Introduction to R

Patrick Breheny

August 25, 2016

Our goal for today is to introduce R, an open-source computing language designed to allow for fluid, interactive data manipulation, analysis, and visualization. R is installed on all computers throughout the College of Public Health Building. If you are interested in installing it at home, go to www.r-project.org. You can run R directly, but it is often more convenient to run R through an integrated development environment; by far the most well-developed of these is RStudio www.rstudio.com, which is also (a) installed throughout the building and (b) open-source and easy to install.

Much of the material here is adapted from *S Programming*, by Venables and Ripley (2000), an excellent book on the details of the S and R languages that goes into far more detail than I do here.

1 R objects

Commands in R are either *expressions*, which are evaluated and printed, or *assignments*, which store the result of an evaluation as an object. Arithmetic operations for the most part work very similar to any calculator (note that # marks the rest of the line as a comment):

```
> (5^2)*(10-8)/3 + 1       ## An expression

[1] 17.66667

> x <- (5^2)*(10-8)/3 + 1  ## An assignment
> x

[1] 17.66667
```

Note that the value of the expression is now stored in an *object* called x. This allows us to use it again in further calculations:

```
> x+1

[1] 18.66667

> n <- 50
> x/n

[1] 0.3533333
```

Objects can be named using any combination of upper- and lower-case letters, digits 0-9 (provided they are not in the initial position), and the period and underscore. Note that R is case sensitive (x and X refer to two different objects).

All objects in R have a *class*, which describes the kind of thing that is stored in the object. For instance,

```
> class(x)

[1] "numeric"
```

tells us that `x` is storing a numeric object at the moment.

1.1 Functions

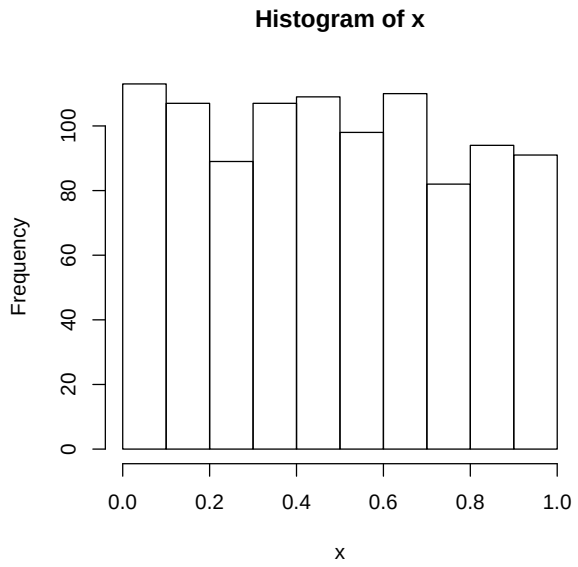
`R` is said to be a functional language, meaning that it is built around calling functions to accomplish tasks:

```
> x <- 1:9  ## Creates a vector of numbers 1, 2, ..., 9
> mean(x)

[1] 5

> median(x)

[1] 5

> sd(x)

[1] 2.738613

> min(x)

[1] 1

> sum(x)

[1] 45

> x^2

[1]  1  4  9 16 25 36 49 64 81

> sum(x^2)

[1] 285
```

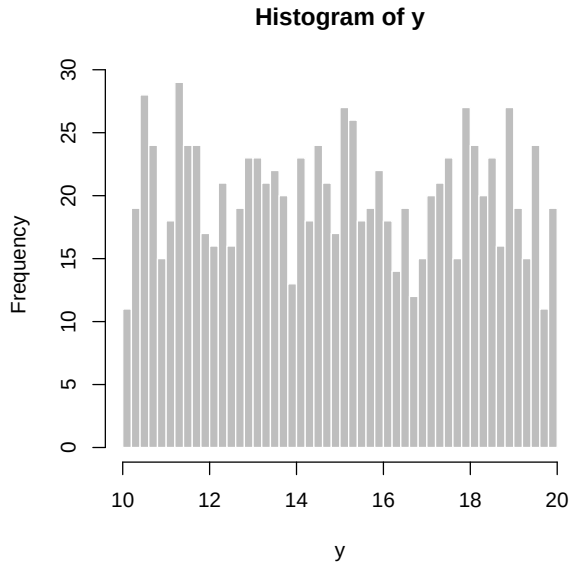
To get more information about any function in `R`, just type `help('sd')` or, more compactly, `?sd`. To search the help files for pages mentioning, say, regression, type `help.search("regression")` or `??regression`. Over time in this course, we'll see a number of functions in `R` and how they are used.

Functions typically have a number of options which may either be specified or left to their default values:

```
> x <- runif(1000)
> y <- runif(1000, min=10, max=20)
> hist(x)
```




```
> hist(y, col="gray", border="white", breaks=40)
```



1.2 Vectors and lists

We have already seen several *vectors* in R; a vector is set of elements that all share the same type. Vectors are typically either numeric, character, or logical:

```
> x <- runif(10)      ## A numeric vector
> y <- letters[1:10] ## A character vector
> z <- x > 0.5        ## A logical vector
```

```
> x

 [1] 0.71537929 0.59062808 0.05495524 0.87832084 0.60352951 0.22114537
 [7] 0.24508131 0.87140221 0.72710437 0.27629105

> y

 [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j"

> z

 [1]  TRUE  TRUE FALSE  TRUE  TRUE FALSE FALSE  TRUE  TRUE FALSE
```

As mentioned earlier, a vector cannot contain elements of different types. We can combine vectors with `c` (for concatenate), but if we try to combine, say, character and numeric vectors, problems may arise:

```
> a <- c(x, y) ## Probably not what you want
> a[1] + a[2]

Error in a[1] + a[2]:  non-numeric argument to binary operator
```

So is there a way to combine elements of different types into a single object? Yes, this is what a *list* is for. Elements of a list may be accessed by number or by name, as in the following example:

```
> a <- list(x=x, y=y, z=z)
> a$x

 [1] 0.71537929 0.59062808 0.05495524 0.87832084 0.60352951 0.22114537
 [7] 0.24508131 0.87140221 0.72710437 0.27629105

> a$x[1] + a$x[2]

[1] 1.306007

> toupper(a$y[3])

[1] "C"

> a[[3]]

 [1]  TRUE  TRUE FALSE  TRUE  TRUE FALSE FALSE  TRUE  TRUE FALSE
```

Note that for lists, you have to put double brackets around the index, as in `myList[[1]]`. Also, note the use of `$` as a separator between the name of the list and the name of the element.

1.3 Data frames

A *data frame* is a special kind of list in `R`, in which each element has the same length and thus the list can be structured as a systematic grid of rows and columns. This is the typical structure of a data set: each row represents a separate observation on a collection of variables (which make up the columns). Note that a list really is necessary here, as a data set could easily contain a mix of different types of variables (continuous, categorical, etc.). In a data frame, each column has its own type (i.e., class).

Data can of course come in a wide variety of formats, but in this class, all data sets will be provided as tab-delimited text files. Let's see an example:

```
> tips <- read.delim("http://myweb.uiowa.edu/pbreheny/data/tips.txt")
> head(tips)

  TotBill  Tip Sex Smoker Day  Time Size
1   18.29 3.76   M    Yes Sat Night    4
2   16.99 1.01   F     No Sun Night    2
3   10.34 1.66   M     No Sun Night    3
4   21.01 3.50   M     No Sun Night    3
5   23.68 3.31   M     No Sun Night    2
6   24.59 3.61   F     No Sun Night    4

> class(tips$TotBill)

[1] "numeric"

> class(tips$Sex)

[1] "factor"
```

(What's a "factor", you ask? We'll cover them in the next section.)

Here, we're reading a data set directly from a web address (obviously, you need an internet connection for this to work). Local addresses can be used as well, either relative to the current directory (`getwd`) or as an absolute path.

It is often cumbersome to type `tips$` repeatedly to access the elements of the data frame. There are two ways around this: `attach` and `with`. The former is permanent (although it can be undone with `detach`) and thus can sometimes lead to unintended side effects, while the latter only acts temporarily:

```
> with(tips, mean(TotBill))

[1] 19.78594

> mean(TotBill)

Error in mean(TotBill):  object 'TotBill' not found

> attach(tips)
> mean(TotBill)

[1] 19.78594

> detach(tips)
```

Note that we can add columns to the data frame after it has been created:

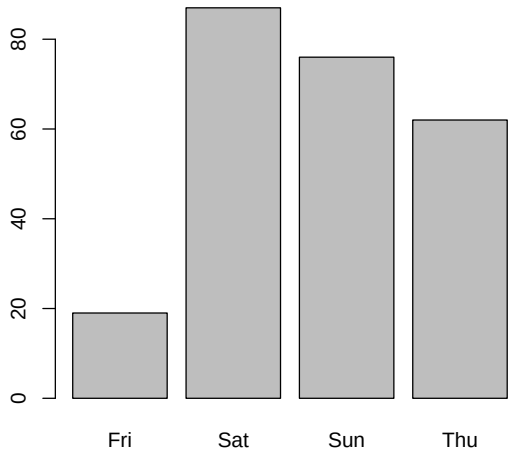
```
> tips$Rate <- with(tips, Tip/TotBill)
```

1.4 Factors

A factor is a special type of vector used to encode levels of a categorical variable (such as `tips$Sex` above).

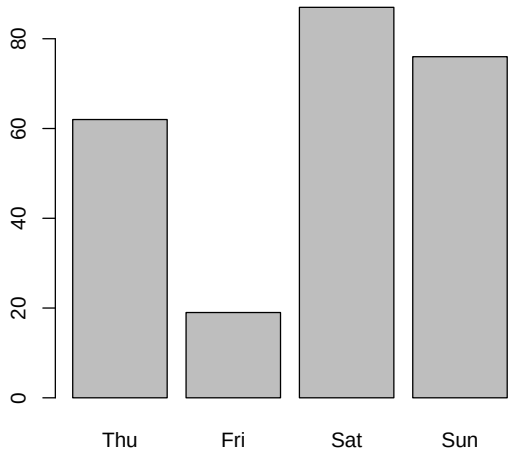
```
> table(tips$Sex)
```

```
  F   M
 87 157

> levels(tips$Sex)

[1] "F" "M"

> barplot(table(tips$Day))
```



It may seem pointless to have `factor` objects as a separate class from `character` objects, but as you will see, it is often very useful. For example, with `factor` objects, you can specify an ordering to the levels:

```
> tips$Day <- factor(tips$Day, levels=c("Thu", "Fri", "Sat", "Sun"))
> barplot(table(tips$Day))
```



2 Indexing

One of the most common tasks in data analysis is subsetting data – picking out certain rows or columns to look at more closely. Because this is a common task and often easier much easier to use one way of subsetting in circumstance and a different method in a different context, `R` provides five different ways to access elements of a vector, which is extremely convenient:

- A logical vector: Specifies, for each element, whether or not to include it
- A vector of positive integers: Lists the elements to include
- A vector of negative integers: Lists the elements to exclude
- A vector of names: Lists the elements to include by name
- Empty: Select all components

```
> Students <- c("Monica", "Michael", "Ming", "Mayra", "Sean", "Jeanette", "Lydia",
+               "Emily", "Devon", "Evan", "Jinli", "Caitlin")
> x <- runif(length(Students), 50, 100)
> names(x) <- Students
> x[x > 75]  ## Logical vector

  Monica    Mayra Jeanette    Emily    Jinli
92.09689 98.90558 94.07367 94.76855 90.86323

> x[1:3]  ## Positive integers

  Monica  Michael     Ming
92.09689 67.59577 58.98153

> x[-(5:10)]  ## Negative integers
```

```
  Monica  Michael     Ming    Mayra    Jinli  Caitlin
92.09689 67.59577 58.98153 98.90558 90.86323 63.67912

> x[c("Lydia", "Evan")]  ## Names

   Lydia     Evan
73.58227 58.49900

> x[]

  Monica  Michael     Ming    Mayra     Sean Jeanette    Lydia    Emily
92.09689 67.59577 58.98153 98.90558 73.65293 94.07367 73.58227 94.76855
   Devon     Evan    Jinli  Caitlin
60.31939 58.49900 90.86323 63.67912
```

The last option may seem pointless, but it is necessary (among other places) when accessing portions of a matrix or data frame (note that we are specifying a subset of rows, but all the columns):

```
> tips[tips$Tip >= 7, ]

    TotBill   Tip Sex Smoker Day  Time Size      Rate
25    39.42  7.58   M     No Sat Night    4 0.1922882
171   50.81 10.00   M    Yes Sat Night    3 0.1968117
213   48.33  9.00   M     No Sat Night    4 0.1862197
```

3 And much more

There is much more to `R` than this, of course: a wide variety of character and mathematical operations, probability distributions, using add-on packages, how to write your own functions, control structures (i.e., `if` statements and `for` loops), how to conduct simulations, and so on. We'll cover these in future labs.

As with any programming language, the only way to learn `R` is to use `R`, and certainly, you'll grow more familiar with `R` and learn many more functions as the semester progresses. Hopefully, this document serves as a useful introduction and reference.