

# **A First Look at R**

Research Methods for Political Science

---

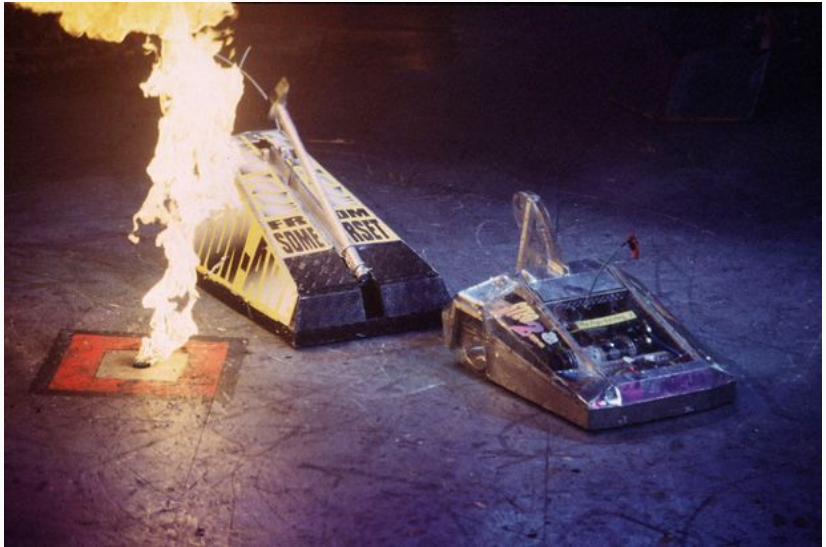
Thomas Chadeaux

Trinity College Dublin

**A fun goal by the end of this class  
(and next week)**

---

# Robot War



Ok, more like that



## Ok, seriously, more like that:

```
winnerRecord <- NULL # Initialize the object
for(i in 1:10){
  winner <- play()
  winnerRecord <- c(winnerRecord, winner)
}
```

```
winnerRecord
```

```
## [1] 0 2 1 1 1 2 2 2 2 2
```

and the winner is...

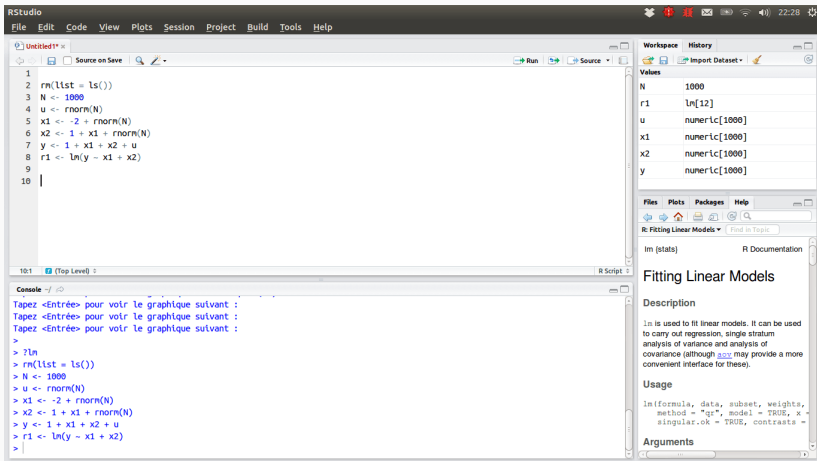
```
## [1] "...Robot 2 !"
```

# Software

---

# What you absolutely need to do (reminder)

- Download R at [www.r-project.org](http://www.r-project.org), install
- Download RStudio at [www.rstudio.com](http://www.rstudio.com) and install (not absolutely necessary, but strongly recommended)



The screenshot displays the RStudio environment. The main editor window shows an R script with the following code:

```
1  
2 rm(list = ls())  
3 N <- 1000  
4 u <- rnorm(N)  
5 x1 <- -2 + rnorm(N)  
6 x2 <- 1 + x1 + rnorm(N)  
7 y <- 1 + x1 + x2 + u  
8 r1 <- lm(y ~ x1 + x2)  
9  
10 |
```

The console window at the bottom shows the execution of the script, with the following output:

```
> ?lm  
> rm(list = ls())  
> N <- 1000  
> u <- rnorm(N)  
> x1 <- -2 + rnorm(N)  
> x2 <- 1 + x1 + rnorm(N)  
> y <- 1 + x1 + x2 + u  
> r1 <- lm(y ~ x1 + x2)  
> |
```

The environment pane on the right shows the following variables:

Variable	Value
N	1000
r1	lm[12]
u	numeric[1000]
x1	numeric[1000]
x2	numeric[1000]
y	numeric[1000]

The right sidebar shows the 'Fitting Linear Models' documentation page, which includes a description of the `lm` function and its usage.

## What you absolutely need to do (reminder)

Both are free and available on most Operating Systems (Mac/Windows/Linux).

You can even run R directly from your browser, but I really don't recommend it beyond just trying a few things ([https://rextester.com/l/r\\_online\\_compiler](https://rextester.com/l/r_online_compiler))



# Why R?

- Free
- Huge community
- Very popular and rising
- Extremely powerful
- Transparent, reproducible, shareable

# The Rstudio environment

---

# The four panes

The screenshot shows the RStudio application window with four main panes. The top-left pane is the Source editor, the top-right is the Console, the bottom-left is the Environment/History pane, and the bottom-right is the Files/Plots/Packages/Help pane. Each pane has a corresponding label and description box overlaid on it.

**1. SOURCE**

This is where you write your code!

Your code will not be evaluated until you “Run” them to the console.

Click “Run” to send your code to the console

**2. CONSOLE**

This is where your code from the Source is evaluated by R.

You can also use the console to perform quick calculations that you don’t need to save

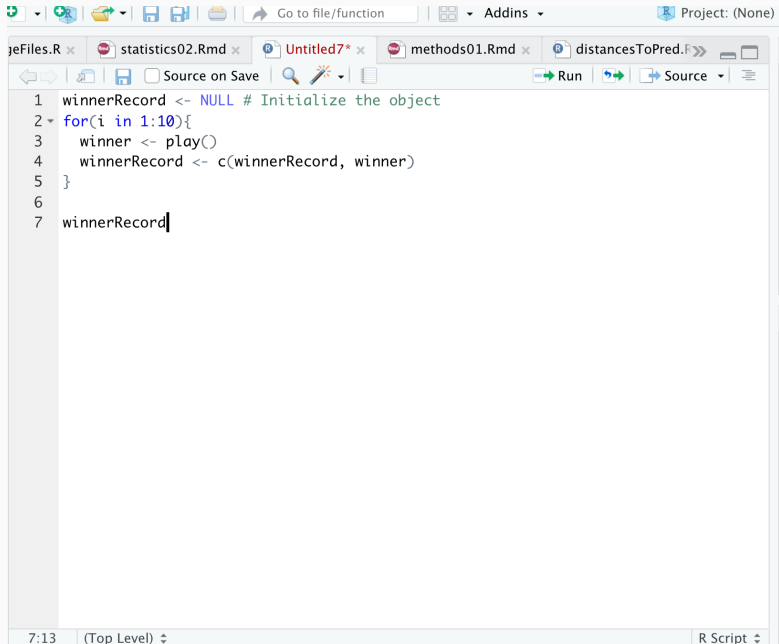
**3. Environment / History**

Here you can see what objects are in your working space (Environment) or view your command history (History)

**4. Files / Plots / Packages / Help**

Here you can see file directories, view plots, see your packages, and access R Help

# The Source pane



# The console pan

Console ~/Desktop/asdf/ ↗

```
R version 3.3.1 (2016-06-21) -- "Bug in Your Hair"
Copyright (C) 2016 The R Foundation for Statistical Computing
Platform: x86_64-apple-darwin13.4.0 (64-bit)
```

```
R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.
```

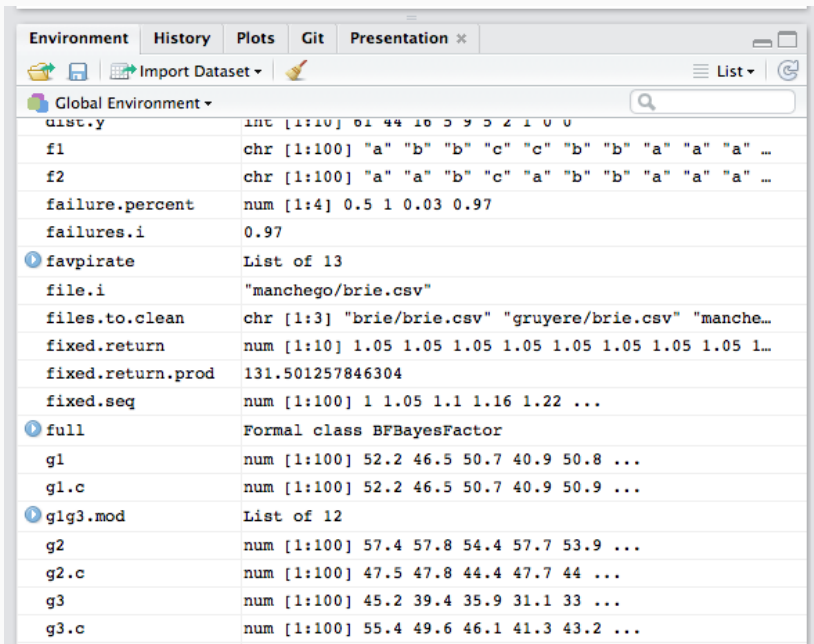
```
  Natural language support but running in an English locale
```

```
R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.
```

```
Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.
```

```
> |
```

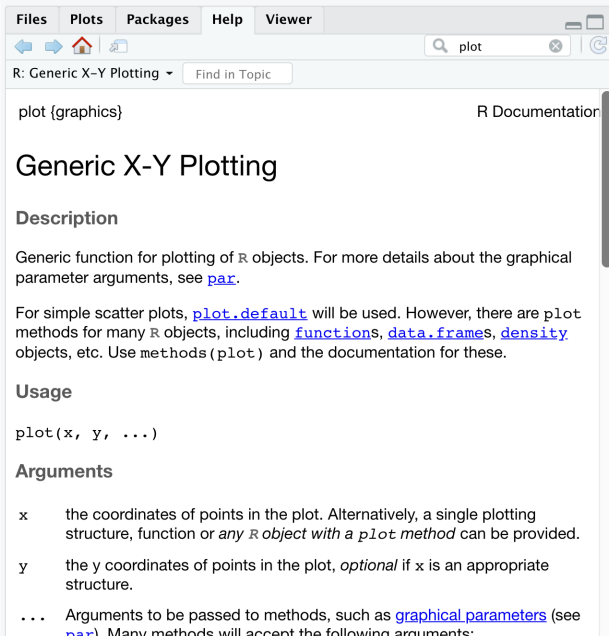
# The Environment pane



The screenshot shows the RStudio Environment pane with the following tabs: Environment, History, Plots, Git, and Presentation. The Environment pane is active, displaying a list of objects in the Global Environment. The objects and their values are as follows:

Object	Value
dist.y	int [1:10] 61 44 16 5 9 5 2 1 0 0
f1	chr [1:100] "a" "b" "b" "c" "c" "b" "b" "a" "a" "a" ...
f2	chr [1:100] "a" "a" "b" "c" "a" "b" "b" "a" "a" "a" ...
failure.percent	num [1:4] 0.5 1 0.03 0.97
failures.i	0.97
favpirate	List of 13
file.i	"manchego/brie.csv"
files.to.clean	chr [1:3] "brie/brie.csv" "gruyere/brie.csv" "manche...
fixed.return	num [1:10] 1.05 1.05 1.05 1.05 1.05 1.05 1.05 1.05 1...
fixed.return.prod	131.501257846304
fixed.seq	num [1:100] 1 1.05 1.1 1.16 1.22 ...
full	Formal class BFBayesFactor
g1	num [1:100] 52.2 46.5 50.7 40.9 50.8 ...
g1.c	num [1:100] 52.2 46.5 50.7 40.9 50.9 ...
g1g3.mod	List of 12
g2	num [1:100] 57.4 57.8 54.4 57.7 53.9 ...
g2.c	num [1:100] 47.5 47.8 44.4 47.7 44 ...
g3	num [1:100] 45.2 39.4 35.9 31.1 33 ...
g3.c	num [1:100] 55.4 49.6 46.1 41.3 43.2 ...

# Files/Plots/Packages/Help Pane



The screenshot shows the RStudio interface with the 'Help' pane active. The top navigation bar includes 'Files', 'Plots', 'Packages', 'Help', and 'Viewer'. Below the navigation bar, there is a search bar containing the text 'plot'. The main content area displays the documentation for the 'plot' function. The title 'Generic X-Y Plotting' is prominently displayed. Below the title, there is a 'Description' section followed by a paragraph explaining the generic function for plotting R objects. This is followed by a 'Usage' section showing the function signature 'plot(x, y, ...)'. Finally, there is an 'Arguments' section listing the parameters 'x', 'y', and '...', each with a description of their role in the plotting process. The 'x' argument is described as the coordinates of points, 'y' as the y coordinates, and '...' as arguments passed to methods.

Files Plots Packages Help Viewer

R: Generic X-Y Plotting Find in Topic

plot {graphics} R Documentation

## Generic X-Y Plotting

### Description

Generic function for plotting of `R` objects. For more details about the graphical parameter arguments, see [par](#).

For simple scatter plots, [plot.default](#) will be used. However, there are `plot` methods for many `R` objects, including [functions](#), [data.frames](#), [density](#) objects, etc. Use methods (`plot`) and the documentation for these.

### Usage

```
plot(x, y, ...)
```

### Arguments

- `x` the coordinates of points in the plot. Alternatively, a single plotting structure, function or *any R object with a `plot` method* can be provided.
- `y` the y coordinates of points in the plot, *optional* if `x` is an appropriate structure.
- `...` Arguments to be passed to methods, such as [graphical parameters](#) (see [par](#)). Many methods will accept the following arguments:

# Basic R

---



# Numbers

Add numbers

```
1 + 2
```

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

```
1 * 2
```

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

```
1 / 2
```

```
## [1] 0.5
```

```
( 2 + 3 ) * ( 2 - 5 )
```

```
## [1] -15
```

# Spaces don't really matter

```
1      +2
```

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

```
1+2
```

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

## You can also write text

For text, you need to use quotation marks

```
"this is a sentence"
```

```
## [1] "this is a sentence"
```

```
'single quotation marks are fine too'
```

```
## [1] "single quotation marks are fine too"
```

## Comments

Write comments with a `#` sign. These are not evaluated by R.  
Helpful to remember what you are doing.

```
1 + 1
```

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

```
# This is ignored
```

```
## 1 + 1 # this is ignored too
```

## Comparison operators

R provides comparison operators that you can use to compare values: `>`, `>=`, `<`, `<=`, `!=` (not equal), and `==` (equal). Each creates a logical test. greater than three?

```
3 > 2
```

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

```
2 > 3
```

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

To test if things are equal, you need to use `=="`

```
3 == 2
```

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

```
3 == 3
```

## Watch out!

An easy mistake to make is to test for equality with `=` instead of `==`.  
When this happens you'll get an error:

```
3=2
```

```
## Error in 3 = 2: invalid (do_set) left-hand side to assign
```

## Boolean operators: &, |, and !

```
3>2 & 4>3
```

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

```
3>2 & 4>5
```

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

```
3>2 | 4>5
```

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

```
!3>2
```

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

## Common mistakes

In R, the order of operations doesn't work like English. You can't write

```
(3 > 4) | 5
```

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

even though you would say: 3 is greater than either 4 or 1. Instead, you need to write:

```
3 > 4 | 3>1
```

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



# Objects

---

## Assign values to an object

An object is like a bucket. You put whatever you want in it using the “<-” sign (< sign and a dash):

```
x <- 1 + 2
```

Note that `x` was not printed. That's because I've put something in it. I didn't ask to see its content. To see the contents of `x`, just type:

```
x
```

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

## You can put (almost) anything you want in an object

```
Iamanoject <- 3  
Iamanojecttoo <- 'Yes I am!'  
I.amAbe.tterOBJECTthan.u <- 3*5  
I.am.probably.a.mistake <- '3*5'
```

It should also be clear that you can name objects almost anything you like, but no spaces are allowed, and you should avoid special characters

## Exercise (easy)

Store the number 32 in a variable called `x`?

## Exercise (easy) solution

```
x <- 32
```

## Exercise (bit harder)

How would you store the result of  $3x$  in a variable called *myprecious*?

```
myprecious <- 3*x
```

## Exercise

What happens if you multiply myprecious with x?

```
myprecious * x
```

```
## [1] 3072
```

Often, we'll want to create a larger collection of numbers? Use the `c()` function (*c* for combine):

```
z <- c(1, 2, 3, 5, 6)
```

Or, for a sequence of consecutive integers, use `:`

```
z <- 1:6
```



Now let's add the number 7 to that vector, without retyping the whole sequence

```
z <- c(z, 7)
```

Notice that I have now overwritten z:

```
z
```

```
## [1] 1 2 3 4 5 6 7
```

## Exercise

Combine two sequences into a variable called 'mycat': the integers from 1 to 15, and the number 100

## operations

You can conduct all kinds of operations on the vector `z`:

```
z*2
```

```
## [1] 2 4 6 8 10 12 14
```

```
(z + 100) / 6
```

```
## [1] 16.83333 17.00000 17.16667 17.33333 17.50000 17.66667
```

Take the square root of  $(z + 1)$ , and then log it:

```
z2 <- sqrt(z + 1)
```

```
z3 <- log(z2)
```

```
z3
```

```
## [1] 0.3465736 0.5493061 0.6931472 0.8047190 0.8958797 0.9502120
```

Or you can do it in one go:

# Recycling

What happens if you add `c(1,2,3,4)` to `c(1,2)`?

```
c(1,2,3,4) + c(1,2)
```

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

The shorter vector gets 'recycled'. However, the code below yields a warning:

```
c(1,2,3,4) + c(1,2,3)
```

```
## Warning in c(1, 2, 3, 4) + c(1, 2, 3): longer object length
```

```
## shorter object length
```

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

## Creating random numbers

There are many ways to do this and we'll cover this later, but for now we'll draw a number from a uniform distribution (i.e., each number is as likely to be picked as any other) between 0 and 100:

```
runif(n = 1,      #we want one number  
      min = 0,    # between 0  
      max = 100) # and 100
```

```
## [1] 8.431574
```

and an integer between 0 and 100:

```
sample(x = 0:100, # draw a number from this list  
       size = 1) # we only want one number
```

```
## [1] 61
```

## Exercise:

- Draw 10 numbers from a uniform distribution, and save them in a variable called 'mrn'
- Sample one number from mrn

## Creating data

The workhorse of data analysis in R is the data frame. To create a data frame, for example:

```
localdat <- data.frame(ID = c(1,2),  
                        gender = c('male', 'female'),  
                        income = c(50000, 60000))
```

You can put complex statements inside that data.frame:

```
localdat <- data.frame(age = round(runif(4,0,100)),  
                        income=round(runif(4, 0, 100000)),  
                        gender=c('male', 'female'))
```

## Importing and exporting data

Instead of creating them, you typically import data frames from your local file system or from the web. `head()` lets you take a look at the first few rows.

```
localdat <- read.csv('mydata.csv')  
head(localdat, 4)
```

```
##   ID age income gender  
## 1  1   4  49672   male  
## 2  2  56  20300 female  
## 3  3  56  73388   male  
## 4  4  99  33374 female
```

Export it to your local file system:

```
write.csv(localdat, file = 'newfile.csv')
```



## Extracting data

Localdat is a data frame. Loosely, a table with data. To access its information, we need to ask R for a row and column number, in that order. For example, to ask for row 1 and column 2, we would write:

```
localdat[1, 2]
```

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

Or perhaps we want to see all columns associated with row 1, in which case we leave the column indicator empty, and similarly if we want all rows associated with a column:

```
localdat[1, ]
```

```
##   ID age income gender  
## 1  1  4  49672   male
```

```
localdat[, 2]
```

## Extracting data (Cont'd)

We can also ask for a specific variable by name in three ways, though the first one is the most common

```
localdat$age
```

```
## [1]  4 56 56 99 22 22 29 14 91
```

```
localdat[, 'age']
```

```
## [1]  4 56 56 99 22 22 29 14 91
```

```
with(localdat, age)
```

```
## [1]  4 56 56 99 22 22 29 14 91
```

## Extracting data (Cont'd)

We might have more specific requests. E.g., we want to see all males younger than 50 with income of less than 20000. In this case there is only one, with ID 7:

```
localdat[localdat$age < 50  
          & localdat$gender == 'male'  
          & localdat$income < 20000, ]
```

```
##   ID age income gender  
## 7  7  29  13124   male
```

## Exercise

Find the ID of all females in localdat?

## Summarizing data

What is the mean income? (NB: we'll talk about means more formally later)

```
mean(localdat$income)
```

```
## [1] 45678.89
```

What is the maximum age?

```
max(localdat$age)
```

```
## [1] 99
```

Also useful:

```
summary(localdat$income)
```

```
##      Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##      4893   20300   49672   45679   73388   84749
```

What is the average income of women under the age of 50?

## Exercise

What is the average income of women under the age of 50?

```
mean(localdat$income[localdat$gender == 'female' &  
                      localdat$age < 50])
```

```
## [1] 65805
```

OR

```
with(localdat, mean(income[gender == 'female' &  
                      age < 50]))
```

```
## [1] 65805
```

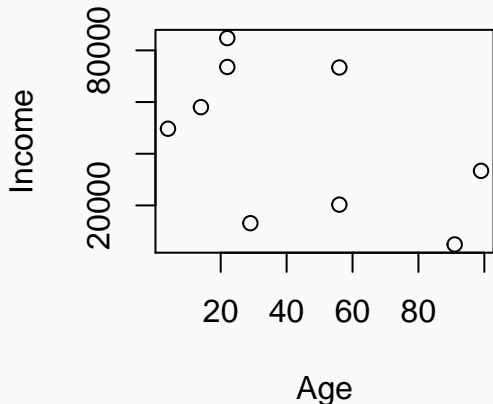
## XY plots

To plot x, y:

```
x <- localdat$age
```

```
y <- localdat$income
```

```
plot(x, y, xlab='Age', ylab='Income')
```





## if statements

Often you will want to check if something is equal, greater, smaller than something else. “if” will tell you if a certain statement is true or not.

```
if(1==2) {print('We need to rethink math')}  
if(1==1) {print('Math is ok')}
```

```
## [1] "Math is ok"
```

We can make this cleaner using “else”:

```
if(1==2){  
  print('We need to rethink math')  
} else  
  print('Math is ok')
```

```
## [1] "Math is ok"
```

# Loops

Often you will want to repeat a certain operation multiple times.  
The way this works is, loosely, as follows:

```
for ( value in sequence ){  
    do something here  
}
```

# Loops

For example, you may want to print all the integers from 1 to 8.

```
for(i in 1:8){  
  print(i)  
}
```

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

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

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

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

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

```
## [1] 6
```

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

```
## [1] 8
```

## Exercise

- Draw 10 numbers from a uniform distribution;
- Calculate and print their mean using the functions `mean()` and `print()`;
- Repeat this operation 100 times.

## Solution

Exercise: draw 10 numbers from a uniform distribution; calculate and print their mean using the functions `mean()` and `print()`; repeat this operation 12 times.

```
for(i in 1:12){  
  mrv <- runif(10)  
  mean.mrv <- mean(mrv)  
  print(mean.mrv)  
}
```

```
## [1] 0.6156325
```

```
## [1] 0.3430734
```

```
## [1] 0.5590263
```

```
## [1] 0.5117077
```

```
## [1] 0.4152132
```

```
## [1] 0.5089787
```

## Exercise

Programme the following situation:

1. Create an “urn” with 100 balls
  - 30 of the balls are black
  - 70 are red.
2. Draw a ball from this urn
3. print the color of that ball.
4. Repeat steps 2 and 3 12 times

NB: the function `rep()` might be useful. For example, `rep('a', 10)` will print a 10 times

```
rep('a', 10)
```

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

# Solution

```
# create balls
blackballs <- rep('black', 30) # create blackballs
redballs <- rep('red', 70) # create redballs
# put them in an urn called "urn"
urn <- c(blackballs, redballs) # create the urn

# Create a loop that will repeat 12 times
for(i in 1:12){
  # Draw a ball from the urn using "sample"
  mydraw <- sample(urn, 1)
  # "Print" the result to the console
  print(mydraw)
}

## [1] "black"
## [1] "red"
## [1] "black"
## [1] "red"
```

Reuse the previous function, but this time save your results (black, red, etc.) in a variable called `mydraws`.

How many red/black balls did you get? (the function `length(x)` calculates the length of vector `x`)



## Solution

```
blackballs <- rep('black', 30) # create blackballs
redballs <- rep('red', 70) # create redballs
urn <- c(blackballs, redballs) # create the urn
```

```
mydraws <- NULL
for(i in 1:12){
  mydraw <- sample(urn, 1)
  print(mydraw)
  mydraws <- c(mydraws, mydraw)
}
```

```
## [1] "red"
## [1] "black"
## [1] "red"
## [1] "red"
## [1] "black"
```

# Functions

---

## Reminder: what is a function?

Often you will want to reuse the same routine. It is then useful to create your own function.

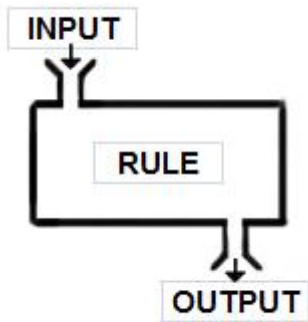
Remember what a function is? A function is basically a machine that eats an input and spits out an output. For example, the function  $f$  defined as

$$f(x) = x^2$$

“eats”  $x$  and “spits out”  $x^2$ .

## Reminder: what is a function?

It's the same in programming, and R is no exception



## Creating your own Functions

E.g., Create a function that prints "I am a hungry function"

*# Create the function, call it ICanPrint*

```
ICanPrint <- function(){  
  "I am a hungry function!"  
}
```

*# Use the function*

```
ICanPrint()
```

```
## [1] "I am a hungry function!"
```

## Creating your own Functions

A slightly more complex example, but still only uses things we already know: 1. generate a sequence of n random numbers drawn from a normal distribution; 2. calculate its mean; 3. find its maximum

*# Create the function*

```
ICanPrintAndMore <- function(){  
  x <- rnorm(100)  
  print(mean(x))  
  print(max(x))  
}  
ICanPrintAndMore()
```

```
## [1] 0.02585902
```

```
## [1] 2.261333
```

PS: You need 'print', because R does not return the results of

## Using arguments in your functions

Often you will want to pass an argument to your function. For example, instead of generating a random sequence of numbers, you want to ask your function to calculate the mean and max of a given sequence of numbers

```
#  
ITakeArguments <- function(x){  
  print(mean(x))  
  print(max(x))  
}  
ITakeArguments(x = 1:10)
```

```
## [1] 5.5
```

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

Exercise: write a function that will return the sum of any two numbers

Type ?function. For example:

```
?c
```

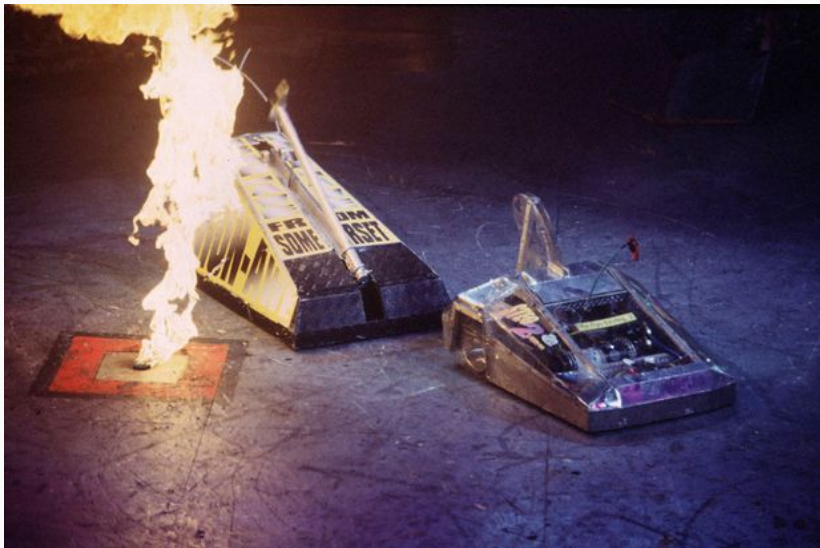
```
?mean
```



# Rock paper scissors

---

## Rock paper scissors



# Rock paper scissors

```
rps <- c('rock', 'paper', 'scissors')

play <- function(mychoice){
  winner <- NULL # initialize the variable
  robotChoice1 <- sample(rps, size = 1) # Robot 1 picks a move (randomly)
  robotChoice2 <- sample(rps, size = 1) # Robot 2 picks a move (randomly)

  # below we define the outcome depending on each of the cases.
  # 0 refers to a draw, otherwise the number is associated with the respective robot
  if(robotChoice1 == 'rock' & robotChoice2 == 'rock') winner <- 0
  if(robotChoice1 == 'rock' & robotChoice2 == 'paper') winner <- 2
  if(robotChoice1 == 'rock' & robotChoice2 == 'scissors') winner <- 1

  if(robotChoice1 == 'paper' & robotChoice2 == 'rock') winner <- 1
  if(robotChoice1 == 'paper' & robotChoice2 == 'paper') winner <- 0
  if(robotChoice1 == 'paper' & robotChoice2 == 'scissors') winner <- 2

  if(robotChoice1 == 'scissors' & robotChoice2 == 'rock') winner <- 2
  if(robotChoice1 == 'scissors' & robotChoice2 == 'paper') winner <- 1
  if(robotChoice1 == 'scissors' & robotChoice2 == 'scissors') winner <- 0

  # we've covered all possible cases,
  # so it's time to declare the winner of this round
  return(winner) # sends this result as the output of the function
}
```

## Let's play!

We're almost ready to play. We just need to 1. create a scoreboard;  
2. repeat the game 10 times and store the winner into the scoreboard every time; and display the result:

```
scoreboard <- NULL # Initialize the object
for(i in 1:10){
  winner <- play()
  scoreboard <- c(scoreboard, winner)
}
```

```
scoreboard
```

```
## [1] 2 2 1 0 0 2 1 1 2 1
```

and so the winner is...

```
## [1] "...Robot 2 !"
```

# R Cheat sheet (for later reference)

## Base R Cheat Sheet

### Getting Help

#### Accessing the help files

**?mean**

Get help of a particular function.

**help.search('weighted mean')**

Search the help files for a word or phrase.

**help(package = 'dplyr')**

Find help for a package.

#### More about an object

**str(iris)**

Get a summary of an object's structure.

**class(iris)**

Find the class an object belongs to.

### Using Packages

**install.packages('dplyr')**

Download and install a package from CRAN.

**library(dplyr)**

Load the package into the session, making all its functions available to use.

**dplyr::select**

Use a particular function from a package.

**data(iris)**

Load a built-in dataset into the environment.

### Working Directory

**getwd()**

Find the current working directory (where inputs are found and outputs are sent).

**setwd('C://file/path')**

Change the current working directory.

Use projects in RStudio to set the working directory to the folder you are working in.

### Vectors

#### Creating Vectors

<code>c(2, 4, 6)</code>	<code>2 4 6</code>	Join elements into a vector
<code>2:6</code>	<code>2 3 4 5 6</code>	An integer sequence
<code>seq(2, 3, by=0.5)</code>	<code>2.0 2.5 3.0</code>	A complex sequence
<code>rep(1:2, times=3)</code>	<code>1 2 1 2 1 2</code>	Repeat a vector
<code>rep(1:2, each=3)</code>	<code>1 1 1 2 2 2</code>	Repeat elements of a vector

#### Vector Functions

<b>sort(x)</b>	<b>rev(x)</b>
Return x sorted.	Return x reversed.
<b>table(x)</b>	<b>unique(x)</b>
See counts of values.	See unique values.

#### Selecting Vector Elements

##### By Position

<code>x[4]</code>	The fourth element.
<code>x[-4]</code>	All but the fourth.
<code>x[2:4]</code>	Elements two to four.
<code>x[-(2:4)]</code>	All elements except two to four.
<code>x[c(1, 5)]</code>	Elements one and five.

##### By Value

<code>x[x == 10]</code>	Elements which are equal to 10.
<code>x[x &lt; 0]</code>	All elements less than zero.
<code>x[x %in% c(1, 2, 5)]</code>	Elements in the set 1, 2, 5.

##### Named Vectors

<code>x['apple']</code>	Element with name 'apple'.
-------------------------	----------------------------

### Programming

#### For Loop

```
for (variable in sequence){  
  Do something  
}
```

##### Example

```
for (i in 1:4){  
  j <- i + 10  
  print(j)  
}
```

#### While Loop

```
while (condition){  
  Do something  
}
```

##### Example

```
while (i < 5){  
  print(i)  
  i <- i + 1  
}
```

#### If Statements

```
if (condition){  
  Do something  
} else {  
  Do something different  
}
```

##### Example

```
if (i > 3){  
  print('Yes')  
} else {  
  print('No')  
}
```

#### Functions

```
function_name <- function(var){  
  Do something  
  return(new_variable)  
}
```

##### Example

```
square <- function(x){  
  squared <- x*x  
  return(squared)  
}
```

### Reading and Writing Data

Also see the **readr** package.

Input	Output	Description
<code>df &lt;- read.table('file.txt')</code>	<code>write.table(df, 'file.txt')</code>	Read and write a delimited text file.
<code>df &lt;- read.csv('file.csv')</code>	<code>write.csv(df, 'file.csv')</code>	Read and write a comma separated value file. This is a special case of read.table/write.table.
<code>load('file.Rdata')</code>	<code>save(df, file = 'file.Rdata')</code>	Read and write an R data file, a file type special for R.

Conditions	a == b	Are equal	a > b	Greater than	a >= b	Greater than or equal to	is.na(a)	Is missing
	a != b	Not equal	a < b	Less than	a <= b	Less than or equal to	is.null(a)	Is null

# R Cheat sheet (for later reference)

## Types

Converting between common data types in R. Can always go from a higher value in the table to a lower value.

<b>as.logical</b>	TRUE, FALSE, TRUE	Boolean values (TRUE or FALSE).
<b>as.numeric</b>	1, 0, 1	Integers or floating point numbers.
<b>as.character</b>	'1', '0', '1'	Character strings. Generally preferred to factors.
<b>as.factor</b>	'1', '0', '1', levels: '1', '0'	Character strings with preset levels. Needed for some statistical models.

## Maths Functions

<b>log(x)</b>	Natural log.	<b>sum(x)</b>	Sum.
<b>exp(x)</b>	Exponential.	<b>mean(x)</b>	Mean.
<b>max(x)</b>	Largest element.	<b>median(x)</b>	Median.
<b>min(x)</b>	Smallest element.	<b>quantile(x)</b>	Percentage quantiles.
<b>round(x, n)</b>	Round to n decimal places.	<b>rank(x)</b>	Rank of elements.
<b>signif(x, n)</b>	Round to n significant figures.	<b>var(x)</b>	The variance.
<b>cor(x, y)</b>	Correlation.	<b>sd(x)</b>	The standard deviation.

## Variable Assignment

```
> a <- 'apple'
> a
[1] 'apple'
```




## The Environment

<b>ls()</b>	List all variables in the environment.
<b>rm(x)</b>	Remove x from the environment.
<b>rm(list = ls())</b>	Remove all variables from the environment.

You can use the environment panel in RStudio to browse variables in your environment.

## Matrices

```
m <- matrix(x, nrow = 3, ncol = 3)
Create a matrix from x.
```

 <b>m[2, ]</b> - Select a row	<b>t(m)</b> Transpose $m^{t \times n}$
 <b>m[, 1]</b> - Select a column	<b>solve(m, n)</b> Matrix Multiplication Find $x$ in: $m \cdot x = n$
 <b>m[2, 3]</b> - Select an element	

## Lists

```
l <- list(x = 1:5, y = c('a', 'b'))
A list is a collection of elements which can be of different types.
```

<b>l[[2]]</b> Second element of l	<b>l[[1]]</b> New list with only the first element.	<b>l\$x</b> Element named x.	<b>l['y']</b> New list with only element named y.
--------------------------------------	--	---------------------------------	--


Also see the **dplyr** package.

## Data Frames



```
df <- data.frame(x = 1:3, y = c('a', 'b', 'c'))
A special case of a list where all elements are the same length.
```

x	y
1	a
2	b
3	c

### Matrix subsetting

<b>df[, 2]</b>	
<b>df[, 1]</b>	
<b>df[2, 2]</b>	

### List subsetting

<b>df\$x</b>		<b>df[[2]]</b>	
Understanding a data frame			
<b>View(df)</b>	See the full data frame.		
<b>head(df)</b>	See the first 6 rows.		

**nrow(df)**  
Number of rows.

**ncol(df)**  
Number of columns.

**dim(df)**  
Number of columns and rows.

**cbind** - Bind columns.



**rbind** - Bind rows.



## Strings

Also see the **stringr** package.

<b>paste(x, y, sep = ' ')</b>	Join multiple vectors together.
<b>paste(x, collapse = ' ')</b>	Join elements of a vector together.
<b>grep(pattern, x)</b>	Find regular expression matches in x.
<b>gsub(pattern, replace, x)</b>	Replace matches in x with a string.
<b>toupper(x)</b>	Convert to uppercase.
<b>tolower(x)</b>	Convert to lowercase.
<b>nchar(x)</b>	Number of characters in a string.

## Factors

**factor(x)**  
Turn a vector into a factor. Can set the levels of the factor and the order.

**cut(x, breaks = 4)**  
Turn a numeric vector into a factor by 'cutting' into sections.

## Statistics

<b>lm(y ~ x, data=df)</b> Linear model.	<b>t.test(x, y)</b> Perform a t-test for difference between means.	<b>prop.test</b> Test for a difference between proportions.
<b>glm(y ~ x, data=df)</b> Generalised linear model.	<b>pairwise.t.test</b> Perform a t-test for paired data.	<b>aov</b> Analysis of variance.
<b>summary</b> Get more detailed information out a model.		

## Distributions

	Random Variables	Density Function	Cumulative Distribution	Quantile
Normal	<b>rnorm</b>	<b>dnorm</b>	<b>pnorm</b>	<b>qnorm</b>
Poisson	<b>rpois</b>	<b>dpois</b>	<b>ppois</b>	<b>qpois</b>
Binomial	<b>rbinom</b>	<b>dbinom</b>	<b>pbinom</b>	<b>qbinom</b>
Uniform	<b>runif</b>	<b>dunif</b>	<b>punif</b>	<b>qunif</b>

## Plotting

Also see the **ggplot2** package.

	<b>plot(x)</b> Values of x in order.		<b>plot(x, y)</b> Values of x against y.		<b>hist(x)</b> Histogram of x.
---	---	---	---	---	-----------------------------------

## Dates

See the **lubridate** package.

**A suggestion: easy way to write  
beautiful assignments (and much  
more)**

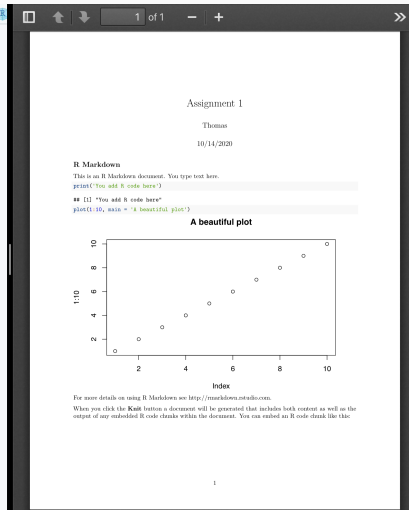
---

# Rmarkdown

```
1 ---
2 title: "Assignment 1"
3 author: "Thomas"
4 date: "10/14/2020"
5 output: pdf_document
6 ---
7
8 ## R Markdown
9
10 This is an R Markdown document. You type text here.
11
12 ```{r}
13 print('You add R code here')
14 plot(1:10, main = 'A beautiful plot')
15 ```
16
17 For more details on using R Markdown see
18 <http://rmarkdown.rstudio.com>.
19
20 When you click the **Knit** button a document will be
21 generated that includes both content as well as the
22 output of any embedded R code chunks within the
23 document. You can embed an R code chunk like this:
```

4:19 Assignment 1 R Markdown

Console





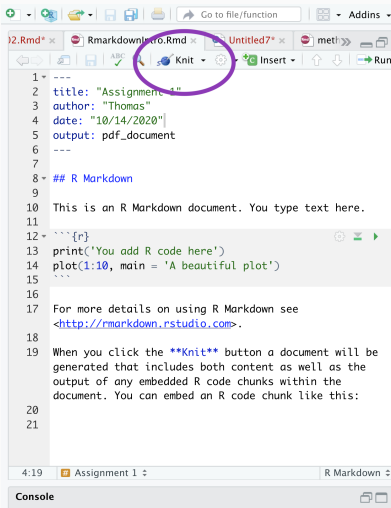
# How to do it?

in Rstudio:

File > New File > R Markdown...

Choose a title, click on PDF, then your markdown file will be created. Then you just need to click on “knit”

# How to do it?



```
1 ---
2 title: "Assignment 1"
3 author: "Thomas"
4 date: "10/14/2020"
5 output: pdf_document
6 ---
7
8 ## R Markdown
9
10 This is an R Markdown document. You type text here.
11
12 ```{r}
13 print('You add R code here')
14 plot(1:10, main = 'A beautiful plot')
15 ```
16
17 For more details on using R Markdown see
18 <http://rmarkdown.rstudio.com>.
19
20 When you click the **Knit** button a document will be
21 generated that includes both content as well as the
22 output of any embedded R code chunks within the
23 document. You can embed an R code chunk like this:
```

4:19 Assignment 1 R Markdown

Console

