**Symbols**

Ruby has an additional data type that is similar to Strings, called **Symbols**. Let's explore what differentiates a Symbol from a String, and how to use them in our code. In Ruby, we can denote a symbol using a colon (:) before writing characters. Where a string is wrapped in quotes, a symbol just has a leading colon. Both strings and symbols contain many characters, but they are not equivalent.

str = "hello" # the string

sym = :hello # the symbol

p str.length # => 5

p sym.length # => 5

p str[1] # => "e"

p sym[1] # => "e"

p str == sym # => false

# a string is different from a symbol!

**Symbols are Immutable**

The most apparent difference between strings and symbols is that strings are mutable, while **symbols are immutable**. This means that string can be "changed", but a symbol can never be "changed":

str = "hello"

sym = :hello

str[0] = "x"

sym[0] = "x"

p str # => "xello"

p sym # => :hello

The utility of a symbol comes from the fact that it can never change over time. The technical implication of this is that a symbol only needs to be "created" once. There is no need to create "copies" of a symbol because we can be certain that it will not change over the course of our programs. Operations such as comparing two symbols is very fast and efficient compared to regular strings.

Under the hood, each time we reference a literal string, Ruby will allocate a piece of our machine's memory to store that string. More memory must always be allocated for a new string, even if it is a duplicate value, because strings are mutable! We must track changes to the strings separately, so we need to store the two instances of the string in distinct memory locations.

Talk of memory locations is pretty abstract, but an easy way to witness this is to use Ruby's object\_id method. This will return the memory address of some data. Notice how duplicate value strings will be stored at different memory locations:

"hello".object\_id # => 70233443667980

"hello".object\_id # => 70233443606440

"hello".object\_id # => 70233443438700

If we don't intend to mutate the string, we can use a symbol to save some memory. A symbol value will be stored in exactly one memory location:

:hello.object\_id # => 2899228

:hello.object\_id # => 2899228

:hello.object\_id # => 2899228

Because of these characteristics, symbols are often used to act as unique identifiers in our code. We'll be able to ensure the identifier will remain intact, without change, while also being efficient with memory.

**Symbols as hash keys**

We'll see the preference of using symbols in a few places in Ruby. For now, one common way to use a symbol is as the key in a hash:

my\_bootcamp = { :name=>"App Academy", :color=>"red", :locations=>["NY", "SF", "ONLINE"] }

p my\_bootcamp # => {:name=>"App Academy", :color=>"red", :locations=>["NY", "SF", "ONLINE"]}

p my\_bootcamp[:color] #=> "red

When initializing a hash with symbol keys, Ruby offers a shortcut. We can drop the rocket (=>) and move the colon (:) to the right of the symbol:

my\_bootcamp = { name:"App Academy", color:"red", locations:["NY", "SF", "ONLINE"] }

p my\_bootcamp # => {:name=>"App Academy", :color=>"red", :locations=>["NY", "SF", "ONLINE"]}

p my\_bootcamp[:color] #=> "red

This shortcut is only allowed when initializing the symbols in the hash. When getting a value from the hash after initialization, we must always put the colon on the left like normal. hash[:key] is the correct syntax. Writing hash[key:] is invalid.