R: The Absolute Basics

Brian Vegetabile

September 25, 2017

Getting Started with R

If you haven't already, install R and RStudio

- https://cran.r-project.org/
- https://www.rstudio.com/products/rstudio/

Why R?

- Why should we utilize the R programming language?
- Most common use in graduate school is for scientific computing.
- Approximately 80 percent of the code you'll write will contain some math of some kind
    - This 80 percent unfortunately will be the easier part of coding!
    - The other 20 percent will take up most of your time

Why R?

- Want a high level "prototyping" langauge to quickly implement mathematical models
- Other languages: Matlab (Expensive), Python (Free, but stats/math come second to language)
- Lower level languages: C, C++, Fortran

Why R?

- From https://www.r-project.org/about.html:
  - "R is a language and environment for statistical computing and graphics."
  - It was meant for this stuff!
  - This can also be a bad thing –> Scientists and Statisticians will prove time and time again that programming is secondary to them

Prototyping: Quick Magic Calculators

- Open an R or RStudio console and type the following code.

```
5+4
```

```
## [1] 9
```

- R and other prototyping languages allow you to get started very quickly

Prototyping: Quick Magic Calculators

- Pros:
  - There is no compiling and automatic type conversions
  - Most libraries come standard (or are easy to import)
  - The tasks you want to do most often have already been coded for you
- Cons:
  - Can be slow, compilation allows speed!

Storing Values

- Storing variables is essential to programming

```r
x <- 10
y <- "strval"
```

- In `R` the assignment opperator is `<-`
- Additionally, `=` can be used for assignment, though most style guides will tell you to use `<-`
  - There are reasons for one over the other which we won't get into here

Accessing Values

- The `print()` function will display the value of a stored variable

```
print(x)
```

```
## [1] 10
```

```
print(y)
```

```
## [1] "strval"
```

Accessing Values

- Using `;` between commands will allow evaluation on the same line

```
print(x); print(y)
```

```
## [1] 10
```

```
## [1] "strval"
```

- Similar to other languages, this tells `R` to evaluate the code and use the next command.

Writing code procedurally

- Most code/simulations you will write will be prodecural.
- Code is executed in the order in which it is received, until the end of the file
- Control functions, can divert execution or tell the computer to exit, but code generally executes "top down".

```
x <- 'Hello';
y <- 'World';
print(paste(x,y, sep='-'))
```

```
## [1] "Hello-World"
```

Data Structures in `R`

- In `R` when you assign a variable, it is really putting that value into a "container".
- These containers in `R` can either be homogeneous or heterogeneous depending on their type
- Types of containers
  - Homogeneous: Atomic Vectors, Matrices, Arrays
  - Heterogeneous: List, Data Frame

Data Structures in R - Examples

```
x <- 10
x[1]
```

```
## [1] 10
```

```
is.vector(x)
```

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

```
typeof(x)
```

```
## [1] "double"
```

Data Structures in `R`- Notes

- The fact that most variable declarations result in an atomic vector can often help explain unintended behavior
- `R` is "one" indexed and not "zero" indexed.
  - Important to consider if writing `C++` functions with `R` code
- The most common data structures we'll use are atomic vectors, matrices, data frames, and lists
  - Lists will generally be returned from a function
  - Similar functionality to dictionaries in other languages

Atomic Vectors

- Atomic vectors can contain values or strings and are one-dimensional and can only contain similar datatypes

```
x <- 'uci_Stats'
typeof(x);
```

```
## [1] "character"
```

```
is.character(x); is.numeric(x)
```

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

```
## [1] FALSE
```

Atomic Vectors

- This will work like you think it will…

```
x <- c(3, 5, 6)
print(x)
```

```
## [1] 3 5 6
```

- This won't…

```
y <- c(3, 'uci', 6)
print(y)
```

```
## [1] "3"   "uci" "6"
```

- Notice that all of the numbers have been converted to strings.

Atomic Vectors: Accessing Values

- To access values you can use subsetting by indexes

```
x <- c(1:9, 10)
print(x)
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10
```

```
x[seq(10,2,-2)]
```

```
## [1] 10  8  6  4  2
```

Matrices

A heirarchy,

- Vectors are one-dimensional and homogeneous
- Matrices are two-dimensional and homogeneous
- Arrays are n-dimensional and homogeneous
    - Check out arrays on your own

Matrices

```
x <- c(1,2,3,4)
X1 <- matrix(x, 2,2, byrow = T)
X2 <- matrix(x, 4,1, byrow = T)
print(X1); print(X2)
```

```
##      [,1] [,2]
## [1,]    1    2
## [2,]    3    4
```

```
##      [,1]
## [1,]    1
## [2,]    2
## [3,]    3
## [4,]    4
```

Matrices are homogeneous

```
x <- c(1,2,'b',4)
X1 <- matrix(x, 2,2, byrow = T)
X2 <- matrix(x, 4,1, byrow = T)
print(X1); print(X2)
```

```
##      [,1] [,2]
## [1,] "1"  "2"
## [2,] "b"  "4"
```

```
##      [,1]
## [1,] "1"
## [2,] "2"
## [3,] "b"
## [4,] "4"
```

Data Frames

- Dataframes are the indispensible data structures of `R`
- Two-dimensional "table" of data
- While matrices are homogeneous, dataframes are heterogeneous.
  - Each column is homogeneous of a dataframe
- So popular with data analysts they have been ported to other languages
  - See `pandas` in Python

Data Frames

```
x <- c(1,2,3)
y <- c('a', 'b', 'c')
z <- data.frame(x,y)
print(z)
```

```
##   x y
## 1 1 a
## 2 2 b
## 3 3 c
```

```
z$y
```

```
## [1] a b c
## Levels: a b c
```

Strings as Factors

- Often when creating a data frame, a column will be a vector of strings
- These are often brought in as "Factors"

```
is.factor(z$y)
```

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

```
levels(z$y)
```

```
## [1] "a" "b" "c"
```

```
z <- data.frame(x,y, stringsAsFactors = F)
levels(z$y)
```

```
## NULL
```

Strings as Factors

- You'll often want to turn this feature off:

```
z <- data.frame(x,y, stringsAsFactors = F)
levels(z$y)
```

```
## NULL
```

Control Functions

- `R` has a variety of control functionality
- Loops:
  - `for(i in seq){expr}` loop,
  - `while(cond){expr}` loop
- Conditions:
  - `if(cond){expr}...else if(cond){expr}...else(cond){expr}`

`If-Else Statments

- Example

```r
x <- 10
if(x <= 5){
    print('This one')
} else {
    print('No that one!')
}
```

```
## [1] "No that one!"
```

For Loops

- Example

```
x <- c(1,3,4,10)
total <- 0
for(i in 1:length(x)){
    total <- total + x[i]
}
message(total)
```

```
## 18
```

While Loops

- Example

```
x <- 1:100
ind <- total <- 0
while(ind < length(x)){
    ind <- ind + 1
    total <- total + x[ind]
}
print(total)
```

```
## [1] 5050
```

```
print(sum(1:100))
```

```
## [1] 5050
```

Functions

- The last programming knowledge we need to get up and running are functions

```
myFunc <- function(val1, val2, val3=NULL){
    print(val1)
    return(val2)
}
myFunc(3,4)
```

```
## [1] 3
```

```
## [1] 4
```

R can be slow

```r
process1 <- function(n=1000){
    x <- c(); for(i in 1:n){x <- c(x, i)};
}
process2 <- function(n=1000){
    x <- rep(NA, n); for(i in 1:n){x[i] <- i};
}
all.equal(process1(), process2())
```

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

```r
rbenchmark::benchmark(process1(), process2())[,1:4]
```

```
##          test replications elapsed relative
## 1 process1()          100   0.230      2.5
## 2 process2()          100   0.092      1.0
```