1: R Crash Course

What is R?

R is alot like python. It is interpreted. R is better for working with data tables and visualization than python. R might be worse than python for some other things though. R is vectorized... we will get to that later. Try going to the console and playing around for a bit

```
1+1
```

```
## [1] 2
```

```
three_is_this_variable <- 3
a.sentence.variable <- "Hello and welcome to the workshop"
```

Note that the period is not a function call (as in python) but just separates words in the variable name. we like to use snake_case in R to name our variables.

types of variables, vectors

one of the core data structures you need to know about is the vector, in R, everything is a vector even if its just one element

```
# types of atomic vectors
int <- 1L          #integers you put an L after the number
text <- 'ace'
int <- c(1L,5L)
text <- c('ace','hearts')
logic <- c(TRUE,FALSE,FALSE)
```

atomic vectors only support one data type. If multiple data types are inserted, then R will coerce everything into one data type based on the hierarchy: boolean -> numeric -> char which preserves the most information.

name attribute

Attributes give some structure to your data

```
die <- 1:6
die
```

```
## [1] 1 2 3 4 5 6
```

```
attributes(die)
```

```
## NULL
```

```
names(die) <- c("one","two","three","four","five","six")
die
```

```
##   one   two three  four  five   six
##     1     2     3     4     5     6
```

dim attribute

you can unpack a vector into a multi-dimensional array

```
dim(die) <- c(2,3)  # reshapes into 2 row, 3 column array
die
```

```
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6
```

```
matrix(die,nrow=2,byrow=TRUE)  # unpack along rows
```

```
##      [,1] [,2] [,3]
## [1,]    1    2    3
## [2,]    4    5    6
```

vector operations

You can do element-wise operations, as well as inner and outer products with vectors as shown below:

```
y <- 1:4
y
```

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

```
y + y         # this is an elementwise operation
```

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

```
y - y         # this is also an elementwise operation
```

```
## [1] 0 0 0 0
```

```
y + 1        # recycling rule for vectors of different lengths
```

```
## [1] 2 3 4 5
```

```
y%*%y         # inner product
```

```
##      [,1]
## [1,]   30
```

```
y%o%y         # outer product (gives a matrix)
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    2    3    4
## [2,]    2    4    6    8
## [3,]    3    6    9   12
## [4,]    4    8   12   16
```

Factors

Factors are variables like gender or eye color which take a limited number of values. A factor's **levels** are always character valued. Example:

```
data = c(1,2,3,2,1,2,3,3,2,1,2,3,3,1,2)
factor(data)
```

```
##  [1] 1 2 3 2 1 2 3 3 2 1 2 3 3 1 2
## Levels: 1 2 3
```

```
factor(data, labels = c('I','II','III'))  # choose levels
```

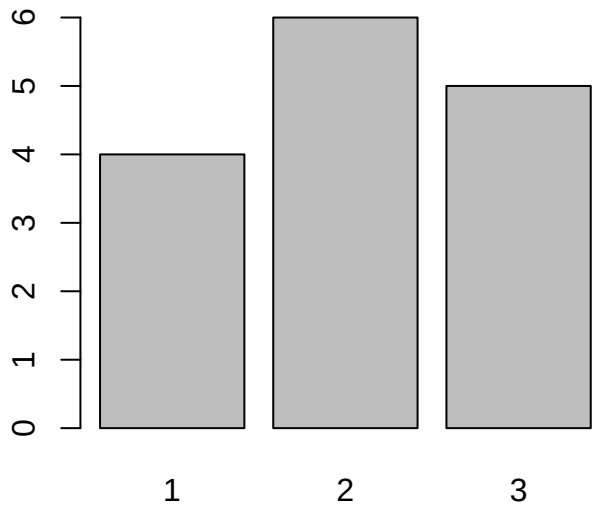
```
##  [1] I   II  III II  I   II  III III II  I   II  III III I   II
## Levels: I II III
```

there are some tricky rules for factors in R (see exercise #1)

Bar Plot

In order to do a bar-plot of categorical data frequencies, you can use table(), or factor. When you pass one argument to plot it tries to do a frequency plot (instead of a scatter if its x,y)

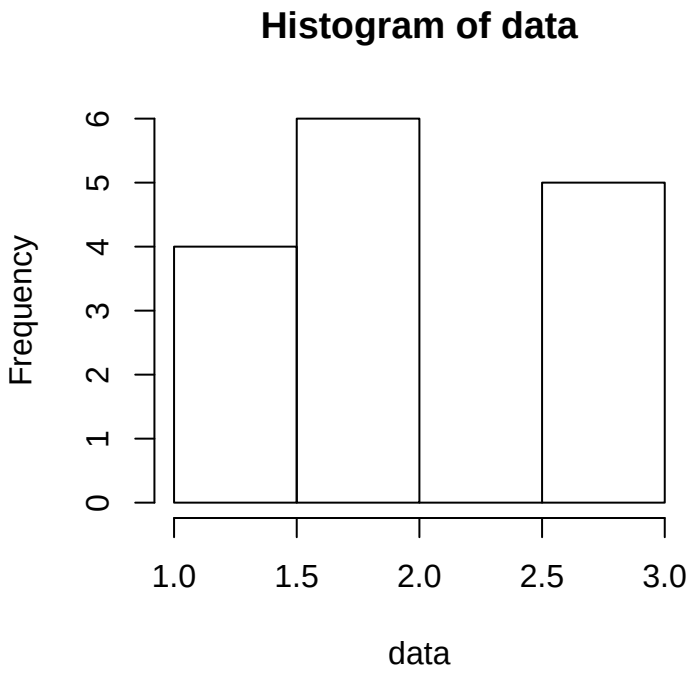
```
plot(factor(data))
```



Histogram

alternatively we could just do a histogram on the vector with no levels. The histogram is for numerical data - can choose bins

```r
hist(data)
```



Lists

Remember, atomic vectors can only hold **ONE** type of data, but lists can hold any type of data including LISTS themselves!

```r
c("ace","hearts",1)    #everything gets coerced into string
```

```
## [1] "ace"    "hearts" "1"
```

```r
list("ace","hearts",1)
```

```
## [[1]]
## [1] "ace"
##
## [[2]]
## [1] "hearts"
##
## [[3]]
## [1] 1
```

data frames

- type of **list** that groups vectors into a two dimensional table
- each vector is a column (variable) of a particular data type
- different columns can be different data types (but same within a column)

- data frames cannot combine columns of different length

```
df <- data.frame(face=c("ace","two","six"),
          suit=c("clubs","clubs","clubs"),value=c(1,2,6))
head(df)
```

```
##   face  suit value
## 1  ace clubs     1
## 2  two clubs     2
## 3  six clubs     6
```

functions for inspecting dataframes

`nrow(my_data_frame)` returns number of rows

`ncol(my_data_frame)` returns number of columns

`dim(my_data_frame)` returns dimensions (rows,columns)

`colnames(my_data_frame)` returns names of the columns

`rownames(my_data_frame)` returns names of the rows

`head(my_data_frame)` returns the first few rows

`tail(my_data_frame)` returns the last few rows

`View(my_data_frame)` opens dataframe in spreadsheet

Not all data is a dataframe

Data can have arbitrary structure. Sometimes you might be looking at data, but it does *NOT* have the structure of a dataframe. You can check whether or not you have a dataframe by doing

```
is.data.frame(data)
```

If your data is not in the structure of dataframe, that's ok, you will learn methods to transform it into one if you need to, various functions such as flatten() etc. will be introduced later.

Read in Tabular Data to DataFrame

We don't want to create data frames manually, we want the computer to create for us. Here we will work with a deck of cards data that has already been in written to a csv file. You can also import data by clicking a button (see Exercise #2).

```
deck <- read.csv("data/deck.csv",stringsAsFactors=FALSE)  # import
head(deck)
```

```
##    face   suit value
## 1  king spades    13
## 2 queen spades    12
## 3  jack spades    11
## 4   ten spades    10
## 5  nine spades     9
## 6 eight spades     8
```

Selecting data

```
deck[1,1]  # elements can be referenced by position row, column
```

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

```
deck[1,]         # leaving blank gives all elements in first row
```

```
##   face   suit value
## 1 king spades    13
```

```
deck[1,c(1,2,3)] # we can also pass vector of indices
```

```
##   face   suit value
## 1 king spades    13
```

$ reference method

Usually, it is recommended not to reference data frames directly using positional arguments, but rather by using the $ to return columns and then slicing those columns as desired. This is because column positions can change, so we want to reference by the name of the column

```
deck$value  # gives you the full column vector - 52 values
```

```
##  [1] 13 12 11 10  9  8  7  6  5  4  3  2  1 13 12 11 10  9  8  7  6  5  4
## [24]  3  2  1 13 12 11 10  9  8  7  6  5  4  3  2  1 13 12 11 10  9  8  7
## [47]  6  5  4  3  2  1
```

```
deck$value[1]  # the first element of the vector
```

```
## [1] 13
```

---

the $ reference gives a vector and not a list/dataframe

```
typeof(deck$value)      # it's an integer vector
```

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

```
class(deck['value'])    # it's a dataframe
```

```
## [1] "data.frame"
```

```
typeof(deck['value'])   # a dataframe is a type of list
```

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

```
mean(deck$value)
```

```
## [1] 7
```

Modifying a vector

```
vec <- c(0,0,0,0,0,0)
vec
```

```
## [1] 0 0 0 0 0 0
```

```
vec[c(1,3,5)] <- c(1,1,1)
vec
```

```
## [1] 1 0 1 0 1 0
```

```
vec[7] <- 1     # can expand a vector (add new element)
vec
```

```
## [1] 1 0 1 0 1 0 1
```

Modify a dataframe

```
deck$new <- 1:52   # create a new column

deck[1,]
```

```
##   face   suit value new
## 1 king spades    13   1
```

```
deck$new <- NULL   # remove the column from your dataframe

deck$value[c(13,26,39,52)] <- c(14,14,14,14)  # change value of aces

deck$value
```

```
##  [1] 13 12 11 10  9  8  7  6  5  4  3  2 14 13 12 11 10  9  8  7  6  5  4
## [24]  3  2 14 13 12 11 10  9  8  7  6  5  4  3  2 14 13 12 11 10  9  8  7
## [47]  6  5  4  3  2 14
```

Logic

Most logic in R is performed ***elementwise***. You should be familiar with the concept of logical AND and logical OR. Examples:

```
x <- TRUE

y <- FALSE

x & y     # logical AND
```

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

```
x | y     # logical OR
```

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

```
1 > 2
```

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

```
1 > c(0,1,2)
```

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

```
c(1,2,3) == c(3,2,1)
```

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

```
c(1,2,3) %in% c(3,4,5)
```

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

Logical Indexing

Logical indexing is very important tool for selecting data that you want based on condition.

Let's see how this works

```
my_data <- c(1,2,3,4,1,2,3,4,1,2,3,4,1,2,3,4,1,2,3,4)

my_data == 2
```

```
##  [1] FALSE  TRUE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE  TRUE FALSE
## [12] FALSE FALSE  TRUE FALSE FALSE FALSE  TRUE FALSE FALSE
```

```
my_data[my_data==2]
```

```
## [1] 2 2 2 2 2
```

logical indexing continued. . .

I'm going to rewrite the last two lines in the following way

```
condition <- my_data == 2

my_data[condition] <- 7   # change all the data that is 2 to 7

my_data
```

```
##  [1] 1 7 3 4 1 7 3 4 1 7 3 4 1 7 3 4 1 7 3 4
```

Notice this is very useful. Instead of manually finding all of the 2's in my vector. I can automatically change all of the 2's in my vector to 7 using logical indexing.

logical indexing continued. . .

```
deck$face == 'ace'   # pay attention to what this does
```

```
##  [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [12] FALSE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [23] FALSE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [34] FALSE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE
## [45] FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE
```

```
deck$value[deck$face=='ace'] <- 100 # find aces, set value to 100
deck$value
```

```
##  [1]  13  12  11  10   9   8   7   6   5   4   3   2 100  13  12  11  10
## [18]   9   8   7   6   5   4   3   2 100  13  12  11  10   9   8   7   6
## [35]   5   4   3   2 100  13  12  11  10   9   8   7   6   5   4   3   2
## [52] 100
```

R Scripts and Functions

You can store functions and other lines of code in R scripts, press the 'source' button when you are done.

```
# function to roll 2 dice and return the sum

roll <- function() {
die = 1:6
dice <- sample(die,size=2,replace=TRUE)  # random sample
sum(dice)                                # return
}

roll()
```

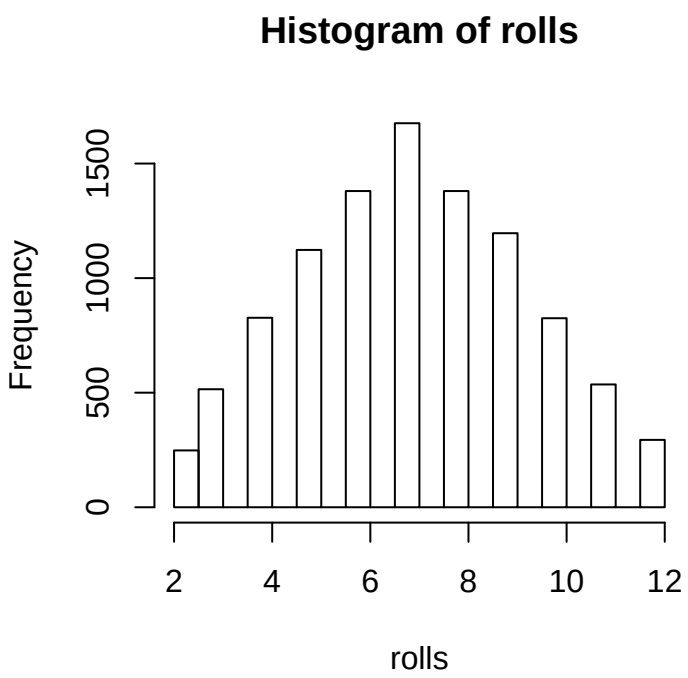
```
## [1] 9
```

Notice that the **last line** inside the curly brackets is what the function returns

Replicate

let's call our roll function to roll two dice 10000 times. The 10000 sums are stored into a vector and a histogram is created

```
rolls <- replicate(10000, roll())
hist(rolls)
```



Functions with arguments

```
deal <- function(cards) {
  cards[1, ]
}

shuffle <- function(cards) {
  random <- sample(1:52, size = 52) # sample WITHOUT replacement!!
  cards[random,]
}

head(shuffle(deck),n=5L)   # notice the order is now random
```

```
##     face     suit value
## 47   six   hearts     6
## 8    six   spades     6
## 45 eight   hearts     8
## 52   ace   hearts   100
## 27  king diamonds    13
```

R Markdown

This is an R Markdown document. Markdown is a simple formatting syntax for authoring HTML, PDF, and MS Word documents. For more details on using R Markdown see http://rmarkdown.rstudio.com.

When you click the **Knit** button a document will be generated that includes both content as well as the output of any embedded R code chunks within the document. You can embed an R code chunk like this:

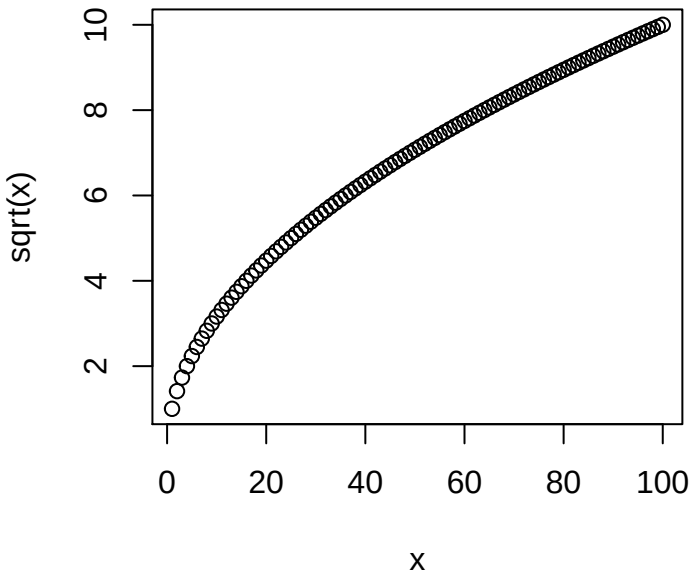
```
x <- 1:100
mean(x)                 # R function to get mean of the vector
```

```
## [1] 50.5
```

Including Plots

You can also embed plots, for example:

```
plot(x,sqrt(x))
```



Practical R Tips

- to clear the console type ctrl + L
- to clear all variables type: rm(list = ls(all.names = TRUE))
- to get info about a vector or dataframe use summary()

```
summary(deck)
```

```
##      face              suit              value       
##  Length:52          Length:52          Min.   :  2.00  
##  Class :character   Class :character   1st Qu.:  5.00  
##  Mode  :character   Mode  :character   Median :  8.00  
##                                        Mean   : 14.62  
##                                        3rd Qu.: 11.00  
##                                        Max.   :100.00
```

Lab 1

(Time allotment ~ 10 minutes)

Exercise 1: More with Factors

1a) Ordered Factor

As an example of an *ordered* factor, consider months. Copy the following lines of code into your console. For each, hit enter to run the code, and then inspect the output.

```
mons = c("March","April","January","November","January","September",
         "October","September","November","August","January",
         "November","February","May","August","July","December",
         "August","August","September","November","February","April")
```

```
mons = factor(mons)
```

```
table(mons)
```

Notice that the proper ordering of months is not reflected in the table

Now we will factor again, but this time choose **BOTH** the levels and the ordering. Again, copy the lines of code below into your console, run each, and inspect each output **carefully**

```
 mons = factor(mons,levels=c("January","February","March","April",
              "May", "June","July","August","September","October",
              "November","December"),ordered=TRUE)
```

```
mons
```

```
table(mons)
```

```
mons[1] < mons[2]
```

you now have ordered data!

Be aware that factors are not vectors and you cannot add a value to a factor if it is not one of the levels, this is why we usually do not choose to import string variable columns as factors when we import tabular data into a data frame, that is, when we build a dataframe it is usually *very important* to set

```
stringsAsFactors=FALSE
```

otherwise R has a default tendency to factor string data columns automatically which might not be desirable

---

1b) Factors from numeric data using cut()

You can use the cut function to create categorical levels from numerical data. Copy the following code into your console:

```
x = c(17,19,22,43,14,8,12,19,20,51,8,12,27,31,44)
```

```
cut(x,3,labels = c("low","medium","high"),ordered=TRUE)
```

you should see categorical factor data now. Think about how useful this could be for splitting up large numerical data sets into different levels. Statisticians do this all the time! You can also choose the way in which you split the numerical data, for example you can do it by quantiles as shown below

```
cut(x,breaks=quantile(x,c(0,0.25,0.5,0.75,1)),labels =
      c("Q1","Q2","Q3","Q4"),include.lowest=TRUE,ordered=TRUE)
```

Exercise 2: Work with a dataframe

2a) Importing data

Download the file deck.csv from the following **url** into a convenient folder on your computer

1. Click Session in the top menubar of RStudio
2. Set the working directory as where the deck.csv file is
3. Import the data file deck.csv by clicking the Import Dataset button in the Environment Pane.
4. Uncheck the String as Factors box - click ok

Inspect the deck dataframe visually by clicking on deck in the environment pane. You should see 52 rows of cards with three columns, face, suit and value

---

2b) practicing select and inspection

You can select data using negative indices. This will return everything EXCEPT the negative indices given. As an example run the code below and inspect the output

```
deck[-2:-52,1:3]
```

you can also pass column names as indices, for example try the following code in the console

```
deck[1,c("face","suit","value")]
```

call each of the dataframe functions on slide 13 above on the deck dataframe that is call nrow(deck), ncol(deck), etc. and *carefully* inspect the outputs, make sure to try

```
View(deck)
```

2c) hearts

In the popular card game hearts, all cards have a value = 0, except hearts which have value = 1, and the queen of spades, which has a value = 13. We will modify our deck dataframe so that the values are consistent with the game hearts. In order to do this you will

1- Create an Rscript: go to file and select new Rscript

2- In this Rscript you will write a function called hearts, which will take as an argument an object called cards

3- Rather than modifying the deck using purely positional arguments use $ and logic

4- First of all you can set cards$value = 0 for the first line of your function (inside the curly brackets) to set all cards value to zero

---

5- Next use logic conditions to filter and assign all the hearts in the deck a value of 1. Use the $ notation, e.g.

```
cards$value[cards$suit == "hearts"] <- 1
```

6- Finally in order to select the queen of spades. You will need to find the card in the deck which has face variable == queen and suit variable == spades. You can use the boolean operator & to do a logical AND test. The final line of your code should return the modified deck.

7- When you are finished writing the function hit the source button and save your Rscript to your computer

8- Finally, go to the console and try running your hearts function on the deck dataframe you imported, that is run hearts(deck) and make sure all of the values are correct - if something is wrong figure out what it is!

2d) Writing output data

You will often need to save data you are working on in R. To practice, run the following code.

```
heartsdeck <- hearts(deck)
write.csv(deck,"hearts-deck.csv", row.names=FALSE)
```

Check that the file hearts-deck.csv appears in your working directory and that all the data looks good. We don't want the row names as these are just labelling the row number which is unimportant for us.