Pass by Reference (For R Users)

Nick Eubank

April 21, 2018

Before you dive into Julia (or most other programming languages like Python, Java, C, etc.), theres one large conceptual change you should be aware of coming from R.

Say you make a new vector as follows:

```
my.vector <- c(1,2,3)
```

In R, theres no difference between a variable (like `my.vector`) and the object (the vector `[1, 2, 3]`) associated with it. But this is actually a slight of hand used by R to hide something fundamental about how computers work, and it does not happen in other languages.

In Julia, when you create an object like a vector, Julia puts that vector somewhere in memory, kind of like you might put something big on a shelf somewhere in a warehouse. The variable associated with that vector (`my.vector`) is not the same as that vector it actually just stores the location on the shelves where you placed that vector. And because this behavior is normal in most languages, you may not see it emphasized in Julia tutorials written by programmers not coming from R.

The reason this matters is that its possible for multiple variables to be pointed at the same item on a shelf, which means if you do something to one variable, it changes the item on the shelf, and so if you call the other variable that points to that item on a shelf, you will find the change affected both items. For example:

```
# Make a new vector
julia> x = [1, 2, 3]
3-element Array{Int64,1}:
 1
 2
 3


 # Make new var Y, and assign it x. In R this would make a copy.
julia> y = x
3-element Array{Int64,1}:
 1
 2
```

```
 3

 # change the last element of x
julia> x[3] = -99
-99

# We see this chnage in x
julia> x
3-element Array{Int64,1}:
   1
   2
 -99

 # But look! It also happened to y!
 # That's because they both point at the same vector on the shelf.
julia> y
3-element Array{Int64,1}:
   1
   2
 -99
```

If what you want to do is make a copy of x, you use the copy() command:

```
julia> x = [1, 2, 3]
3-element Array{Int64,1}:
 1
 2
 3

julia> y_copy = copy(x)
3-element Array{Int64,1}:
 1
 2
 3

julia> x[3] = -99
-99

julia> y_copy
3-element Array{Int64,1}:
 1
 2
 3
```

If you want to see if two variables point to the same thing, you can use the `===` (triple equal sign) operator, which tests whether two variables are pointed at the same place in

memory / same shelf:

```
julia> x = [1, 2, 3]
3-element Array{Int64,1}:
 1
 2
 3

julia> y = x
3-element Array{Int64,1}:
 1
 2
 3

julia> y_copy = copy(x)
3-element Array{Int64,1}:
 1
 2
 3

julia> x === y
true

julia> x === y_copy
false
```

0.1 Mutable versus Immutable Types

However, there is an exception to this behavior. Certain data types in Julia are called "immutable," meaning that if you try to change them, Julia cant modify the item thats already sitting on the shelf; instead, it has to create a new item on a different shelf and redirect the variable to point at that new item. The most common are plain numbers, strings, and tuples.

```
# Make x a simple number
julia> x = 5
5

julia> y = x
5

julia> x = x + 1
6
```

```
# y is unchanged because x + 1 actually created a new "6" on a new shelf, and x char
# to 5 to pointing to 6
julia> y
5
```

OK, thats it thats the one big, weird conceptual change to be aware of!