Reading Guide

Dan Maxwell

May 3, 2017

Section Notes for *R in Action*

4.4 Renaming Variables
Kabocoff only discusses the `fix()` function in this section. This function works the same way in the RStudio environment. It opens a window that allows you to edit individual cells in a dataframe. The `edit()` function does essentially the same thing, though Kabocoff does not discuss it here. And finally, you can use the `View()` function to examine and filter a dataframe without modifying it. Please note: this function is mixed case with a capital 'V'. Because R is case-sensitive, you'll get an error if you type it as all lower-case.

4.7 Type Conversions R is a *dynamically* typed language as opposed to a *statically* typed language like C. What this means is that the datatype of a variable is set at assignment and can be changed on the fly as demonstrated here.

```
                    # Static typed languages do not allow this...

myvar <- 15         # Set myvar to a numeric datatype.
is.numeric(myvar)   # Is it numeric? (TRUE/FALSE).
```

```
## [1] TRUE
```

```
myvar <- "String"   # Now dynamically recast myvar as a string.
is.character(myvar) # Is it a character? (TRUE/FALSE).
```

```
## [1] TRUE
```

A *statically* typed language like C would not compile this code. Instead, it would want us to define myvar once, as either a number or a character. After that, any assignment to myvar would need to match its underlying datatype.

So why is this important? Well, it becomes an issue when you write a custom function that accepts arguments. Because R is *dynamically* typed, the interpreter does not check the datatype of arguments upfront, when the function is called. Thus you'll want to verify argument datatypes inside the function, using one of the `is.` functions from Table 4.5. Consider the following function that accepts a single numeric argument.

```
# Create function tst() that expects one numeric argument.

tst <- function(num) {
                  # Fail to check the datatype of the argument with is.numeric().
  print(num)      # This command executes properly, whatever the datatype.

  num <- num + 1  # Add 1 to the variable -- we're assuming this is a numeric!

  return(num)
}
```

In this example, we fail to verify the argument's datatype, using the `is.numeric()` function. Now, try calling this function with a character string, like this: `tst("chars")`. R throws an error. But strangely, the `print()` statement displays our argument to the console, meaning that we are in the function and the

interpreter is happy with everything up to that point. It complains only when it encounters the next line which assumes a numeric variable. Thus R is not checking the datatype of passed arguments. You must do that yourself in a *dynamically* typed language.

4.10.4 The subset() Function

I favor the `subset()` function over the use of indexes as outlined in the previous sections. It has an intuitive feel to it and the code is easier to read. Even so, this is a personal preference and you might favor the use of indexes in your code.

Compare and contrast the two approaches. Which do you prefer?

```
myvar <- roster[roster$MAJOR.CODE == "CEX",]
myvar <- subset(roster, MAJOR.CODE == "CEX")
```