**R Primer and Cheat Sheet**

This document is intended to introduce you to the basics of base R. This does not cover all of R programming, but does cover enough to allow basic data manipulation and analysis. This should teach you enough R to make it easier to learn the rest. This document (particularly the last page) can also serve as a reference guide as you continue to learn R in the future.

---

**R in a nutshell:**

R uses functions to manipulate data structures.

You supply R with the data structure you want manipulated and the function you want R to use and R happily obliges.

Note that "#" in R indicates notes (and text following a "#" is not read by R as a command).

---

**Data Structures**

Data structures are "containers" for your data.

Three types of data structures that will be highly relevant to you:

Scalars: holds a single number. "1", "2", "3"

Vectors: holds lists of scalars. "1, 2, 3, 4, 5, 6, 7, 8, 9, 10"

Data frames: holds lists of vectors (like a spreadsheet)

To put data into a data structure, use **<-**:

```
structureName<-data
#You can call your data structures whatever you like. R doesn't care.
#It just can't start with a number or contain certain symbols (e.g. %)
#Some tips: make your structure names memorable, easy to type, and informative
#Try to follow a convention. Some people use snake_casing. I usually use camelCasing.
```

```
#To create a scalar:
        scalar1<-1
        scalar2<-2
        scalar3<-109387958125096128995408

#To create a vector, use the c() command:
        vector1<-c(1,2,3,4,5,6,7,8,9,10)

#(you can also use 1:10 to tell R "All of the numbers between 1 and 10")
        vector2<-25:34
        vector3<-c(100,4,21.5,16^2,33,8,1286,444,0,1/4)

#Or, if you already have data inside some scalar data structures:
        vector4<-c(scalar1,scalar2,scalar3)

#To create a data frame, use the data.frame() command:
        yourData<-data.frame(vector1, vector2, vector3)
#For this to work, your vectors need to be "of the same length" (i.e., have the same number
of scalars in them) or the longer vectors need to be multiples of the shorter vectors

#To change specific elements of your data structures, you use "indexing"
#Indexing is done with brackets in R: []
#yourdata[row,column] pulls out the specified row and column of yourdata.

#To change the item in the 5th row and 32nd column, you would:
        yourData[5,32]<-newData
```

**#To refer to all columns and rows, you just leave that section of the index reference blank**

**#Every column in the 3rd row:**

**yourData[3,]**

**#Every row in the 50th column:**

**yourData[,50]**

---

When using R for data analysis, you will most often assign your data to a data frame (using **data.frame()**) with each of your variables as a vector within this dataframe.

There are lots of ways to import data into R. We'll talk about some more later. But for now, an easy way is to combine the **read.csv()** and **file.choose()** commands:

**yourData<-read.csv(file.choose())**

**#This will open a new explorer/finder window on your computer and let you navigate to your file. When you select it, it will automatically be imported into R and stored as a dataframe in your global environment.**

**#Make sure the file you're importing is .csv format**

---

**R and Functions**

In R, you ask R to perform different functions on your data structures.

The basic structure of R functions is as follows:

functionName(dataStructure,options)

**functionName** would be the name of your given function.

**dataStructure** would be the name of the data structure you want to manipulate.

> This will almost always be a vector. Isolated vectors can be referred to by name. To refer to a vector within a dataframe, use a "$" as follows: **dataframe$vectorName**

**options** refer to any additional options you want from that function. You can see what options are available for a function by using "?" as follows: **?functionName**. This will pull up the help dialog for that function, including a description of all available options.

**Common R Functions and Operators**.

Below are commonly used functions and operators in R, organized into various sections. Just type this code into R and replace things like “x” and “y” with your data structures.

**Basic Arithmetic**

These operators do basic arithmetic calculations on your data.

Addition:

```
3+2

10190835+91730513
```

Subtraction:

```
400-300

0-12
```

Multiplication:

```
100*100

.5*5
```

Division:

```
1200/600

0/40
```

Exponents:

```
2^2

5^2.5
```

Square root:

```
sqrt(4)
```

#Or

```
4^.5
```

Summation:

```
sum(c(1,2,3,4,5))
```

Sum all columns or rows:

**colSums(dataframe)**

**rowSums(dataframe)**

---

**Comparisons and Logicals**

These operators compare data structures or evaluate logical statements, but do not necessarily calculate new values. They generally return either "TRUE" if the statement is true and "FALSE" if it is not.

Handy tip: R treats "TRUE" as being equivalent to 1 and "FALSE" as being equivalent to 0. So **TRUE + TRUE = 2**; **TRUE + FALSE = 1**; and **FALSE + FALSE = 0**

Equivalence:

**4 == 4**

**4 == 1**

**4 == c(1,2,3,4)**

Non-equivalence:

**4 != 4**

**4 != 1**

Greater or Less Than:

**4 > 4**

**4 < 1**

Greater/Less Than or Equal To:

**4 >= 4**

**4 <= 1**

Containment:

**4 %in% c(1,2,3,4)**

**4 %in% c(5,6,7,8)**

AND:

```
#Are both of these statements true?

    4 > 3 & 4 < 5

    4 < 1 & 4 < 7
```

OR:

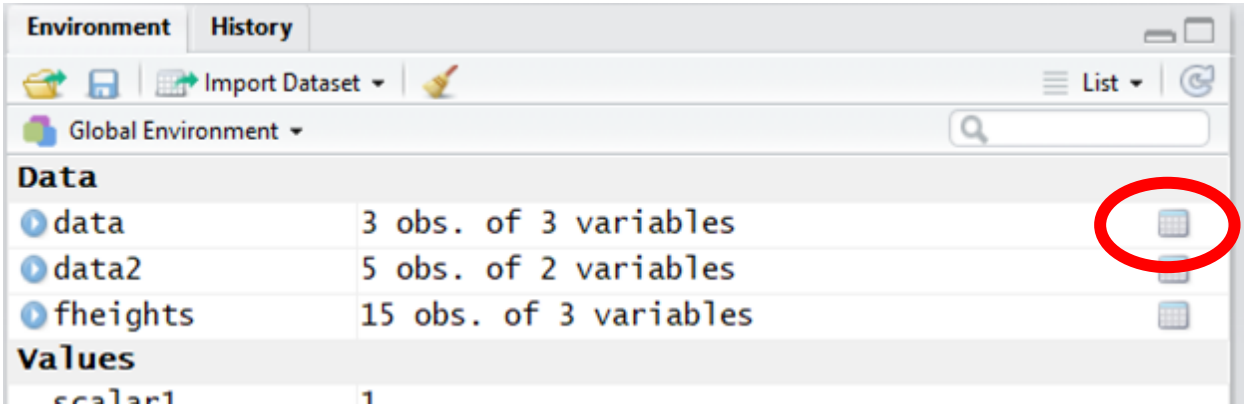
```
#Is at least one of these statements true?

    4 < 1 | 4 < 7

    4 < 0 | 4 > 100
```

---

**Viewing Your Data**

In order to work with your data, it's often helpful to be able to look at it; both to inspect the individual cases for issues (e.g. missing data), and to see what your variables are actually called.

In RStudio, you have two easy options to do this. First, you can click the table icon next to your dataframe in the "Environment" tab:




This is good if you *really* need to be able to look at the observations. If you just need to remember the name of a variable, you can use the **names()** function:

```
names(yourData)
```

This will show you all of the vector names in yourdata in order.

**Summary Statistics**

These functions generally take a vector as an input and each compute some common summary statistic value.

Sample size:

**#For sample size, you have two options. Which you use depends on what kind of data structure you are using.**

**#For dataframes:**

```
nrow(dataframe)
```

**#For vectors:**

```
length(vector)
```

Mean:

```
mean(x)
```

**#Note: R really hates missing data. If you have any missing values in your vector, this will return "NA". To avoid this, use the "na.rm=TRUE" option, which means "Remove any missing values before you calculate the mean". You can use this for all summary functions.**

```
mean(x,na.rm=T)
```

Mean of column or row

```
colMeans(x)
```

```
rowMeans(x)
```

Median

```
median(x)
```

Standard Deviation:

```
sd(x)
```

Range:

**#This function returns the highest and lowest values in the vector**

```
range(x)
```

Summary:

```
#This function outputs a number of useful summary values
    summary(x)
```

---

**Plotting data**

The below functions allow you to visualize your data. They generally take one or two vectors as inputs.

Histogram:

```
    hist(x)
```

Plot:

```
    plot(y~x,data=data,co="color",type="p,l")
#Base plotting function
#col changes the line color, type changes the type of plot (p = points, l = lines)
```

Better plotting

```
#The base plotting function is good for quick-and-dirty plots
#But for more complex and publication-quality plots, you'll want to use the ggplot2 package
#This is more complicated, but allows you to create great figures
#The most basic and most useful function in ggplot2 is the "ggplot" function:
    ggplot(data=data,aes(x=x,y=y))+
        geom_point()

#ggplot2 plots work by continually adding options onto a basic plot. Some options include:
    ggplot(data=data,aes(x=x,y=y))+
        geom_point()+
        theme_classic()
#theme_classic() basically makes a plot APA style (and prettier)
```

```
        ggplot(data=data,aes(x=x,y=y))+
                geom_point()+
                theme_classic()+
                geom_smooth(method="lm",se=F)
#geom_smooth adds a best-fit line to the plot

#Other kinds of plots are possible too:
        ggplot(data=data,aes(x=x,y=y))+
                geom_bar(stat="summary",fun="mean")

#ggplot2's documentation is also fantastically helpful:
https://ggplot2.tidyverse.org/index.html
```

---

**Data Wrangling**

R is really good at analyzing data. It's also really useful for manipulating data: combining, labeling, rearranging, repeating, etc. Below are a number of functions commonly used to wrangle data.

Generate a sequence of data:

```
#All values between 1 and 10
        1:10
        #Or
        seq(1,10,1)

#All values between 1 and 10 skipping by 2
        seq(1,10,2)
```

Repeat a sequence of data:

```
#Repeat 5 times
```

```
rep(1,times=5)
```

**#Repeat each vector element twice**

```
rep(c(1,2),each=2)
```

**#Repeat the vector twice while also repeating each element twice!**

```
rep(c(0,1),times=2,each=2)
```

Put vectors or dataframes together:

**#Smoosh two vectors or dataframes together side-by-side (by columns):**

```
cbind(vector1,vector2)
```

**#Smoosh two vectors or dataframes together top-to-bottom (by rows):**

```
rbind(vector1,vector2)
```

Pick a random subset of data:

```
sample(data,samplesize,replace=TRUE/FALSE)
```

**#Data is the data you want to sample from, samplesize is the size of the sample you want, replace determines whether you sample with replacement**

```
sample(c(1,2,3,4,5),1,replace=F)
```

**#Sample can also be used to generate data. For example, this will generate a vector of 5 values, randomly equal to 1 or 2**

```
sample(c(1,2),5,replace=T)
```

Random Binomial Data

**#Simulates draws from a binomial distribution**

**#Good for simulating coin flips**

```
flips<-rbinom(numbercoins, numberflips, probabilityheads)
```

```
flips<-rbinom(1,1,.5)
```

```
flips<-rbinom(100,5,.75)
```

Random Uniform Data

```
#Simulates draws from a uniform distribution
#Good for simulating dice rolls
        rolls<-runif(numberdice,minvalue,maxvalue)
        rolls<-runif(1,1,6)
        rolls<-runif(100,10,90713593)
```

Random Normal Data

```
#Simulates draws from a random normal distribution
#Good for simulating psychological variables

        norm<-rnorm(samplesize,mean,sd)
        height<-rnorm(100,70,15)
```

---

**Control Flow**

R also contains a number of functions that allow you to control how R programs progress with conditional loops and branches.

If function

```
#An if function executes a section of code only if some statement is true
        if(statement){
                code
        }

#For example:
        if(4>3){
                print('Yep! ')
        }
```

```
#Note that maintaining white space and indentation is very important for keeping if() and similar functions readable. To automatically format an if statement (or any chunk of R code), select the code and hit CTRL+Shift+A on a PC or CMD+Shift+A on a Mac
```

Else function

```
#An else function executes a section of code if a statement is false

    if(statement){
        code1
    } else{
        code2
    }

#For example:

    if(3>4){
        print('Yep! ')
    } else{
        print('Nope! ')
    }
```

Ifelse function

```
#An ifelse function just crams an if and else function into one line. Very handy!

    ifelse(statement,codeiftrue,codeiffalse)

    ifelse(4>3, 'Yep! ', 'Nope! ')

    ifelse(3>4, 'Yep! ', 'Nope! ')
```

For loop:

```
#A for loop repeats a section of code for a pre-specified number of iterations.

    for(i in 1:iterations){
        code
    }
```

```
#The "i" acts as a counter that updates itself after each iteration. You can replace "i" with any letter or valid data structure name that you like. Try to make it something that makes sense to you.

#You can use this to, for example, fill a vector step by step:

    vector<-rep(0,100)

    for(step in 1:100){

        vector[step]<-step

    }
```

While loop:

```
#A while loop repeats a section of code so long as a statement is true.

    while(statement){

        code

    }

#For example:

    a<-0

    b<-1000

    while(sum(a)<b){

      #This will keep appending values to a until the sum of all values is greater than b

      a<-c(a,length(a))

    }
```

Apply function:

```
#An apply function repeats a function over each row and column of a dataframe

    apply(data,row/column,function(x)

        function

        )
```

```
#Data refers to the data you want to manipulate, row/column refers to whether you want the function to repeat across rows (1 for rows) or columns (2 for columns), function refers to the function you want repeated. You can replace "x" with whatever data structure name you like.

    apply(data,1,function(x)

        sum(x)

        )

#Takes the sum of every row. Equivalent to rowSums(data)
```

Other apply functions:

```
    sapply(vector,function(x)

        function

    )

#Sapply applies a function to each element of a vector

    lapply(vector1,vector2,function(x)

        function

    )

#Lapply applies a function to vector1 separately for each level of vector 2
```

---

**Custom Functions**

Finally, if all else fails and R simply doesn't have a function to do what you want to do, you can write your own custom function! This function can take whatever inputs you desire, process those inputs however you want, and output whatever you like.

```
#To write a custom function:

    functionName<-function(inputs){

        code

        return(output)

    }
```

```
#To use your function, run this code and then call your function like any other:
        functionName(inputs)
#For example:
        addTwoValues<-function(value1,value2){
                output<-value1 + value2
                return(output)
        }
        addTwoValues(1,2)
```

# Base R Cheat Sheet

**Data Structures**
scalar = single number
vector = list of scalars
dataframe = list of vectors

**Assignment**
structureName<-data

**Indexing**
structureName[row,col]
data$variable

**Specify data type**
vector<-as.numeric(vector)
vector<-as.factor(vector)

**Install and load a package**
Install.packages("packagename")
library(packagename)

**Import data (from CSV)**
data<-read.csv(filepath)
data<-read.csv(file.choose())

**Save data (to CSV)**
write.csv(data,filepath,row.names=F)
write.csv(data,file.choose(),row.names=F)

---

**Arithmetic operators**
+, -, *, /, ^, sqrt(x), sum(x)

**Comparisons and logicals**
== ,!=, >, >=, <, <=, %in%, &, |

**Length and Sample Size**
length(vector)
nrow(dataframe)

**Mean**
mean(vector,na.rm=T)
colMeans(dataframe,na.rm=T)
rowMeans(dataframe,na.rm=T)

**Median**
median(vector)

**Standard deviation**
sd(vector)

**Range**
range(vector)

**Summary**
summary(vector)

**Single sample t-test**
t1<-t.test(data$y,mu)
t1

**Between subjects t-test**
t1<-t.test(data$y~data$x)
t1

**Correlation**
cor(data$x,data$y,use="pairwise.complete.obs")
cor.test(data$x,data$y)

**Regression**
r1<-lm(y~x1+x2,data=data)
summary(r1)

**Regression (with interaction)**
r1<-lm(y~x1*x2,data=data)
summary(r1)

**One-way ANOVA**
a1<-lm(y~x,data=data)
anova(a1)

**Two-way ANOVA**
a1<-lm(y~x1*x2,data=data)
anova(a1)

**Repeated Measures ANOVA (with long format data)**
a1<-aov(y~x+Error(participantnumber/x),data=data)
anova(a1)

**Chi-squared test**
chisq.test(data)

---

**Histogram**
hist(vector)

**Basic Plot**
Qplot(y~x,data=data)

**Ggplot2 Plot**
ggplot(data=data,
aes(x,y,color=z))+theme_classic()+geom_smooth(method="lm",se=F)

---

**Generate a Sequence of Data**

```
vector<-1:10
seq(start,finish,by=skip)
```

**Repeat data**

```
vector<-
rep(data,times=t,each=e)
```

**Merge data structures**

```
new<-
cbind(vector1,vector2)
new<-
rbind(vector1,vector2)
```

**Sample from data**

```
sample(data,size,replace=T/
F)
```

**Random Binomial Data**

```
rbinom(ncoins,nflips,prob)
```

**Random Uniform Data**

```
runif(ndice,min,max)
```

**Random Normal Data**

```
rnorm(n,mean,sd)
```

---

**If function**

```
If(statement){
        code
}
```

**Else function**

```
if(statement){
        code1
} else{
        code2
}
```

**Ifelse function**

```
ifelse(statement,code1,cod
e2)
```

**For loop**

```
for(i in 1:iterations){
        code
}
```

**While loop**

```
while(statement){
        code
}
```

**Apply functions**

```
apply(data,row/column,fun
ction(x)
function
)

sapply(vector,function(x)
function
)

lapply(vector1,vector2,funct
ion(x)
function
)
```

**Custom function**

```
functionName<-
function(inputs){
        code
        return(output)
}
```