Basic R

“Processing large dataset with R”

Instructors

Silvia
BOTTINI

silvia.bottini@univ-cotedazur.fr

Theoretic classes

Giulia
MARCHELLO

giulia.marchello@inria.fr

Practical classes

Syllabus

	Topic	Teacher	Location	Date
1.	Introduction, advanced R and programming basis	SB	On line	02/10
2.	Handling massive data (dplyr)	SB	On line	5/10
3.	Workshop1	GM	room	8/10
4.	Data visualization (ggplot2)	SB	On line	12/10
5.	Workshop2	GM	room	15/10 (13h30)

Syllabus

Topic	Teacher	Location	Date
6. Reporting with R (Rmarkdown & Shiny)	SB	On line	
7. Workshop3	GM	room	
8. Simulation/Parallel computation	SB	On line	
9. workshop4	GM	room	
10. exam	GM	room	

Why R?

Interactivity ✓

Scripting ✓

Open Source ✓

Multi platform ✓

Drawbacks

Not user friendly @ sta

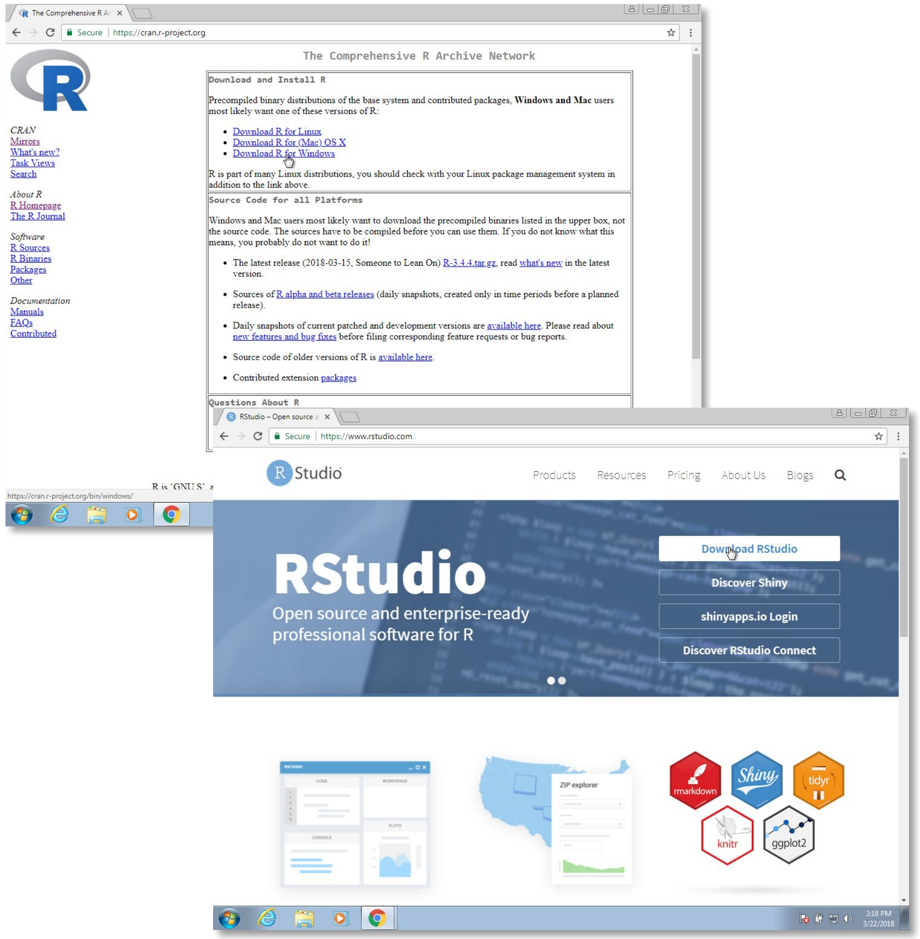
Data preparatio

No easy debugging

Working with larg
datasets is limited by
RAM

Let’s start

1. Download R: Console & RStudio



The Comprehensive R Archive Network

CRAN
Mirrors
What's new?
Task Views
Search

About R
R Homepage
The R Journal

Software
R Sources
R Binaries
Packages
Other

Documentation
Manuals
FAQs
Contributed

Download and Install R

Precompiled binary distributions of the base system and contributed packages, **Windows and Mac** users most likely want one of these versions of R:

- Download R for Linux
- Download R for (Mac) OS X
- Download R for Windows

R is part of many Linux distributions, you should check with your Linux package management system in addition to the link above.

Source Code for all Platforms

Windows and Mac users most likely want to download the precompiled binaries listed in the upper box, not the source code. The sources have to be compiled before you can use them. If you do not know what this means, you probably do not want to do it!

- The latest release (2018-03-15, Someone to Lean On) R-3.4.4.tar.gz, read what's new in the latest version.
- Sources of R alpha and beta releases (daily snapshots, created only in time periods before a planned release).
- Daily snapshots of current patched and development versions are available here. Please read about new features and bug fixes before filing corresponding feature requests or bug reports.
- Source code of older versions of R is available here.
- Contributed extension packages

Questions About R

R is ‘GNU S’

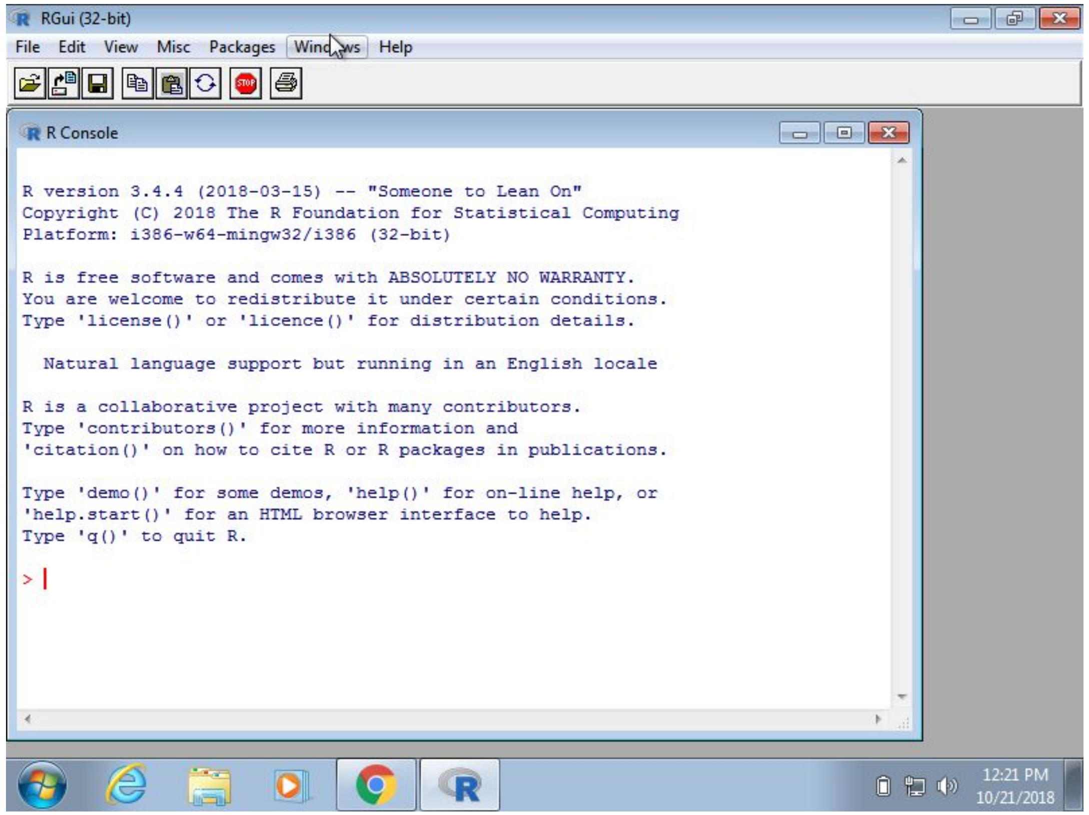
https://cran.r-project.org/bin/windows/

RStudio

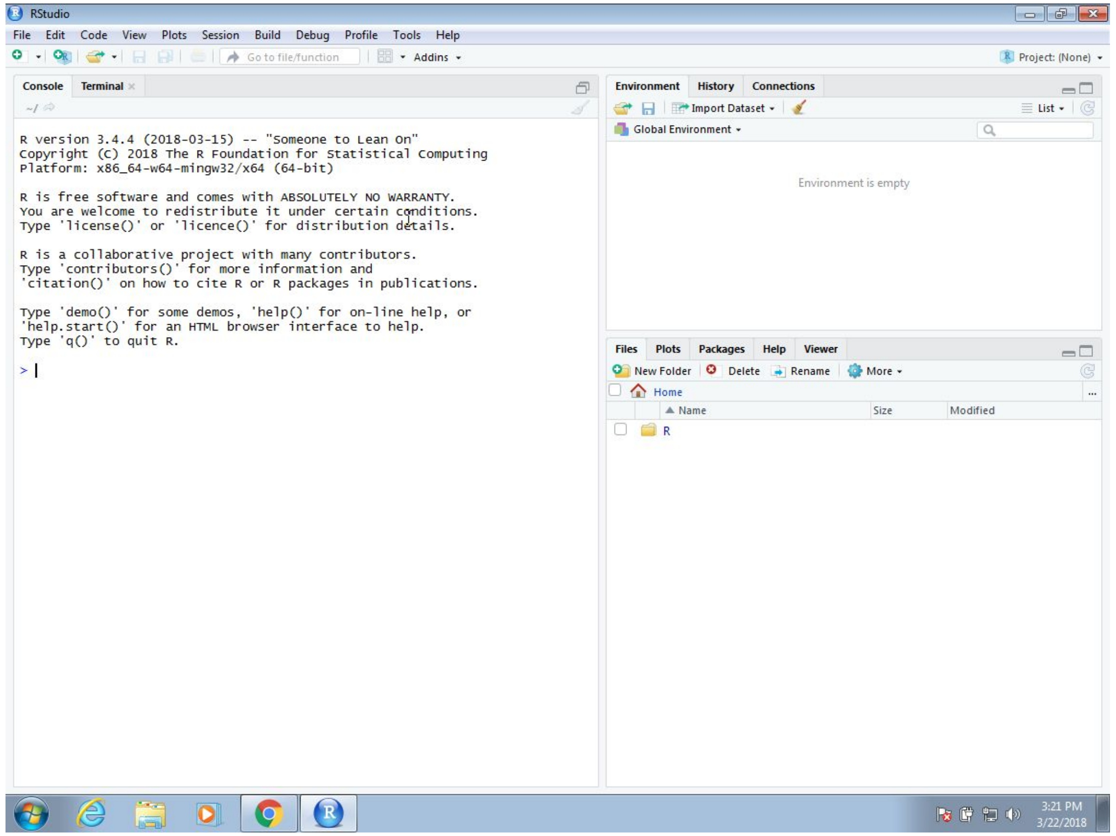
Open source and enterprise-ready professional software for R

Let’s start

1. Download R: Console & Rstudio
2. Open the console



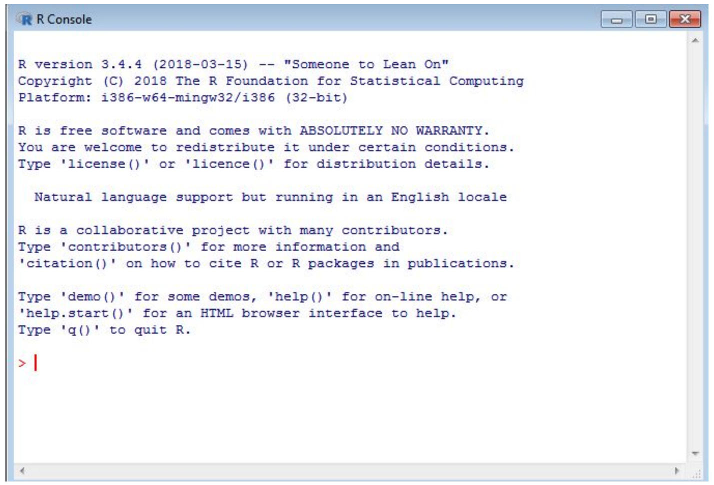
```
R version 3.4.4 (2018-03-15) -- "Someone to Lean On"
Copyright (C) 2018 The R Foundation for Statistical Computing
Platform: i386-w64-mingw32/i386 (32-bit)

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.

> 
```



```
R version 3.4.4 (2018-03-15) -- "Someone to Lean On"
Copyright (C) 2018 The R Foundation for Statistical Computing
Platform: x86_64-w64-mingw32/x64 (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.

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.

> 
```

Let’s start

1. Download R: Console & Rstudio
2. Open the console
3. Make simple calculation

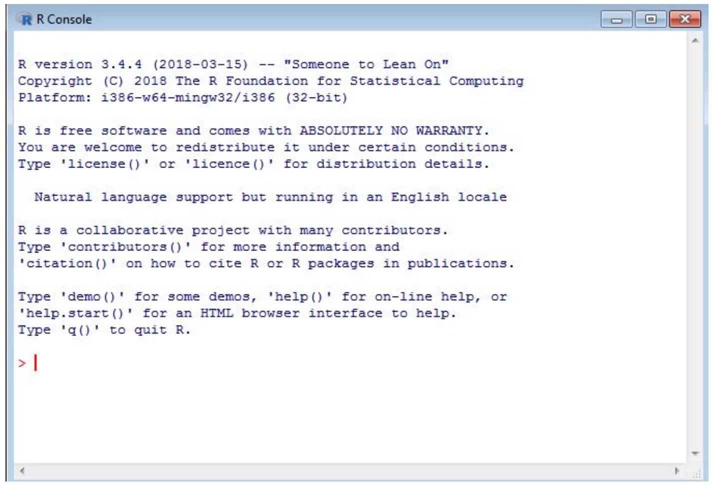


```
R Console

R version 3.4.4 (2018-03-15) -- "Someone to Lean On"
Copyright (C) 2018 The R Foundation for Statistical Computing
Platform: i386-w64-mingw32/i386 (32-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.

> |
```

```
> 2 * 2
? [1] 4
> 0.15 * 19.71
? [1] 2.96
> exp(-2)
? [1] 0.1353353
```

Let’s start

1. Download R: Console & Rstudio
2. Open the console
3. Make simple calculation
4. Make more complex calculation



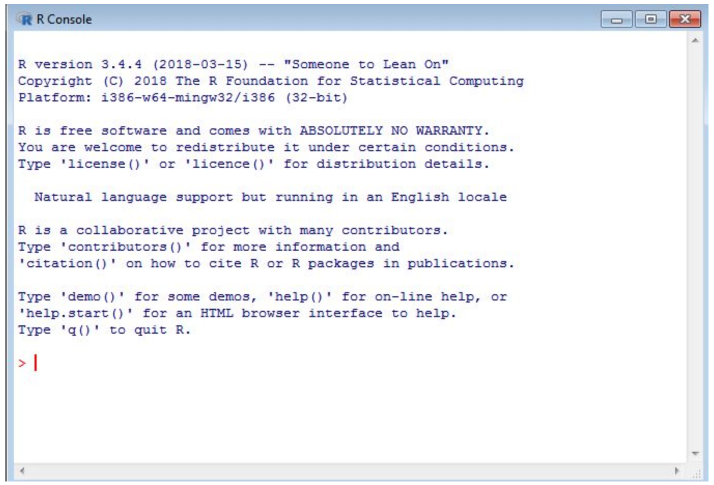
```
R version 3.4.4 (2018-03-15) -- "Someone to Lean On"
Copyright (C) 2018 The R Foundation for Statistical Computing
Platform: i386-w64-mingw32/i386 (32-bit)

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.

> |
```

```
> ?t.test
```

⍰Description: a short description in English of the method implemented in the function
Usage: the way the function should be used, with all input parameters and their default values
Arguments: a detailed explenation of the input parameters
Values: a detailed explenation of the output fields

Let’s start

1. Download R: Console & Rstudio
2. Open the console
3. Make simple calculation
4. Make more complex calculation
5. Install libraries



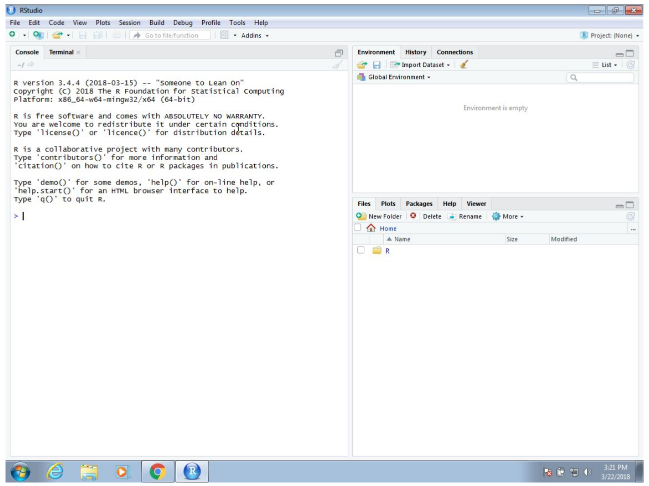
```
R version 3.4.4 (2018-03-15) -- "Someone to Lean On"
Copyright (C) 2018 The R Foundation for Statistical Computing
Platform: i386-w64-mingw32/i386 (32-bit)

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.

> 
```

```
# installing the package
> install.packages("dslabs")
```

⍰ The downloaded binary packages are in
	/var/folders/0t/35l1qfw53xqbdy1ldsnljgmc0000gn/T//Rtmp9XgRWo/downloaded_packages

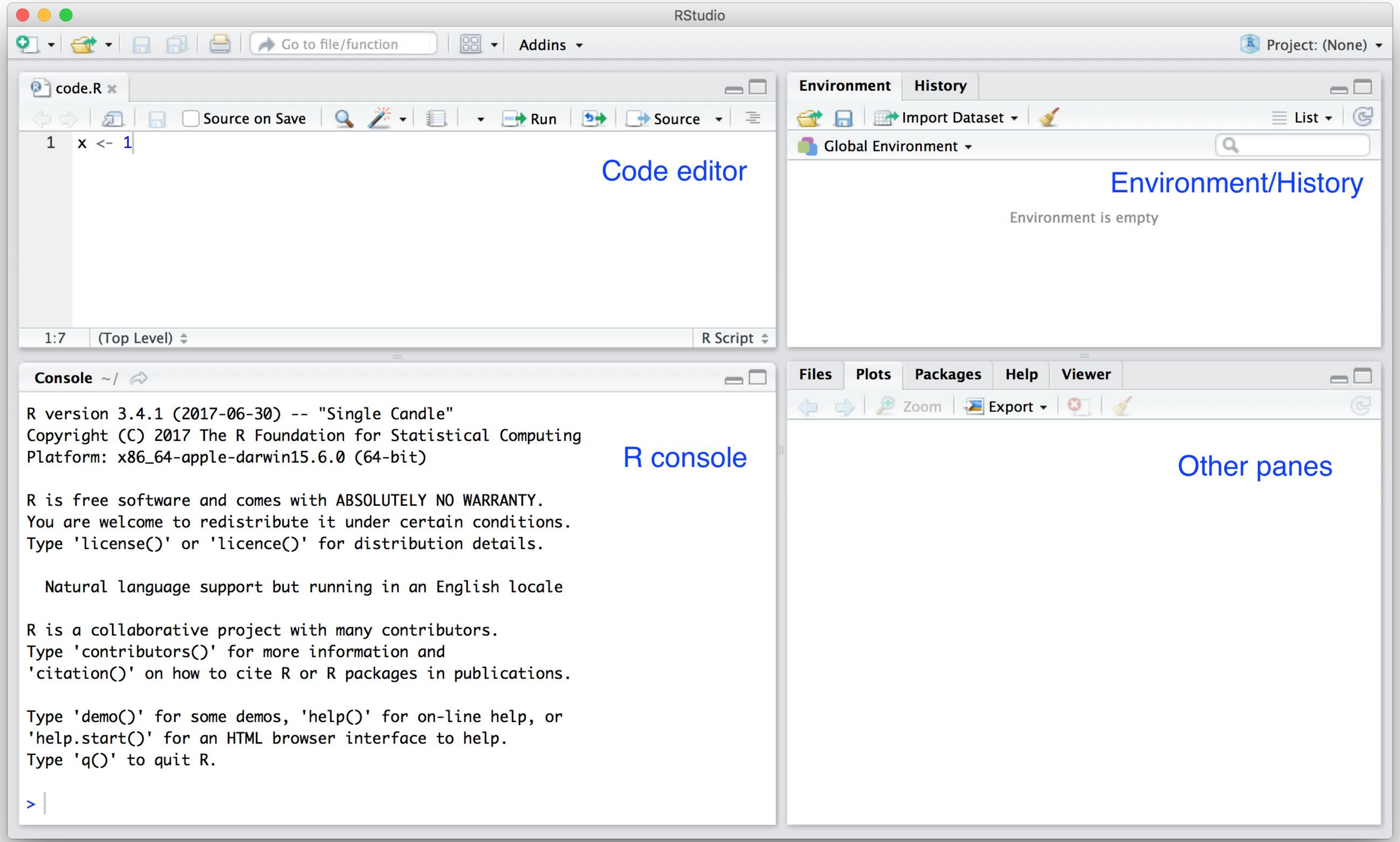
```
# loading the package
> library(dslabs)
```

Let’s do some scripting



```
R version 3.4.4 (2018-03-15) -- "Someone to Lean On"
Copyright (C) 2018 The R Foundation for Statistical Computing
Platform: x86_64-w64-mingw32/x64 (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.

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.

> 
```

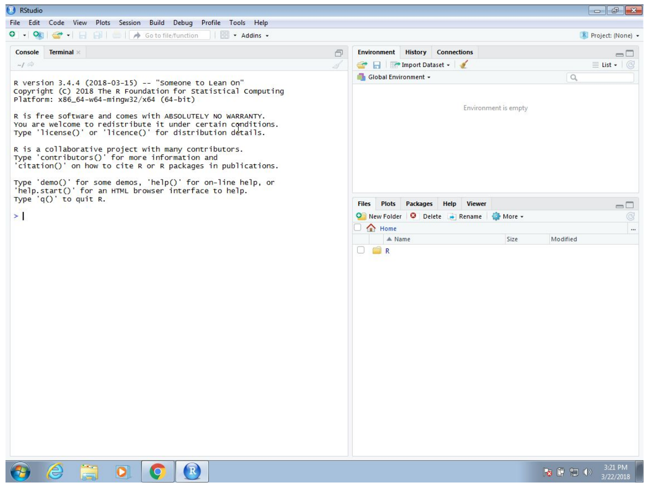
□ New File □ Rscript □



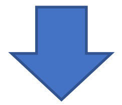
```
x <- 1
```

```
R version 3.4.1 (2017-06-30) -- "Single Candle"
Copyright (C) 2017 The R Foundation for Statistical Computing
Platform: x86_64-apple-darwin15.6.0 (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.

> 
```

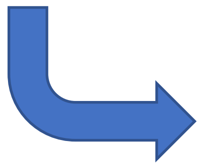
Let’s do some scripting

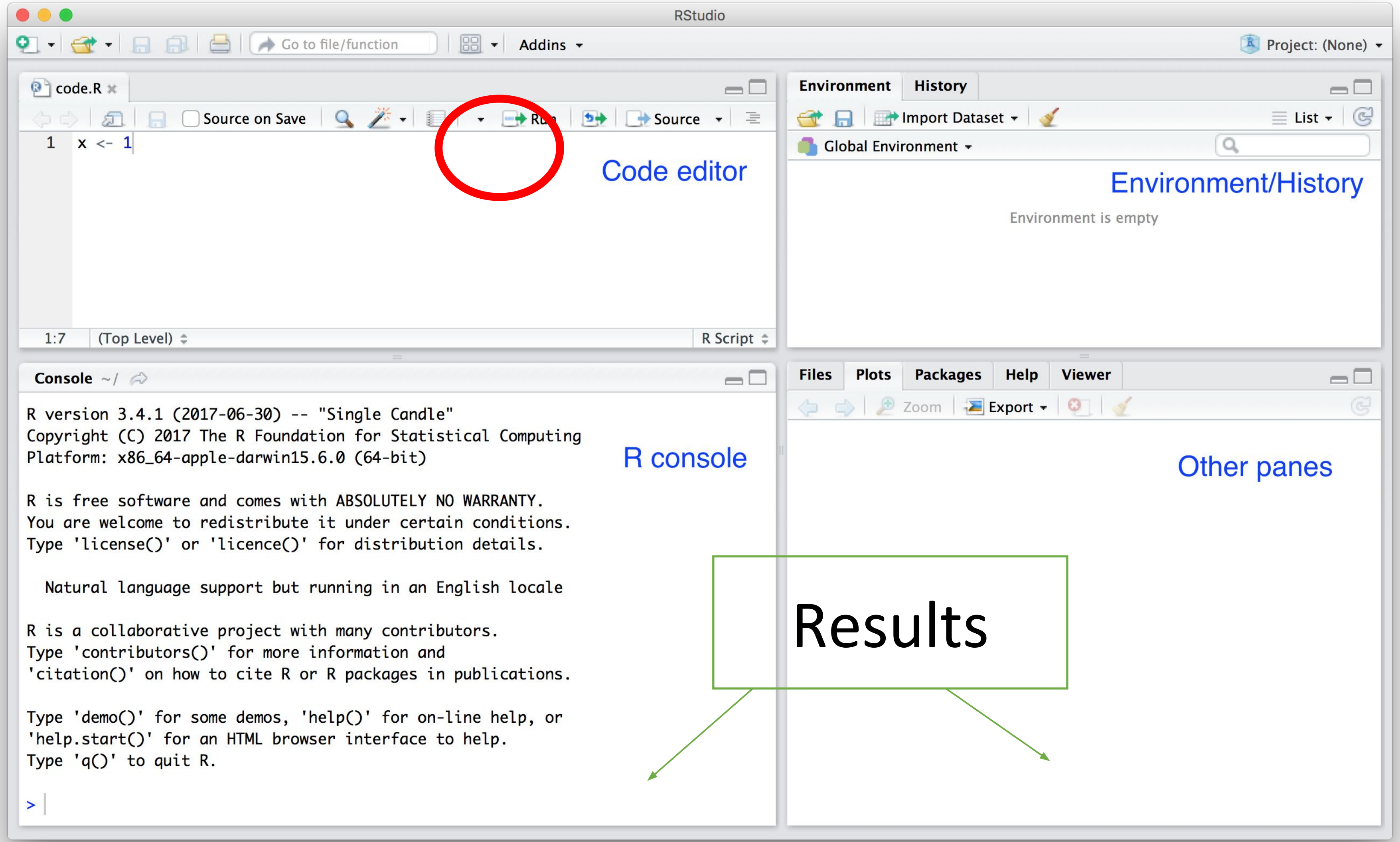


```
R version 3.4.4 (2018-03-15) -- "Someone to Lean On"
Copyright (C) 2018 The R Foundation for Statistical Computing
Platform: x86_64-w64-mingw32/x64 (64-bit)

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.

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.

> |
```



 New File Rscript





```
x <- 1
```

```
R version 3.4.1 (2017-06-30) -- "Single Candle"
Copyright (C) 2017 The R Foundation for Statistical Computing
Platform: x86_64-apple-darwin15.6.0 (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 very basic: variables & functions

```
> a <- 1
> a
 [1] 1
> print a
 [1] 1
```

The very basic: variables & functions

```
> a <- 1
> a
 [1] 1
> print a
 [1] 1
```

```
> log(8)
 [1] 2.08
> help("log")
 > ?log
> args(log)
 function (x, base = exp(1))
```

The very basic: variables & functions

$$ax^2+bx+c=0$$

```
> a <- 1
> b <- 1
> c <- -1
> (-b + sqrt(b^2 - 4*a*c) ) / ( 2*a )
[1] 0.618
> (-b + sqrt(b^2 - 4*a*c) ) / ( 2*a )
[1] -1.62
```

```
## Code to compute solution to quadratic equation of the form ax^2 + bx + c
## define the variables
a <- 3
b <- 2
c <- -1
## now compute the solution
Sol1 <- (-b + sqrt(b^2 - 4*a*c)) / (2*a)
Sol2 <- (-b - sqrt(b^2 - 4*a*c)) / (2*a)
```

Exercise

What is the sum of the first 100 positive integers? The formula for the sum of integers 1 through n is n(n+1)/2. Define n=100 and then use R to compute the sum of 1 through 100 using the formula. What is the sum?

Data types & data frame

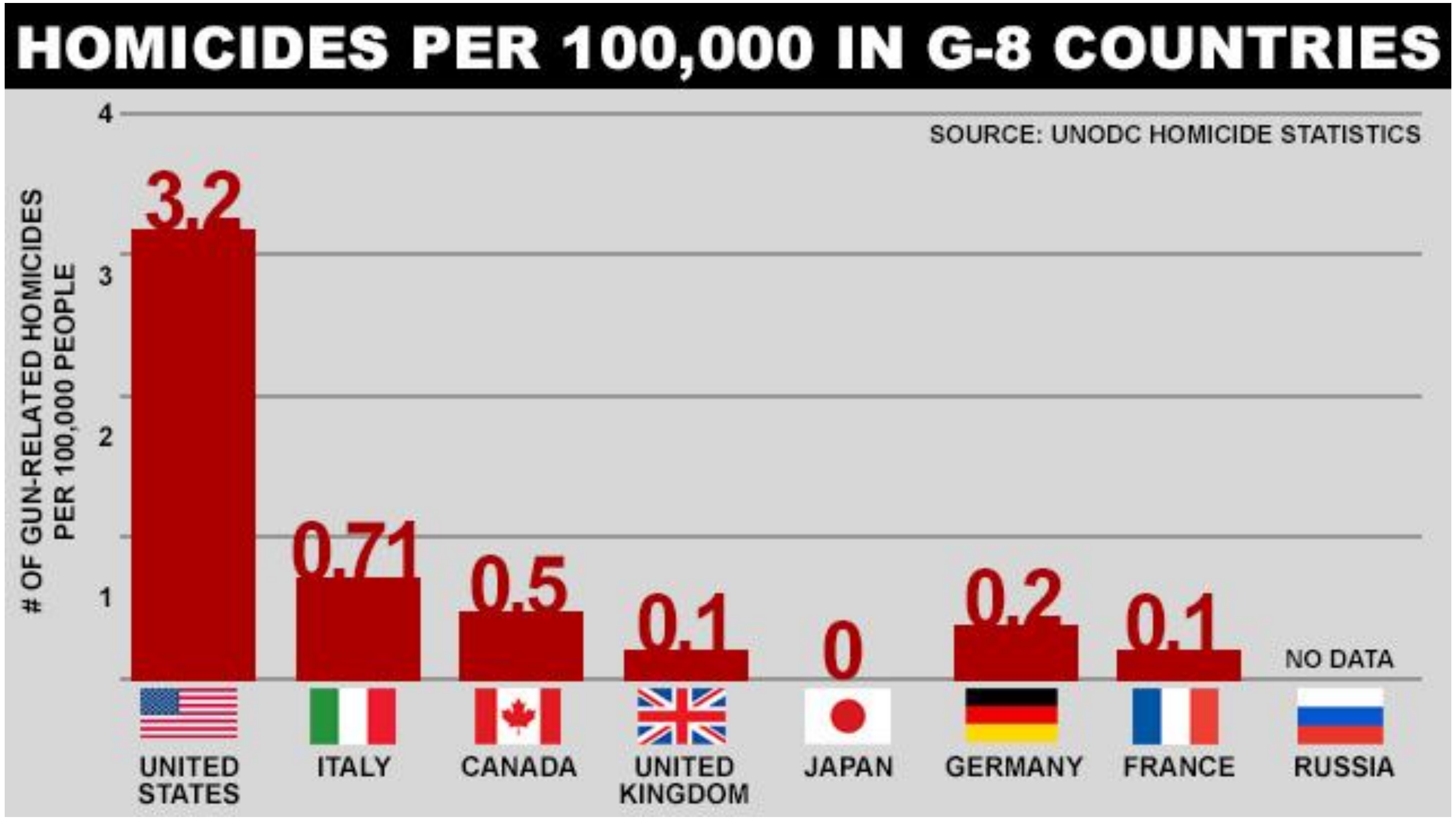
```
> a <- 1
> class(a)
? [1] “numeric”
> print a
? [1] 1
```

Data types & data frame

```
> a <- 1
> class(a)
⍰ [1] “numeric”
> print a
⍰ [1] 1
```

```
> library(dslabs)
> data(murders)
> class(murders)
⍰ [1]
"data.frame"
```

The data frame "murders"



(Source: Ma'ayan Rosenzweigh/ABC News, Data from UNODC Homicide Statistics)

Examining data frame

The function *str* is useful for finding out more about the structure of an object:

```
> str(murders)
```

We can show the first six lines using the function ***head***:

```
> head(murders)
```

To reveal the names for each of the variables stored in this table we use the function *names :*

```
> names(murders)
```

To access the different variables represented by columns included in this data frame, we use the accessor operator $

```
> murders$population
```

Vectors

```
> class(murders$population)
 [1]  " numeric  "
> length(murders$population)
 [1]  51
> class(murders$states)
 [1]  " character  "
> class(murders$region)
 [1]  " factor      "
> levels(murders$region)
[1] "Northeast" "South" "North Central" "West"
> z <- 3 == 2
 [1]  FALSE
> class(z)
 [1]  " logical  "
```

The object murders$population is not one number but several.
We call these types of objects *vectors*.
A single number is technically a vector of length 1, but in general we use the term vectors to refer to objects with several entries.

Lists

```
> mylist = list(name='Charles',age=28,married=TRUE)
> mylist
## $name
## [1] "Charles"
## $age
## [1] 28
## $married
## [1] TRUE
> out = t.test(1:10, y = c(7:20))
> out
## Welch Two Sample t-test
## data: 1:10 and c(7:20)
## t = -5.4349, df = 21.982, p-value = 1.855e-05
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
## -11.052802 -4.947198
## sample estimates:
## mean of x mean of y
## 5.5 13.5
```

Lists are objects able to store information of different types. All functions return lists, so it is important to be able to manipulate them!

Matrices

```
> mat <- matrix(1:12, 4, 3)
#> [,1] [,2] [,3]
#> [1,] 1 5 9
#> [2,] 2 6 10
#> [3,] 3 7 11
#> [4,] 4 8 12
> mat[2, 3]
#> [1] 10
> mat[2, ]
 #> [1] 2 6 10
> mat[, 3]
 #> [1] 9 10 11 12
> mat[, 2:3]
#> [,1] [,2]
 #> [1,] 5 9
#> [2,] 6 10
#> [3,] 7 11
#> [4,] 8 12
> as.data.frame(mat)
```

Matrices are similar to data frames in that they are two-dimensional: they have rows and columns. However, like numeric, character and logical vectors, entries in matrices have to be all the same type.
Yet matrices have a major advantage over data frames: we can perform a matrix algebra operations, a powerful type of mathematical technique.

Exercise

Load the US murders dataset.

1. Use the accessor $ to extract the state abbreviations and assign them to the object a. What is the class of this object?

2. Now use the square brackets to extract the state abbreviations and assign them to the object b. Use the identical function to determine if a and b are the same.

3. The function table takes a vector and returns the frequency of each element. You can quickly see how many states are in each region by applying this function. Use this function in one line of code to create a table of states per region.

Vectors

In R, the most basic objects available to store data are *vectors*. As we have seen, complex datasets can usually be broken down into components that are vectors. For example, in a data frame, each column is a vector.

Create vectors

```
> codes <- c(380, 124, 818)
#> [1] 380 124 818
> country <- c("italy", "canada", "egypt")
> codes <- c(italy = 380, canada = 124, egypt = 818)
 #> italy canada egypt
#> 380 124 818
> class(codes)
#> [1] "numeric"]
> names(codes)
 #> [1] "italy" "canada" "egypt"
```

We can create vectors using the function `c`, which stands for *concatenate*.

Create vectors

```
> codes <- c(380, 124, 818)
#> [1] 380 124 818
> country <- c("italy", "canada", "egypt")
> codes <- c(italy = 380, canada = 124, egypt = 818)
 #> italy canada egypt
#> 380 124 818
> class(codes)
#> [1] "numeric"]
> names(codes)
 #> [1] "italy" "canada" "egypt"
> seq(1, 10)
#> [1] 1 2 3 4 5 6 7 8 9 10
```

Another useful function for creating vectors generates sequences:

Create vectors

```
> codes <- c(380, 124, 818)
#> [1] 380 124 818
> country <- c("italy", "canada", "egypt")
> codes <- c(italy = 380, canada = 124, egypt = 818)
 #> italy canada egypt
#> 380 124 818
> class(codes)
#> [1] "numeric"]
> names(codes)
 #> [1] "italy" "canada" "egypt"
> seq(1, 10)
#> [1] 1 2 3 4 5 6 7 8 9 10
> codes[2]
#> canada
#> 124
> codes[c(1,3)]
#> italy egypt
#> 380 818
> codes[1:2]
#> italy canada
#> 380 124
```

Subsetting a vector

Coercion

In general, *coercion* is an attempt by R to be flexible with data types. When an entry does not match the expected, some of the prebuilt R functions try to guess what was meant before throwing an error. This can also lead to confusion. Failing to understand *coercion* can drive programmers crazy when attempting to code in R since it behaves quite differently from most other languages in this regard.

```
> x <- c(1, "canada", 3)
```

You expected an error message!

Coercion

In general, *coercion* is an attempt by R to be flexible with data types. When an entry does not match the expected, some of the prebuilt R functions try to guess what was meant before throwing an error. This can also lead to confusion. Failing to understand *coercion* can drive programmers crazy when attempting to code in R since it behaves quite differently from most other languages in this regard.

```
> x <- c(1, "canada", 3)
> x
#> [1] "1" "canada" "3"
> class(x)
#> [1] "character  "
```

R *coerced* the data into characters.

Coercion

In general, *coercion* is an attempt by R to be flexible with data types. When an entry does not match the expected, some of the prebuilt R functions try to guess what was meant before throwing an error. This can also lead to confusion. Failing to understand *coercion* can drive programmers crazy when attempting to code in R since it behaves quite differently from most other languages in this regard.

```
> x <- c(1, "canada", 3)
> x
#> [1] "1" "canada" "3"
> class(x)
#> [1] "character  "
```

R *coerced* the data into characters.

```
> x <- 1:5
> y <- as.character(x)
> y
#> [1] "1" "2" "3" "4" "5"
> as.numeric(y)
#> [1] 1 2 3 4 5
```

R also offers functions to change from one type to another.

Exercise

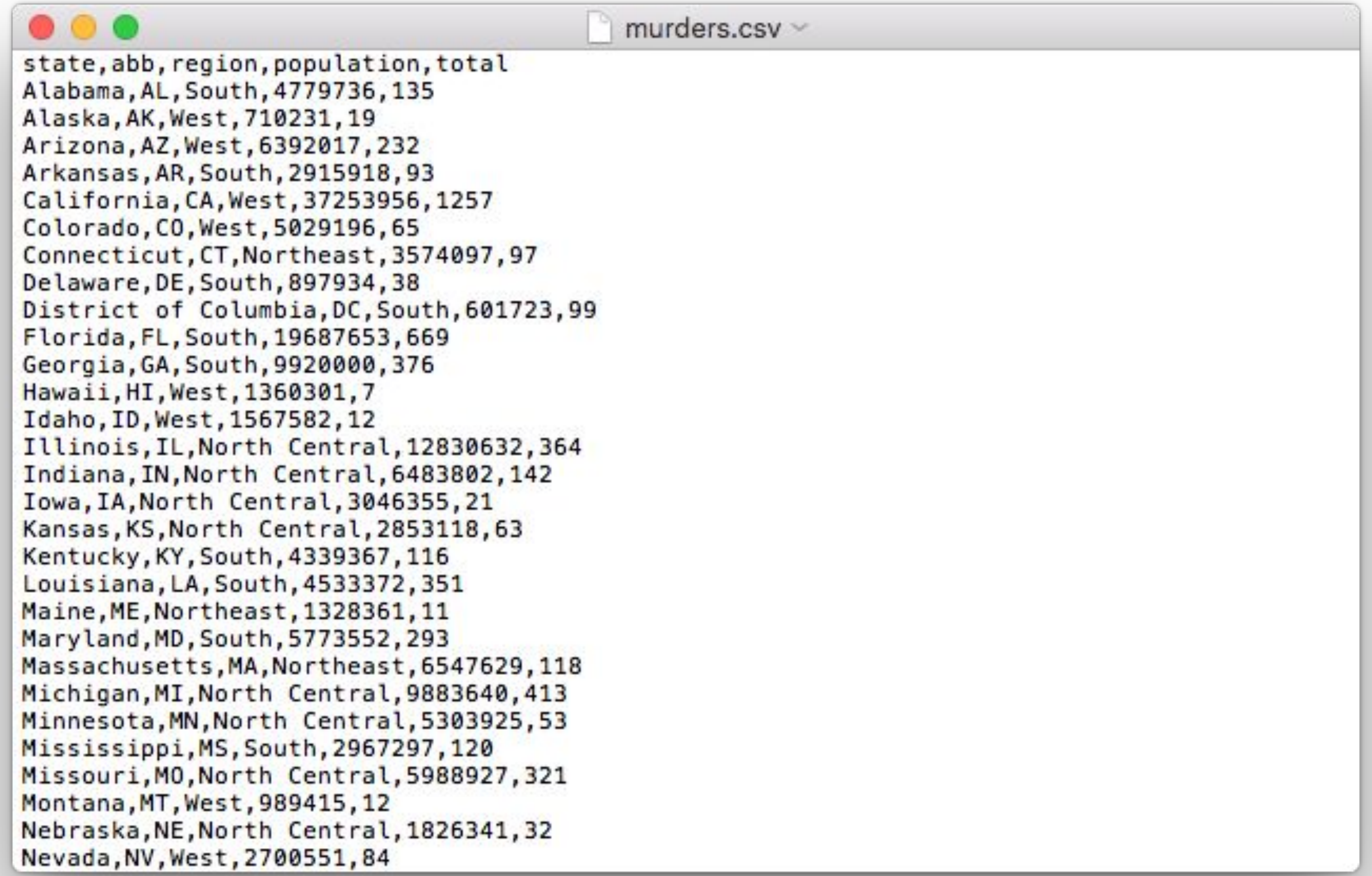
1. Create two vectors of different dimensions and insert the second one in the first one between the 2nd and 3rd elements.

2. Draw 100 numbers from a Uniform distribution on [0,1] and count how many values are larger than 0.5

3. Compute the per 100,000 murder rate for each state and store it in the object murder_rate. Then compute the average murder rate for the US using the function mean. What is the average?

Importing data

When creating spreadsheets with text files, like the ones created with a simple text editor, a new row is defined with return and columns are separated with some predefined special character. The most common characters are comma (,), semicolon (;), space (`) and tab ( or`, a preset number of spaces).



murders.csv

```
state,abb,region,population,total
Alabama,AL,South,4779736,135
Alaska,AK,West,710231,19
Arizona,AZ,West,6392017,232
Arkansas,AR,South,2915918,93
California,CA,West,37253956,1257
Colorado,CO,West,5029196,65
Connecticut,CT,Northeast,3574097,97
Delaware,DE,South,897934,38
District of Columbia,DC,South,601723,99
Florida,FL,South,19687653,669
Georgia,GA,South,9920000,376
Hawaii,HI,West,1360301,7
Idaho,ID,West,1567582,12
Illinois,IL,North Central,12830632,364
Indiana,IN,North Central,6483802,142
Iowa,IA,North Central,3046355,21
Kansas,KS,North Central,2853118,63
Kentucky,KY,South,4339367,116
Louisiana,LA,South,4533372,351
Maine,ME,Northeast,1328361,11
Maryland,MD,South,5773552,293
Massachusetts,MA,Northeast,6547629,118
Michigan,MI,North Central,9883640,413
Minnesota,MN,North Central,5303925,53
Mississippi,MS,South,2967297,120
Missouri,MO,North Central,5988927,321
Montana,MT,West,989415,12
Nebraska,NE,North Central,1826341,32
Nevada,NV,West,2700551,84
```

```
> wd <- getwd()
```

The filesystem

The *path* of a file is a list of directory names that can be thought of as instructions on what folders to click on, and in what order, to find the file.

```
> system.file(package = "dslabs")
#> [1] "/Library/Frameworks/R.framework/Versions/3.5/Resources/library/dslabs »
> filename <- "murders.csv"
> dir <- system.file("extdata", package = "dslabs")
> fullpath <- file.path(dir, filename)
```

We can use the function list.files to see examples of relative path

```
> dir <- system.file(package = "dslabs") list.files(path = dir)
#> [1] "data" "DESCRIPTION" "extdata" "help" "html" #> [6] "INDEX" "Meta" "NAMESPACE" "R" "script"
```

Copying files using paths

```
> file.copy(fullpath, "murders.csv")
 #> [1] TRUE
```

The readr and readxl packages

readr

Function	Format	Typical suffix
read_table	white space separated values	txt
read_csv	comma separated values	csv
read_csv2	semicolon separated values	csv
read_tsv	tab delimited separated values	tsv

readxl

Function	Format	Typical suffix
read_excel	auto detect the format	xls, xlsx
read_xls	original format	xls
read_xlsx	new format	xlsx

```
> library(readr)
> read_lines("murders.csv", n_max = 3)
#> [1] "state,abb,region,population,total" "Alabama,AL,South,4779736,135"
#> [3] "Alaska,AK,West,710231,19 »
> dat <- read_csv(filename)
> head(dat)
```

R-base also provides import functions. These have similar names to those in the **tidyverse**, for example read.table, read.csv and read.delim

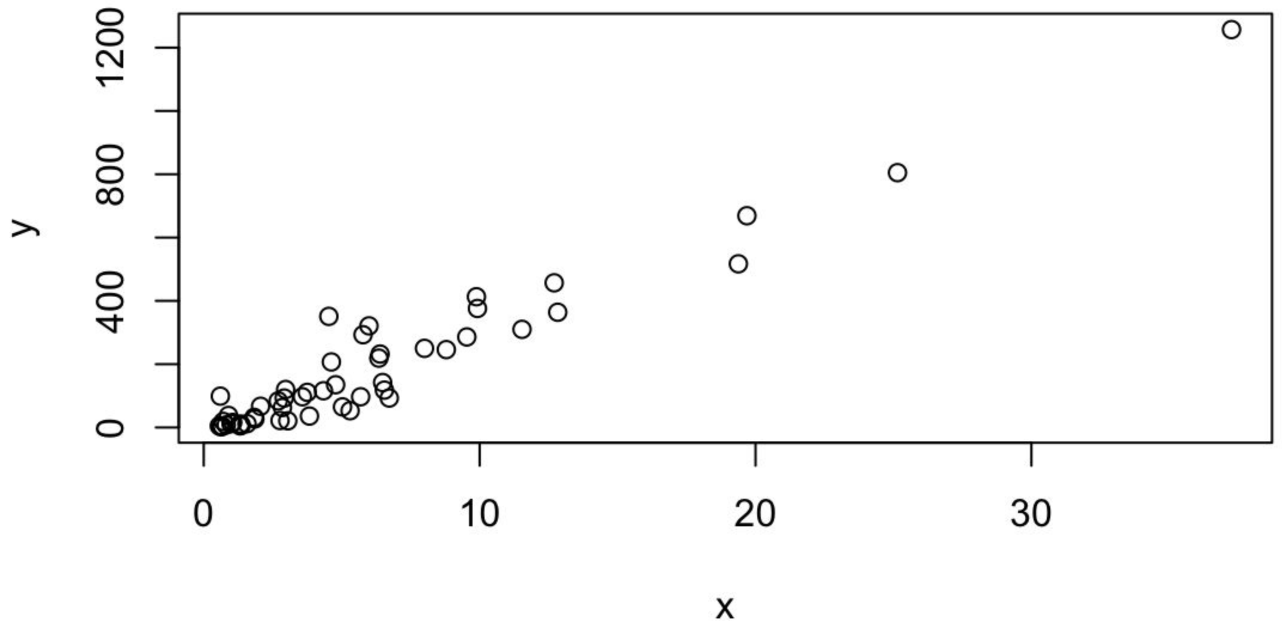
Exercises

Write a script allowing to load a vector file and “remove” the missing values.

Basic plots

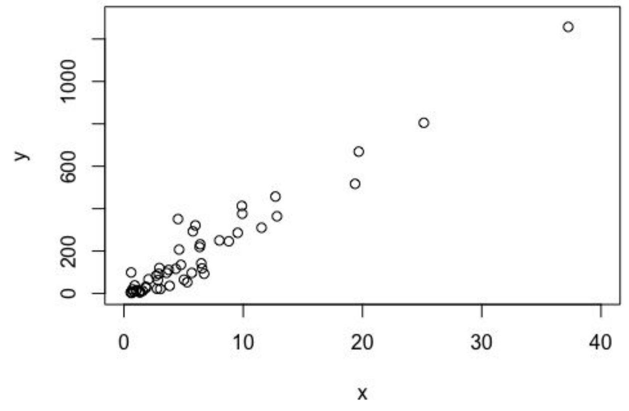
The plot function can be used to make scatterplots.

```
> x <- murders$population / 10^6
> y <- murders$total
> plot(x, y)
```

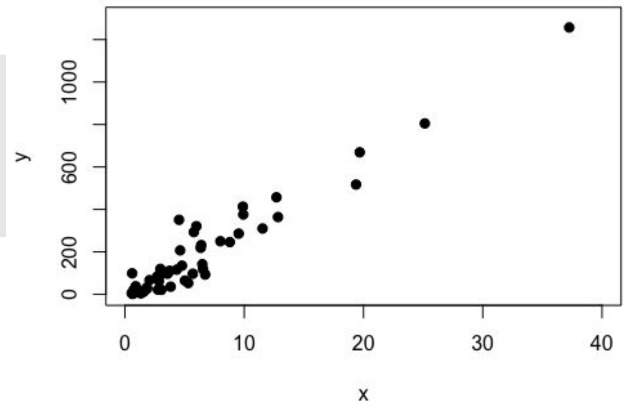


Basic plots

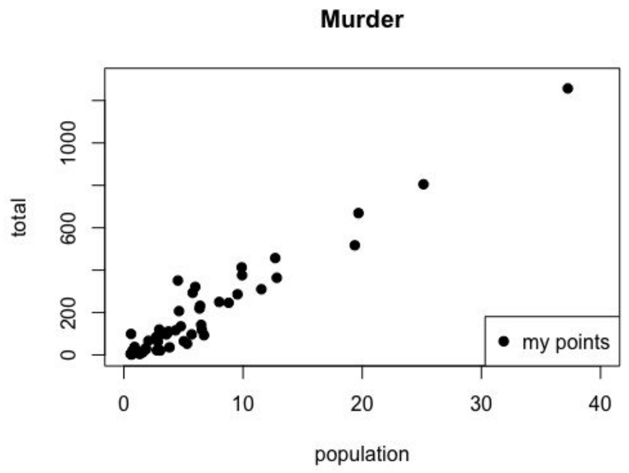
```
> plot(x, y, xlim=c(0, 40),
ylim=c(0,1300))
```



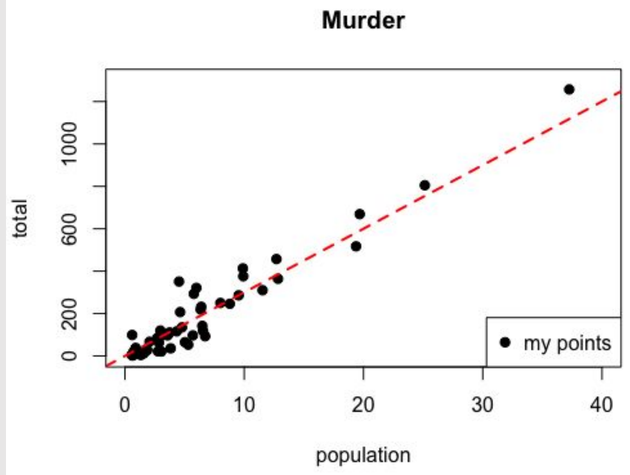
```
> plot(x, y, xlim=c(0, 40),
ylim=c(0,1300), pch=19)
```



```
> plot(x, y, xlim=c(0, 40),
ylim=c(0,1300), pch=19,
xlab="population",
ylab="total",
main="Murder")
> legend("bottomright",
legend = 'my
points',col=1,pch=19)
```



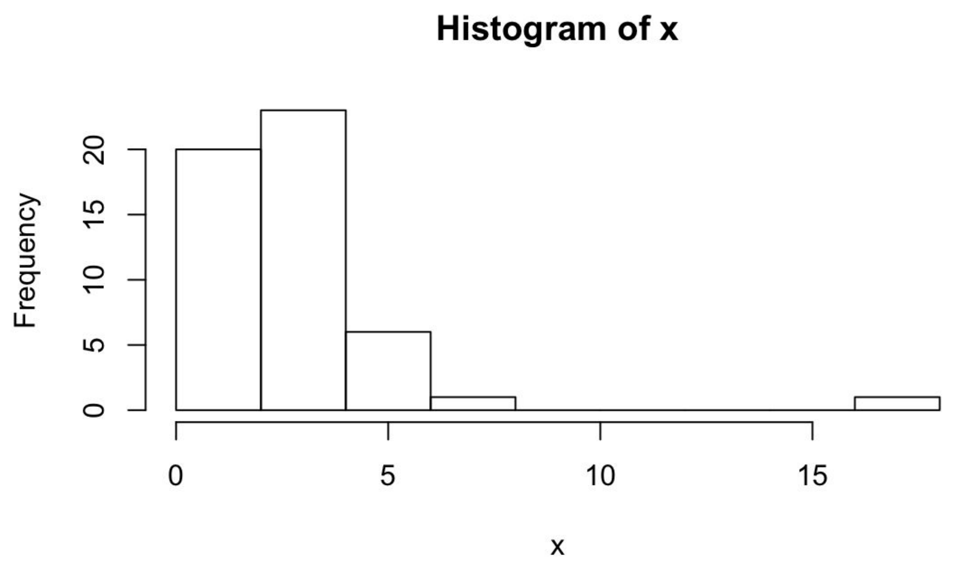
```
> plot(x, y, xlim=c(0, 40),
ylim=c(0,1300), pch=19,
xlab="population",
ylab="total",
main="Murder")
> legend("bottomright",
legend = 'my
points',col=1,pch=19)
> abline(a=0,b=30,lty=2,
lwd=2,col='red')
```



Basic Statistics: histograms

We can make a histogram of our murder rates by simply typing:

```
> x <- with(murders, total / population * 100000)
> hist(x)
```

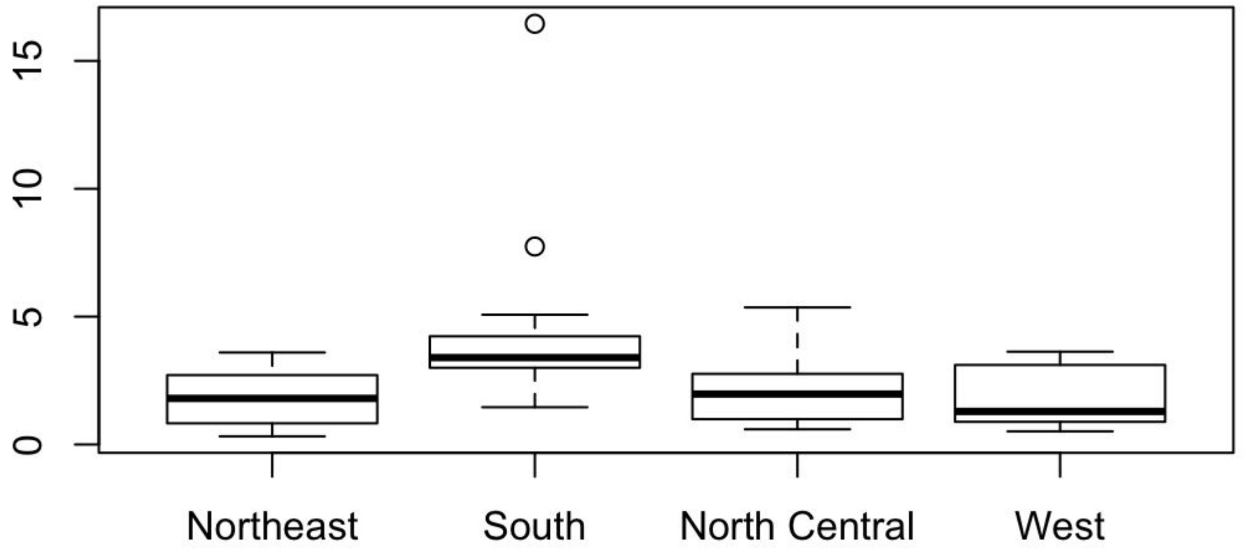


```
> murders$state[which.max(x)]
#> [1] "District of Columbia"
```

Basic Statistics: boxplots

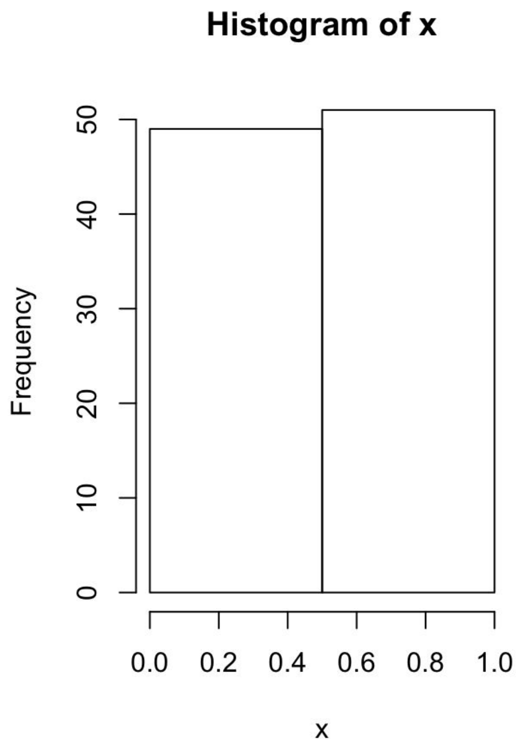
We can use boxplots to compare the murder rates of different regions:

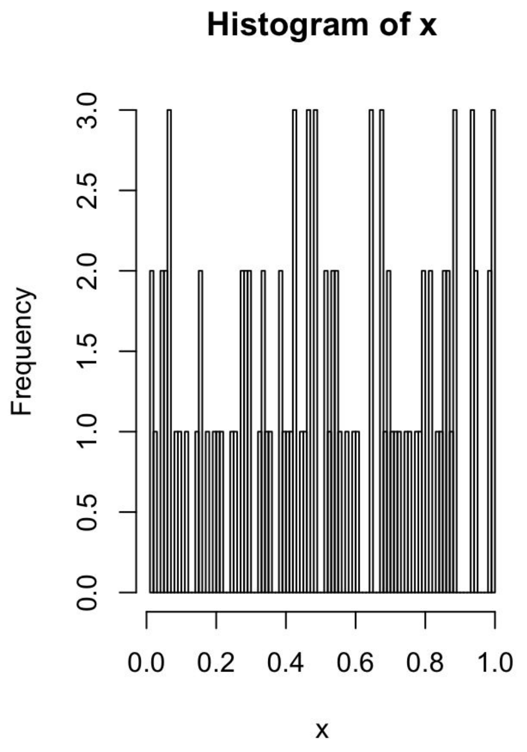
```
> murders$rate <- with(murders, total / population * 100000)
> boxplot(rate~region, data = murders)
```



Histograms

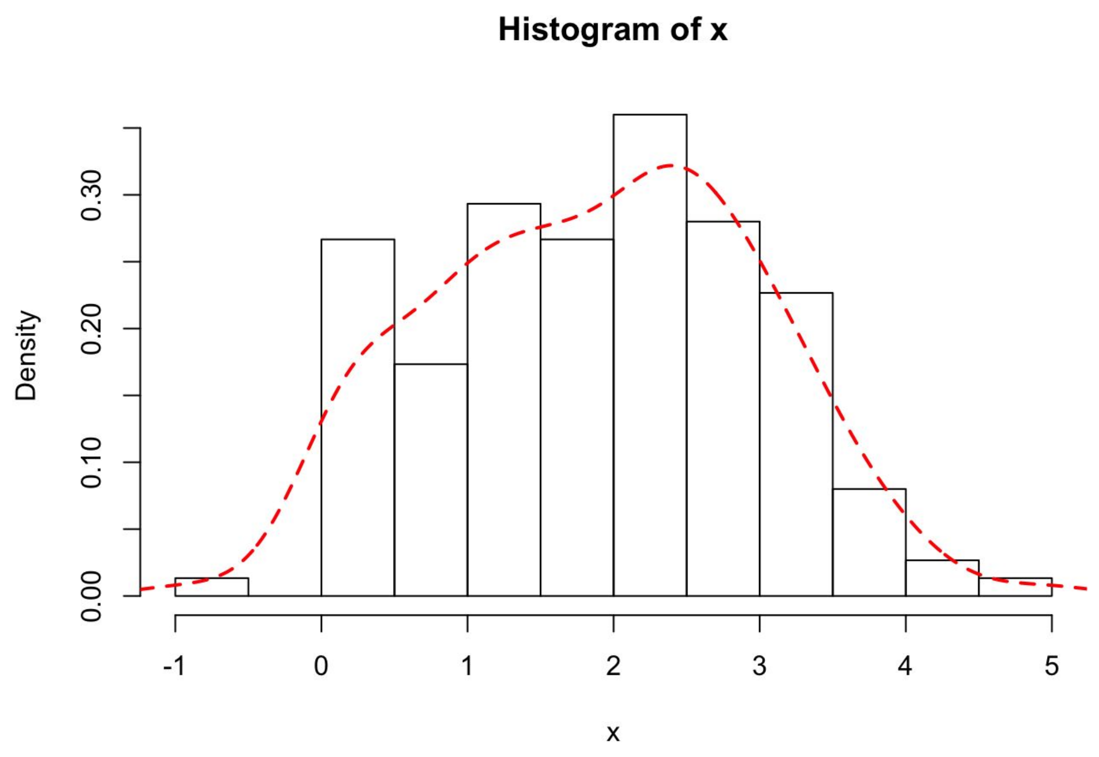
```
x = runif(100,0,1)
par(mfrow=c(1,2))
hist(x,breaks=2)
hist(x,breaks=100)
```





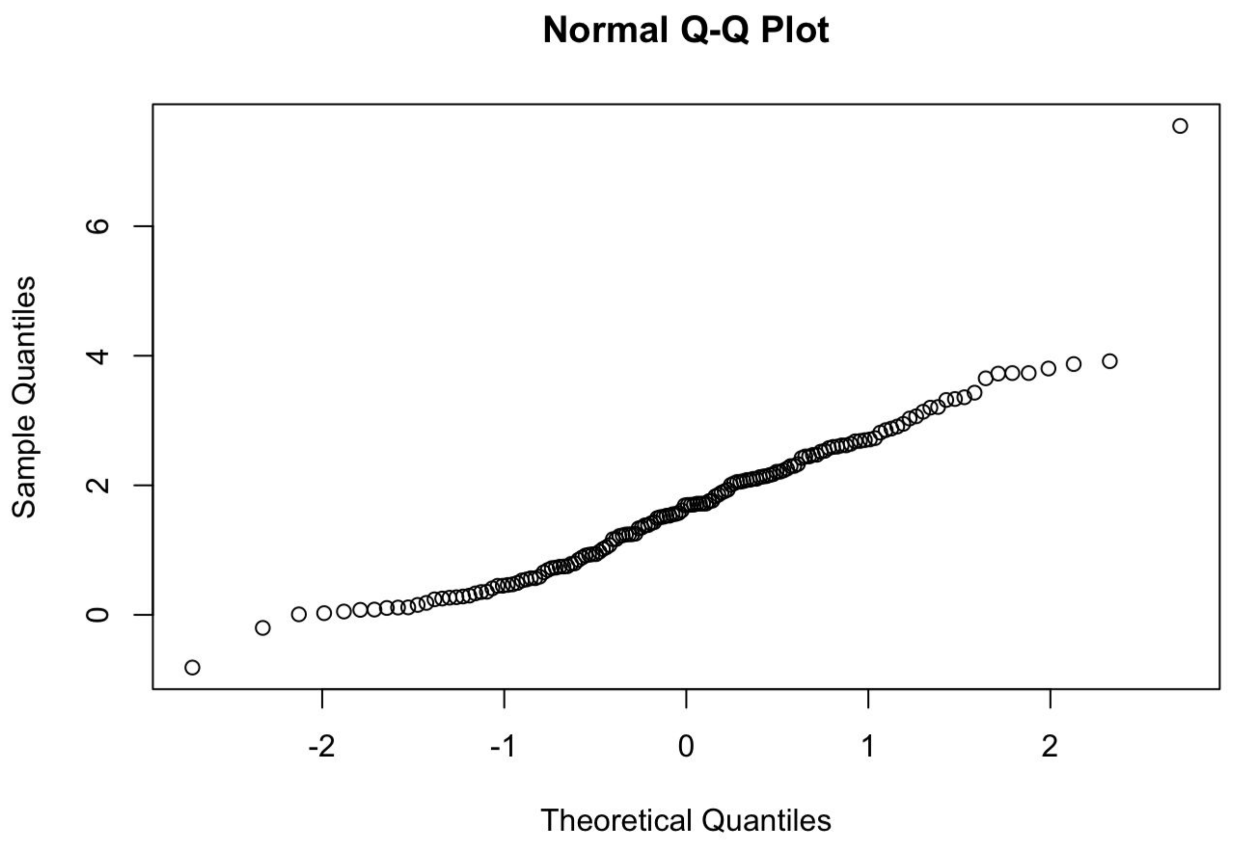
Density plot

```
x = c(rnorm(100,2,1),rgamma(50,shape = 1))
hist(x,freq = FALSE)
f = density(x)
lines(f,lwd=2,col='red',lty=2)
```



Compare distributions

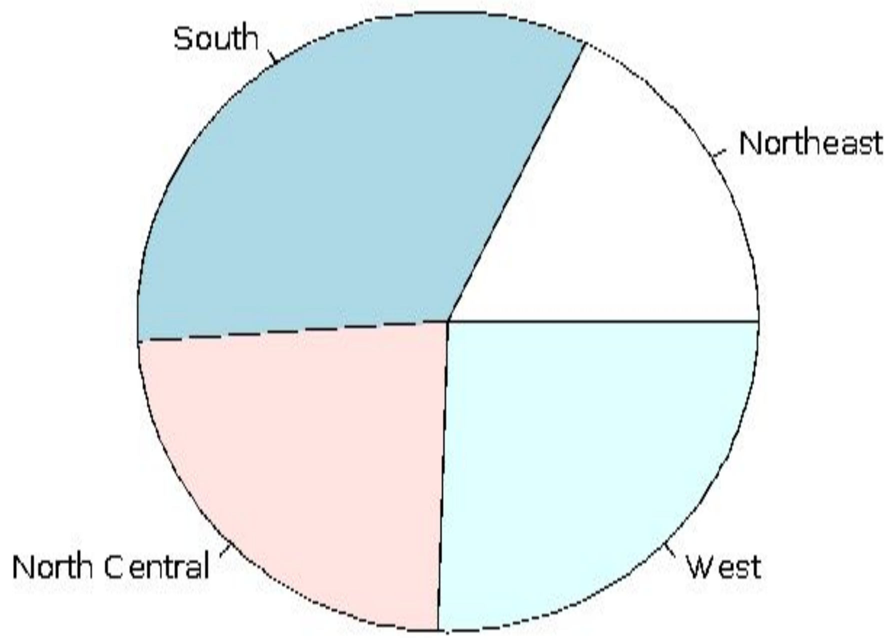
```
x = c(rnorm(100,2,1),rgamma(50,shape = 1))
qqnorm(x)
```



Basic Statistics: piecharts

Finally, for categorical variables, you can draw pie plots:

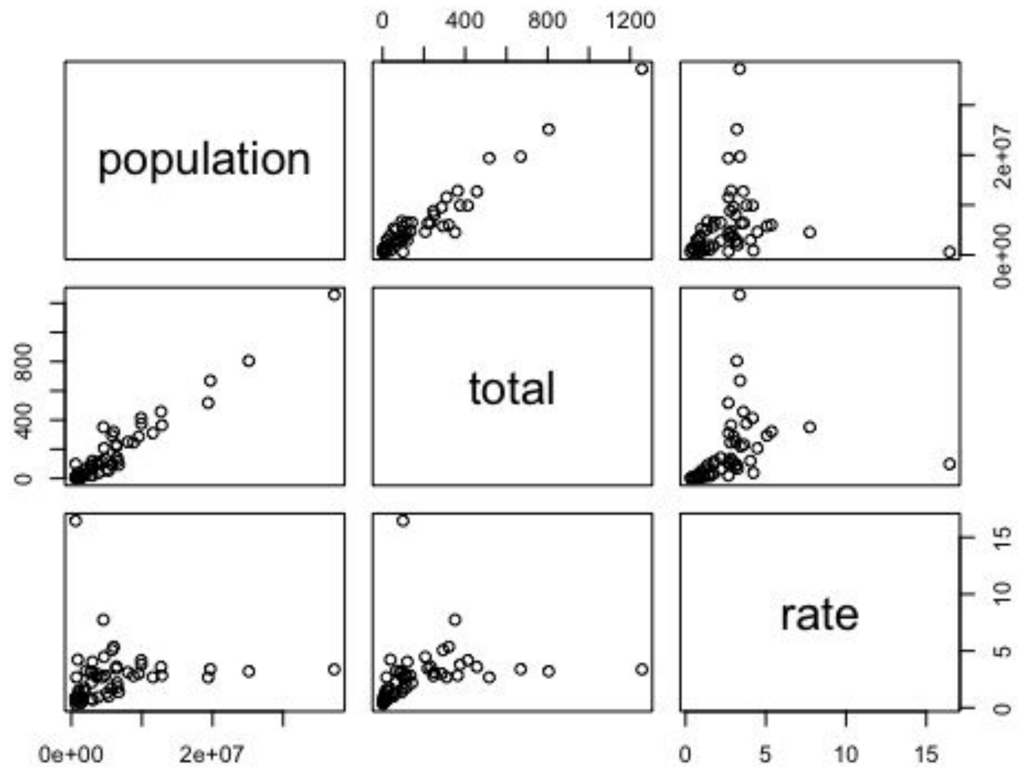
```
> pie(summary(murders$region))
```

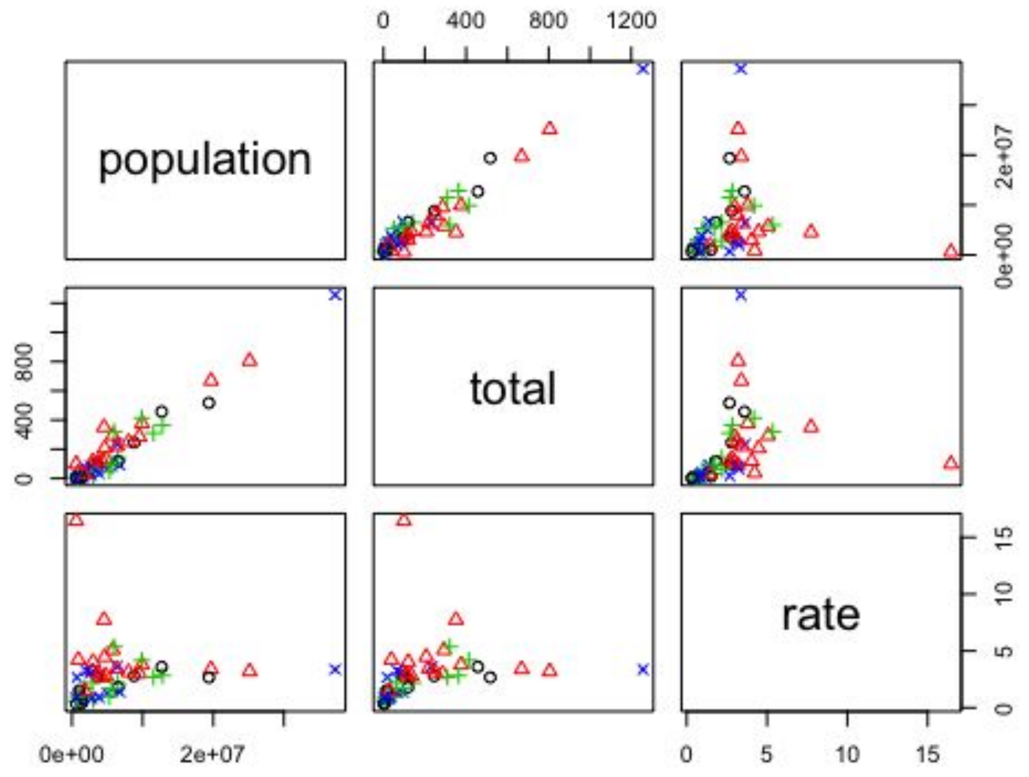


Basic Statistics: pairplot

```
> x<-murders[,4:6]
> pairs(x)
```

```
> x<-murders[,4:6]
> pairs(x, col=as.numeric(murders$region),
pch=as.numeric(murders$region))
```





Exercises

1. Create a histogram of the state populations.
2. Generate boxplots of the state populations by region.

Functions

Writing functions is a core activity of an R programmer. It represents the key step of the transition from a mere "user" to a developer who creates new functionality for R. Functions are often used to encapsulate a sequence of expressions that need to be executed numerous times, perhaps under slightly different conditions. Functions are also often written when code must be shared with others or the public.

The writing of a function allows a developer to create an interface to the code, that is explicitly specified with a set of parameters. This interface provides an abstraction of the code to potential users. This abstraction simplifies the users' lives because it relieves them from having to know every detail of how the code operates. In addition, the creation of an interface allows the developer to communicate to the user the aspects of the code that are important or are most relevant.

Functions in R

Functions in R are “first class objects”, which means that they can be treated much like any other R object.

Importantly,

- **Functions can be passed as arguments to other functions. This is very handy for the various apply functions, like lapply() and sapply().**

- **Functions can be nested, so that you can define a function inside of another function**

If you’re familiar with common language like C, these features might appear a bit strange. However, they are really important in R and can be useful for data analysis.

Your First Function

Functions are defined using the function() directive and are stored as R objects just like anything else. In particular, they are R objects of class “function”.

Here’s a simple function that takes no arguments and does nothing.

```
> f <- function() {
+       ## This is an empty function
+ }
> ## Functions have their own class
> class(f)
[1] "function"
> ## Execute this function
> f()
NULL
```

Here we create a function that actually has a non-trivial *function body*.

```
> f <- function() {
+       cat("Hello, world!\n")
+ }
> f()
Hello, world!
```

Your First Function

The last aspect of a basic function is the *function arguments*. These are the options that you can specify to the user that the user may explicity set.

For this basic function, we can add an argument that determines how many times “Hello, world!” is printed to the console.

```
> f <- function(num) {
+       for(i in seq_len(num)) {
+               cat("Hello, world!\n")
+       }
+ }
> f(3)
Hello, world!
Hello, world!
Hello, world!
```

This next function returns the total number of characters printed to the console.

```
> f <- function(num) {
+       hello <- "Hello, world!\n"
+       for(i in seq_len(num)) {
+               cat(hello)
+       }
+       chars <- nchar(hello) * num
+       chars
+ }
> meaningoflife <- f(3)
Hello, world!
Hello, world!
Hello, world!
> print(meaningoflife)
[1] 42
```

Your First Function

Here, for example, we could set the default value for num to be 1, so that if the function is called without the num argument being explicitly specified, then it will print “Hello, world!” to the console once.

```
> f <- function(num = 1) {
+        hello <- "Hello, world!\n"
+        for(i in seq_len(num)) {
+               cat(hello)
+        }
+        chars <- nchar(hello) * num
+        chars
+ }
> f()    ## Use default value for 'num'
Hello, world!
[1] 14
> f(2)   ## Use user-specified value
Hello, world!
Hello, world!
[1] 28
```

At this point, we have written a function that

- has one *formal argument* named num with a *default value* of 1. The *formal arguments* are the arguments included in the function definition. The formals() function returns a list of all the formal arguments of a function
- prints the message “Hello, world!” to the console a number of times indicated by the argument num
- *returns* the number of characters printed to the console

Argument Matching

Calling an R function with arguments can be done in a variety of ways. This may be confusing at first, but it’s really handing when doing interactive work at the command line. R functions arguments can be matched *positionally* or by name. Positional matching just means that R assigns the first value to the first argument, the second value to second argument, etc. So in the following call to rnorm()

```
> str(rnorm)
function (n, mean = 0, sd = 1)
> mydata <- rnorm(100, 2, 1)          ## Generate some data
```

100 is assigned to the `n` argument, 2 is assigned to the `mean` argument, and 1 is assigned to the `sd` argument, all by positional matching.

Argument Matching

The following calls to the `sd()` function (which computes the empirical standard deviation of a vector of numbers) are all equivalent. Note that `sd()` has two arguments: `x` indicates the vector of numbers and `na.rm` is a logical indicating whether missing values should be removed or not.

```
> ## Positional match first argument, default for 'na.rm'
> sd(mydata)
[1] 1.014325
> ## Specify 'x' argument by name, default for 'na.rm'
> sd(x = mydata)
[1] 1.014325
> ## Specify both arguments by name
> sd(x = mydata, na.rm = FALSE)
[1] 1.014325
```

Lazy Evaluation

Arguments to functions are evaluated *lazily*, so they are evaluated only as needed in the body of the function.
In this example, the function f() has two arguments: a and b.

```
> f <- function(a, b) {
+       a^2
+ }
> f(2)
[1] 4
```

This function never actually uses the argument `b`, so calling `f(2)` will not produce an error because the 2 gets positionally matched to `a`. This behavior can be good or bad. It’s common to write a function that doesn’t use an argument and not notice it simply because R never throws an error.

Lazy Evaluation

This example also shows lazy evaluation at work, but does eventually result in an error.

```
> f <- function(a, b) {
+       print(a)
+       print(b)
+ }
> f(45)
[1] 45
Error in print(b): argument "b" is missing, with no default
```

Notice that “45” got printed first before the error was triggered. This is because `b` did not have to be evaluated until after `print(a)`. Once the function tried to evaluate `print(b)` the function had to throw an error.

The ... Argument

There is a special argument in R known as the ... argument, which indicate a variable number of arguments that are usually passed on to other functions. The ... argument is often used when extending another function and you don’t want to copy the entire argument list of the original function
For example, a custom plotting function may want to make use of the default plot() function along with its entire argument list. The function below changes the default for the type argument to the value type = "l" (the original default was type = "p").

```
myplot <- function(x, y, type = "l", ...) {
        plot(x, y, type = type, ...)        ## Pass '...' to 'plot' function
}
```

Generic functions use ... so that extra arguments can be passed to methods.

```
> mean
function (x, ...)
UseMethod("mean")
<bytecode: 0x7fe2e9aead40>
<environment: namespace:base>
```

Exercise

Data normalization

1. Create a function that normalizes a vector:

   x_new = (x-mean(x))/std(x)
2. Use this function on the iris dataset so that each column is normalized
3. You can also make the function more general

   x_new = (x-a)/b

and use it to preprocess the data with min/(max-min) instead of mean/std

Trainig/test sets

Create a new file with a function that split a dataframe into train-valid-test set, given 3 ratios

Summary

- Functions can be defined using the function() directive and are assigned to R objects just like any other R object
- Functions have can be defined with named arguments; these function arguments can have default values
- Functions arguments can be specified by name or by position in the argument list
- Functions always return the last expression evaluated in the function body
- A variable number of arguments can be specified using the special ... argument in a function definition.