Starter Guide to **R**

Alan Jenn

February 15, 2015

1 R Layout and Workspace

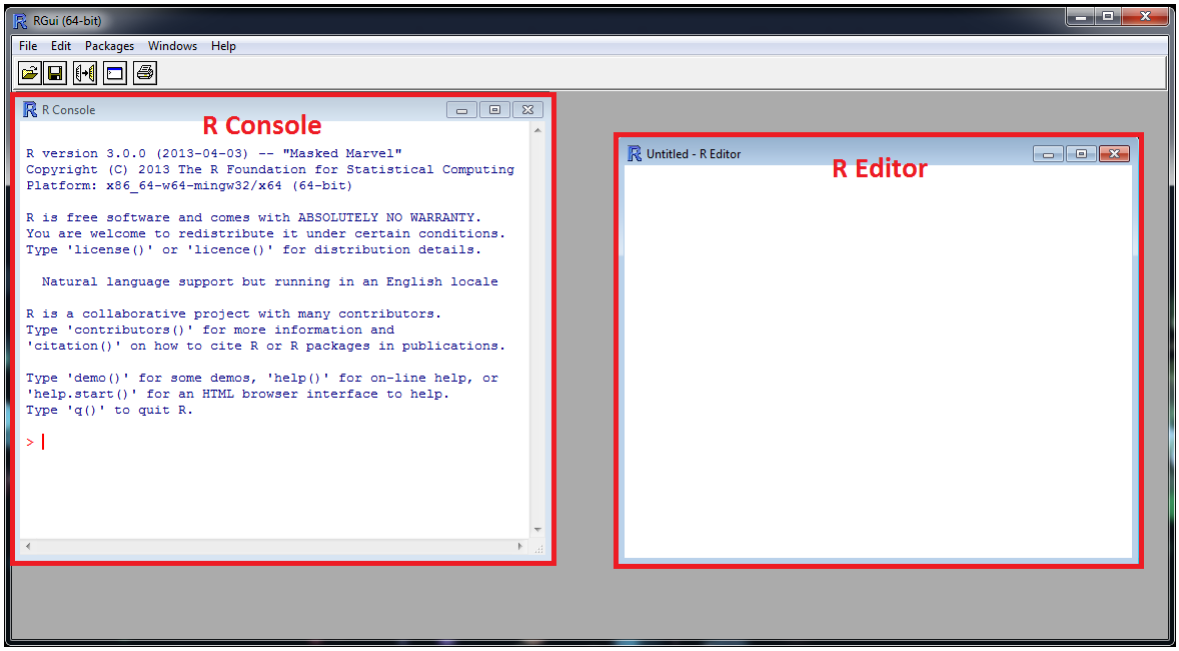

Figure 1: The two main components of the **R** are the console and the editor.

- Console - This is analogous to a calculator, you can perform line-by-line operations here. Commands executed from the editor will also be displayed as they run and it serves as a memory display for older commands.
- Editor - This is analogous to your notebook, you can write and store commands (and entire models) into the editor. The contents of the editor can be fed into the console line-by-line, by selection, or in its entirety.

2 Basic commands and objects

The following commands can be executed within the console with their corresponding outputs displayed.

```
> 1+2
[1] 3

> 2*3
```

```
[1] 6

> 3^4
[1] 81
```

Another useful command is the `concatenate` function `c()`, which generates a vector of numbers.

```
> c(1,2,3,4)
[1] 1 2 3 4
```

Suppose we want to store an object for use later, **R** can store objects (as well as overwrite objects). The objects can be stored with either the `<-` command or the `=` command.

```
> x <- 2
> y <- 3
> x
[1] 2
> y
[1] 3
> x+y
[1] 5
> x <- 3
> x+y
[1] 6
```

Let's examine some counter-intuitive vector behavior in **R**. In the first example, what happens when we add or subtract vectors from each other?

```
> x1 <- c(1,2)
> x2 <- c(1,2)
> x3 <- c(1,2,3)
> x1-x2
[1] 0 0
> x3-x2
[1] 0 0 2
Warning message:
In x3 - x2 :
  longer object length is not a multiple of shorter object length
> x2-x3
[1]  0  0 -2
Warning message:
In x2 - x3 :
  longer object length is not a multiple of shorter object length
```

What about for multiplication?

```
> x1 <- c(1,2)
> x2 <- c(1,2)
> x3 <- c(1,2,3)
> x1*x2
[1] 1 4
> x2*x3
[1] 1 4 3
Warning message:
```

```
In x2 * x3 :
  longer object length is not a multiple of shorter object length
> x1/x2
[1] 1 1
> x3*4
[1]  4  8 12
```

Actual matrix operations use a slightly different operator:

```
> x1%*%x2
     [,1]
[1,]    5
```

3 Functions in R

3.1 Built-in functions

There are a number of prebuilt functions in **R** (as well as a very large library of downloadable packages). Functions generally follow a straightforward procedure: `functionName(input1,input2,...)`. Here is a list of common and useful functions:

- `help(input)` - Where `input` is the name of a function. This will open the R documentation for any of the below functions, and any function that exists in R.
- `ls()` - This lists the objects stored on the current console.
- `rm(input)` - Where `input` is an object in the current workspace. This removes the object from the workspace.
- `matrix(input1,input2,input3)` - Where `input1` is a vector of numbers representing the entries into the matrix going left-to-right and top-to-bottom *in that order*. `input2` is the number of rows and `input3` is the number of columns.
- `mean(input)` - Where `input` is a vector of values, this returns the mean of the vector.
- `length(input)` - Where `input` is a vector of values, this returns the number of entries in the vector.
- `nrow(input)` - Where `input` is a matrix or data frame, this returns the number of rows in the object.
- `ncol(input)` - Where `input` is a matrix or data frame, this returns the number of columns in the object.
- `max(input)` - Where `input` is an object containing only numeric values, this returns the largest value of all the numeric values.
- `sample(input1,input2,input3)` - Where `input1` is the object to sample from, `input2` is the number of samples to take, and `input3` is a TRUE/FALSE value indicating whether or not you sample with replacement.
- `rep(input1,input2)` - Where `input1` is an number or vector and `input2` is the number of times to repeat that number or vector.
- `rnorm(input1,input2,input3)` - Where `input1` is the number of samples to draw from a normal distribution with mean equal to `input2` and a standard deviation equal to `input3`.
- `runif(input1,input2,input3)` - Where `input1` is the number of smaples to draw from a uniform distribution with a minimum value of `input2` and a maximum value of `input3`.

- `seq(input1,input2,by=input3)` - The function generates a sequence of numbers where `input1` is the number to start, `input2` is the number to end, and `input3` is the interval to generate numbers on.

Is there a common function that you can't find? Chances are someone made it already (aka there's a function for it), this is one of the biggest benefits of an open-source software. For Windows users just go to *Packages → Install Packages → Select a mirror → Select a package*. For Mac users go to *Packages & Data → Package Installer*. Note you have to load the package (but not re-install it) whenever you open a new console using `library(packageName)`.

3.2 Custom functions

Let's build some of our own functions.

```
imTheRealMean <- function(input) {
    x <- sum(input)
    y <- length(input)
    z <- x/y
    return(z)
}
```

Note that `x` within the function doesn't change the `x` that was previously defined. This is due to the difference between *global* and *local* variables. Variables contained in a function are considered to be local variables and essentially exist in a different "world". Functions can call global variables but you generally cannot call local variables in the global environment. It is technically possible to write to the global environment within a function but this is generally frowned upon. In more canonical programming languages, the use of global variables is avoided as much as possible but in the **R** environment this is less true. Try to use good practices, don't use the same names in your local and global environment!

The following is an example of multiplying the expectation of samples from two distributions to find whether their product is equal to the mean parameters of those distributions.

```
testTheDifference <- function() {
    hold1 <- rnorm(1000,100,10)
    #print(hold1)
    hold2 <- runif(1000,0,10)
    #print(hold2)
    output <- 100*5-mean(hold1)*mean(hold2)
    return(output)
}
replicate(1000,testTheDifference())
```

4 For Loops in R

Loops are very powerful tools in programming languages, let's take a look at a basic example of a loop in **R**:

```
for(x in 1:10) {
    print(x)
}
```

Or perhaps a more complex example, using loops to determine whether a number is prime (%% is the modulus operator):

```
isPrime <- function(x) {
    if(x < 2) {
        return(FALSE)
```

```
    }
    for(factor in 2:(x-1)) {
        if(x %% factor == 0) {
            return(FALSE)
        }
    }
    return(TRUE)
}
```

Now use a loop to test it:

```
for(n in 1:100) {
    print(n)
    print(isPrime(n))
}
```

5 Managing Working Directories

R operates in a specific working directory which can be viewed in the console via the function `getwd()`. However, if your code is for a project or larger model where you often have inputs (calling in data) and outputs (generating data or figures), it may be a good idea to change your working directory using the function `setwd()`.

```
#returns path
> getwd()

> setwd('/genericPath')
> getwd()
[1] "/genericPath"
```

Additionally, to quickly view what files there are in the working directory, use the function: `list.files()`.

6 Loading data

The function `read.table()` is the general purpose data importing tool in **R**. It allows for the import of a wide variety of files including text files and CSV files.

```
#example read table format
read.table(file='import.csv',header=TRUE,sep=',')
help(read.table) #for further information
```

Exercise #1

*Create a working directory with two subfolders: inputs and outputs. Save the **R** work file in the working directory and download the data CSV file into the inputs folder. Set the **R** console working directory to the newly created directory. Load the data CSV file into the console saved as the object "data". The data should look something like:*

```
> data <- read.table(file='anonymizedData.csv',header=TRUE,sep=',')

> summary(data)
   firstName     lastName       age         heightInches   gender      eyeColor
 a      : 1   aa     : 1   Min.   :23.00   Min.   :63.00   F: 7   black    : 3
 b      : 1   bb     : 1   1st Qu.:25.25   1st Qu.:66.00   M:15   blue     : 2
```

```
c      : 1   cc     : 1   Median :26.00   Median :69.50          blue grey: 1
e      : 1   ee     : 1   Mean   :27.45   Mean   :69.14          brown    :11
f      : 1   ff     : 1   3rd Qu.:28.75   3rd Qu.:72.00          grey     : 1
g      : 1   gg     : 1   Max.   :39.00   Max.   :75.00          hazel    : 4
(Other):16   (Other):16
numberOfClasses favoriteNumber    birthMonth          gpa
Min.   :0.000   Min.   : 1.00   Min.   : 1.000   Min.   :1.269
1st Qu.:3.000   1st Qu.: 6.00   1st Qu.: 5.000   1st Qu.:2.541
Median :3.000   Median : 7.00   Median : 8.500   Median :3.191
Mean   :3.045   Mean   :14.95   Mean   : 7.409   Mean   :3.108
3rd Qu.:4.000   3rd Qu.:14.75   3rd Qu.:10.750   3rd Qu.:3.756
Max.   :5.000   Max.   :99.00   Max.   :12.000   Max.   :4.000
```

7 Data Frames

Data frames are the backbone of **R**'s data management. They are an easy way of storing, calling, and visualizing data. Let's examine how to create a data frame:

```
> studentData <- data.frame('names'=c('student1','student2','student3'),
    'ages'=c(24,25,27),'gender'=c('M','M','F'))
> studentData
     names ages gender
1 student1   24      M
2 student2   25      M
3 student3   27      F
```

We can run some quick summary statistics on the data:

```
> summary(studentData)
      names        ages       gender
 student1:1   Min.   :24.00   F:1
 student2:1   1st Qu.:24.50   M:2
 student3:1   Median :25.00
              Mean   :25.33
              3rd Qu.:26.00
              Max.   :27.00
```

7.1 Searching in a data frame

There are several different methods of searching for specific entries in a data frame: by column, by row, by a specific value, or using a query:

```
#Search by row
> data[3,]      #this returns a data frame w/ one row

#Search by multiple rows
> data[c(3,4),]     #this returns a data frame w/ two rows

#Search by column (all produce the same results)
> data[,2]      #this returns a vector of data
> data$lastName
> data[,'lastName']

#Search by multiple columns (produces same results)
```

```
> data[,c(1,2)]     #this returns a data frame w/ two columns
> data[,c('firstName','lastName')]

#Search by row and column
> data[1,2]

#Search by values using a logical vector
> data$lastName
> data$lastName=='aa'
> data[data$lastName=='aa',]      #this returns the full row
> data$firstName[data$lastName=='aa']     #this returns only the first name

#Searching with logicals
> data[data$age>25&data$age<=32,]        #returns everyone above 25 and 32 or below
```

7.2 Other useful operations

In addition to the above, the search functions can be combined to select data for specific operations such as dropping nonsensical data or replacing data.

```
> df <- data.frame('col1'=1:5,'col2'=rnorm(5))
> na.omit(df)       #this function drops rows containing NA values
> df[-1,]           #this function omits the first row of the df
> df[,-1]           #this function omits the first column of the df
> df <- df[-1,]     #this function drops the first row of the df
> df[,1] <- df[,1]*2    #this function multiplies all values of the first column by 2
```

Exercise #2

Clean the data: remove unreasonable ages, drop rows with NA values, and correct any alphabet case issues (change to either all upper case or all lower case).

```
> data <- na.omit(data)
> data$gender <- as.factor(toupper(data$gender))
> data$eyeColor <- as.factor(tolower(data$eyeColor))
> data <- data[data$age<100,]
> summary(data)
   firstName     lastName       age         heightInches   gender      eyeColor
 a      : 1   aa     : 1   Min.   :23.00   Min.   :63.00   F: 7   black    : 3
 b      : 1   bb     : 1   1st Qu.:25.25   1st Qu.:66.00   M:15   blue     : 2
 c      : 1   cc     : 1   Median :26.00   Median :69.50          blue grey: 1
 e      : 1   ee     : 1   Mean   :27.45   Mean   :69.14          brown    :11
 f      : 1   ff     : 1   3rd Qu.:28.75   3rd Qu.:72.00          grey     : 1
 g      : 1   gg     : 1   Max.   :39.00   Max.   :75.00          hazel    : 4
 (Other):16   (Other):16
 numberOfClasses favoriteNumber    birthMonth          gpa
 Min.   :0.000   Min.   : 1.00   Min.   : 1.000   Min.   :1.269
 1st Qu.:3.000   1st Qu.: 6.00   1st Qu.: 5.000   1st Qu.:2.541
 Median :3.000   Median : 7.00   Median : 8.500   Median :3.191
 Mean   :3.045   Mean   :14.95   Mean   : 7.409   Mean   :3.108
 3rd Qu.:4.000   3rd Qu.:14.75   3rd Qu.:10.750   3rd Qu.:3.756
 Max.   :5.000   Max.   :99.00   Max.   :12.000   Max.   :4.000
```

8 Other data structures

There are other data structures in **R** that allow for storage of data in more dimensions and can have different functional properties from data frames. An array can be used as a matrix greater than 2 dimensions and a list is one of the more flexible storage devices.

```
> matrix1 <- matrix(1,nrow=3,ncol=3)
> matrix2 <- matrix(2,nrow=3,ncol=3)
> matrix3 <- matrix(3,nrow=3,ncol=3)
>
> arrayExample <- array(data=c(matrix1,matrix2,matrix3),dim=c(3,3,3))
> listExample <- list(matrix1,matrix2,matrix3,data)
```

9 Looping and the Apply Family

Exercise #3

Using either for loops or an apply function, create an extra column in the data frame named "gpa" and fill in the values using the provided `gpaFunction`.

```
#This function takes a single age input and outputs a single GPA value.
gpaFunction <- function(age) {
    randomGPA <- rnorm(1,mean=3,sd=.5)
    ageWeight <-rnorm(1,mean=((age-27.5)/100),sd=.5)
    weightedGPA <- randomGPA+ageWeight
    if(weightedGPA > 4) {finalGPA <- 4}
    else if(weightedGPA < 0) {finalGPA <- 0}
    else {finalGPA <- weightedGPA}
    return(finalGPA)
}

data$gpa <- sapply(data$age,gpaFunction)
```

```
> ptm <- proc.time()
> for(row in 1:nrow(data)) {
+   gpaFunction(data$age[row])
+ }
> proc.time()-ptm
   user  system elapsed
  0.012   0.003   0.038

> ptm <- proc.time()
> sapply(data$age,gpaFunction)
> proc.time()-ptm
   user  system elapsed
  0.010   0.002   0.036
```

For most purposes, for loops and apply functions can be used as substitutes. However, for larger scale functions and problems where computation time becomes an issue, the apply function can often be superior to inefficient for loops.

10 Plotting in R

The plotting functionality in **R** allows for a great deal of flexibility but there are a large number of commands to learn in order to take advantage of the numerous available tools. For the purposes of this tutorial, we will

examine some of the basic functions:

10.1 The Plot

The basic plot function:

```
> plot(x=data[,'age'],y=data[,'gpa'])
```

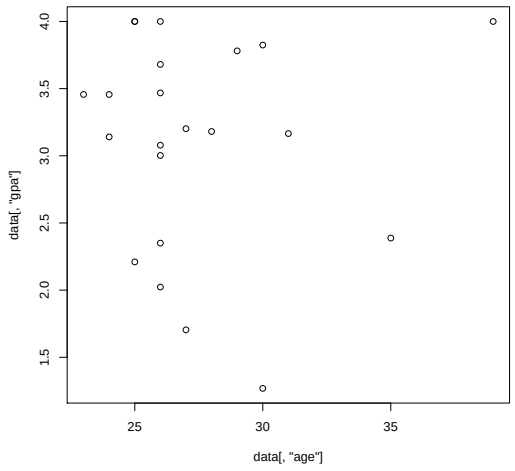

Figure 2: Unformatted plot of age versus GPA

There are a number of formatting commands:

- type: Type of plot; l-lines, p-points, b-both, etc.
- main: Title of the plot
- xlab: x-axis label
- ylab: y-axis label
- xlim: x-axis range
- ylim: y-axis range
- cex: Size of labels and points/lines (cex.axis, cex.lab)

Now we can make our plot much prettier:

```
> plot(x=data[,'age'],y=data[,'gpa'],
    xlab='Age',ylab='GPA',ylim=c(0,4),cex.lab=1.5,cex.axis=1.5)
```

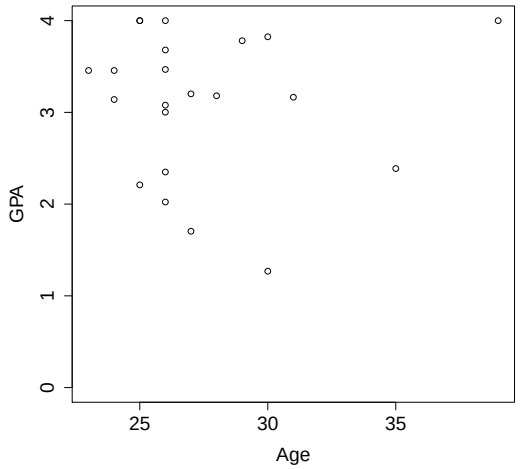

Figure 3: Unformatted plot of age versus GPA

Plots can be saved using a wrapper function around the plot, starting with `pdf(filename,opts)` (for pdf format). Other file formats are similar: `png(filename,opts)`, `tiff(filename,opts)`, and `jpeg(filename,opts)`. Each file format has their own pros and cons, I personally prefer pdfs due to their vectorization properties (can be resized without losing quality).

```
> pdf('output/age_vs_gpa.pdf')
> plot(x=data[,'age'],y=data[,'gpa'],
    xlab='Age',ylab='GPA',ylim=c(0,4),cex.lab=1.5,cex.axis=1.5)
> dev.off()
```

10.2 Histograms

```
> hist(data[,'birthMonth'],main=NULL)
```

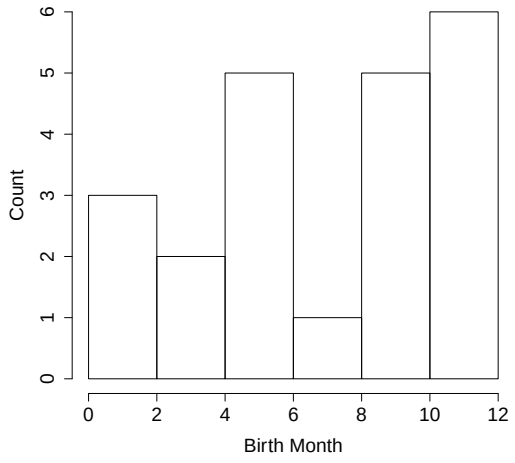

Figure 4: Histogram of birth months

10.3 Bar Plots

```
> eyeColorCounts <- table(data[,'eyeColor'])
> eyeColorCounts

    black      blue blue grey     brown      grey     hazel
        3         2         1        11         1         4

> barplot(eyeColorCounts,names.arg=names(eyeColorCounts))
```

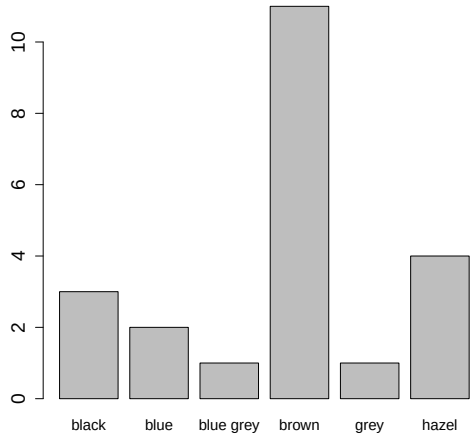

Figure 5: Barplot of eye colors

11 Statistical Analysis in R

Perform a basic statistical analysis of our data in **R**, you can run a simple linear regression and basic diagnostics as follows:

```
> #running a regression model
> model <- lm(gpa~age,data=data)
> summary(model)

Call:
lm(formula = gpa ~ age, data = data)

Residuals:
    Min      1Q  Median      3Q     Max
-1.1150 -0.3643 -0.1484  0.4091  1.1139

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept) 2.832744   0.982327   2.884  0.00918 **
age         0.002052   0.035466   0.058  0.95443
---
Signif. codes:  0 *** 0.001 ** 0.01 * 0.05 . 0.1   1
```

```
Residual standard error: 0.6096 on 20 degrees of freedom
Multiple R-squared:  0.0001674, Adjusted R-squared:  -0.04982
F-statistic: 0.003349 on 1 and 20 DF,  p-value: 0.9544

> #plotting age vs gpa with fitted line
> plot(x=data[,'age'],y=data[,'gpa'],
    xlab='Age',ylab='GPA',ylim=c(0,4),cex.lab=1.5,cex.axis=1.5)
> lines(x=data[,'age'],y=predict(model),col='red')
```

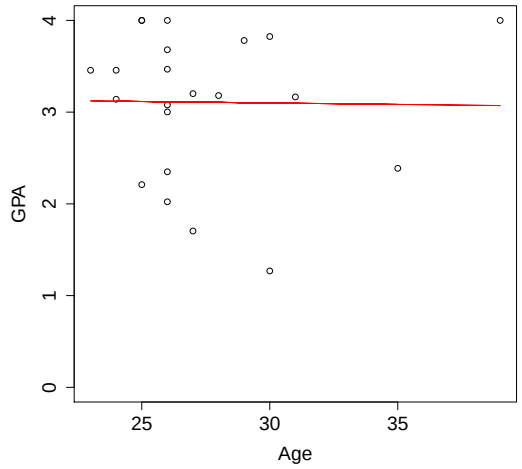

Figure 6: Age versus GPA with fitted line from regression model

```
> #plotting age vs residuals
> plot(x=data[,'age'],y=model$residuals,
    xlab='Age',ylab='Residuals',cex.lab=1.5,cex.axis=1.5)
```

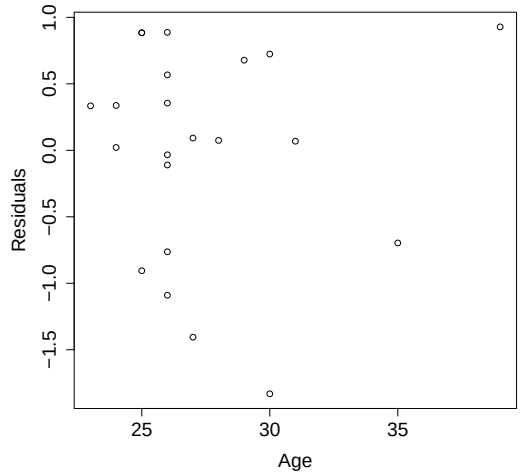

Figure 7: Age versus Residuals from regression model