Introduction

Andrew Roth

2024-04-24

The R ecosystem

R

R is a programming language originally developed for doing statistical analysis. R is different than many other statistics software packages such as JMP, SPSS, Excel etc. as it requires writing code or programming instead of clicking through a graphical interface. The downside, is that there is a learning curve when starting to use R. The upside, is that R can be significantly more powerful than other tools. For example, if you write code to do an analysis on one data set and produce plots, it is trivial to reuse your code and do the same analysis on a new dataset.

R provides an environment to execute code and some built in functionality for analyzing data. Later we will see how to extend the basic built in capabilities of R using *libraries*.

By default there are two ways to work with R and write code.

1. The first way is to work interactively with the R interpreter. When doing this you will type commands one after the other into a terminal and run them sequentially. This can be great for testing out new functions or quickly doing something with the data. You can save pieces of information from you session using the `save` function. You can also save the entire *workspace* using the `save.image` function. > When working interactively you can use the tab button to autocomplete commands and the up arrow to see the previous command.

2. The second way to work with R is to write *scripts* which are text files with contain code. In practice this is the preferred way to work with R as you can easily rerun any analysis you have performed.

Over time a lot of additional tools have been developed to make using R easier.

R Studio

For example, RStudio is an integrated development environment that makes writing R code and working with data easier. RStudio has a lot of functionality and we won't cover it all. Starting out one of the biggest strengths of RStudio is the ability to run pieces of your script interactively. This allows you to blend the two standard ways of working with R so you can write code in your script run it in the terminal and then write some more code in your script. This approach makes it easy to incrementally write a full analysis.

R Markdown

Another example is R Markdown which is a tool which allows you to combine text and code to form "runnable" documents. R Markdown can be a great way to write reports, as the code for doing all the analysis is included in the document.

This tutorial is written in R Markdown.

Getting started with R

Let's start with a simple example of R code that adds 2 + 2 and stores it in a *variable*.

```r
x <- 2 + 2
print(x)
```

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

We have done three things here.

1. We added 2 and 2 together.
2. We have saved the result into a variable called `x`
3. We have printed out the result.

The idea of storing something in a variable is one of fundamental concepts in programming. In the previous example the value stored in our variable was a simple number. In practice you will be storing more complex things in variables. In programming terms these more complex things are often referred to as *objects*. The most import type of *class* of object you will work with in R is the *data frame*.

A data frame is basically a table that you can interact with in R. A data frame has rows and columns. Typically each row in a data frame corresponds to an observation e.g. a mouse you have observed. Each column in a data frame then corresponds to some feature of the observation you have recored e.g. sex, weight, treatment. All values for a given column must be of the same type i.e. weight is a number. However, each column can store a different type of data.

Below we show an example of a data set stored in a data frame. This data set describes features of different car models is builtin to R so we do not need to load it.

```r
print(mtcars)
```

```
##                      mpg cyl  disp  hp drat    wt  qsec vs am gear carb
## Mazda RX4           21.0   6 160.0 110 3.90 2.620 16.46  0  1    4    4
## Mazda RX4 Wag       21.0   6 160.0 110 3.90 2.875 17.02  0  1    4    4
## Datsun 710          22.8   4 108.0  93 3.85 2.320 18.61  1  1    4    1
## Hornet 4 Drive      21.4   6 258.0 110 3.08 3.215 19.44  1  0    3    1
## Hornet Sportabout   18.7   8 360.0 175 3.15 3.440 17.02  0  0    3    2
## Valiant             18.1   6 225.0 105 2.76 3.460 20.22  1  0    3    1
## Duster 360          14.3   8 360.0 245 3.21 3.570 15.84  0  0    3    4
## Merc 240D           24.4   4 146.7  62 3.69 3.190 20.00  1  0    4    2
## Merc 230            22.8   4 140.8  95 3.92 3.150 22.90  1  0    4    2
## Merc 280            19.2   6 167.6 123 3.92 3.440 18.30  1  0    4    4
## Merc 280C           17.8   6 167.6 123 3.92 3.440 18.90  1  0    4    4
## Merc 450SE          16.4   8 275.8 180 3.07 4.070 17.40  0  0    3    3
## Merc 450SL          17.3   8 275.8 180 3.07 3.730 17.60  0  0    3    3
## Merc 450SLC         15.2   8 275.8 180 3.07 3.780 18.00  0  0    3    3
## Cadillac Fleetwood  10.4   8 472.0 205 2.93 5.250 17.98  0  0    3    4
## Lincoln Continental 10.4   8 460.0 215 3.00 5.424 17.82  0  0    3    4
## Chrysler Imperial   14.7   8 440.0 230 3.23 5.345 17.42  0  0    3    4
## Fiat 128            32.4   4  78.7  66 4.08 2.200 19.47  1  1    4    1
## Honda Civic         30.4   4  75.7  52 4.93 1.615 18.52  1  1    4    2
## Toyota Corolla      33.9   4  71.1  65 4.22 1.835 19.90  1  1    4    1
## Toyota Corona       21.5   4 120.1  97 3.70 2.465 20.01  1  0    3    1
## Dodge Challenger    15.5   8 318.0 150 2.76 3.520 16.87  0  0    3    2
## AMC Javelin         15.2   8 304.0 150 3.15 3.435 17.30  0  0    3    2
## Camaro Z28          13.3   8 350.0 245 3.73 3.840 15.41  0  0    3    4
## Pontiac Firebird    19.2   8 400.0 175 3.08 3.845 17.05  0  0    3    2
## Fiat X1-9           27.3   4  79.0  66 4.08 1.935 18.90  1  1    4    1
## Porsche 914-2       26.0   4 120.3  91 4.43 2.140 16.70  0  1    5    2
## Lotus Europa        30.4   4  95.1 113 3.77 1.513 16.90  1  1    5    2
## Ford Pantera L      15.8   8 351.0 264 4.22 3.170 14.50  0  1    5    4
```

```
## Ferrari Dino        19.7   6 145.0 175 3.62 2.770 15.50  0  1    5    6
## Maserati Bora       15.0   8 301.0 335 3.54 3.570 14.60  0  1    5    8
## Volvo 142E          21.4   4 121.0 109 4.11 2.780 18.60  1  1    4    2
```

The above code prints out the entire dataset. If your data set is big, you usually do not want to do this. However, it is often useful to see the first few rows of the data to see what columns you have and what values they take, or to make sure your data got loaded correctly into R. The `head` function allows you to see the first few rows of your data frame.

```r
head(mtcars)
```

```
##                    mpg cyl disp  hp drat    wt  qsec vs am gear carb
## Mazda RX4         21.0   6  160 110 3.90 2.620 16.46  0  1    4    4
## Mazda RX4 Wag     21.0   6  160 110 3.90 2.875 17.02  0  1    4    4
## Datsun 710        22.8   4  108  93 3.85 2.320 18.61  1  1    4    1
## Hornet 4 Drive    21.4   6  258 110 3.08 3.215 19.44  1  0    3    1
## Hornet Sportabout 18.7   8  360 175 3.15 3.440 17.02  0  0    3    2
## Valiant           18.1   6  225 105 2.76 3.460 20.22  1  0    3    1
```

There is also a `tail` function that lets you look at the last few rows.

```r
tail(mtcars)
```

```
##                 mpg cyl  disp  hp drat    wt qsec vs am gear carb
## Porsche 914-2  26.0   4 120.3  91 4.43 2.140 16.7  0  1    5    2
## Lotus Europa   30.4   4  95.1 113 3.77 1.513 16.9  1  1    5    2
## Ford Pantera L 15.8   8 351.0 264 4.22 3.170 14.5  0  1    5    4
## Ferrari Dino   19.7   6 145.0 175 3.62 2.770 15.5  0  1    5    6
## Maserati Bora  15.0   8 301.0 335 3.54 3.570 14.6  0  1    5    8
## Volvo 142E     21.4   4 121.0 109 4.11 2.780 18.6  1  1    4    2
```

Basic statistics can be obtained using the `summary` function.

```r
summary(mtcars)
```

```
##       mpg             cyl             disp             hp       
##  Min.   :10.40   Min.   :4.000   Min.   : 71.1   Min.   : 52.0  
##  1st Qu.:15.43   1st Qu.:4.000   1st Qu.:120.8   1st Qu.: 96.5  
##  Median :19.20   Median :6.000   Median :196.3   Median :123.0  
##  Mean   :20.09   Mean   :6.188   Mean   :230.7   Mean   :146.7  
##  3rd Qu.:22.80   3rd Qu.:8.000   3rd Qu.:326.0   3rd Qu.:180.0  
##  Max.   :33.90   Max.   :8.000   Max.   :472.0   Max.   :335.0  
##       drat             wt             qsec             vs        
##  Min.   :2.760   Min.   :1.513   Min.   :14.50   Min.   :0.0000  
##  1st Qu.:3.080   1st Qu.:2.581   1st Qu.:16.89   1st Qu.:0.0000  
##  Median :3.695   Median :3.325   Median :17.71   Median :0.0000  
##  Mean   :3.597   Mean   :3.217   Mean   :17.85   Mean   :0.4375  
##  3rd Qu.:3.920   3rd Qu.:3.610   3rd Qu.:18.90   3rd Qu.:1.0000  
##  Max.   :4.930   Max.   :5.424   Max.   :22.90   Max.   :1.0000  
##        am              gear            carb      
##  Min.   :0.0000   Min.   :3.000   Min.   :1.000  
##  1st Qu.:0.0000   1st Qu.:3.000   1st Qu.:2.000  
##  Median :0.0000   Median :4.000   Median :2.000  
##  Mean   :0.4062   Mean   :3.688   Mean   :2.812  
##  3rd Qu.:1.0000   3rd Qu.:4.000   3rd Qu.:4.000  
##  Max.   :1.0000   Max.   :5.000   Max.   :8.000  
```

You can also compute the statistics individually for specific columns. For example the following code gets the mean of the mpg column.

```
mean(mtcars$mpg)
```

```
## [1] 20.09062
```

> In the previous example we introduce a new notation the `$` *operator* which allows us to access a column of a data frame.

There are a number of other built in functions such as `range` and `quantile`. Let's take a look at the `quantile` function.

```
quantile(mtcars$mpg)
```

```
##     0%    25%    50%    75%   100%
## 10.400 15.425 19.200 22.800 33.900
```

By default quantile will report the 0%, 25%, 50%, 75% and 100% quantiles (percentiles?). What if you want different quantiles? We can look into the documentation for the `quantile` function. R has quite a good documentation built in. You can access it with the `?` operator.

> You can also just search the web :)

For example to get help for the `quantile` operator we would do the following.

```
?quantile
```

The documentation reveals that `quantile` takes one mandatory *argument* and several optional arguments. If we want different quantiles we would need to change the optional probs argument. Let's get the 33% and 66% percentiles.

```
quantile(mtcars$mpg, probs=c(0.33, 0.66))
```

```
##    33%    66%
## 16.607 21.400
```

We have introduced another new function, `c`, which is the combine function. This function will take a collection of values and combine them into a *vector*. Vectors are another class of objects in R, which store a one dimensional collection of items. Columns in a data frame are actually vectors themselves (**Andy is 90% sure of this**).

Let us try using `c` to make a vector.

```
y <- c(1, 10, 42)
print(y)
```

```
## [1]  1 10 42
```

In the previous example all the items we combined were of the same type, numbers. We can see this using the `class` function.

```
class(y)
```

```
## [1] "numeric"
```

What happens if we combine different types into a vector? In the next example we will combine some numbers and some "strings".

```
z <- c(1, "a", 42, "Andy")
class(z)
```

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

You will see that the vector we created `z` has type character. When programming the concept of type is quite important. Variables have types, the type of a variable can impact how or even if a function works with a variable.

```
mean(z)
```

```
## Warning in mean.default(z): argument is not numeric or logical: returning NA
```

```
## [1] NA
```

Programming languages generally have a hierarchy of types which goes from most specific to most general. In the previous example because we combined numbers and strings R a decided to make the vector of the type character (string). This can cause unexpected behaviour sometimes. Let's look at the first value of our vector `z`.

```
z[1]
```

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

We can see that the number 1 now has quotation marks around it indicating it is a string.

In the previous example we have done something new, specifically accessing a specific element of a vector by its numbered position. This is done using the `[]` syntax.

> R uses one based indexing i.e. the first element of a vector is 1. This differs from some other programming languages which use 0.

We can use the a similar same syntax access elements in a data frame. For example if we want the first row we could do the following.

```
mtcars[1,]
```

```
##           mpg cyl disp  hp drat   wt  qsec vs am gear carb
## Mazda RX4  21   6  160 110  3.9 2.62 16.46  0  1    4    4
```

Unlike a vector, a data frame is two dimensional so we use a `,` with the `[]` syntax to access different dimensions. If we wanted to get the third column from the second row we would do the following.

```
mtcars[2, 3]
```

```
## [1] 160
```

We had previously accessed the column by name using `mtcars$mpg`. Both numerical and named indexing can be useful, but generally you will use named indexing which is less error prone. We could achieve the same things as the previous code using named indexing as follows.

```
mtcars$disp[2]
```

```
## [1] 160
```

The `colnames` function lets you see the names of the columns.

```
colnames(mtcars)
```

```
##  [1] "mpg"  "cyl"  "disp" "hp"   "drat" "wt"   "qsec" "vs"   "am"   "gear"
## [11] "carb"
```

Rows can also have names in a data frame, and we can see them using the `rownames` function.

```
rownames(mtcars)
```

```
##  [1] "Mazda RX4"           "Mazda RX4 Wag"       "Datsun 710"
##  [4] "Hornet 4 Drive"      "Hornet Sportabout"   "Valiant"
##  [7] "Duster 360"          "Merc 240D"           "Merc 230"
## [10] "Merc 280"            "Merc 280C"           "Merc 450SE"
```

```
## [13] "Merc 450SL"          "Merc 450SLC"         "Cadillac Fleetwood"
## [16] "Lincoln Continental" "Chrysler Imperial"   "Fiat 128"
## [19] "Honda Civic"         "Toyota Corolla"      "Toyota Corona"
## [22] "Dodge Challenger"    "AMC Javelin"         "Camaro Z28"
## [25] "Pontiac Firebird"    "Fiat X1-9"           "Porsche 914-2"
## [28] "Lotus Europa"        "Ford Pantera L"      "Ferrari Dino"
## [31] "Maserati Bora"       "Volvo 142E"
```

We can also access rows by name.

```
mtcars["Mazda RX4",]
```

```
##           mpg cyl disp  hp drat   wt  qsec vs am gear carb
## Mazda RX4  21   6  160 110  3.9 2.62 16.46  0  1    4    4
```

And we can get specific elements by row and column name.

```
mtcars["Porsche 914-2", "gear"]
```

```
## [1] 5
```

Finally we can get multiple items using the `[]` syntax and the `c` function.

```
mtcars["Porsche 914-2", c("gear", "mpg")]
```

```
##               gear mpg
## Porsche 914-2    5  26
```

Packages and libraries

Often you will want to get some additional functionality that is not built into R. This where packages can be useful. A package is additional code someone has written that you can install and use. Most often you will install packages from CRAN.

We will install the `EnvStats` package which provides the `summaryFull` function to compute a richer set of descriptive statistics. You can install packages from CRAN using the `install.packages` function. You can also install packages using RStudio which might be slightly easier when you are starting out.

Once you have the package installed you will need to load using the it `library` function before you can use the functions it provides.

```
library(EnvStats)
```

```
##
## Attaching package: 'EnvStats'

## The following objects are masked from 'package:stats':
##
##     predict, predict.lm
```

Now we can use the `summaryFull` command.

```
summaryFull(mtcars)
```

```
##                              am       carb    cyl      disp     drat     gear
## N                             32       32      32       32       32       32
## Mean                           0.4062   2.812   6.188  230.7      3.597    3.688
## Median                         0        2       6      196.3      3.695    4
## 10% Trimmed Mean               0.3846   2.654   6.231  222.5      3.579    3.615
## Skew                           0.4008   1.157  -0.1923   0.4202   0.2928   0.5823
## Kurtosis                      -1.967    2.02   -1.763   -1.068   -0.4504  -0.8953
```

```
## Min                         0         1        4        71.1       2.76      3
## Max                         1         8        8       472         4.93      5
## Range                       1         7        4       400.9       2.17      2
## 1st Quartile                0         2        4       120.8       3.08      3
## 3rd Quartile                1         4        8       326         3.92      4
## Standard Deviation          0.499     1.615    1.786   123.9       0.5347    0.7378
## Interquartile Range         1         2        4       205.2       0.84      1
## Median Absolute Deviation   0         1.483    2.965   140.5       0.7042    1.483
##                           hp        mpg        qsec      vs        wt
## N                          32        32         32        32        32
## Mean                      146.7      20.09      17.85     0.4375    3.217
## Median                    123        19.2       17.71     0         3.325
## 10% Trimmed Mean          141.2      19.7       17.83     0.4231    3.153
## Skew                        0.7994    0.6724     0.4063   0.2645    0.4659
## Kurtosis                    0.2752   -0.02201    0.8649  -2.063     0.4166
## Min                        52        10.4       14.5      0         1.513
## Max                       335        33.9       22.9      1         5.424
## Range                     283        23.5        8.4      1         3.911
## 1st Quartile               96.5      15.42      16.89     0         2.581
## 3rd Quartile              180        22.8       18.9      1         3.61
## Standard Deviation         68.56      6.027      1.787    0.504     0.9785
## Interquartile Range        83.5       7.38       2.01     1         1.029
## Median Absolute Deviation  77.1       5.411      1.416    0         0.7672
```

When we move onto plotting we will need some libraries, in particular `ggplot`. There is a collection of tools caled the tidyverse which includes `ggplot` and other useful packages like `dplyr`.

Loading data

So far we have been working with a toy data set built into R. But in practice you will want to use your own data sets. The first step we need to tackle is loading data into R. There are several functions to help with this depending on the file format you store your data in. If you are working in the lab it is likely you are using a spreadsheet program like Excel. Excel files can be loaded into R and we will see how in a second. But it is useful to note that a lot of data is often stored in another format called a `csv` file, which stands for comma separated values file. The key difference between a csv file and the standard format Excel uses is that the csv only contains your data. An Excel file will store your data, but also lots of other information like any formulas or colouring you have done. You can easily export to csv files from Excel and similar programs, **but only your data will be exported**. So if you are doing more than keeping data, than Excel's default format is fine.

The point of the long discussion above is that R has built in support for reading csv files, because they are simple. For using other formats you might need to install additional packages.

Let's start by loading a file saved in csv format. We will use the diabetes dataset from out last lecture which we converted to csv.

```
diabetes <- read.csv("/home/andrew/Desktop/path/Diabetes_Full.csv")
head(diabetes)
```

```
##   Random.Blood.Glucose.mg.dL Random.Blood.Glucose.Binary
## 1                        151                         Low
## 2                         75                         Low
## 3                        141                         Low
## 4                        206                        High
## 5                        135                         Low
## 6                         97                         Low
##   Random.Blood.Glucose.Ordinal Age Sex  BMI  BP Total.Cholesterol   LDL HDL TCH
```

```
## 1                       Medium  59   2 32.1  101              157  93.2  38   4
## 2                          Low  48   1 21.6   87              183 103.2  70   3
## 3                          Low  72   2 30.5   93              156  93.6  41   4
## 4                         High  24   1 25.3   84              198 131.4  40   5
## 5                          Low  50   1 23.0  101              192 125.4  52   4
## 6                          Low  23   1 22.6   89              139  64.8  61   2
##      LTG Fasting.Glucose
## 1 4.8598              87
## 2 3.8918              69
## 3 4.6728              85
## 4 4.8903              89
## 5 4.2905              80
## 6 4.1897              68
```

The previous code reads the data in our file using the `read.csv` function and stores the resulting data frame into a variable called `diabetes`. Next we use `head` to look at the first few rows and make sure everything loaded correctly.

Now it is good practice to determine what type of values R thinks are stored in each column. We can use `typeoff` to see the exact type R believes each column is. Let's try this on one column first.

```r
typeof(diabetes$BMI)
```

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

Now we could go through head column and check their type by running the above code with each column name. But this would really tedious. This is where using a programing language can be helpful as we can automate that tasks. We are going to use the `sapply` function which go through each column and `apply` a specified function to it.

```r
sapply(diabetes, typeof)
```

```
##   Random.Blood.Glucose.mg.dL  Random.Blood.Glucose.Binary
##                    "integer"                  "character"
## Random.Blood.Glucose.Ordinal                          Age
##                  "character"                    "integer"
##                          Sex                          BMI
##                    "integer"                     "double"
##                           BP            Total.Cholesterol
##                     "double"                    "integer"
##                          LDL                          HDL
##                     "double"                     "double"
##                          TCH                          LTG
##                     "double"                     "double"
##              Fasting.Glucose
##                    "integer"
```

Without knowing a lot about this data, it seems like most values are what you would expect.

We see the types:

- integer - Which is a number like 1, 2, 3, -1, 0 etc
- double - Which is decimal number like 3.45
- character - Which is letters are text

One potential issue is that R believes sex is an integer. Another issue is that there binary and ordinal blood glucose columns which R thinks are characters. We will see later how to let R know that we expect columns to be of a certain type. This will become important when we start plotting items. For now the key concept to understand is that there are types and they impact the way R treats your data.