A First Look at R

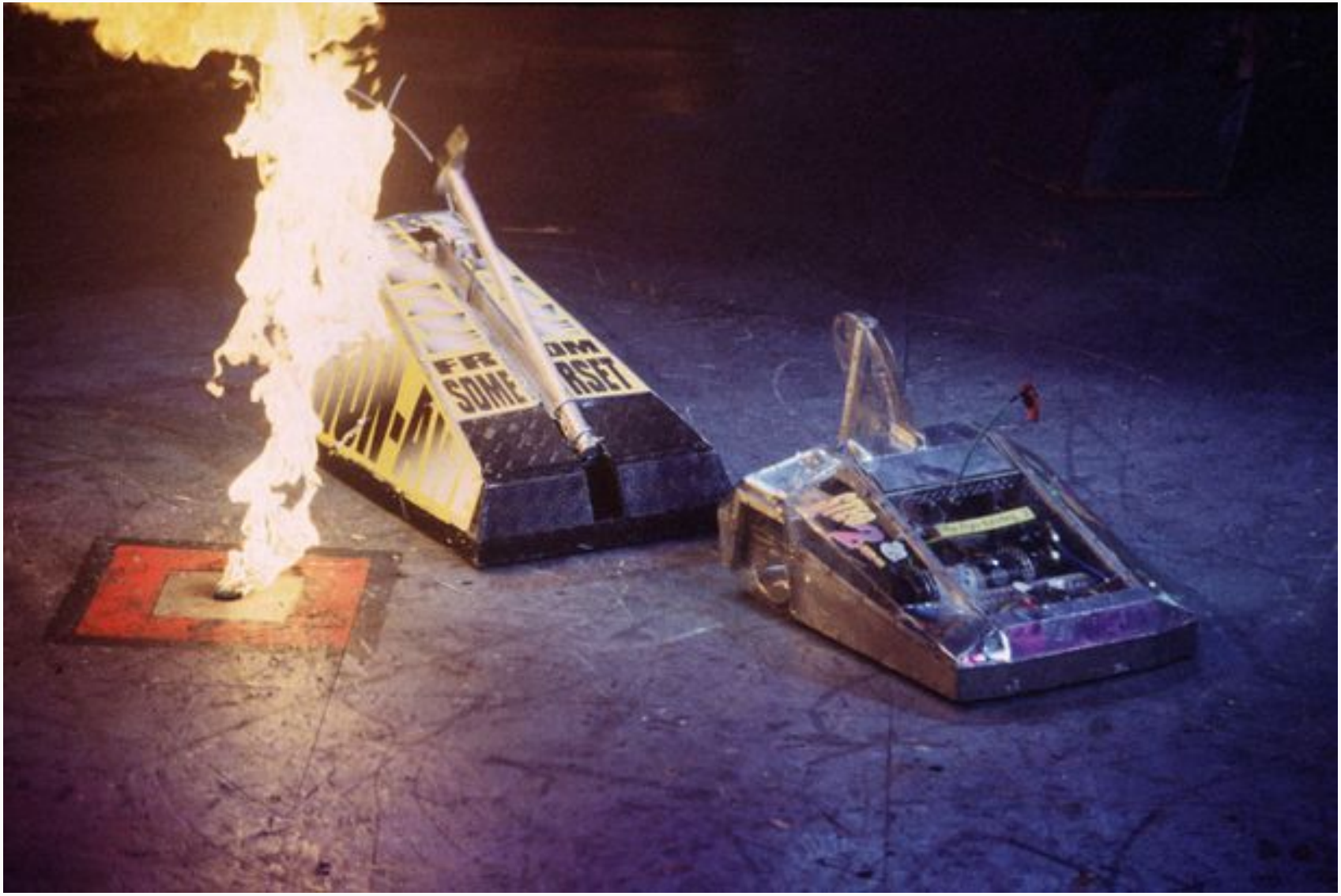
Research Methods for Political Science

Thomas Chadefaux

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
```

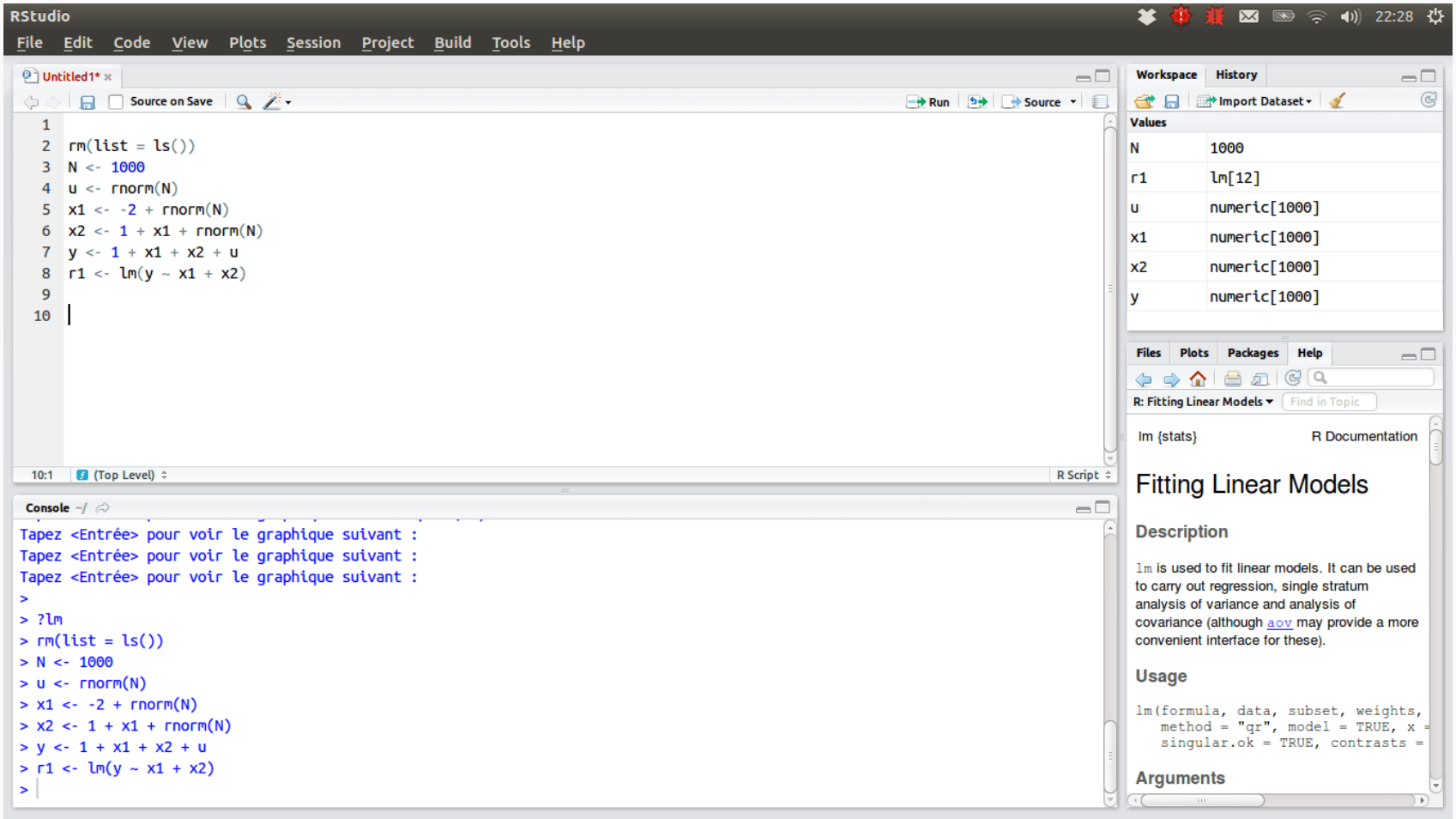
and the winner is...

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

Software

What you absolutely need to do (reminder)

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



What you absolutely need to do (reminder)

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

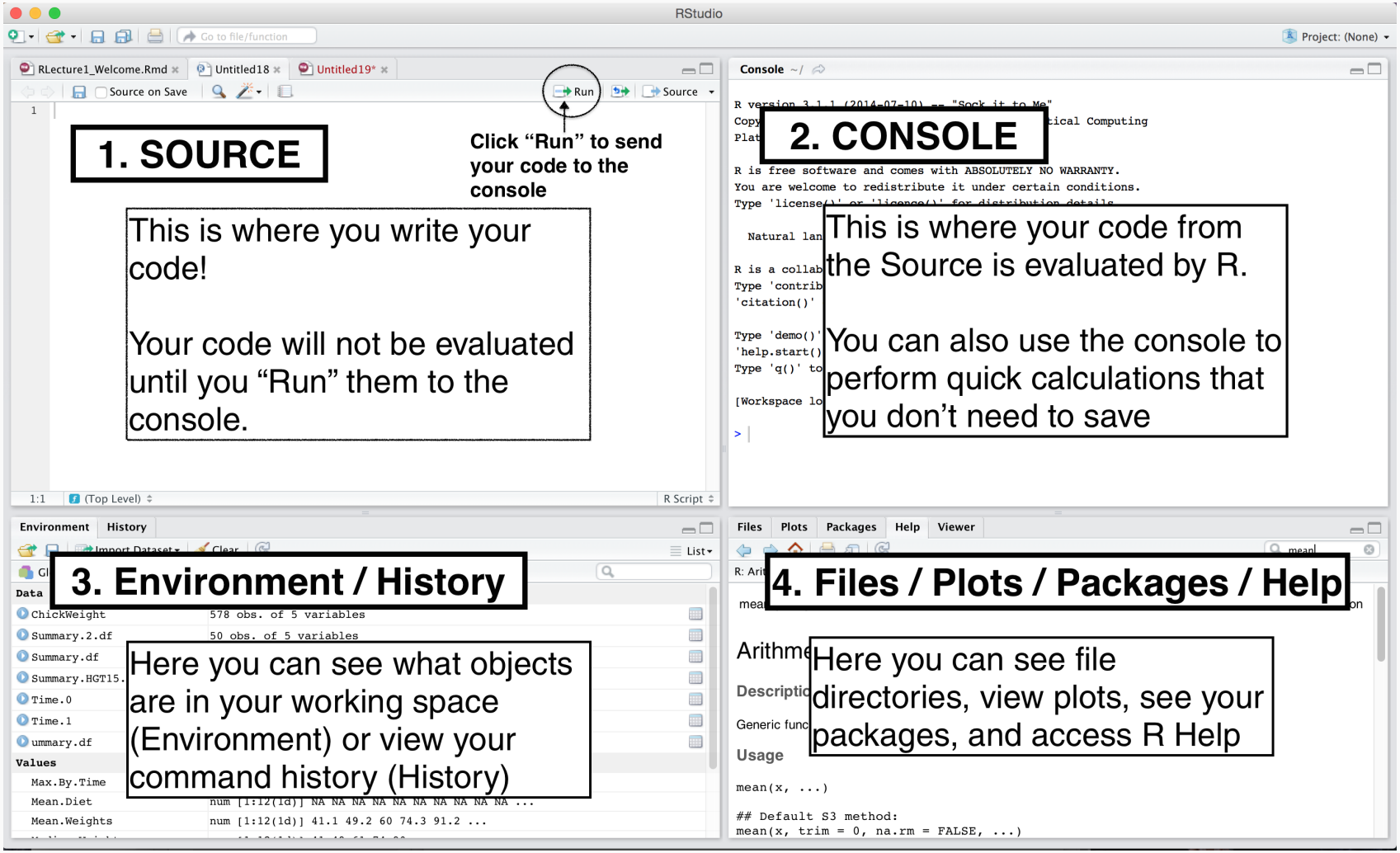
You can even run R directly from your browser, but I really don't recommend it beyond just trying a few things (https://rextester.com/l/r_online_compiler)

Why R?

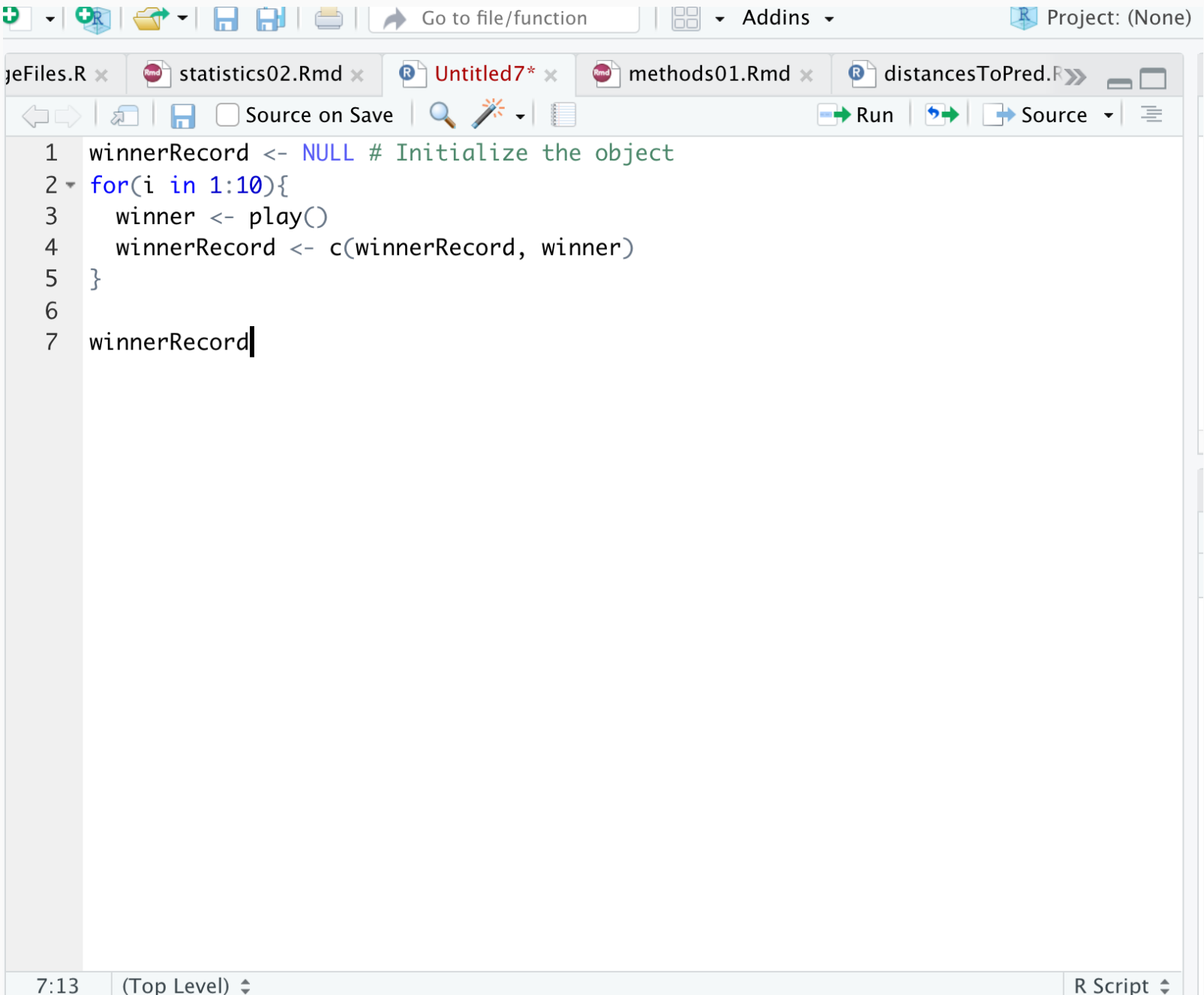
- Free
- Huge community
- Very popular and rising
- Extremely powerful
- Transparent, reproducible, shareable

The Rstudio environment

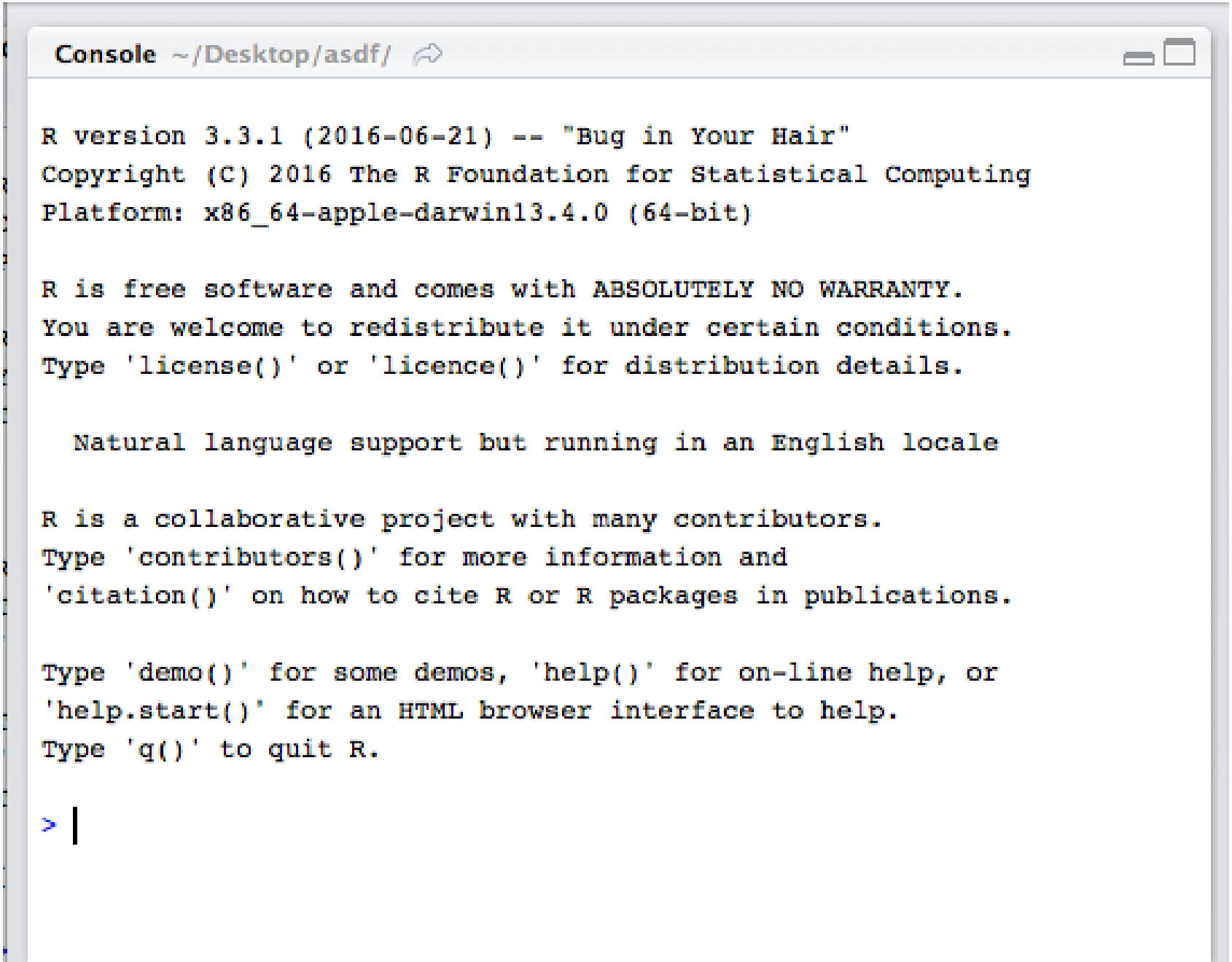
The four panes



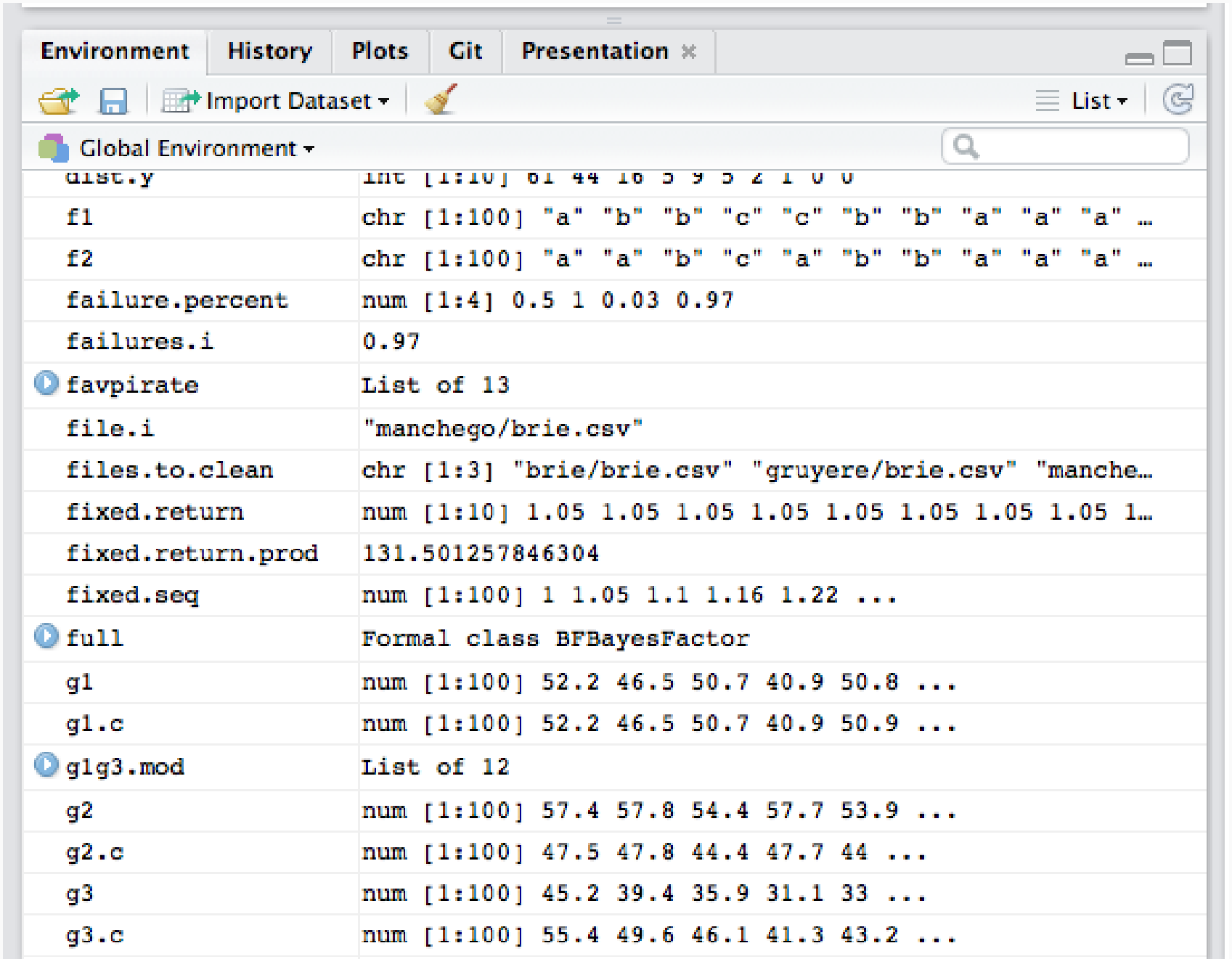
The Source pane



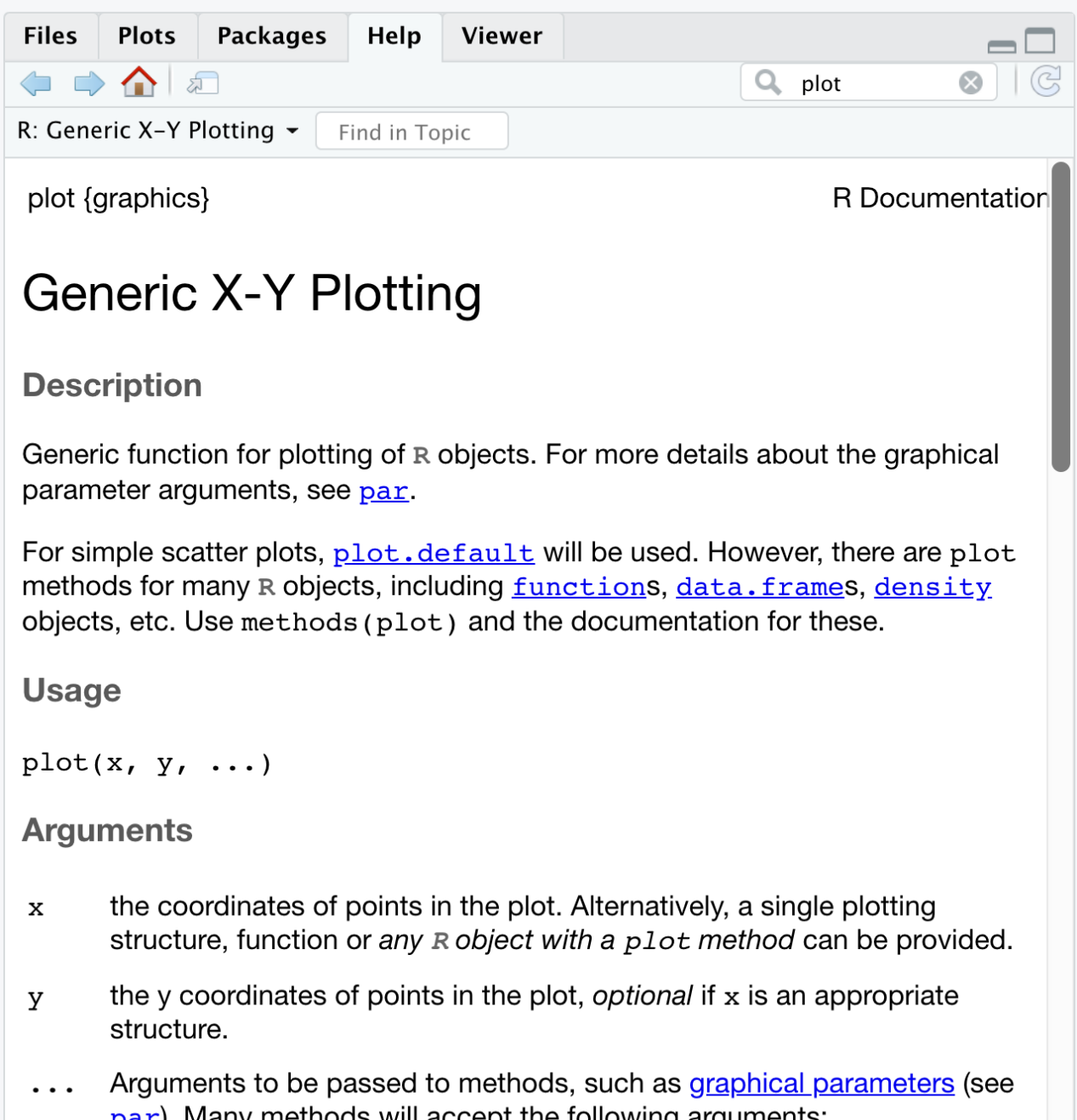
The console pan



The Environment pane



Files/Plots/Packages/Help Pane



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 assig
```

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

```
Iamanobject <- 3
Iamanobjecttoo <- '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
```

Vectors

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

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

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

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

Vectors

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 17.666
```

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.
```

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 len
## 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 it 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
```

Exercise

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

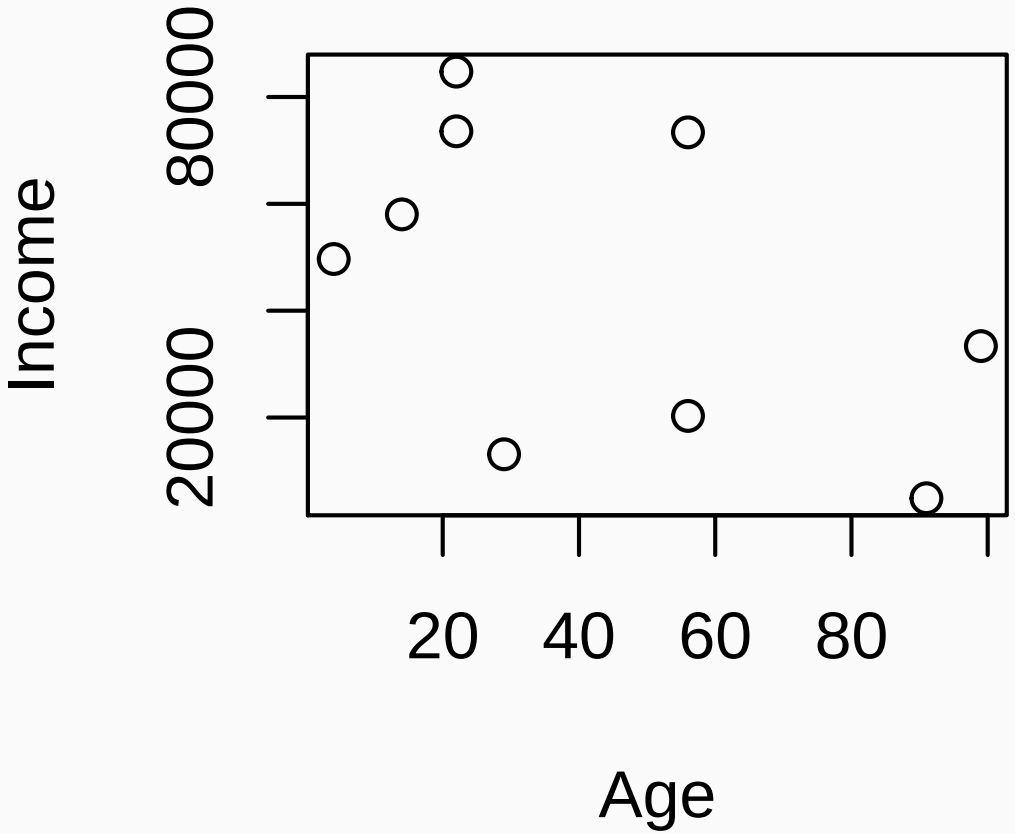
```
with(localdat, mean(income[gender == 'female' &
                           age < 50]))
```

```
## [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

```r
# 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"
```

Extension

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.

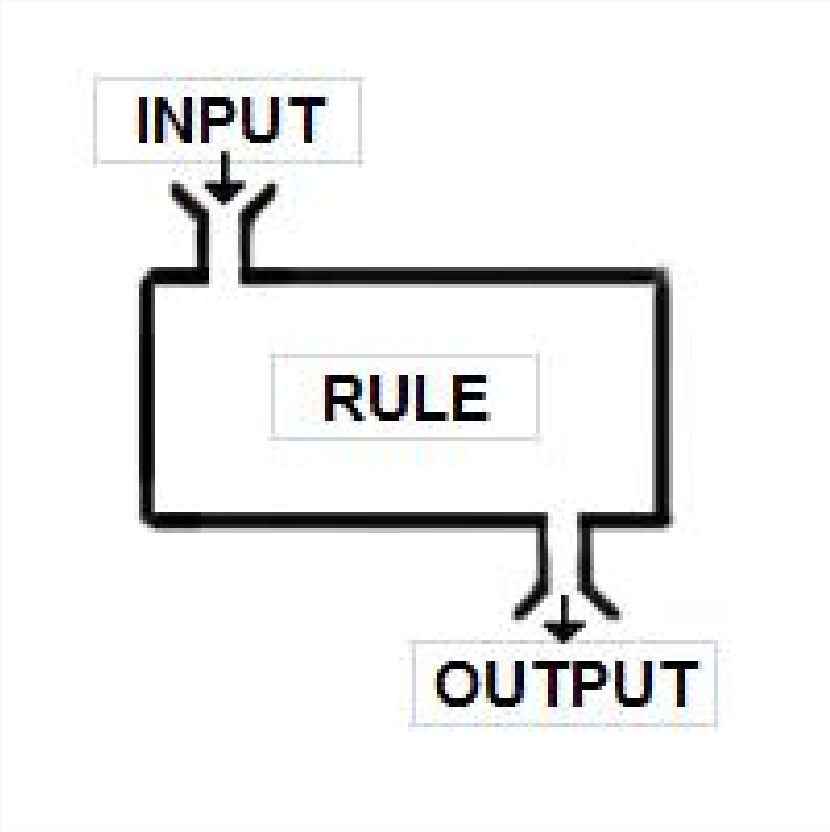
Remember what a function is? A function is basically a machine that eats an input and spits out an output. For example, the function f defined as

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

"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

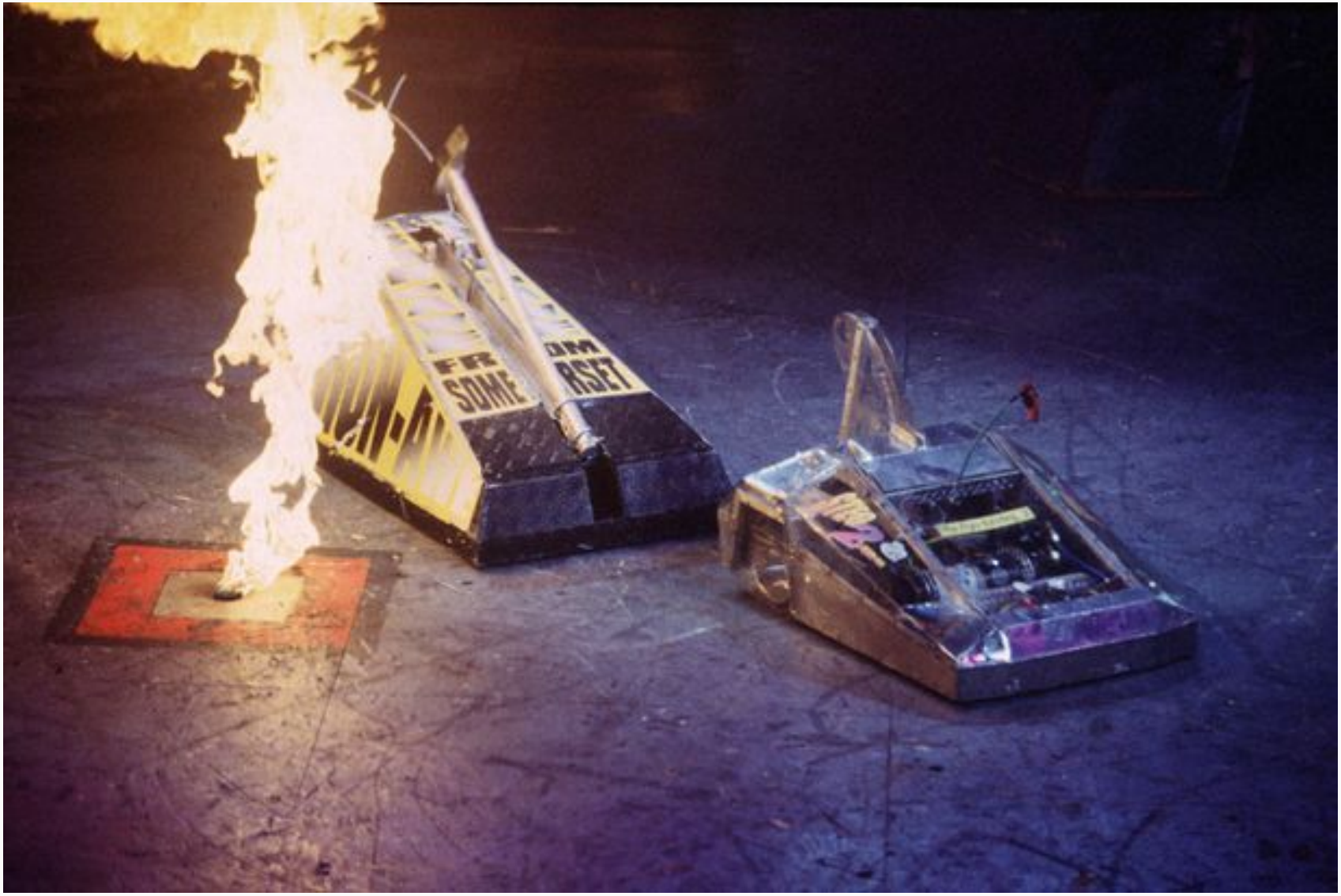
Getting help

Type ?function. For example:

```
?c
?mean
```

Rock paper scissors

Rock paper scissors



Rock paper scissors

```r
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

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

Vector Functions

sort(x)
Return x sorted.
rev(x)
Return x reversed.
table(x)
See counts of values.
unique(x)
See unique values.

Selecting Vector Elements

By Position

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

By Value

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

Named Vectors

x['apple']	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	Ouput	Description
df <- read.table('file.txt')	write.table(df, 'file.txt')	Read and write a delimited text file.
df <- read.csv('file.csv')	write.csv(df, 'file.csv')	Read and write a comma separated value file. This is a special case of read.table/ write.table.
load('file.RData')	save(df, file = 'file.Rdata')	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.

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

Maths Functions

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

Variable Assignment

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

The Environment

`ls()`	List all variables in the environment.
`rm(x)`	Remove x from the environment.
`rm(list = ls())`	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.


`m[2, ]` - Select a row


`m[ , 1]` - Select a column


`m[2, 3]` - Select an element

`t(m)`
Transpose
`m %*% n`
Matrix Multiplication
`solve(m, n)`
Find x in: m * x = n

Lists

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

`l[[2]]`	`l[1]`	`l$x`	`l['y']`
Second element of l.	New list with only the first element.	Element named x.	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

List subsetting

`df$x` 
`df[[2]]` 

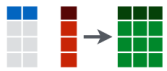
Understanding a data frame

`View(df)`	See the full data frame.
`head(df)`	See the first 6 rows.

Matrix subsetting

`df[ , 2]` 
`df[2, ]` 
`df[2, 2]` 

`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.

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

`lm(y ~ x, data=df)`
Linear model.

`glm(y ~ x, data=df)`
Generalised linear model.

`summary`
Get more detailed information out a model.

`t.test(x, y)`
Perform a t-test for difference between means.

`pairwise.t.test`
Perform a t-test for paired data.

`prop.test`
Test for a difference between proportions.

`aov`
Analysis of variance.

Distributions

	Random Variates	Density Function	Cumulative Distribution	Quantile
Normal	`rnorm`	`dnorm`	`pnorm`	`qnorm`
Poisson	`rpois`	`dpois`	`ppois`	`qpois`
Binomial	`rbinom`	`dbinom`	`pbinom`	`qbinom`
Uniform	`runif`	`dunif`	`punif`	`qunif`

Plotting

Also see the **ggplot2** package.


`plot(x)`
Values of x in order.


`plot(x, y)`
Values of x against y.


`hist(x)`
Histogram of x.

Dates

See the **lubridate** package.

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

Rmarkdown

```
---
title: "Assignment 1"
author: "Thomas"
date: "10/14/2020"
output: pdf_document
---

## R Markdown

This is an R Markdown document. You type text here.

```{r}
print('You add R code here')
plot(1:10, main = 'A beautiful plot')
```

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:
```

Assignment 1

Thomas

10/14/2020

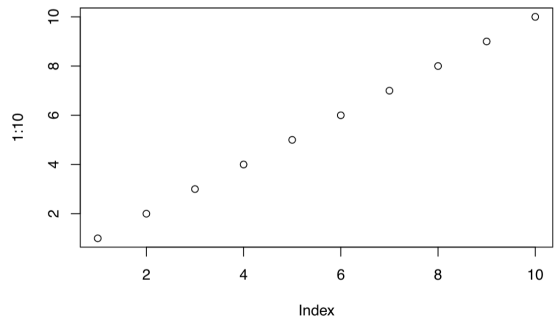
R Markdown

This is an R Markdown document. You type text here.

```
print('You add R code here')
```

```
## [1] "You add R code here"
```

```
plot(1:10, main = 'A beautiful plot')
```



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:

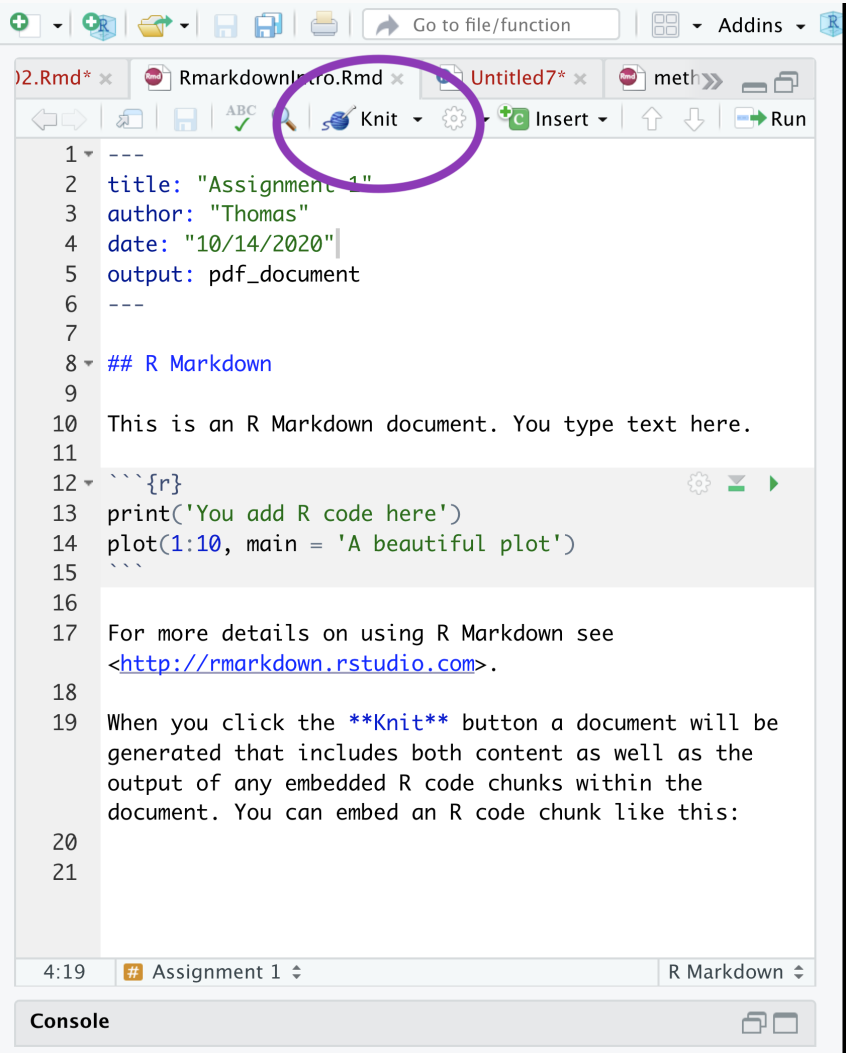
How to do it?

in Rstudio:

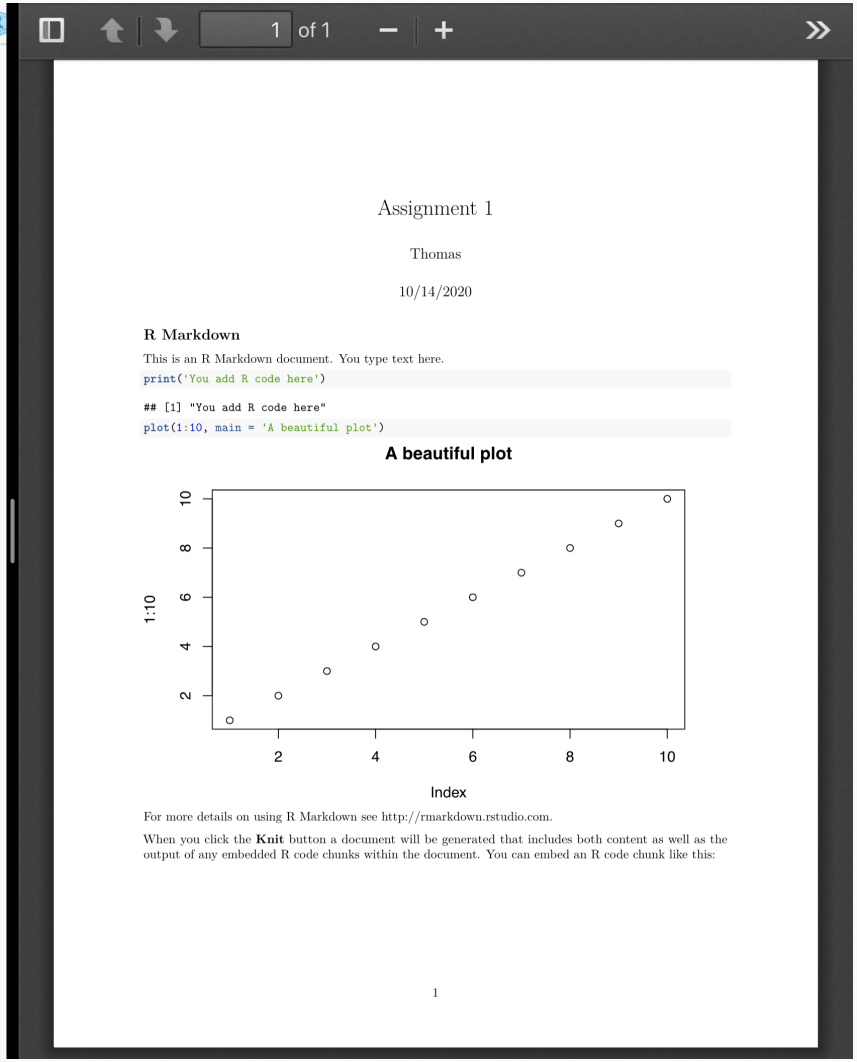
File > New File > R Markdown...

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?


Go to file/function
Addins
Knit
Insert
Run
---
title: "Assignment 1"
author: "Thomas"
date: "10/14/2020"
output: pdf_document

R Markdown
This is an R Markdown document. You type text here.
```{r}
print('You add R code here')
plot(1:10, main = 'A beautiful plot')
```
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:
4:19
Assignment 1
R Markdown
Console


1 of 1
Assignment 1
Thomas
10/14/2020
R Markdown
This is an R Markdown document. You type text here.
print('You add R code here')
[1] "You add R code here"
plot(1:10, main = 'A beautiful plot')
A beautiful plot
1:10
Index
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:
1