luke => "Farmboy turned Jedi")
2 hash.each do |key, value|
3       puts value
4 end
```

If I felt like offing Luke, I could do something like this:

```
1 hash.delete(:luke)
```

Now Luke's no longer in the hash. Or lets say I just had a vendetta against farmboys in general. I could do this:

```
1 hash.delete_if {|key, value| value.downcase.match("farmboy")}
```

This iterates through each key-value pair and deletes it, but only if the block of code following it returns `true`. In the block I made the value lowercase (in case the farmboys decided to start doing stuff like "FaRmBoY!1!") and then checked to see if "farmboy" matched anything in its contents. I could have used a regular expression, but that's another story entirely.

I could add Lando into the mix by assigning a new value to the hash:

```
1 hash[:lando] = "Dashing and debonair city administrator."
```

I can measure the hash with `hash.length`. I can look at only keys with the `hash.inspect` method, which returns the hash's keys as an `Array`. Speaking of which...

Arrays

Arrays are a lot like Hashes, except that the keys are always consecutive numbers, and always starts at 0. In an Array with five items, the *last* element would be found at array[4] and the *first* element would be found at array[0]. In addition, all the methods that you just learned with Hashes can also be applied to Arrays.

Here are two ways to create an Array:

```
1 array1 = ["hello", "this", "is", "an", "array!"]
2 array2 = []
3 array2 << "This"   # index 0
4 array2 << "is"     # index 1
5 array2 << "also"   # index 2
6 array2 << "an"     # index 3
7 array2 << "array!" # index 4
```

As you may have guessed, the << operator pushes values onto the end of an Array. If I were to write puts array2[4] after declaring those two Arrays the output would be array!. Of course, if I felt like simultaneously getting array! and deleting it from the array, I could just Array.pop it off. The Array.pop method returns the *last* element in an array and then immediately removes it from that array:

```
1 string = array2.pop
```

Then string would hold array! and array2 would be an element shorter.
If I kept doing this, array2 wouldn't hold any elements. I can check for this condition by calling the Array.empty? method. For example, the following bit of code moves all the elements from one Array to another:

```
1 array1 << array2.pop until array2.empty?
```

Here's something that really excites me: Arrays can be subtracted from, and added to, each other. I can't vouch for *every* language that's out there, but I know that Java, C++, C# and perl would all look at me like I was a crazy person if I tried to execute the following bit of code:

```
1 array3 = array1 - array2
2 array4 = array1 + array2
```

After that code is evaluated, all of the following are true:

- array3 contains all of the elements that array1 did, except the ones that were also in array2.
- All the elements of array1, minus the elements of array2, are now contained within array3.
- array4 now contains all the elements of both array1 and array2.

You may search for a particular value in variable array1 with the Array.include? method: array1.include?("Is this in here?")

If you just wanted to turn the whole Array into a String, you could:

```
1 string = array2.join(" ")
```

If array2 had the value that we declared in the last example, then string's value would be

```
This is also an array!
```

We could have called the `Array.join` method without any arguments:

```
1 string = array2.join
```

`string`'s value would now be

```
Thisisalsoanarray!
```

Strings

I would recommend reading the chapters on strings and alternate quotes now if you haven't already. This chapter is going to cover some pretty spiffy things with `String`s and just assume that you already know the information in these two chapters.

In Ruby, there are some pretty cool built-in functions where `String`s are concerned. For example, you can multiply them:

```
"Danger, Will Robinson!" * 5
```

yields

```
Danger, Will Robinson!Danger, Will Robinson!Danger, Will Robinson!Danger, Will Robinson!Danger, Will Robinson!
```

`String`s may also be compared:

```
"a" < "b"
```

yields

```
true
```

The preceding evaluation is actually comparing the ASCII values of the characters. But what, I hear you ask, is the ASCII value of an given character? With ruby versions prior to 1.9 you can find the ASCII value of a character with:

```
1 puts ?A
```

However, With Ruby version 1.9 or later that no longer works (http://stackoverflow.com/questions/1270209/getting-an-ascii-character-code-in-ruby-fails). Instead, you can try the `String.ord` method:

```
1 puts "A".ord
```

Either approach will display

```
65
```

which is the ASCII value of `A`. Simply replace `A` with whichever character you wish to inquire about.

To perform the opposite conversion (from `65` to `A`, for instance), use the `Integer.chr` method:

```
1 puts 65.chr
```

displays

```
A
```

Concatenation works the same as most other languages: putting a `+` character between two `String`s will yield a new `String` whose value is the same as the others, one after another:

```
1 "Hi, this is " + "a concatenated string!"
```

yields

```
Hi, this is a concatenated string!
```

For handling pesky `String` variables without using the concatenate operator, you can use interpolation. In the following chunk of code `string1`, `string2`, and `string3` are identical:

```
1 thing1 = "Red fish, "
2 thing2 = "blue fish."
3 string1 = thing1 + thing2 + " And so on and so forth."
4 string2 = "#{thing1 + thing2} And so on and so forth."
5 string3 = "#{thing1}#{thing2} And so on and so forth."
```

If you need to iterate through (that is, step through each of) the letters in a `String` object, you can use the `String.scan` method:

```
1 thing = "Red fish"
2 thing.scan(/./) {|letter| puts letter}
```

Displays each letter in `thing` (puts automatically adds a newline after each call):

```
R
e
d

f
i
s
h
```

But what's with that weird "`/./`" thing in the parameter? That, my friend, is called a *regular expression*. They're helpful little buggers, quite powerful, but outside the scope of this discussion. All you need to know for now is that `/./` is "regex" speak for "any *one* character." If we had used `/../` then ruby would have iterated over each group of two characters, and missed the last one since there's an odd number of characters!

Another use for regular expressions can be found with the `=~` operator. You can check to see if a `String` matches a regular expression using the match *operator*, `=~`:

```
1 puts "Yeah, there's a number in this one." if "C3-P0, human-cyborg relations" =~ /[0-9]/
```

displays

```
Yeah, there's a number in this one.
```

The `String.match` method works much the same way, except it can accept a `String` as a parameter as well. This is helpful if you're getting regular expressions from a source outside the code. Here's what it looks like in action:

```
1 puts "Yep, they mentioned Jabba in this one." if "Jabba the Hutt".match("Jabba")
```

Alright, that's enough about regular expressions. Even though you can use regular expressions with the next two examples, we'll just use regular old `String`s. Lets pretend you work at the Ministry of Truth and you need to replace a word in a `String` with another word. You can try something like:

```
1 string1 = "2 + 2 = 4"
2 string2 = string1.sub("4", "5")
```

Now `string2` contains `2 + 2 = 5`. But what if the `String` contains lots of lies like the one you just corrected? `String.sub` only replaces the first occurrence of a word! I guess you could iterate through the `String` using `String.match` method and a **`while`** loop, but there's a much more efficient way to accomplish this:

```
1 winston = %q{   Down with Big Brother!
2         Down with Big Brother!
3         Down with Big Brother!
4         Down with Big Brother!
5         Down with Big Brother!}
6 winston.gsub("Down with", "Long live")
```

Big Brother would be *so* proud! `String.gsub` is the "*g*lobal *sub*stitute" function. Every occurrence of "`Down with`" has now been replaced with "`Long live`" so now `winston` is only proclaiming its love for Big Brother, not its disdain thereof.
On that happy note, lets move on to `Integer`s and `Float`s. If you want to learn more about methods in the `String` class, look at the end of this chapter for a quick reference table.

Numbers (Integers and Floats)

You can skip this paragraph if you know all the standard number operators. For those who don't, here's a crash course. `+` adds two numbers together. `-` subtracts them. `/` divides. `*` multiplies. `%` returns the remainder of two divided numbers.

Alright, integers are numbers with no decimal place. Floats are numbers with decimal places. `10 / 3` yields `3` because dividing two integers yields an integer. Since integers have no decimal places all you get is `3`. If you tried `10.0 / 3` you would get `3.33333...` If you have even one float in the mix you get a float back. Capice?

Alright, let's get down to the fun part. Everything in Ruby is an object, let me reiterate. That means that pretty much everything has at least one method. Integers and floats are no exception. First I'll show you some integer methods.
Here we have the venerable `times` method. Use it whenever you want to do something more than once. Examples:

```
puts "I will now count to 99..."
100.times {|number| puts number}
5.times {puts "Guess what?"}
puts "I'm done!"
```

This will print out the numbers 0 through 99, print out `Guess what?` five times, then say `I'm done!` It's basically a simplified `for` loop. It's a little slower than a `for` loop by a few hundredths of a second or so; keep that in mind if you're ever writing Ruby code for NASA. ;-)

Alright, we're nearly done, six more methods to go. Here are three of them:

```
# First a visit from The Count...
1.upto(10) {|number| puts "#{number} Ruby loops, ah-ah-ah!"}

# Then a quick stop at NASA...
puts "T-minus..."
10.downto(1) {|x| puts x}
puts "Blast-off!"

# Finally we'll settle down with an obscure Schoolhouse Rock video...
5.step(50, 5) {|x| puts x}
```

Alright, that should make sense. In case it didn't, `upto` counts up from the number it's called from to the number passed in its parameter. `downto` does the same, except it counts down instead of up. Finally, `step` counts from the number its called from to the first number in its parameters by the second number in its parameters. So `5.step(25, 5) {|x| puts x}` would output every multiple of five starting with five and ending at twenty-five.

Time for the last three:

```
string1 = 451.to_s
string2 = 98.6.to_s
int = 4.5.to_i
float = 5.to_f
```

`to_s` converts floats and integers to strings. `to_i` converts floats to integers. `to_f` converts integers to floats. There you have it. All the data types of Ruby in a nutshell. Now here's that quick reference table for string methods I promised you.

Additional String Methods

```
# Outputs 1585761545
"Mary J".hash

# Outputs "concatenate"
"concat" + "enate"
```

```ruby
# Outputs "Washington"
"washington".capitalize

# Outputs "uppercase"
"UPPERCASE".downcase

# Outputs "LOWERCASE"
"lowercase".upcase

# Outputs "Henry VII"
"Henry VIII".chop

# Outputs "rorriM"
"Mirror".reverse

# Outputs 810
"All Fears".sum

# Outputs cRaZyWaTeRs
"CrAzYwAtErS".swapcase

# Outputs "Nexu" (next advances the word up one value, as if it were a number.)
"Next".next

# After this, nxt == "Neyn" (to help you understand the trippiness of next)
nxt = "Next"
20.times {nxt = nxt.next}
```

Retrieved from "https://en.wikibooks.org/w/index.php?title=Ruby_Programming/Data_types&oldid=3067497"
