# Basic R

# Instructors

Silvia BOTTINI

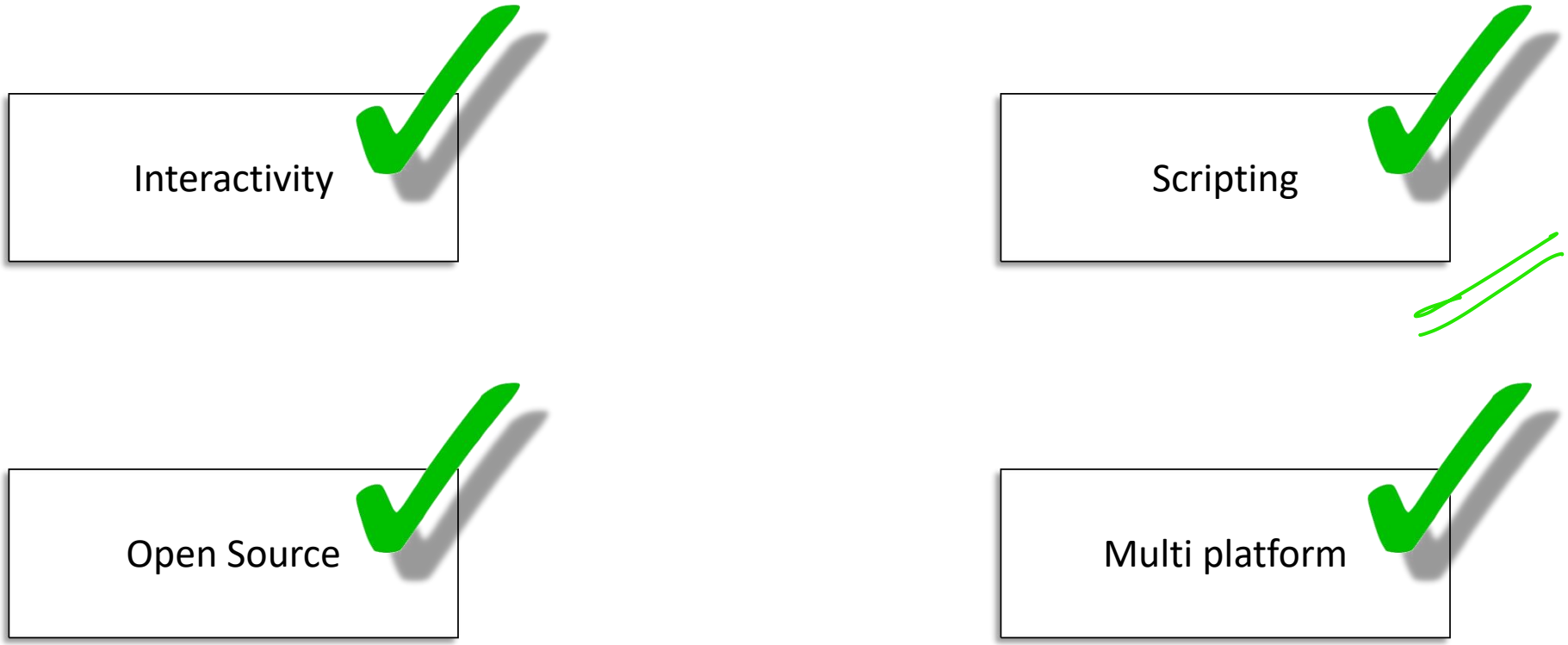
silvia.bottini@univ-cotedazur.fr

Justine LABORY

Justine.labory@etu.univ-cotedazur.fr

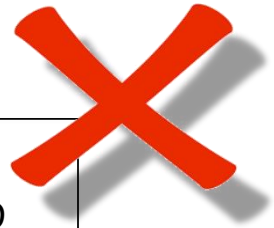
# Materials

# Why R?

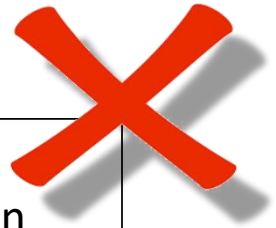


# Drawbacks

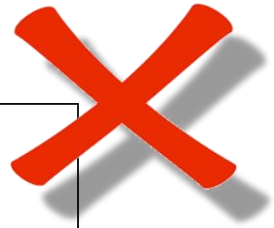
Not user friendly @ start



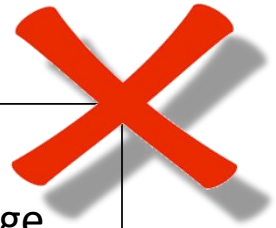
Data preparation



No easy debugging

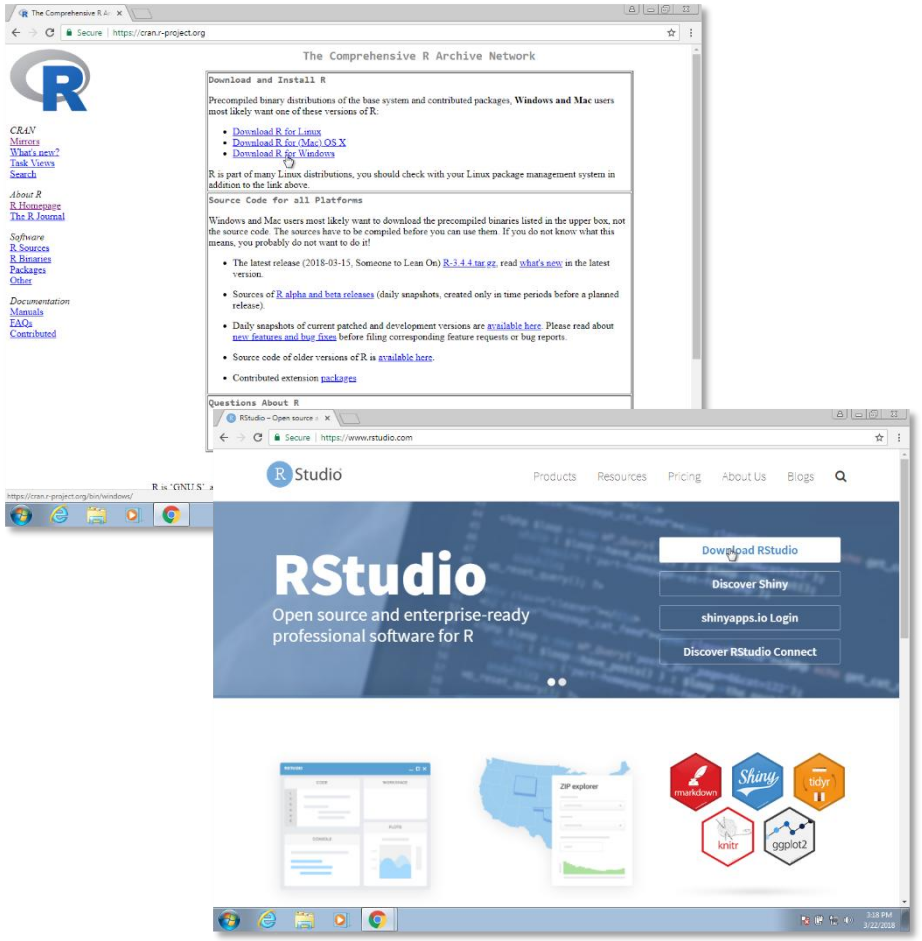


Working with large datasets is limited by RAM



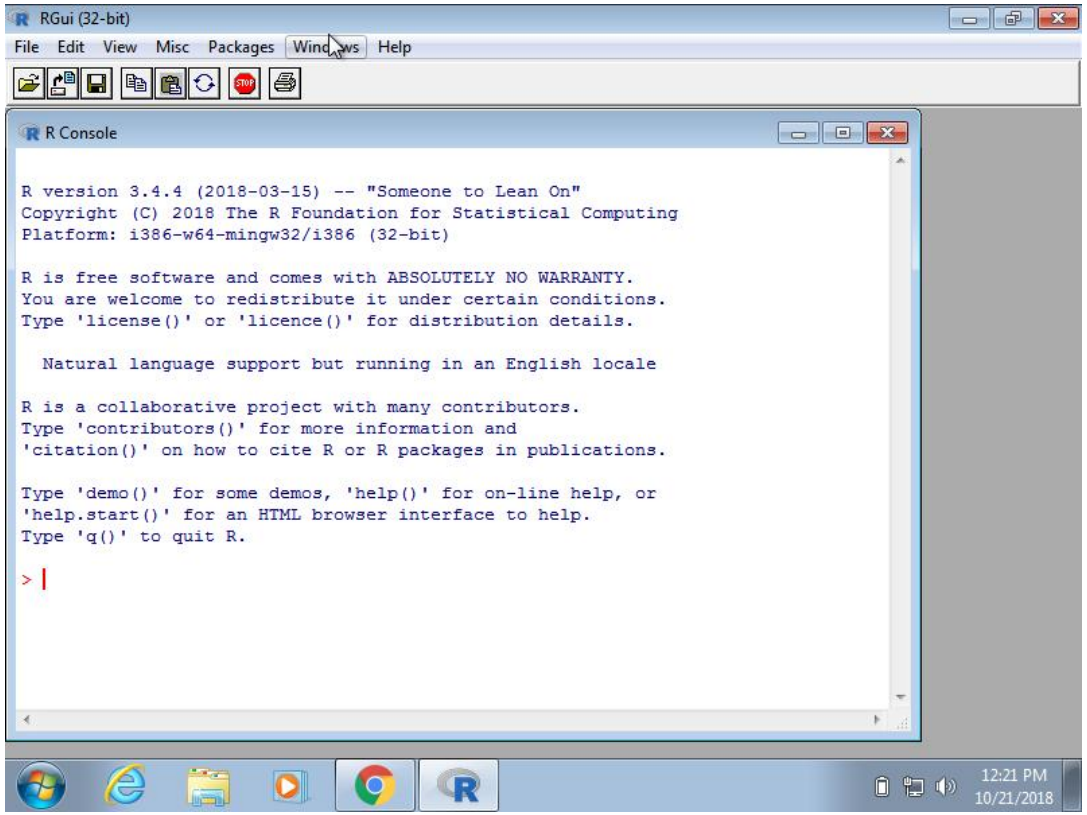
# Let’s start

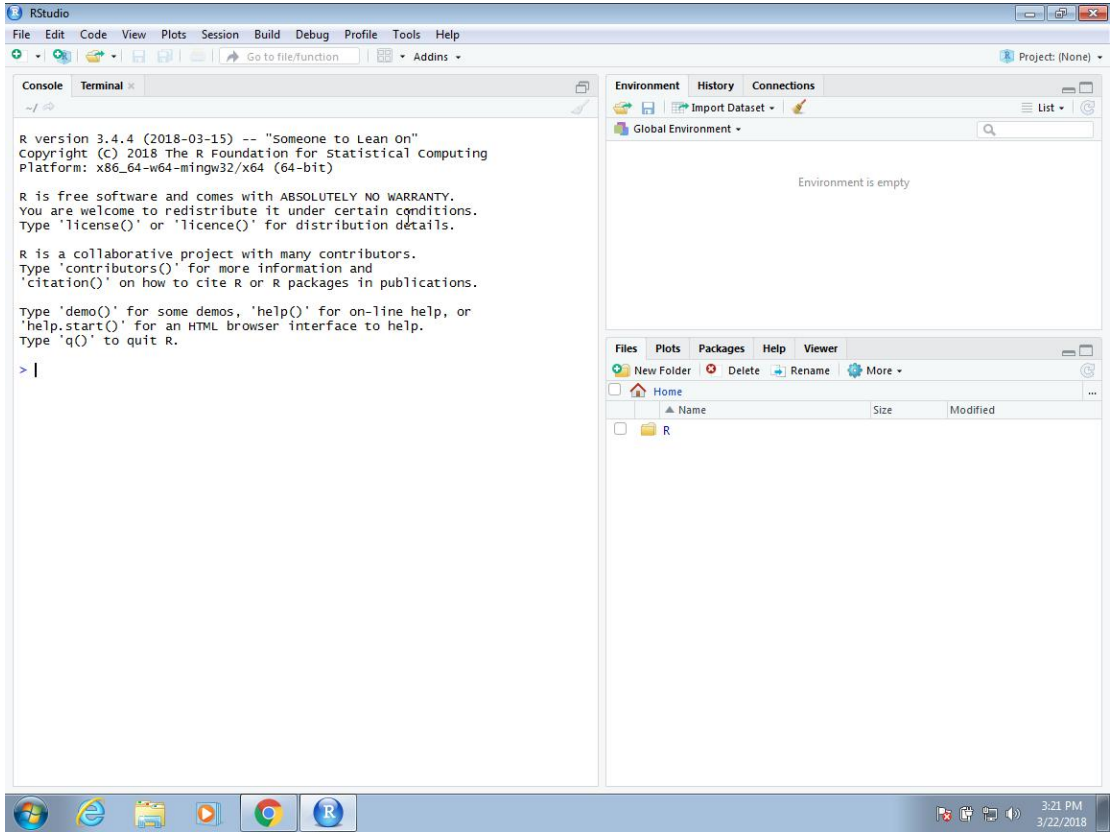
1. Download R: Console & RStudio



# Let’s start

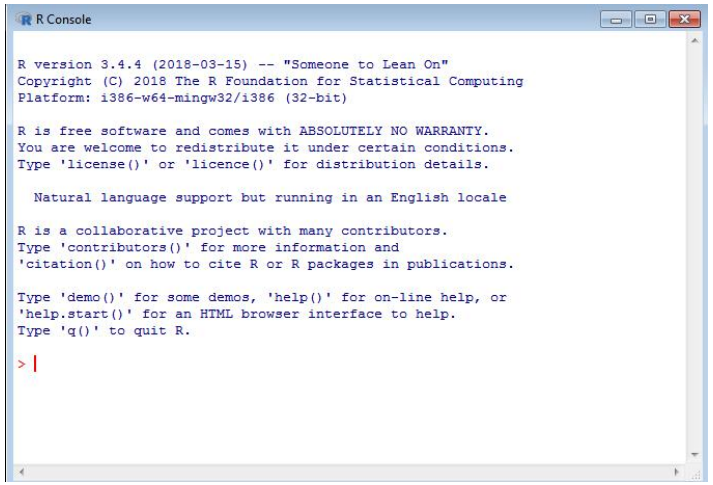
1. Download R: Console & Rstudio
2. Open the console



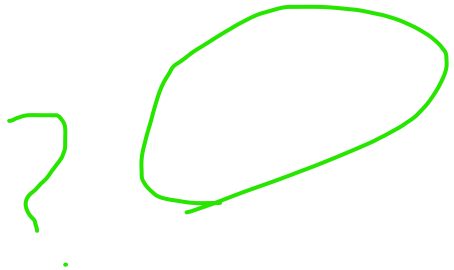
# Let’s start

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

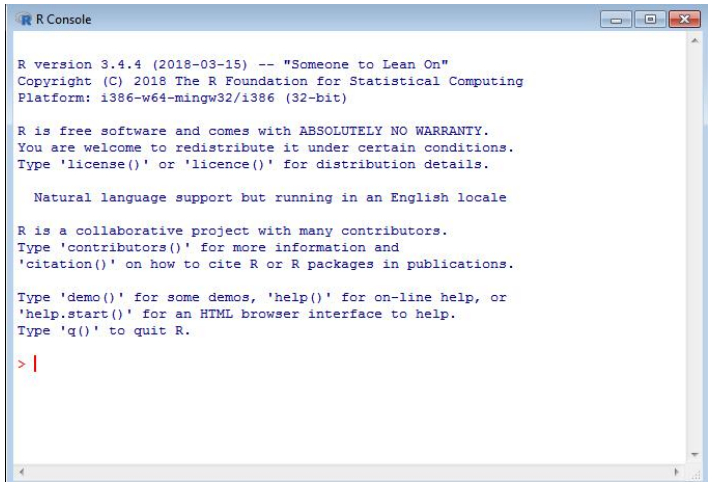



```
> 2 * 2
   [1] 4
> 0.15 * 19.71
   [1] 2.96
> exp(-2)
   [1] 0.1353353
```

# Let’s start



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




> ?t.test

Description: a short description in English of the method implemented in the function

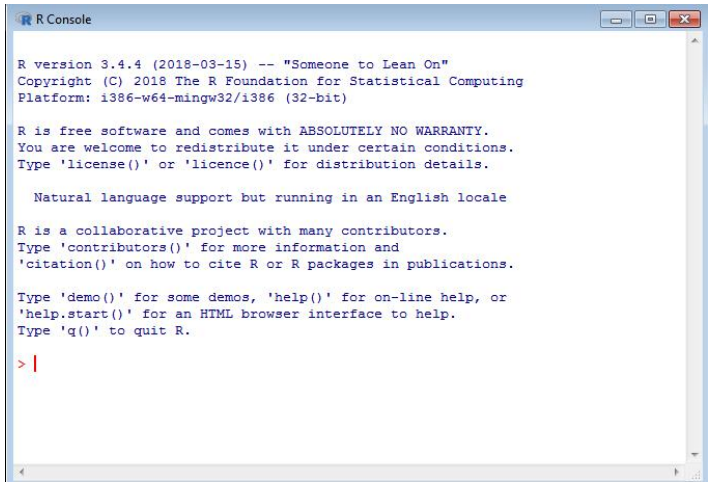
Usage: the way the function should be used, with all input parameters and their default values

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



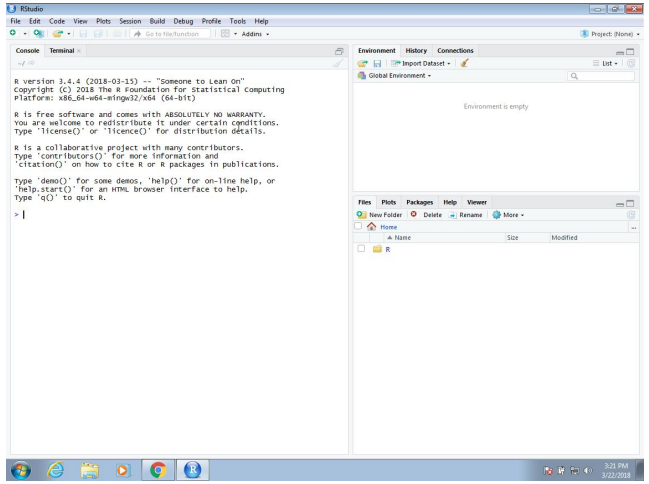

```
# installing the package
> install.packages("dslabs")
```

The downloaded binary packages are in

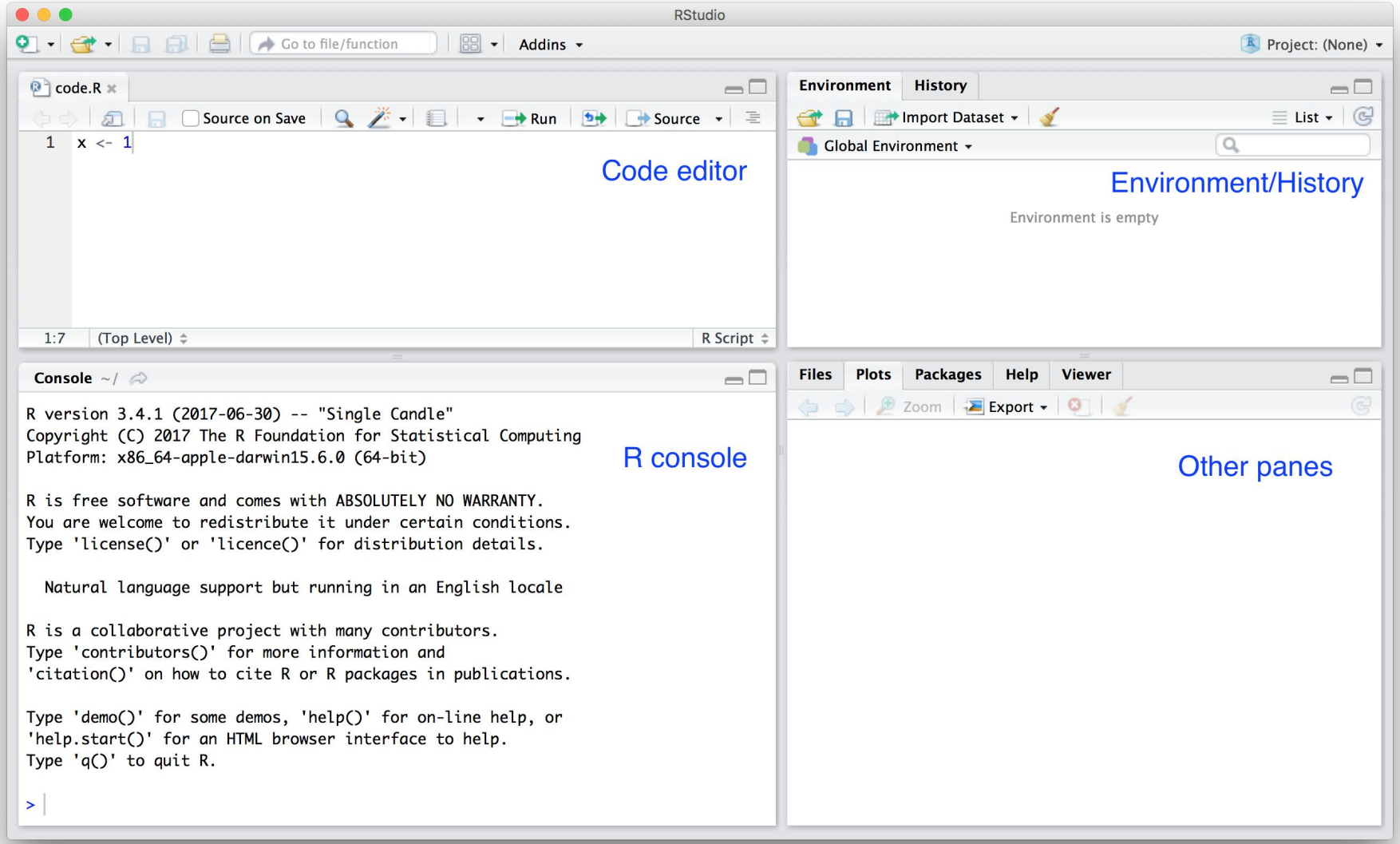
/var/folders/0t/35l1qfw53xqbdy1ldsnljgmc0000gn/T//Rtmp9XgRWo/downloaded_packages

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

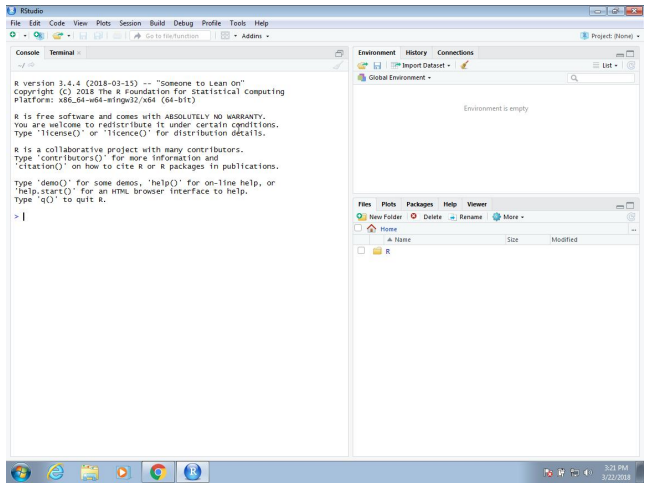
# Let’s do some scripting



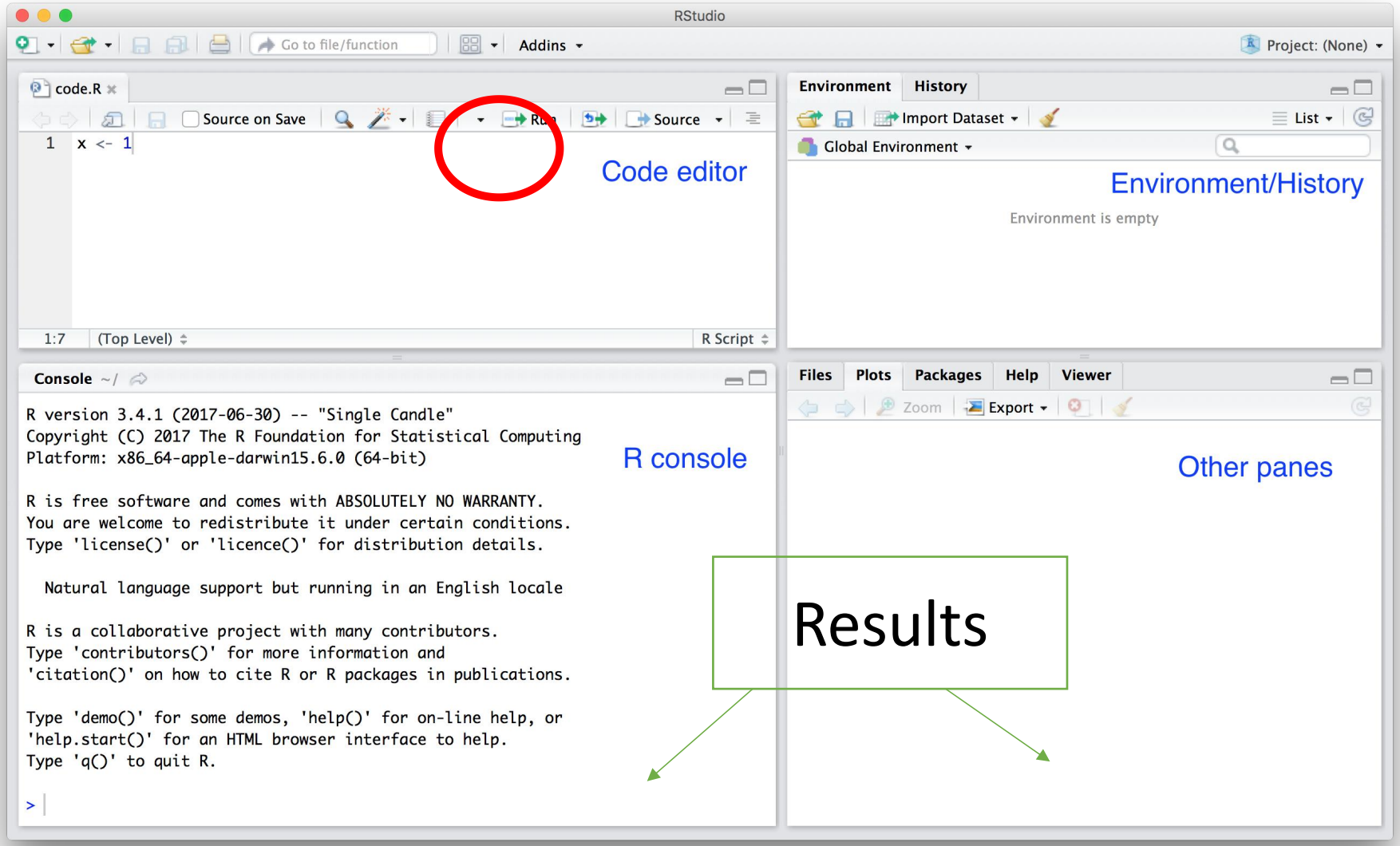
→ New File → Rscript →



# Let’s do some scripting



→ New File → Rscript →



# 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))
```

# Arithmetic Operators

Arithmetic Operators in R

| Operator | Description |
|---|---|
| + | Addition |
| – | Subtraction |
| * | Multiplication |
| / | Division |
| ^ | Exponent |
| %% | Modulus (Remainder from division) |
| %/% | Integer Division |

# Arithmetic Operators

```
> x <- 5
> y <- 16
> x+y
[1] 21
> x-y
[1] -11
> x*y
[1] 80
> y/x
[1] 3.2
> y%/%x
[1] 3
> y%%x
[1] 1
> y^x
[1] 1048576
```

# Relational Operators

*Relational operators are used to compare between values.*

Relational Operators in R

| Operator | Description |
|---|---|
| < | Less than |
| > | Greater than |
| <= | Less than or equal to |
| >= | Greater than or equal to |
| == | Equal to |
| != | Not equal to |

# Relational Operators

```
> x <- 5
> y <- 16
> x<y
[1] TRUE
> x>y
[1] FALSE
> x<=5
[1] TRUE
> y>=20
[1] FALSE
> y == 16
[1] TRUE
> x != 5
[1] FALSE
```

# Logical Operators

*Logical operators are used to carry out Boolean operations like* AND, OR *etc.*

Logical Operators in R

| Operator | Description |
|---|---|
| ! | Logical NOT |
| & | Element-wise logical AND |
| && | Logical AND |
| \| | Element-wise logical OR |
| \|\| | Logical OR |

Operators `&` and `|` perform element-wise operation producing result having length of the longer operand.

But `&&` and `||` examines only the first element of the operands resulting into a single length logical vector.

Zero is considered `FALSE` and non-zero numbers are taken as `TRUE`. An example run.

# Logical Operators

```
> x <- c(TRUE,FALSE,0,6)
> y <- c(FALSE,TRUE,FALSE,TRUE)
> !x
[1] FALSE  TRUE  TRUE FALSE
> x&y
[1] FALSE FALSE FALSE  TRUE
> x&&y
[1] FALSE
> x|y
[1]  TRUE  TRUE FALSE  TRUE
> x||y
[1] TRUE
```

# Assignment Operators

Assignment Operators in R

| Operator | Description |
| --- | --- |
| <-, <<-, = | Leftwards assignment |
| ->, ->> | Rightwards assignment |

The operators `<–` and `=` can be used, almost interchangeably, to assign to variable in the same environment.

The `<<–` operator is used for assigning to variables in the parent environments (more like global assignments). The rightward assignments, although available are rarely used.

# Assignment Operators

```
> x <- 5
> x
[1] 5
> x = 9
> x
[1] 9
> 10 -> x
> x
[1] 10
```

# 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)
```

# Data types

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

# 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

# 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.

# 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 "
```

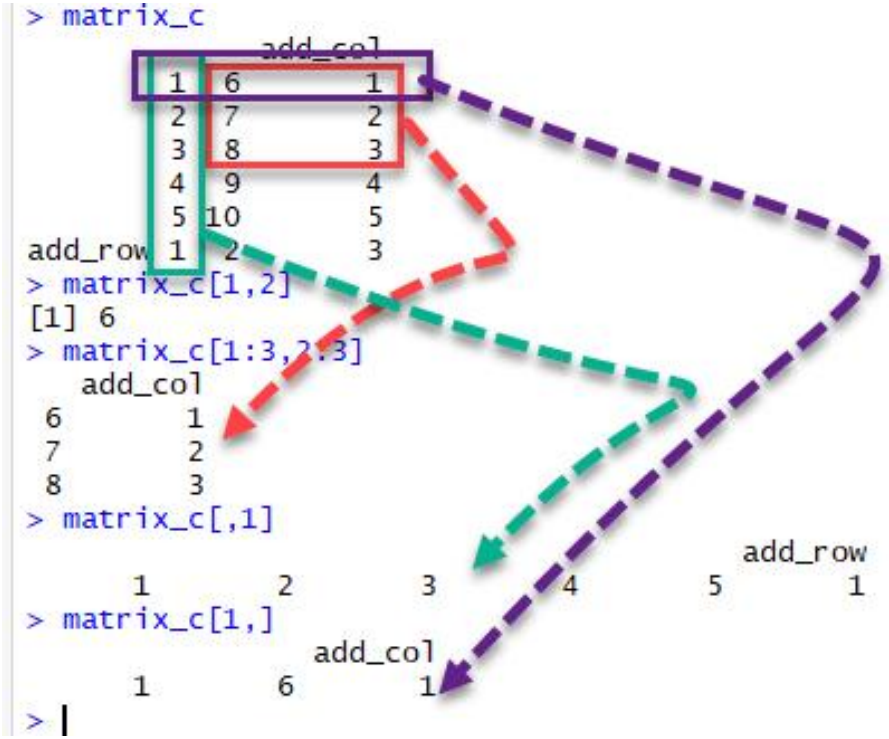
R *coerced* the data into characters.

```
> 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.

# 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.

# 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!

# data frame

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

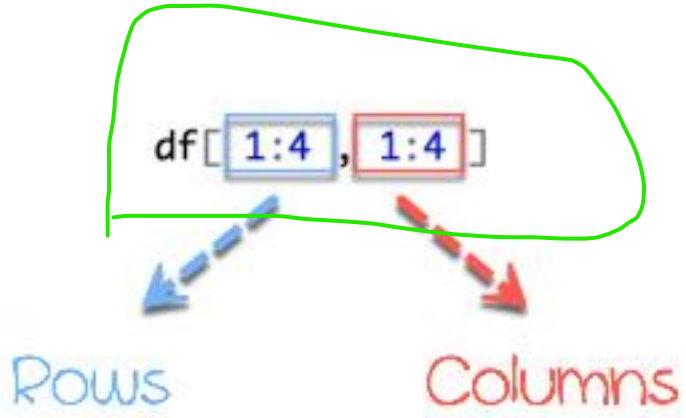
# How to Create a Data Frame

We can create a dataframe in R by passing the variable a,b,c,d into the data.frame() function. We can R create dataframe and name the columns with name() and simply specify the name of the variables.

```
data.frame(df, stringsAsFactors = TRUE)
```

```
# Create a, b, c, d variables
a <- c(10,20,30,40)
b <- c('book', 'pen', 'textbook', 'pencil_case')
c <- c(TRUE,FALSE,TRUE,FALSE)
d <- c(2.5, 8, 10, 7)
# Join the variables to create a data frame
df <- data.frame(a,b,c,d)
df
```

```
## a b c d
## 1 10 book TRUE 2.5
## 2 20 pen FALSE 8.0
## 3 30 textbook TRUE 10.0
## 4 40 pencil_case FALSE 7.0
```



# Slice Data Frame

```
## Select row 1 in column 2

df[1,2]
```

```
## [1] book
## Levels: book pen pencil_case textbook
```

| | ID | items | store | price |
|---|---|---|---|---|
| 1 | 10 | book | TRUE | 2.5 |
| 2 | 20 | pen | FALSE | 8.0 |
| 3 | 30 | textbook | TRUE | 10.0 |
| 4 | 40 | pencil_case | FALSE | 7.0 |

```
## Select Rows 1 to 3 and columns 3 to 4

df[1:3, 3:4]
```

```
## store price
## 1 TRUE 2.5
## 2 FALSE 8.0
## 3 TRUE 10.0
```

```
## Select Rows 1 to 2

df[1:2,]
```

```
## ID items store price
## 1 10 book TRUE 2.5
## 2 20 pen FALSE 8.0
```

```
## Select Column 1

df[,1]
```

```
## [1] 10 20 30 40
```

# Append a Column to Data Frame

```
# Create a new vector
quantity <- c(10, 35, 40, 5)
# Add `quantity` to the `df` data frame
df$quantity <- quantity
df
```

Output:

```
## ID items store price quantity
## 1 10 book TRUE 2.5 10
## 2 20 pen FALSE 8.0 35
## 3 30 textbook TRUE 10.0 40
## 4 40 pencil_case FALSE 7.0 5
```

# Subset a Data Frame

We use the subset() function.

subset(x, condition)

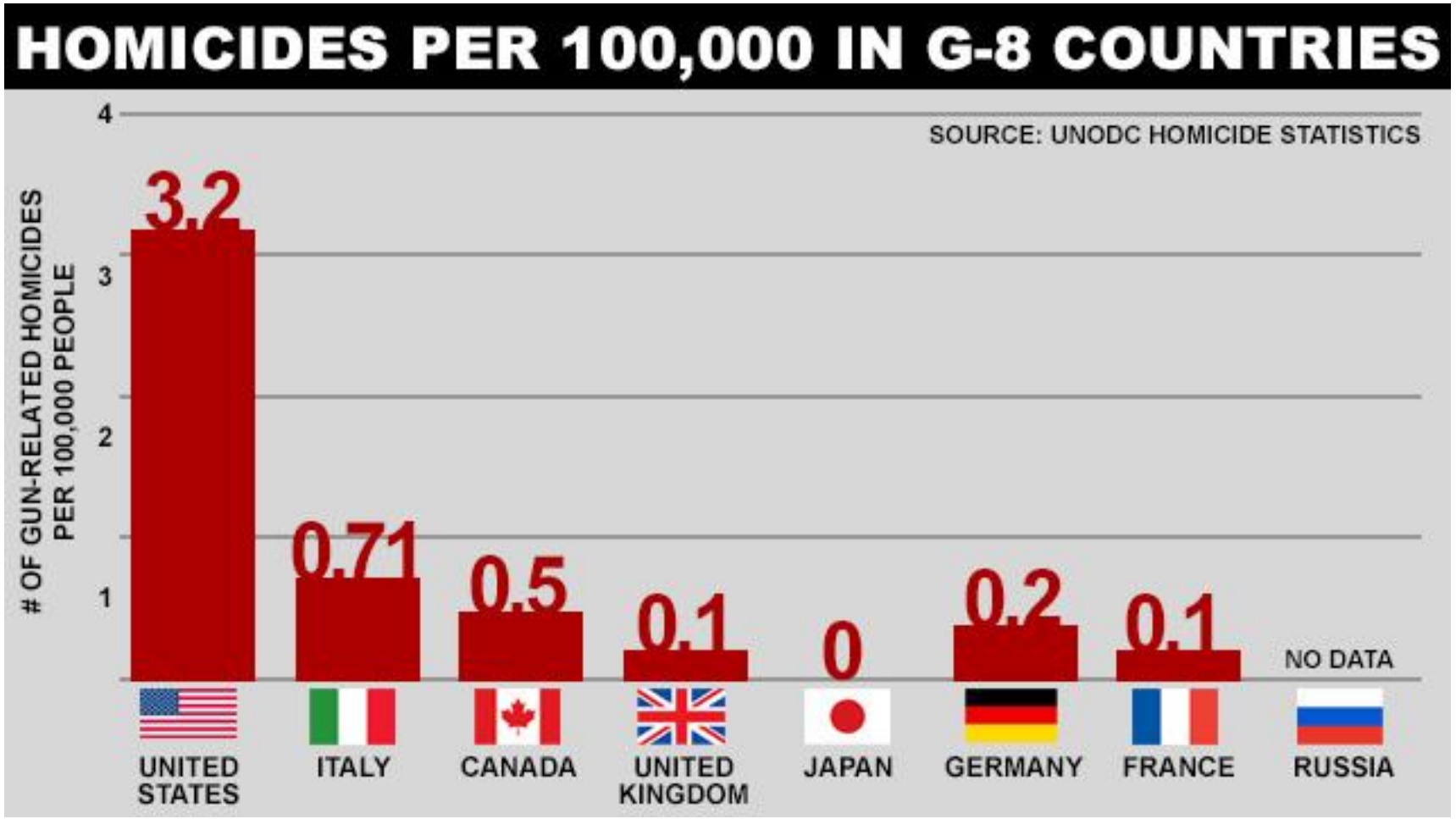
arguments:
- x: data frame used to perform the subset
- condition: define the conditional statement

```
# Select price above 5
subset(df, subset = price > 5)
```

*Output:*

```
ID items store price
2 20 pen FALSE 8
3 30 textbook TRUE 10
4 40 pencil_case FALSE 7
```

# 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* :

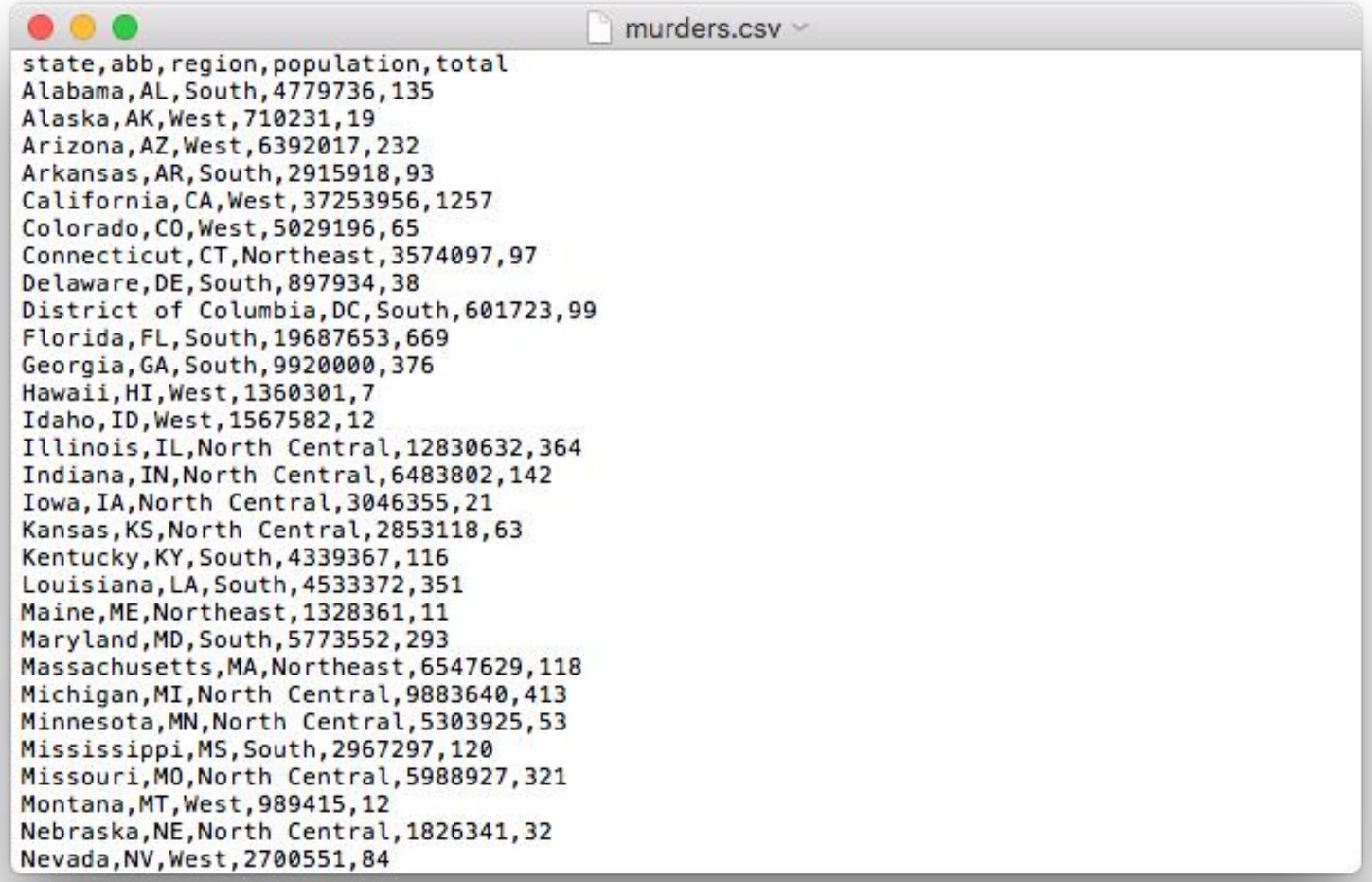
```
> names(murders)
```

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

```
> murders$population
```

# 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).



# 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 |
| read_delim | general text file format, must define delimiter | txt |

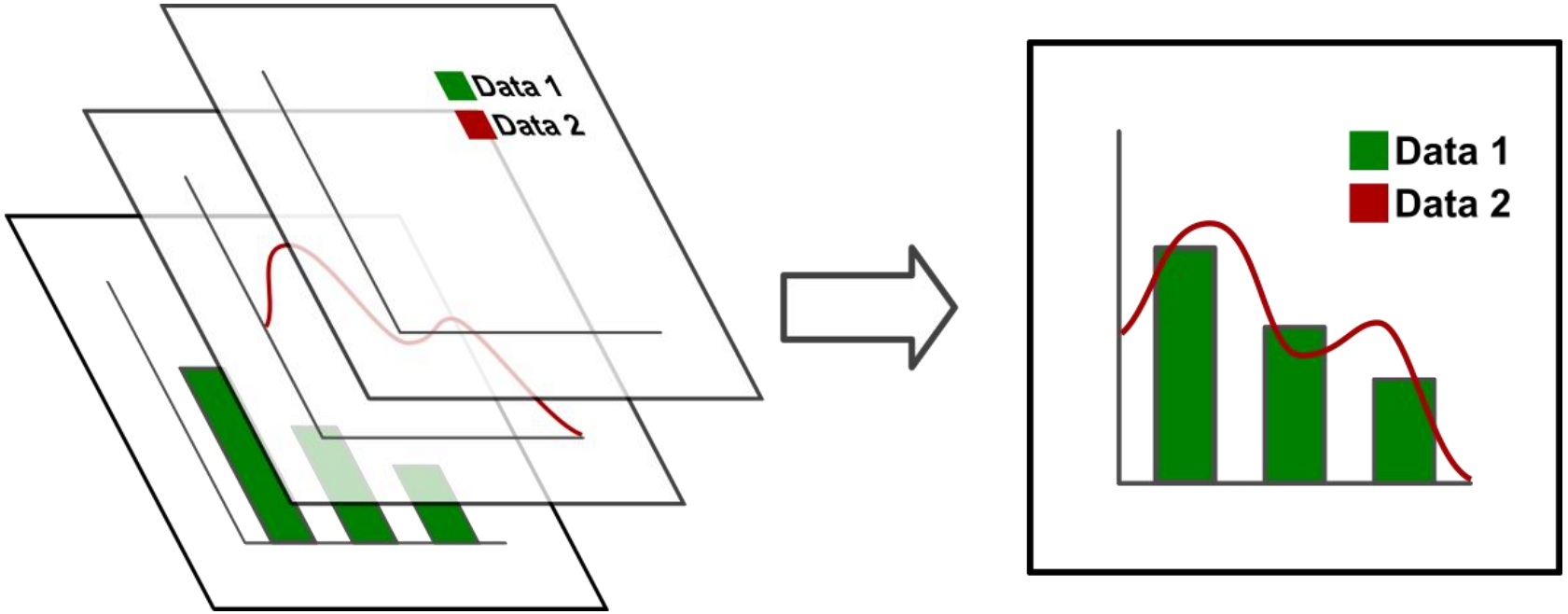
readxl

| 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)
```

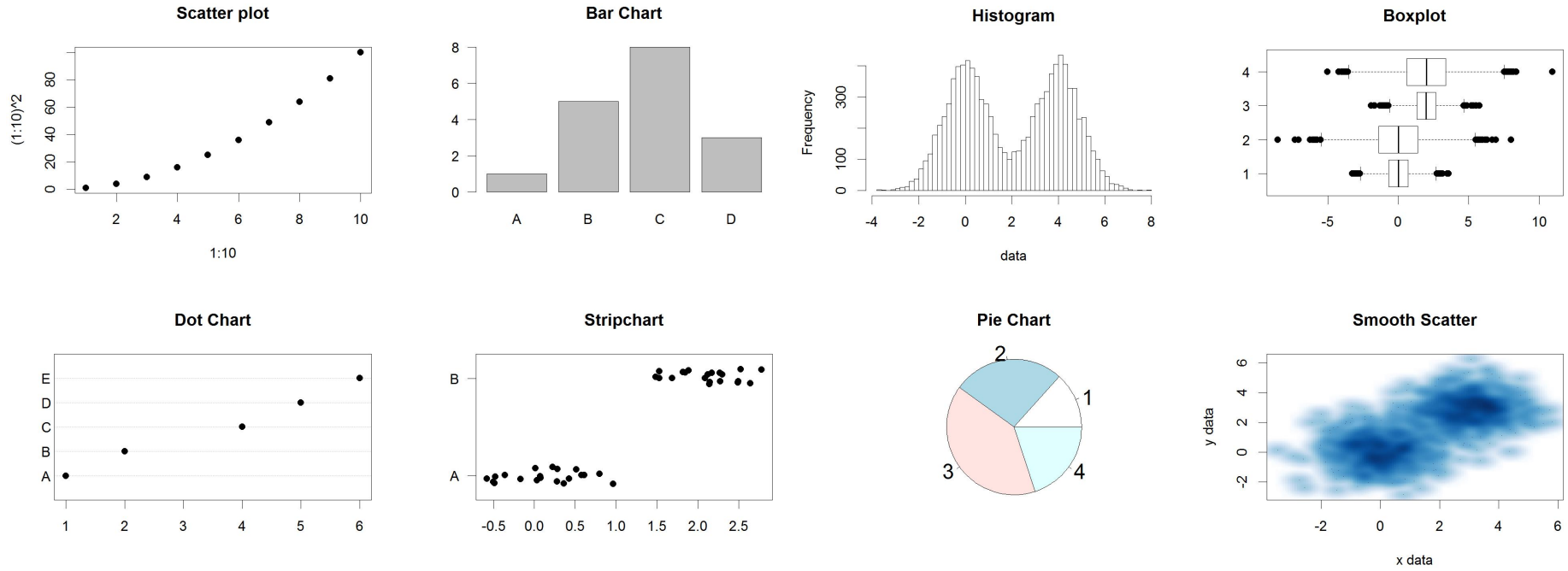
R-base also provides import functions. These have similar names to those in the **tidyverse**, for example **read.table**, **read.csv** and **read.delim**

# The R Painters Model




- Plot area
- Base plot
- Overlays

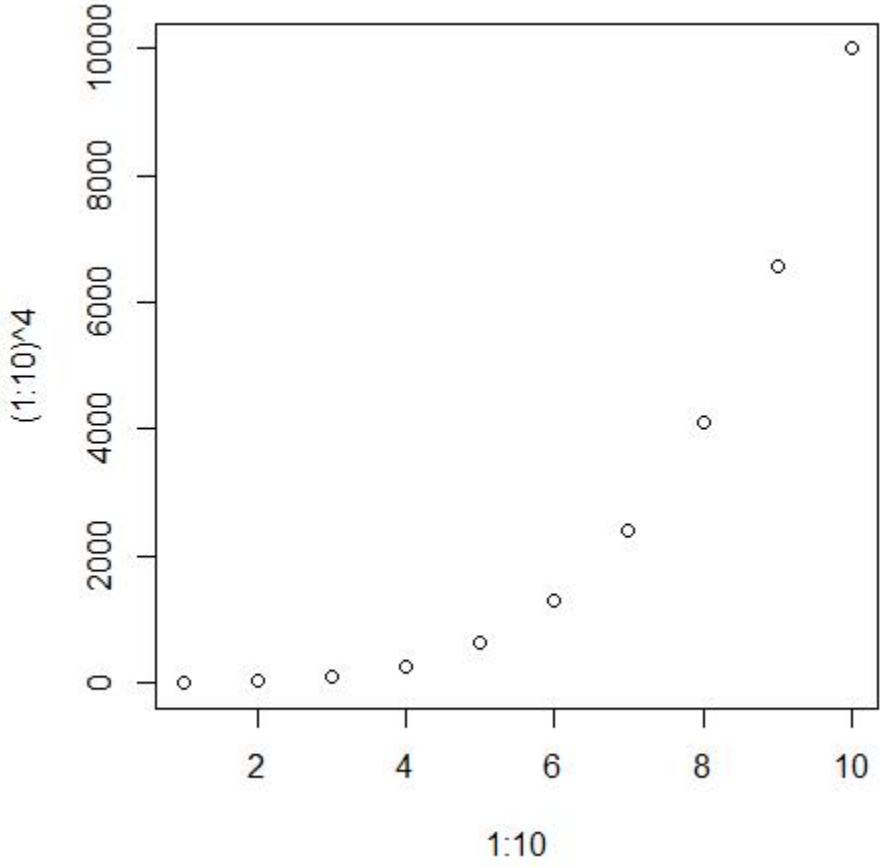
# Core Graph Types




- Local options to change a specific plot
- Global options to affect all graphs

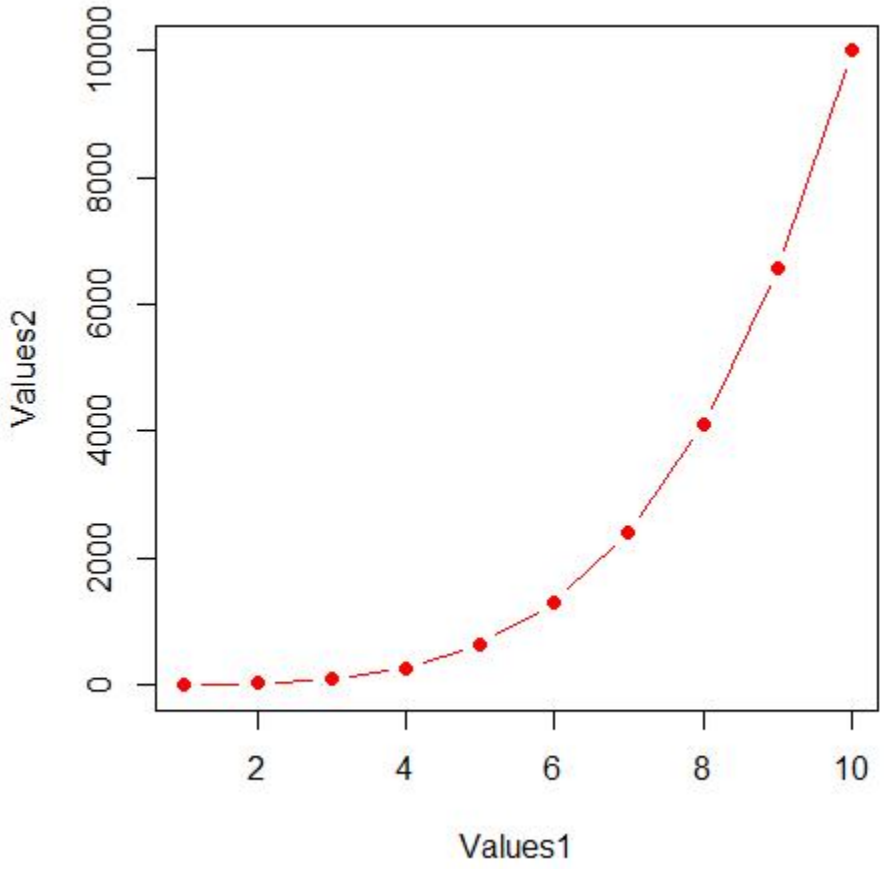
# Figures are configured based on the options passed to them

```
plot(
  1:10,(1:10) ^ 4
  )
```



# Figures are configured based on the options passed to them

```
plot(
  1:10,(1:10) ^ 4,
  pch=19,
  type="b",
  xlab="Values1",
  ylab="Values2",
  col="red"
  )
```

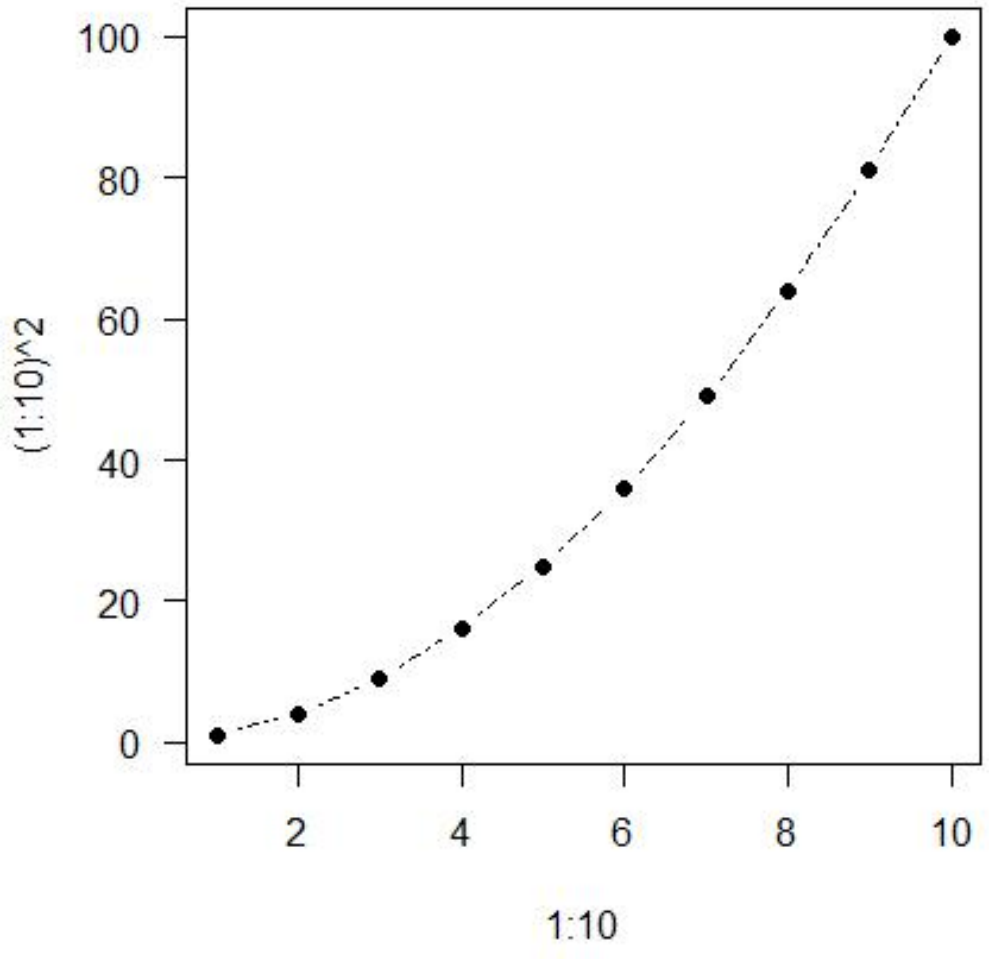


# Some options are common to many plot types

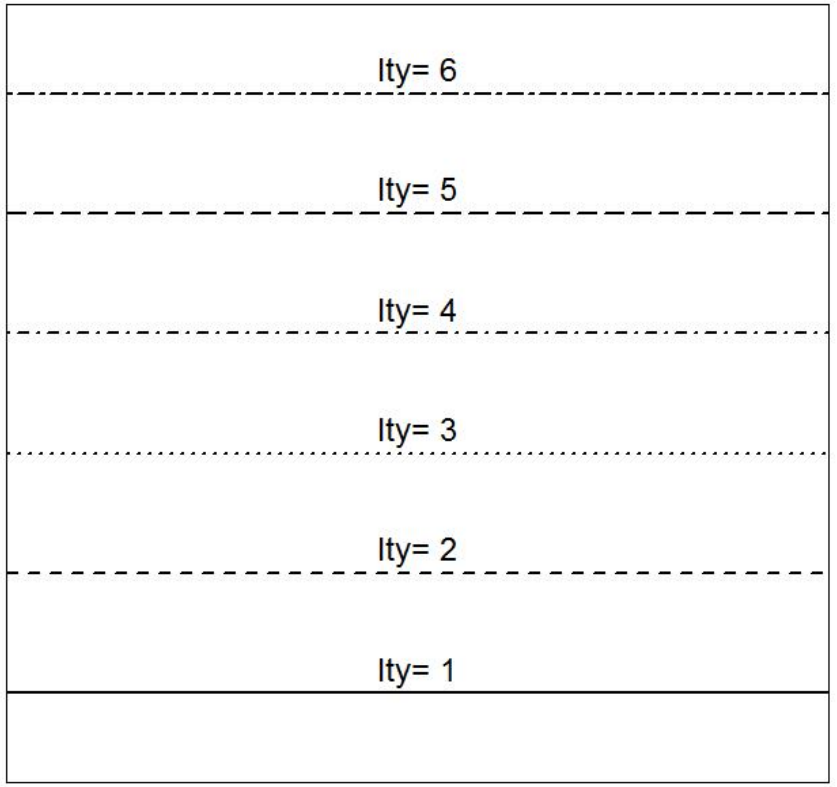
- Axis scales
  - `xlim c(min,max)`
  - `ylim c(min,max)`
- Axis labels
  - `xlab(text)`
  - `ylab(text)`
- Plot titles
  - `main(text)`
  - `sub(text)`
- Plot characters
  - `pch(number)`
  - `cex(number)`

# Some options take 'magic' numbers

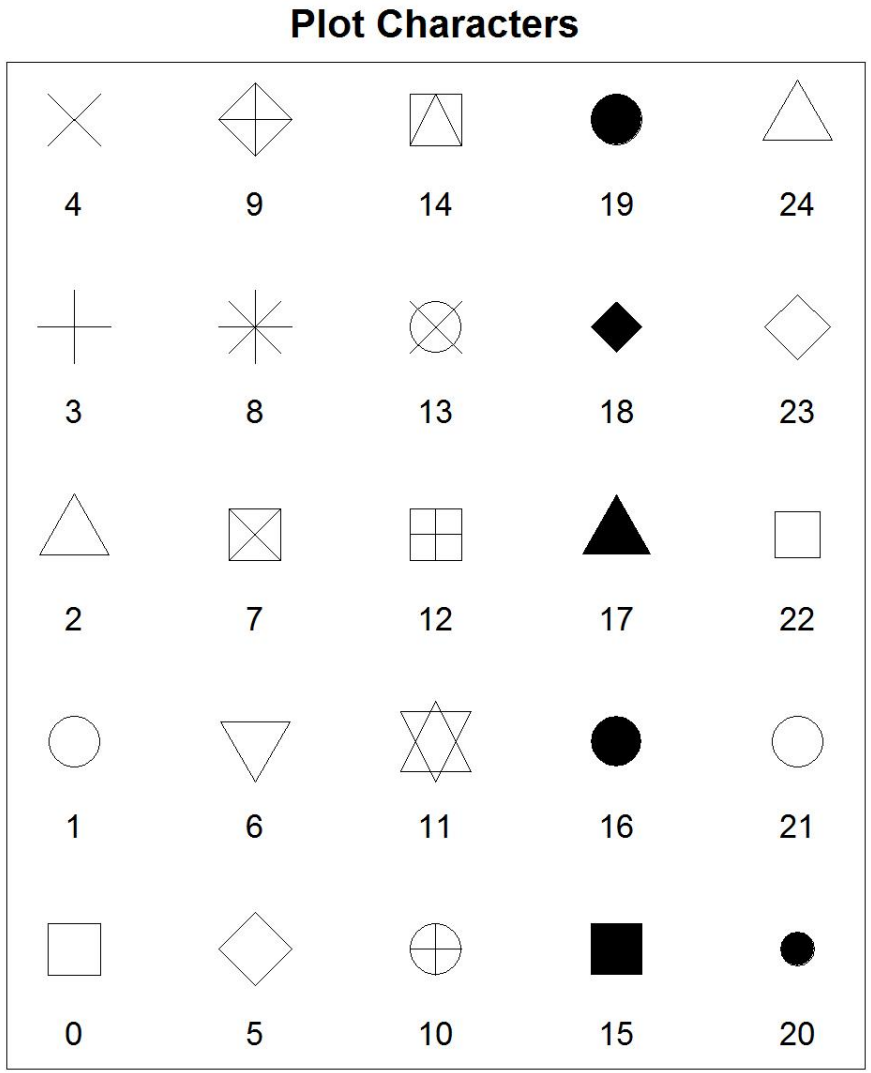
```
plot(
  1:10,
  (1:10)^2,
  type="b",
  lty=2,
  pch=19
)
```



# Line types

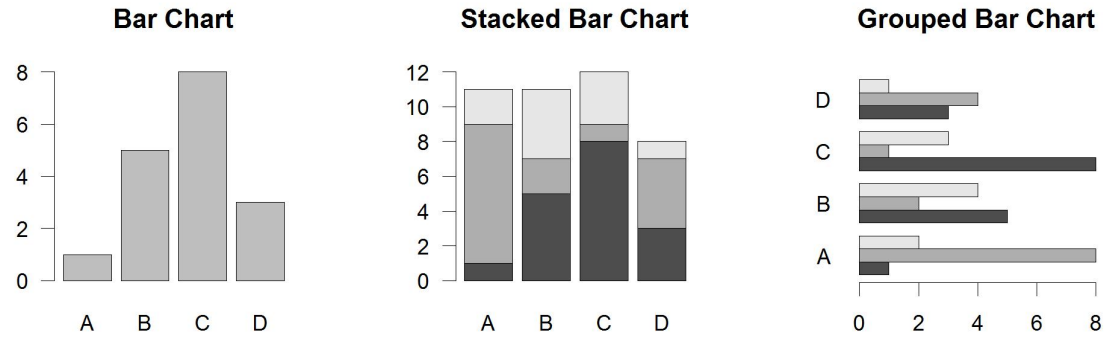


# Plot Characters




pch=

# Some options are specific to one graph type (eg barplot)




- Options:
  - `names.arg` Bar labels (if not from data)
  - `horiz=TRUE` Plot horizontally
  - `beside=TRUE` Plot multiple series as a group rather than stacked

# Par

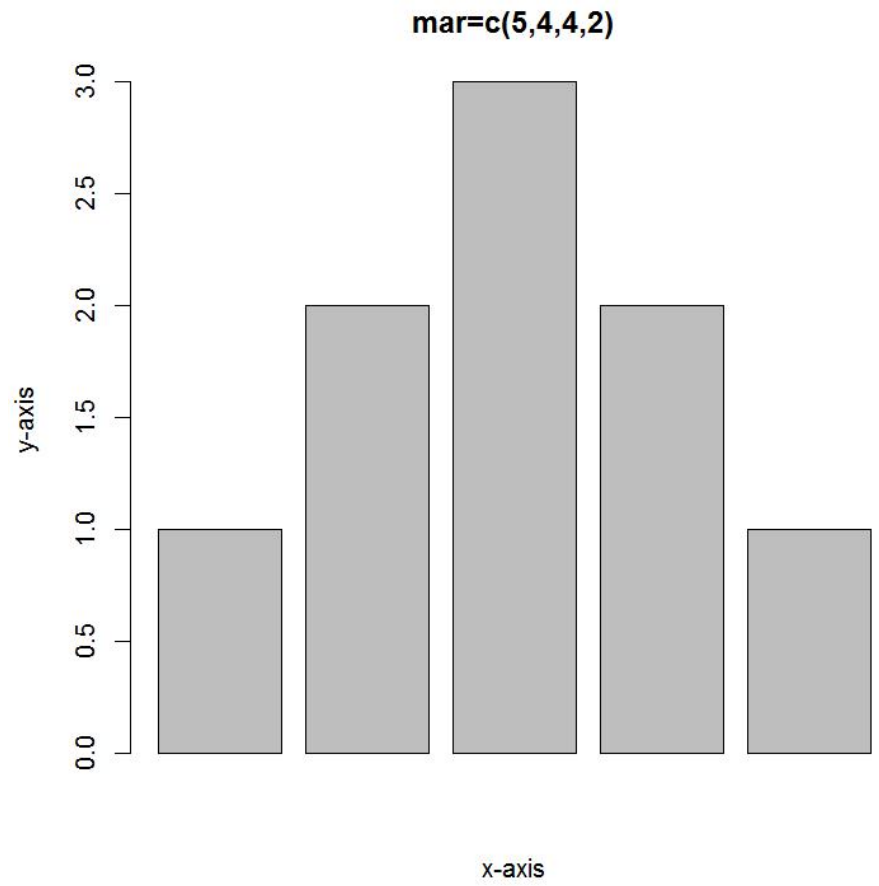
- The `par` function controls global parameters affecting all plots in the current plot area
- Changes affect all subsequent plots
- Many par options can also be passed to individual plots

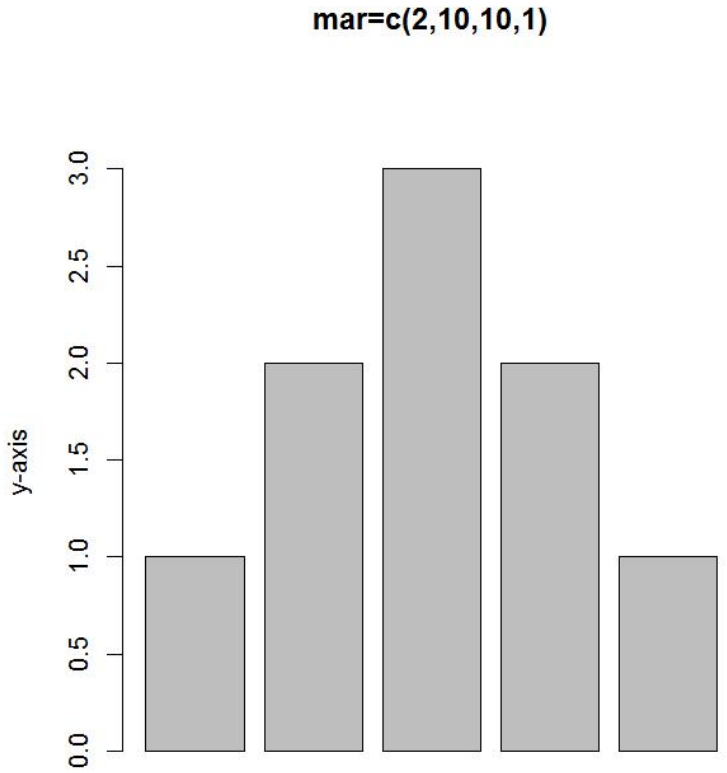
# Par examples

- Reading current value
  - `par()$cex`
- Setting a value
  - `par(cex=1.5) -> old.par`
- Restoring a value
  - `par(old.par)`
  - `dev.off()`

# Par options

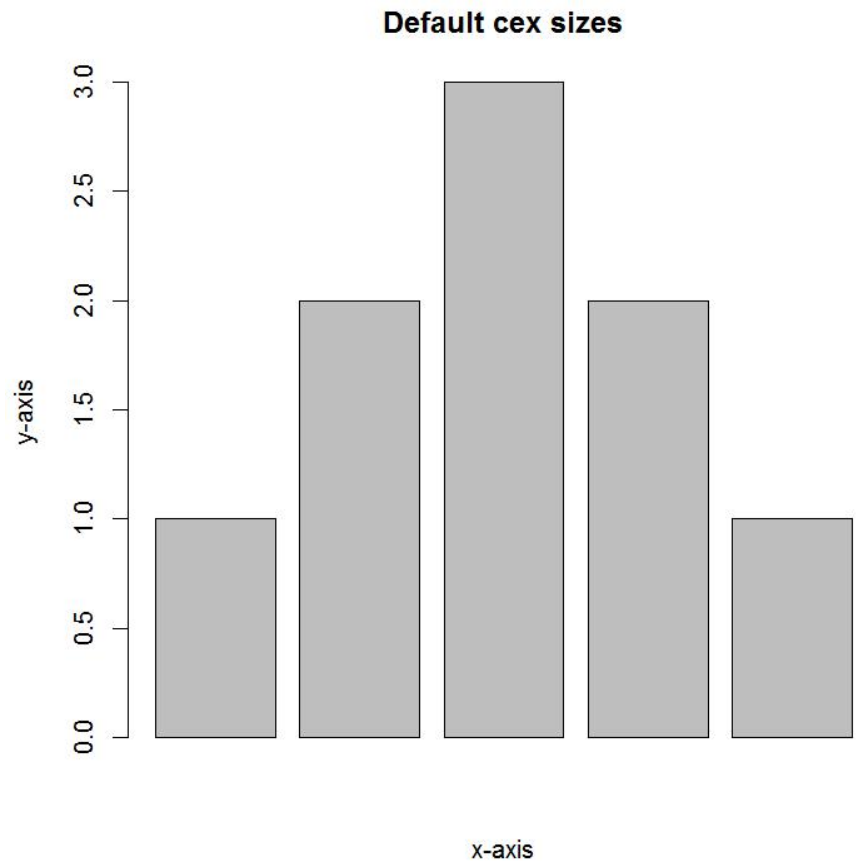
- Margins
  - `mai` (set margins in inches)
  - `mar` (set margins in number of lines)
  - `mex` (set lines per inch)
  - 4 element vector (bottom, left, top, right)
- Warning
  - `Error in plot.new() : figure margins too large`

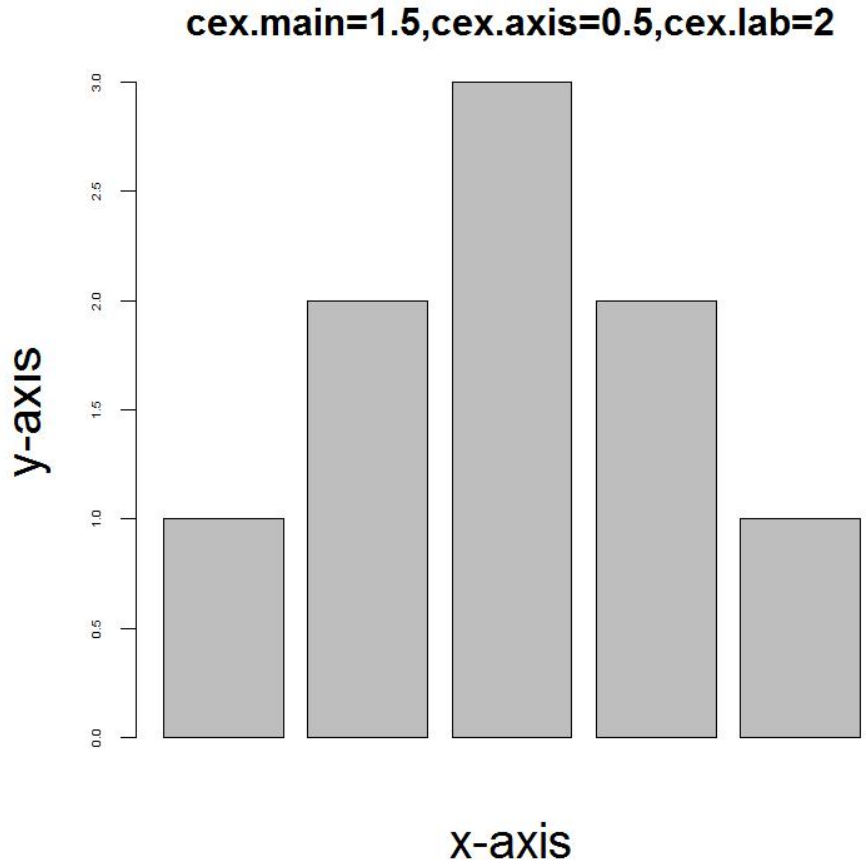

mar=c(5,4,4,2)
y-axis
0.0
0.5
1.0
1.5
2.0
2.5
3.0
x-axis


mar=c(2,10,10,1)
y-axis
0.0
0.5
1.0
1.5
2.0
2.5
3.0

# Par options

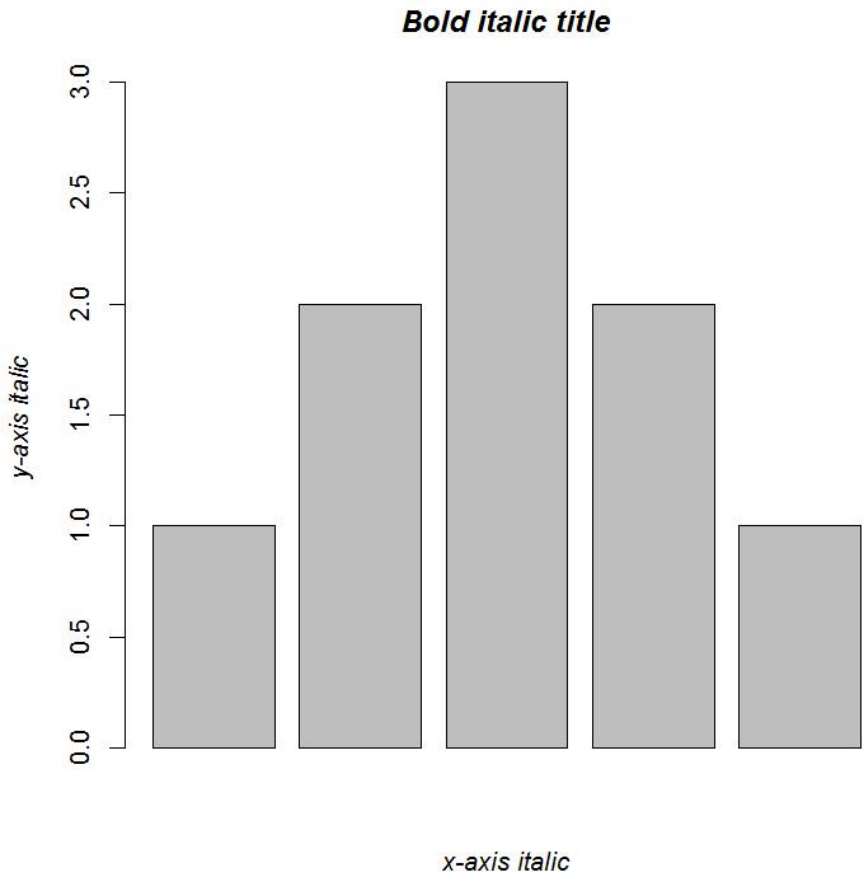
- Fonts and labels
  - `cex` – global char expansion
    - `cex.axis`
    - `cex.lab`
    - `cex.main`
    - `cex.sub`

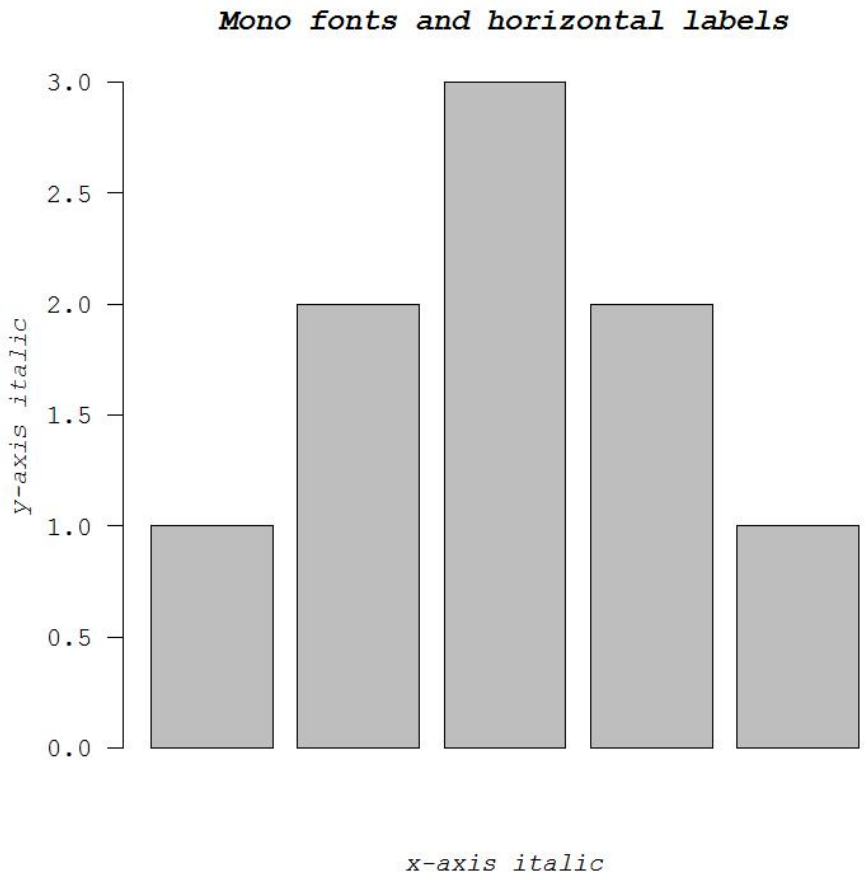

Default cex sizes
y-axis
0.0
0.5
1.0
1.5
2.0
2.5
3.0
x-axis


cex.main=1.5,cex.axis=0.5,cex.lab=2
y-axis
0.0
0.5
1.0
1.5
2.0
2.5
3.0
x-axis

# Par options

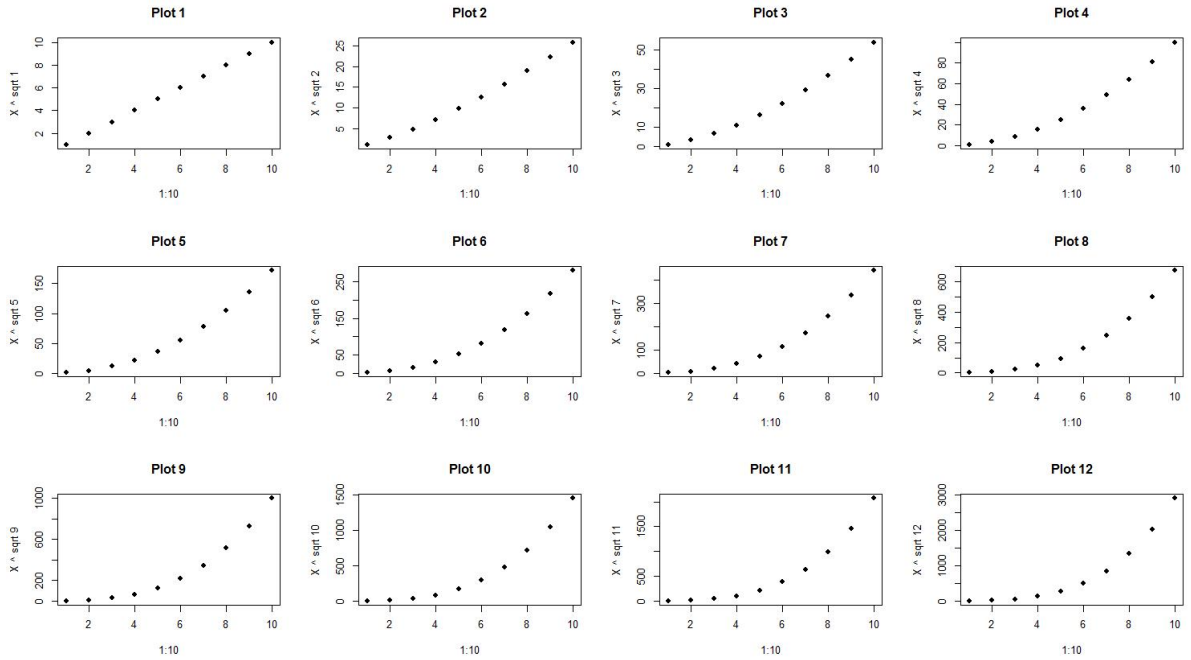
- Font style
  - `font` (`font.axis`, `font.main`, `font.sub`, `font.lab`)
    - 1 = Plain text
    - 2 = Bold text
    - 3 = Italic text
    - 4 = Bold italic text
  - `las` (label orientation)
    - 0 = Parallel to axis
    - 1 = Horizontal
    - 2 = Perpendicular
    - 3 = Vertical


Bold italic title
y-axis italic
0.0
0.5
1.0
1.5
2.0
2.5
3.0
x-axis italic


Mono fonts and horizontal labels
y-axis italic
0.0
0.5
1.0
1.5
2.0
2.5
3.0
x-axis italic

# Par options

- Multi-panel
  - `mfrow(rows,cols)`
  - Not supported by some packages

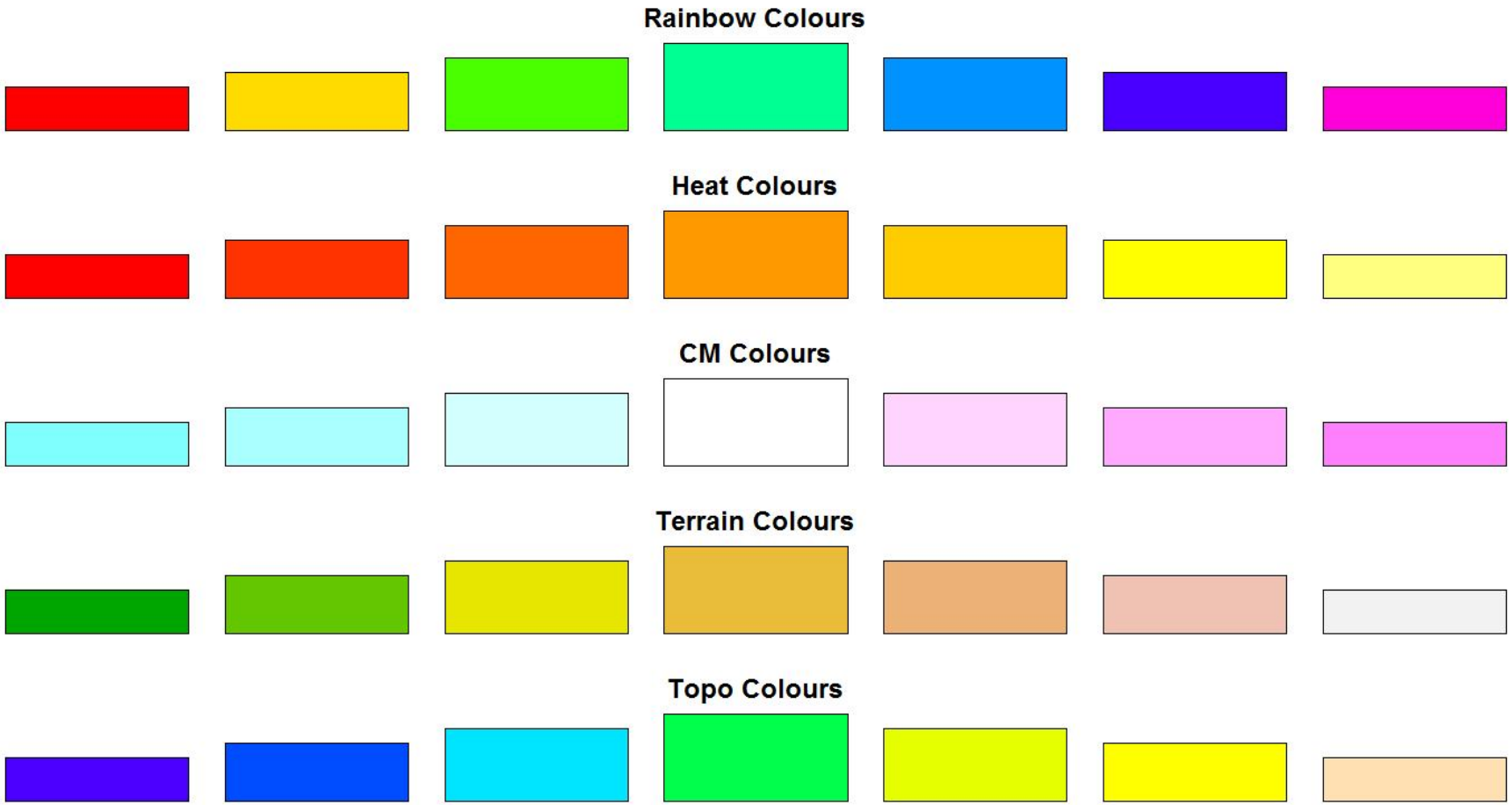


# Specifying colours

- Hexadecimal strings
  - `#FF0000` (red)
  - `#0000FF` (blue)
  - `#CC00CC` (purple)
- Controlled names
  - `“red” “green”` etc.
  - `colors()`

# Built in colour schemes

- Functions to generate colours
- Pass in number of colours to make
- Functions:
  - `rainbow`
  - `heat.colors`
  - `cm.colors`
  - `terrain.colors`
  - `topo.colors`


Rainbow Colours
Heat Colours
CM Colours
Terrain Colours
Topo Colours

# Colour Packages

- Color Brewer
  - Set of pre-defined, optimised palettes
  - `library(RColorBrewer)`
  - `brewer.pal(no colours, palette)`

- ColorRamps
  - Create smooth palettes for ramped colour
  - Generates a function to make actual colour vectors
  - `colorRampPalette(c("red","white","blue"))`
  - `colorRampPalette(c("red","white","blue"))(5)`

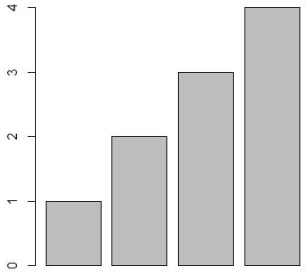
# Applying Colour to Plots

- Vector of colours passed to the `col` parameter

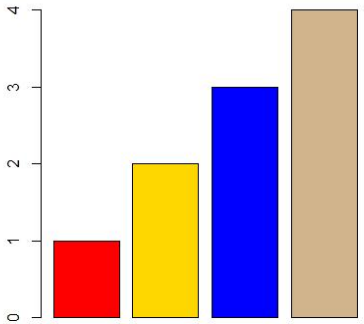
- Vector of factors used to divide the data
  - Colours taken from palette
  - Can read or set using palette function
    - `palette()`
    - `palette(brewer.pal(9,"Set1")`
    - Ordered by levels of factor vector

# Applying Colour to Plots

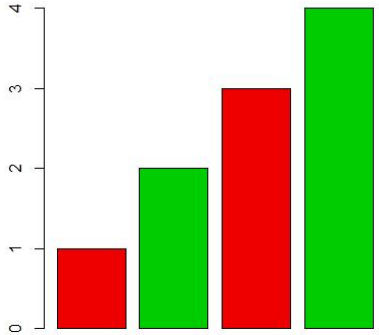
```
barplot(1:4)
```



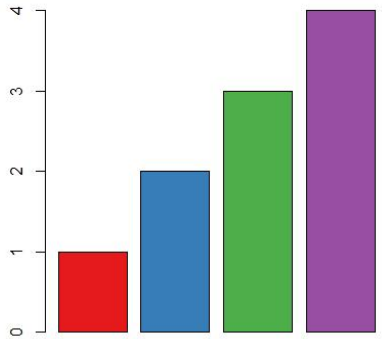

```
barplot(
  1:4,
  col=c("red","gold","blue","tan")
)
```




```
barplot(
  1:4,
  col=c("red2","green3")
)
```




```
library(RColorBrewer)
barplot(
  1:4,
  col=brewer.pal(4,"Set1")
)
```



# Applying Colour to Plots

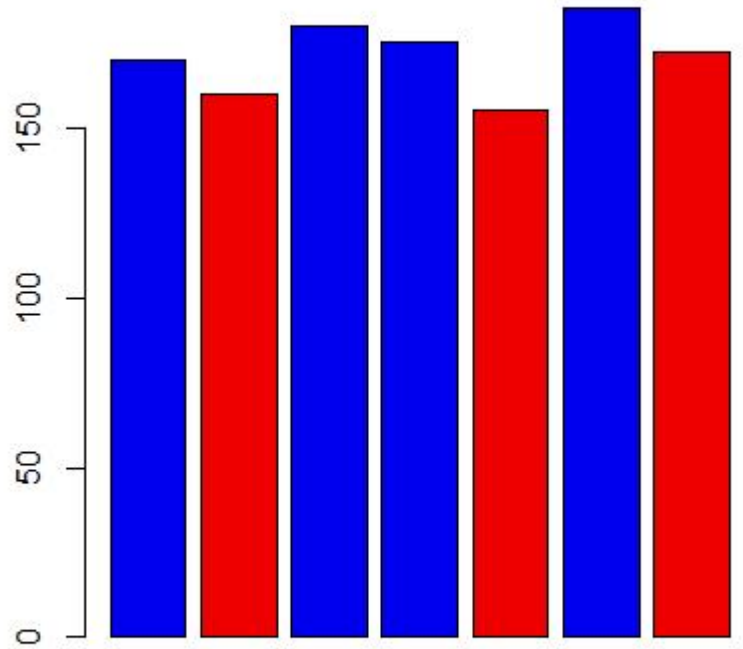
```
> height.data
  height sex
1    170   M
2    160   F
3    180   M
4    175   M
5    155   F
6    185   M
7    172   F
```

```
> palette()
[1] "black" "red"     "green3"  "blue"
[5] "cyan"  "magenta" "yellow"  "gray"
```

```
> levels(height.data$sex)
[1] "F" "M"
```

```
> palette(c("red2","blue2"))
```

```
barplot(height.data$height,col=height.data$sex)
```



# Dynamic use of colour

- Colouring by density
  - Pass data and palette to `densCols`
  - Vector of colours returned

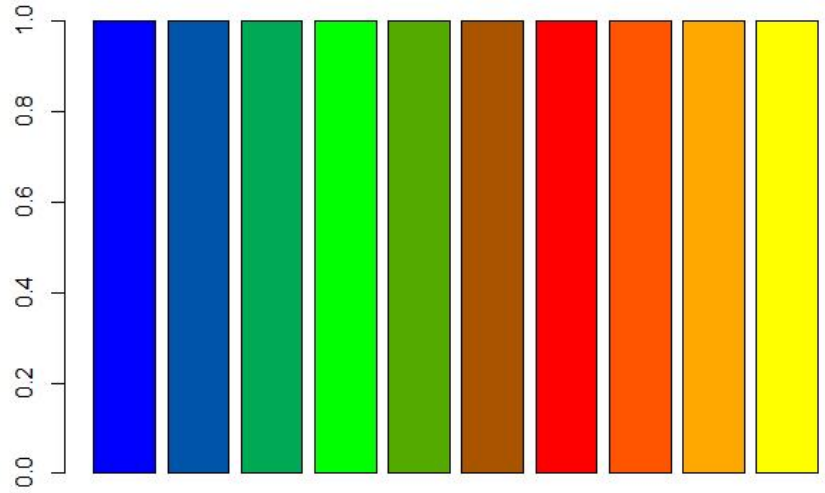
- Colouring by value
  - Need function to map values to colours

# Making colour ramps

```
> colorRampPalette(c("blue","green","red","yellow"))
function (n)
{
    x <- ramp(seq.int(0, 1, length.out = n))
    if (ncol(x) == 4L)
        rgb(x[, 1L], x[, 2L], x[, 3L], x[, 4L], maxColorValue = 255)
    else rgb(x[, 1L], x[, 2L], x[, 3L], maxColorValue = 255)
}
```

```
> colorRampPalette(c("blue","green","red","yellow"))(10)
 [1] "#0000FF" "#0055AA" "#00AA55" "#00FF00" "#55AA00" "#AA5400" "#FF0000" "#FF5400" "#FFA900" "#FFFF00"
```

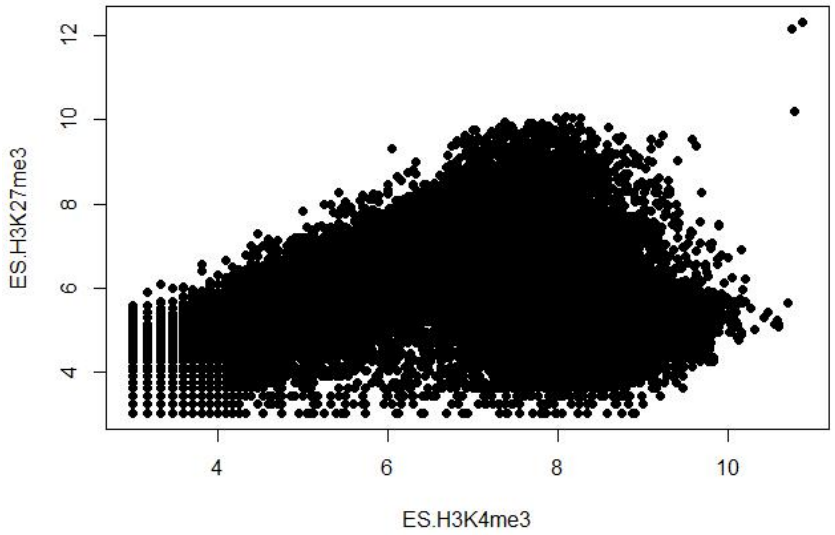
```
> barplot(
  rep(1,10),
  col=colorRampPalette(
        c("blue","green","red","yellow")
       )(10)
  )
```

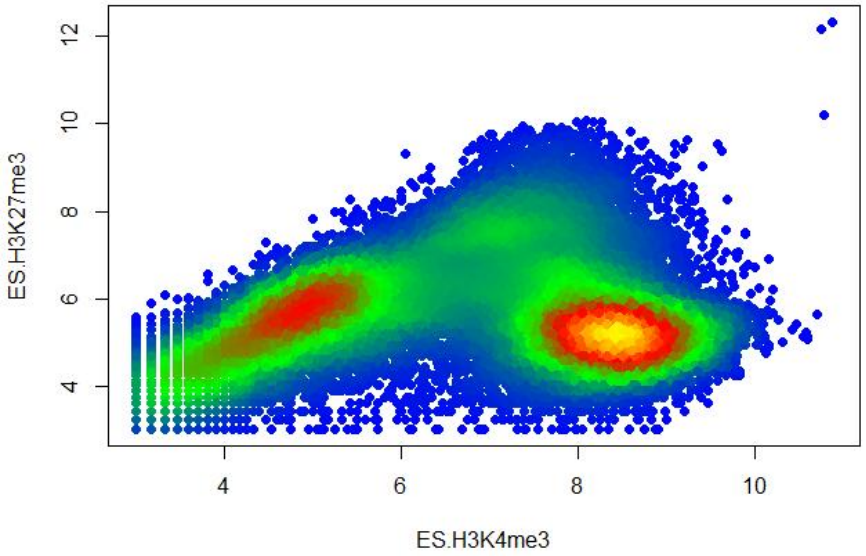


# Using colour to plot density

```
plot(lots.of.data, pch=19)
```

```
plot(
  lots.of.data,
  pch=19,
  col=densCols(
    lots.of.data,
    colramp=colorRampPalette(c(
      "blue","green","red","yellow")
      )
    )
)
```



# Colour Mapping Function

```
map.colours <- function(values,palette) {

  range <- range(values)

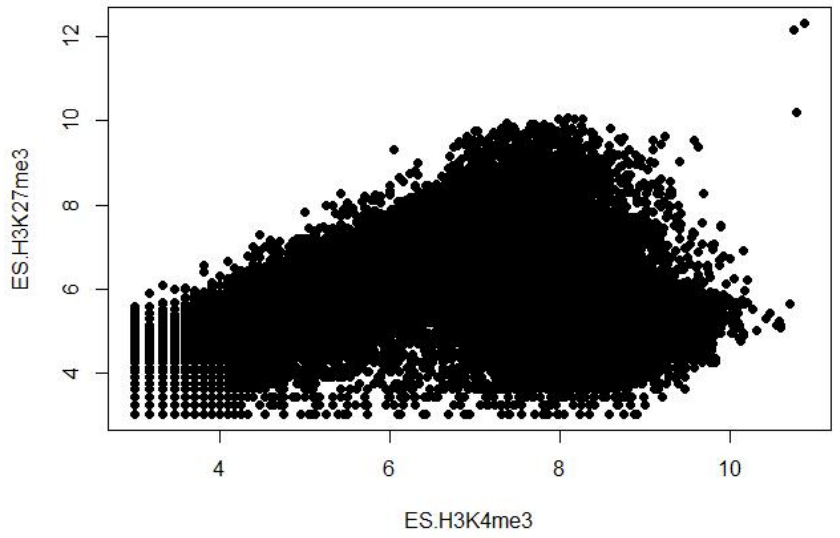
  proportion <- (values-range[1])/(range[2]-range[1])
  index <- round((length(palette)-1)*proportion)+1

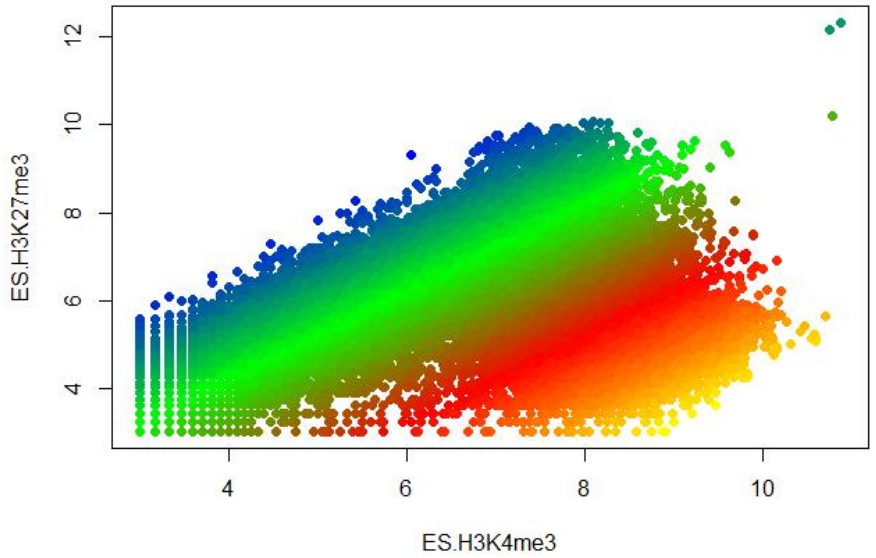
  return(palette[index])
}
```

# Plotting Quantitative Colour

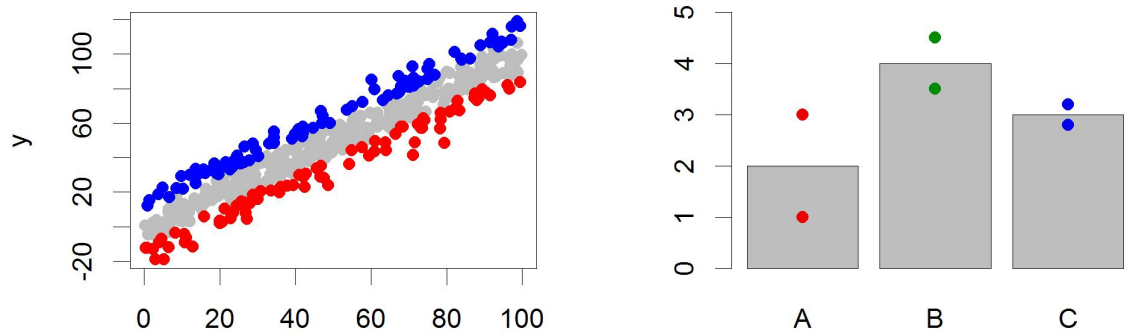
```
plot(lots.of.data, pch=19)
```

```
plot(
  lots.of.data,
  pch=19,
  col=map.colours(
    lots.of.data$K4 - lots.of.data$K27,
    colorRampPalette(c(
      "blue","green","red","yellow")
    )(100)
  )
)
```



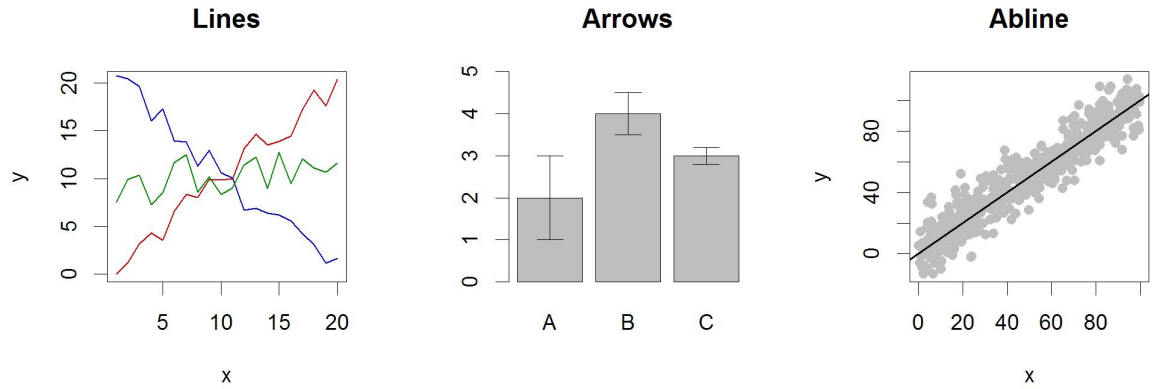



# Points




- Input: 2 Vectors (x and y positions)
- Options:
  - `pch`
  - `cex`

# Lines / Arrows / Abline



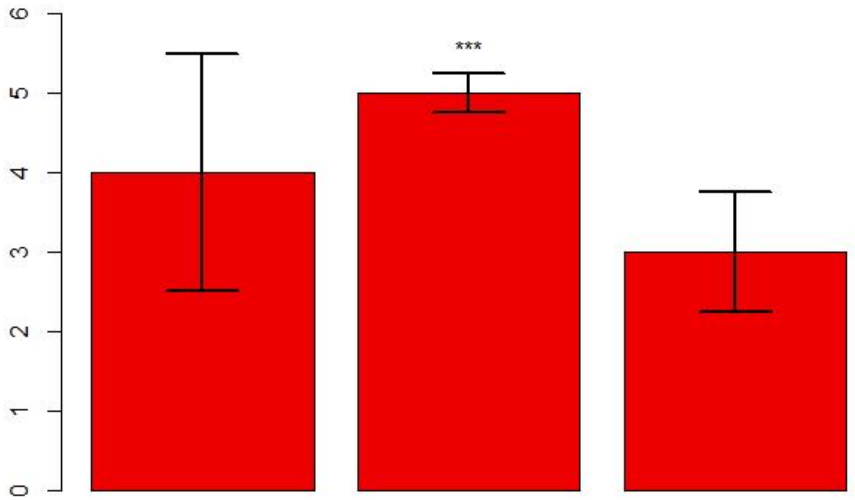

- Input:
  - Lines 2 vectors (x and y)
  - Arrows 4 vectors (x0,y0,x1,y1)
  - Abline Intercept and slope (or correlation object)
- Options:
  - `lwd`
  - `angle` (arrows)

# Example multi-layer plot

```
barplot(
  error.data$values,
  col="red2",
  ylim=(c(0,6))
) -> bar.centres

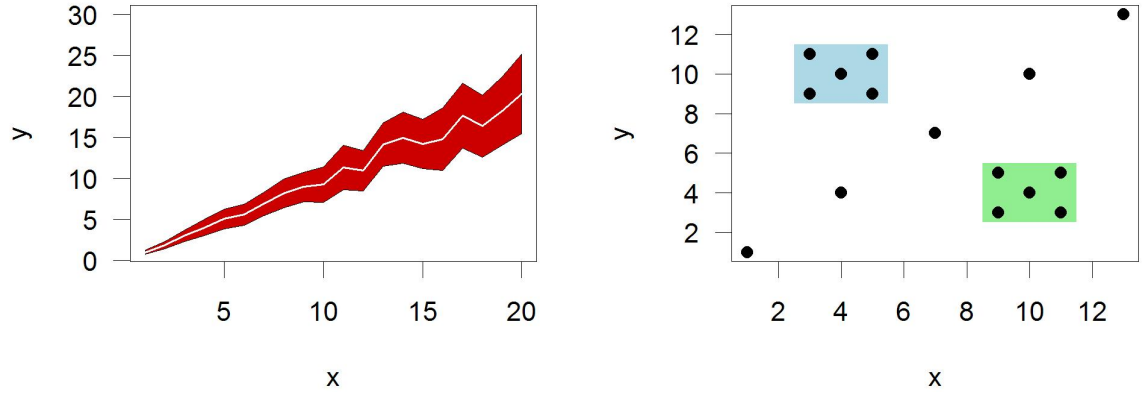
arrows(
  x0=bar.centres,
  y0=error.data$values - error.data$sem,
  x1=bar.centres,
  y1=error.data$values + error.data$sem,
  angle=90,
  code = 3,
  lwd=2
)

text(
  bar.centres[2],
  y = error.data$values[2] + error.data$sem[2],
  labels = "***",
  pos=3
)
```



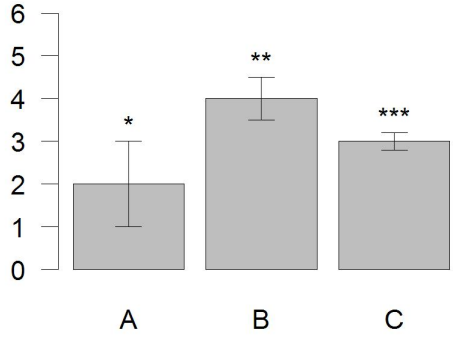

```
> error.data
  values    sem
1      4   1.50
2      5   0.25
3      3   0.75
```

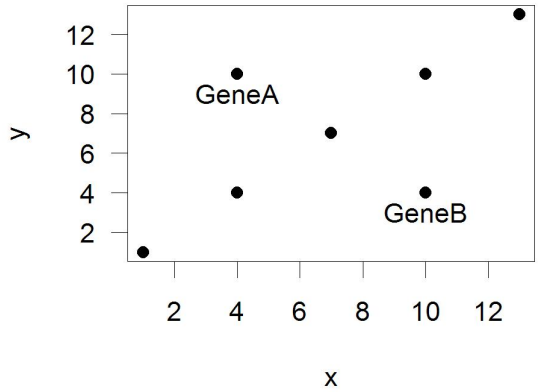
# Polygon (shaded areas)




- Input:
  - 2 vectors (x and y) for bounding region
- Options:
  - `col`

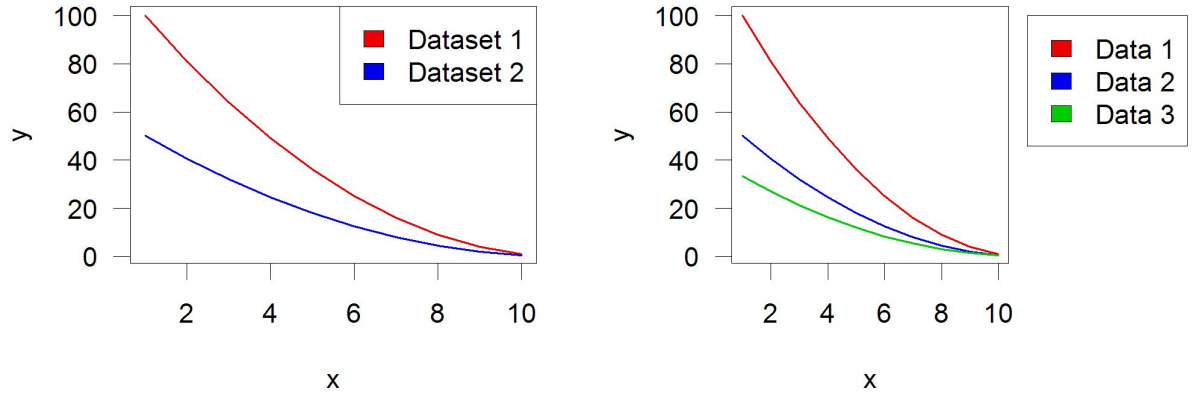
# Text (in plot text)




- Input:
  - Text, x, y
- Options:
  - `adj` (x and y offsets)
  - `pos` (auto offset 1=below,2=left,3=above, 4=right)

# Legend

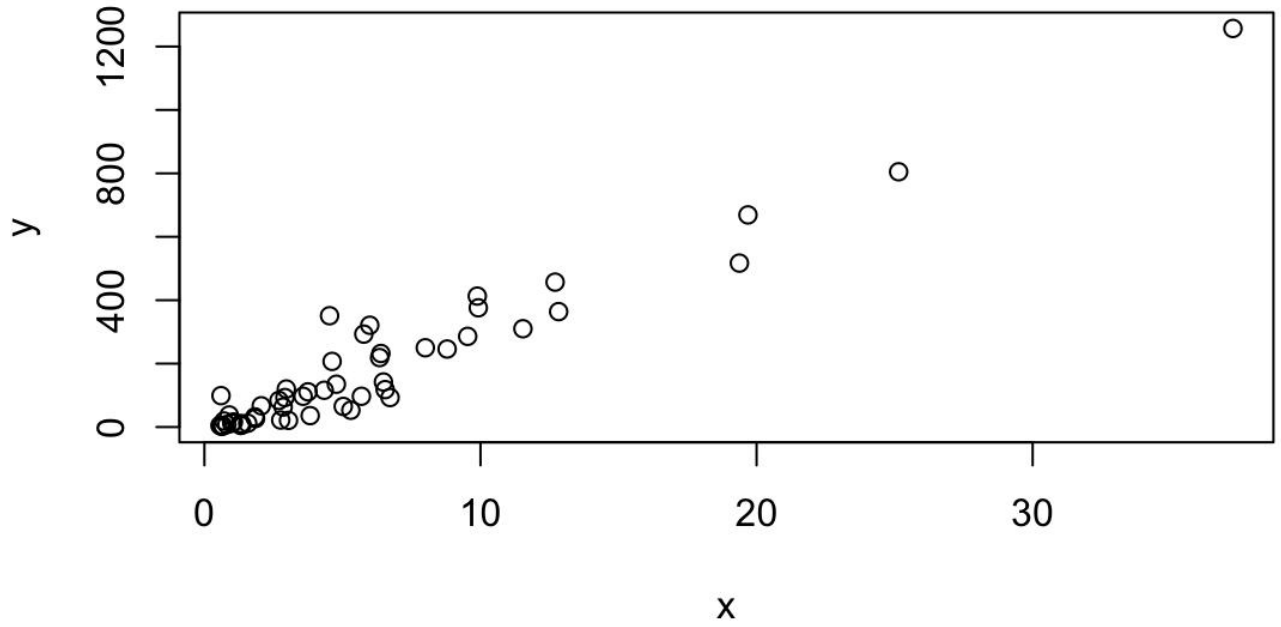



- Input:
  - Position (x,y or “topright”,”bottomleft” etc)
  - Text labels
- Options:
  - `fill` (colours for shaded boxes)
  - `xpd=NA` (draw outside plot area)

# Scatterplot

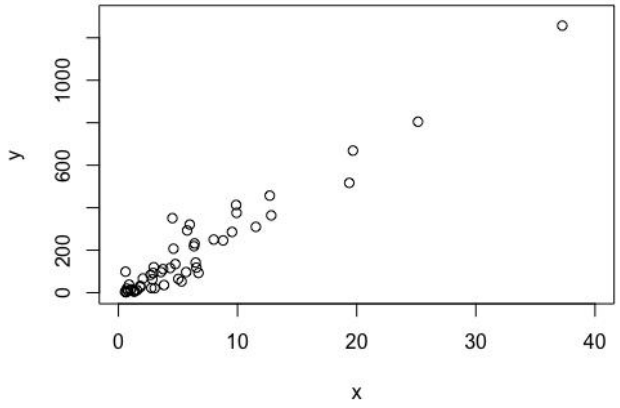
The plot function can be used to make scatterplots.

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

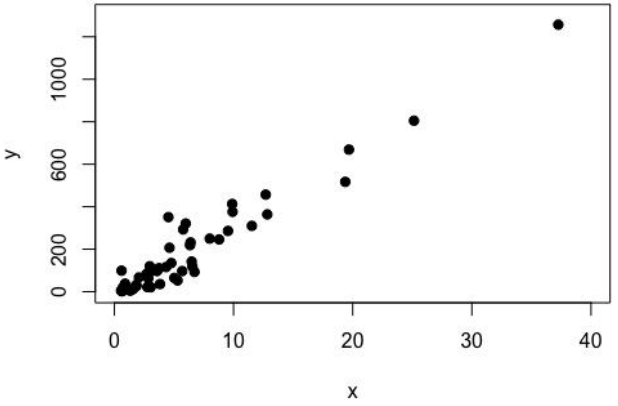


# Scatterplot

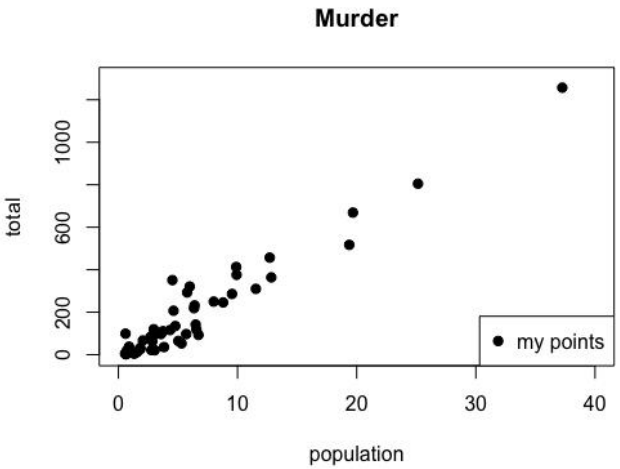
```
> plot(x, y, xlim=c(0, 40),
ylim=c(0,1300))
```



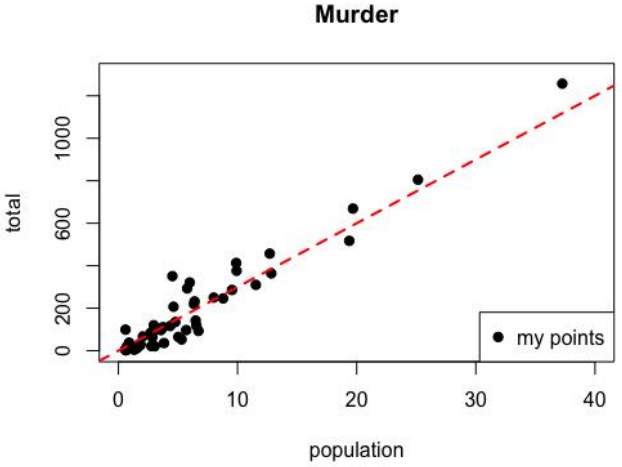

```
> plot(x, y, xlim=c(0, 40),
ylim=c(0,1300), pch=19)
```




```
> 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)
```



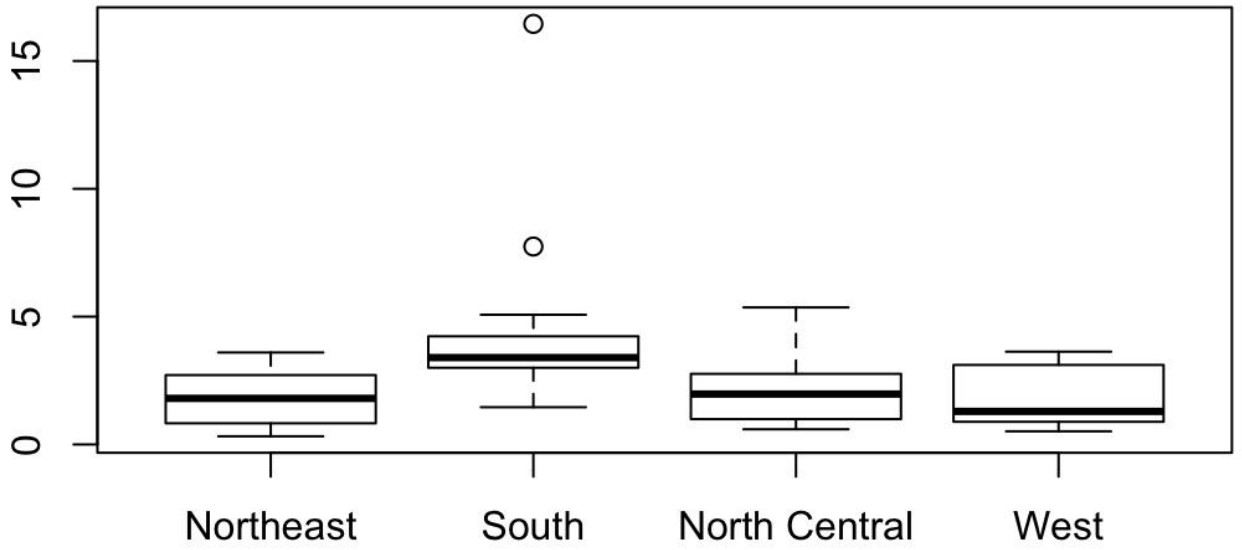

```
> plot(x, y, xlim=c(0, 40),
ylim=c(0,1300), pch=19,
xlab="population",
ylab="total",
main="Murder")
> legend("bottomright",
legend = 'my
points',col=1,pch=19)
> abline(a=0,b=30,lty=2,
lwd=2,col='red')
```



# boxplots

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

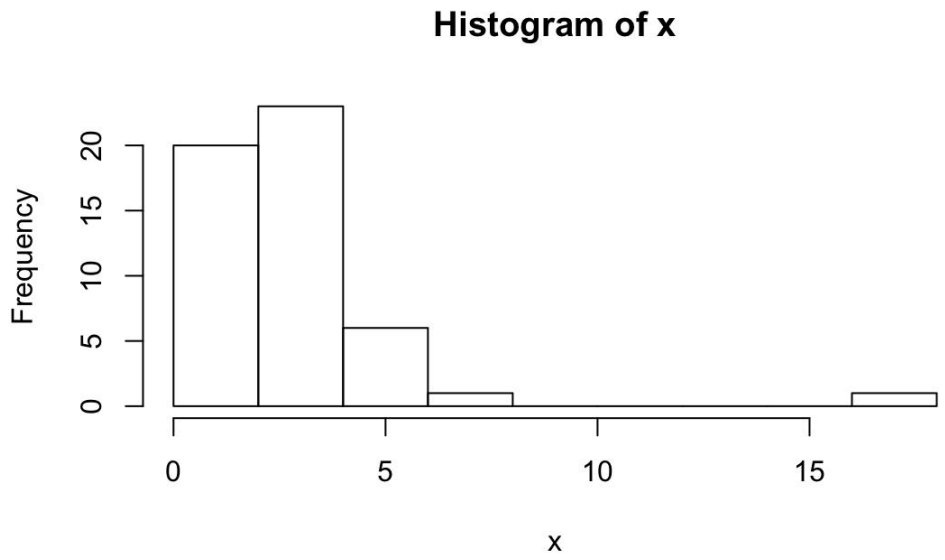
```
> murders$rate <- with(murders, total / population * 100000)
> boxplot(rate~region, data = murders)
```



# histograms

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

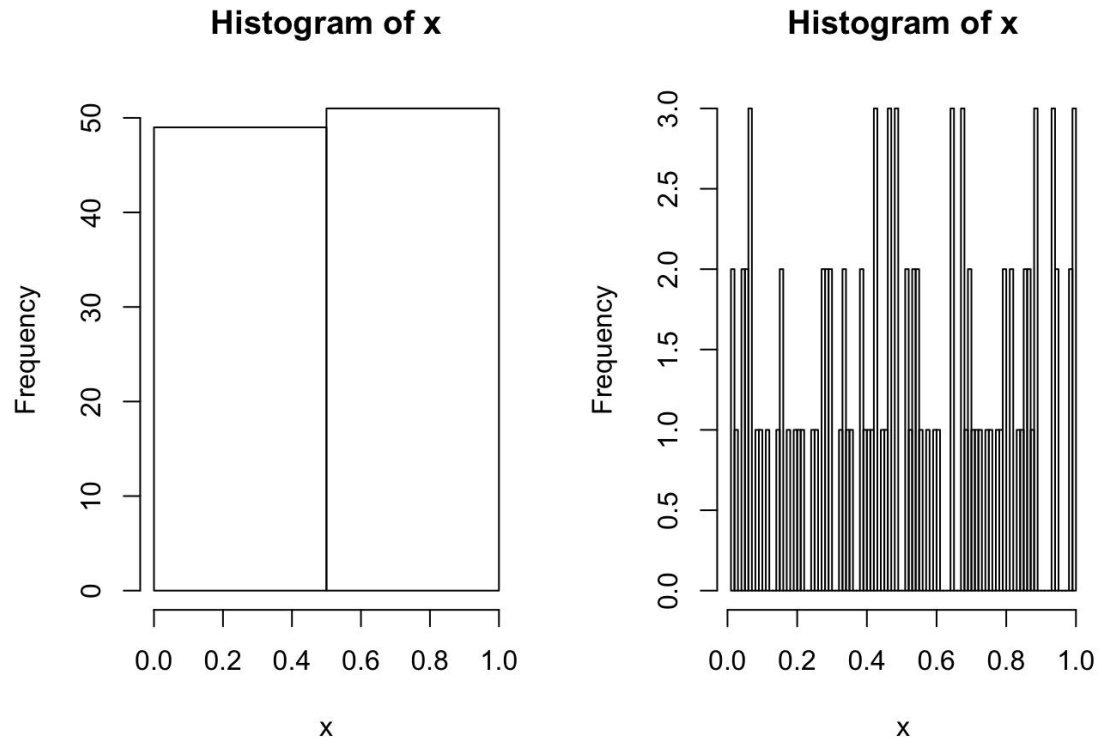
```
> x <- with(murders, total / population * 100000)
> hist(x)
```




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

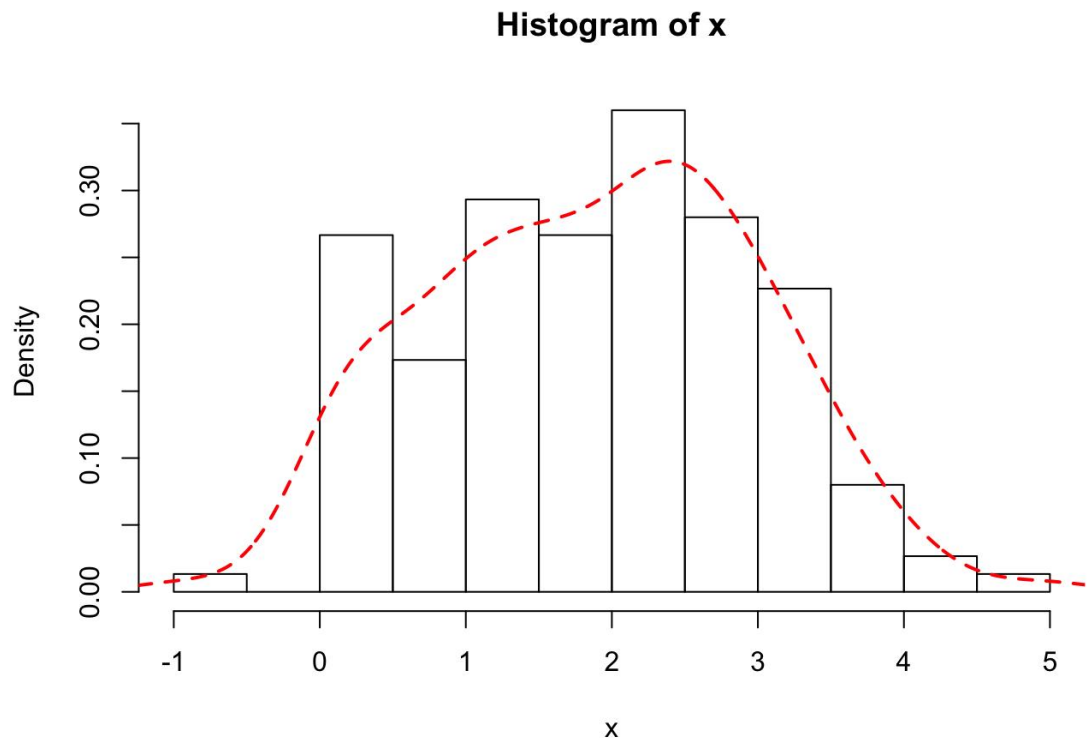
# Histograms

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



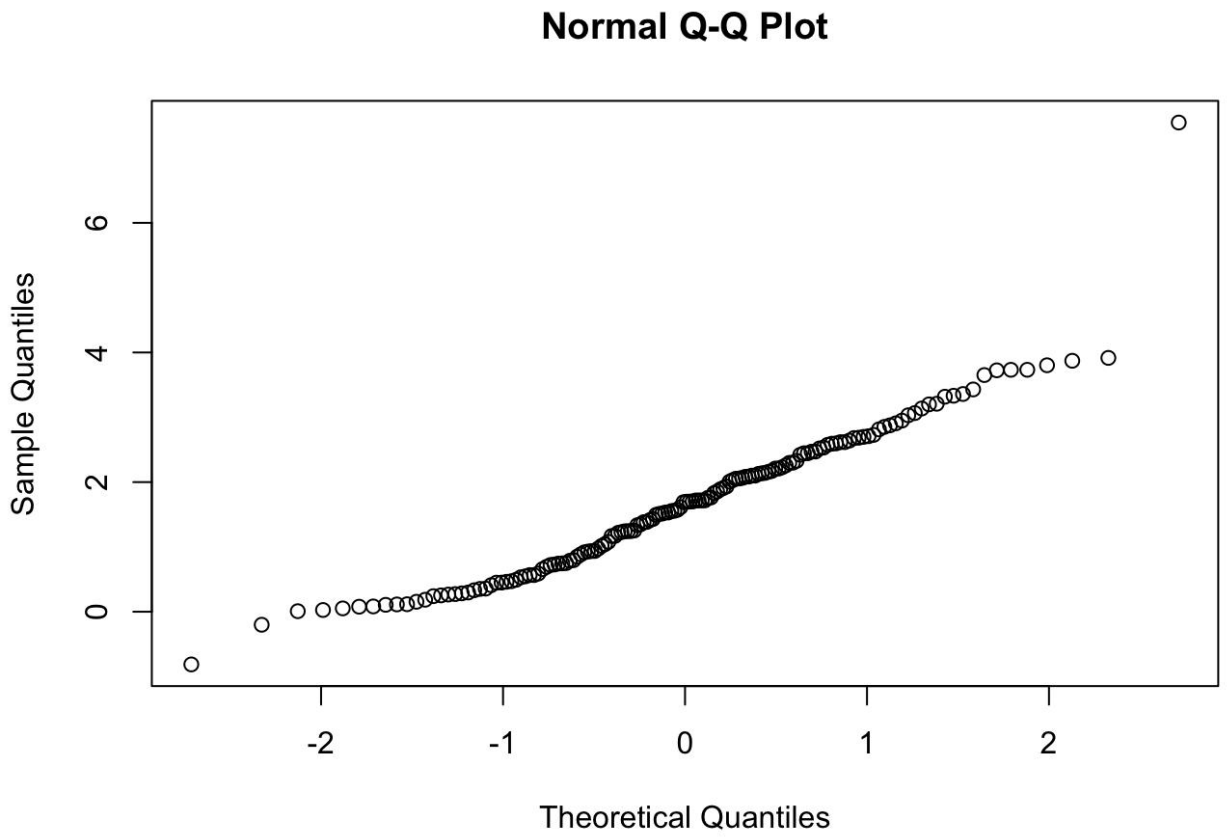
# Density plot

```
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

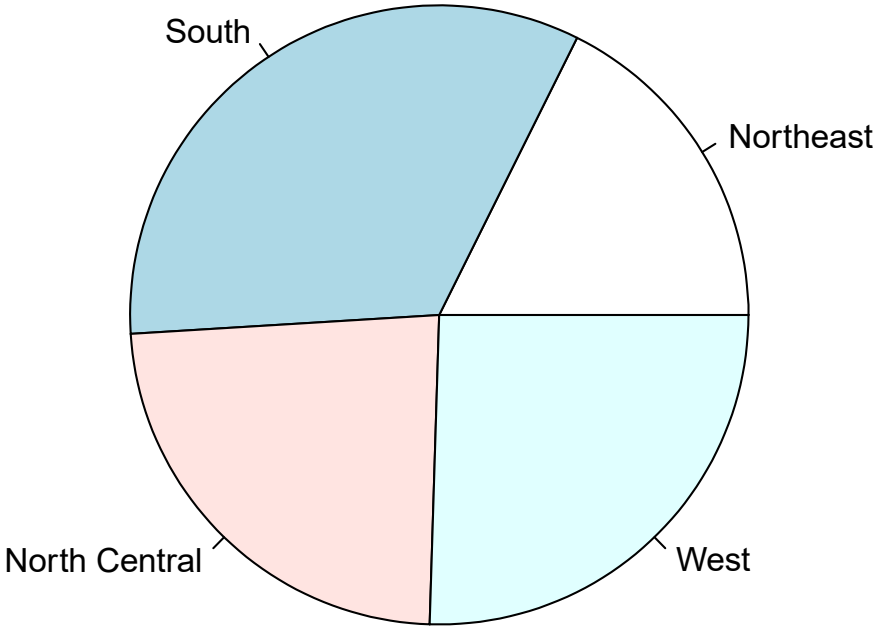
```
x = c(rnorm(100,2,1),rgamma(50,shape = 1))
qqnorm(x)
```



# Piecharts

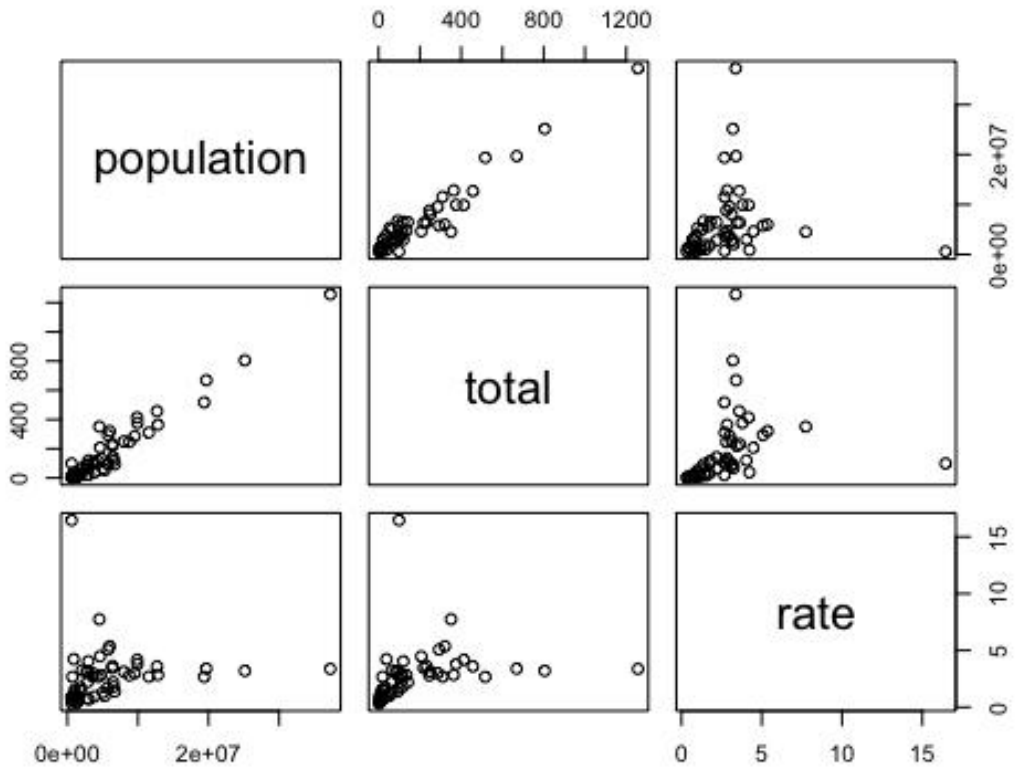
Finally, for categorical variables, you can draw pie plots:

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

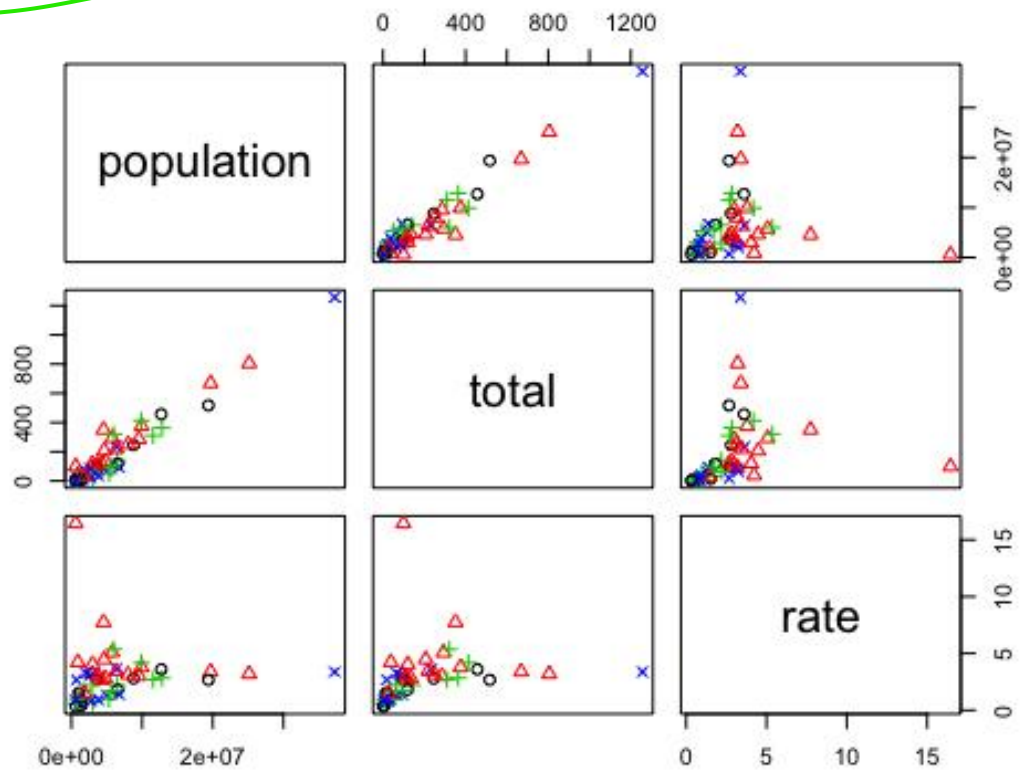


# Pairplot

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




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



# Control Structures

Control structures in R allow you to control the flow of execution of a series of R expressions. Basically, control structures allow you to put some “logic” into your R code, rather than just always executing the same R code every time. Control structures allow you to respond to inputs or to features of the data and execute different R expressions accordingly.

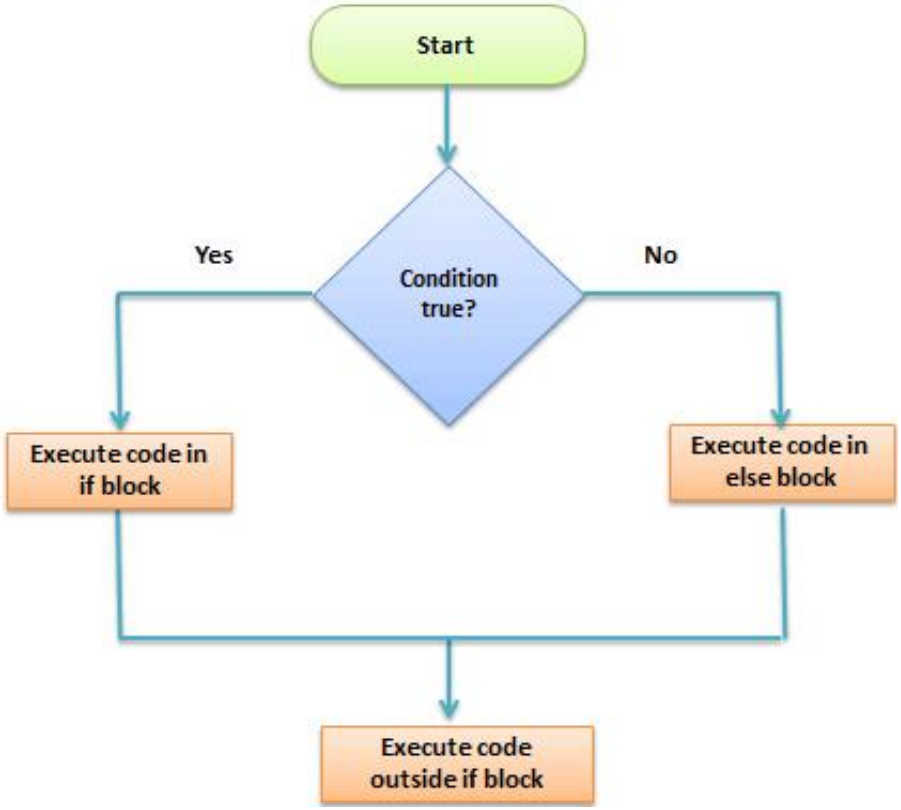
Commonly used control structures are
- if and else: testing a condition and acting on it
- for: execute a loop a fixed number of times
- while: execute a loop *while* a condition is true
- repeat: execute an infinite loop (must break out of it to stop)
- break: break the execution of a loop
- next: skip an iteration of a loop

Most control structures are not used in interactive sessions, but rather when writing functions or longer expressions. However, these constructs do not have to be used in functions and it’s a good idea to become familiar with them before we delve into functions.

# if-else

The if-else combination is probably the most commonly used control structure in R (or perhaps any language). This structure allows you to test a condition and act on it depending on whether it's true or false.

```
if(<condition>) {
        ## do something
}
## Continue with rest of code
```




You can have a series of tests by following the initial if with any number of else ifs.

```
if(<condition1>) {
        ## do something
} else if(<condition2>)  {
        ## do something different
} else {
        ## do something different
}
```

# If – else: example

```
## Generate a uniform random number
x <- runif(1, 0, 10)
if(x > 3) {
      y <- 10
} else {
      y <- 0
}
```

```
y <- if(x > 3) {
      10
} else {
      0
}
```

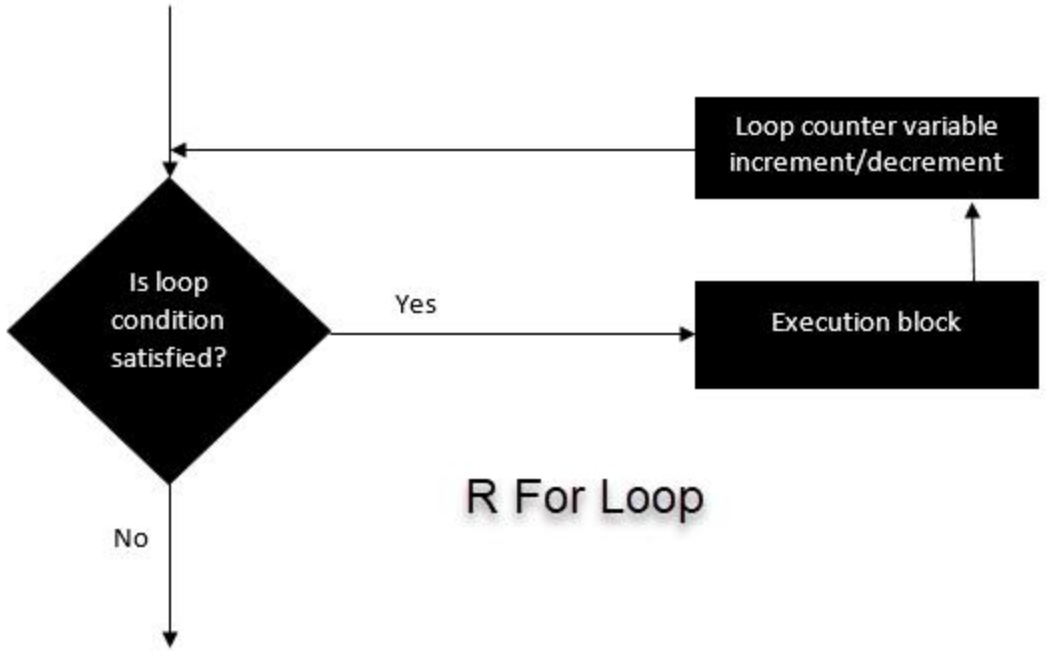
The value of **y** is set depending on whether **x > 3** or not. This expression can also be written a different, but equivalent, way in R.

# for Loops

For loops are pretty much the only looping construct that you will need in R. While you may occasionally find a need for other types of loops, in my experience doing data analysis, I've found very few situations where a for loop wasn't sufficient.
In R, for loops take an interator variable and assign it successive values from a sequence or vector. For loops are most commonly used for iterating over the elements of an object (list, vector, etc.)

```
> for(i in 1:10) {
+       print(i)
+ }
[1] 1
[1] 2
[1] 3
[1] 4
[1] 5
[1] 6
[1] 7
[1] 8
[1] 9
[1] 10
```

This loop takes the i variable and in each iteration of the loop gives it values 1, 2, 3, …, 10, executes the code within the curly braces, and then the loop exits.



# for Loops

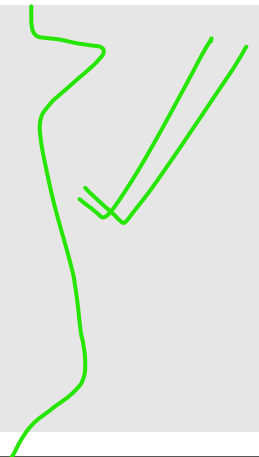
```
> x <- c("a", "b", "c", "d")
>
> for(i in 1:4) {
+       ## Print out each element of 'x'
+       print(x[i])
+ }
[1] "a"
[1] "b"
[1] "c"
[1] "d"
```

The seq_along() function is commonly used in conjunction with for loops in order to generate an integer sequence based on the length of an object (in this case, the object x).

```
> ## Generate a sequence based on length of 'x'
> for(i in seq_along(x)) {
+       print(x[i])
+ }
[1] "a"
[1] "b"
[1] "c"
[1] "d"
```

It is not necessary to use an index-type variable.

```
> for(letter in x) {
+       print(letter)
+ }
[1] "a"
[1] "b"
[1] "c"
[1] "d"
```



For one line loops, the curly braces are not strictly necessary.

```
> for(i in 1:4) print(x[i])
[1] "a"
[1] "b"
[1] "c"
[1] "d"
```

# For Loop over a list

```
# Create a list with three vectors
fruit <- list(Basket = c('Apple', 'Orange', 'Passion fruit', 'Banana’),
Money = c(10, 12, 15), purchase = FALSE)
for (p in fruit) { print(p) }
```

Output:

```
## [1] "Apple" "Orange" "Passion fruit" "Banana"
## [1] 10 12 15
## [1] FALSE
```

# Nested for loops

Nested loops are commonly needed for multidimensional or hierarchical data structures (e.g. matrices, lists). Be careful with nesting though. Nesting beyond 2 to 3 levels often makes it difficult to read/understand the code. If you find yourself in need of a large number of nested loops, you may want to break up the loops by using functions (discussed later).

```
x <- matrix(1:6, 2, 3)

for(i in seq_len(nrow(x))) {
      for(j in seq_len(ncol(x))) {
            print(x[i, j])
      }
}
```

# For Loop over a matrix

A matrix has 2-dimension, rows and columns. To iterate over a matrix, we have to define two for loop, namely one for the rows and another for the column.

```
# Create a matrix
mat <- matrix(data = seq(10, 20, by=1), nrow = 6, ncol =2)
# Create the loop with r and c to iterate over the matrix
for (r in 1:nrow(mat))
        for (c in 1:ncol(mat))
                print(paste("Row", r, "and column",c, "have
                values of", mat[r,c]))
```

Output:

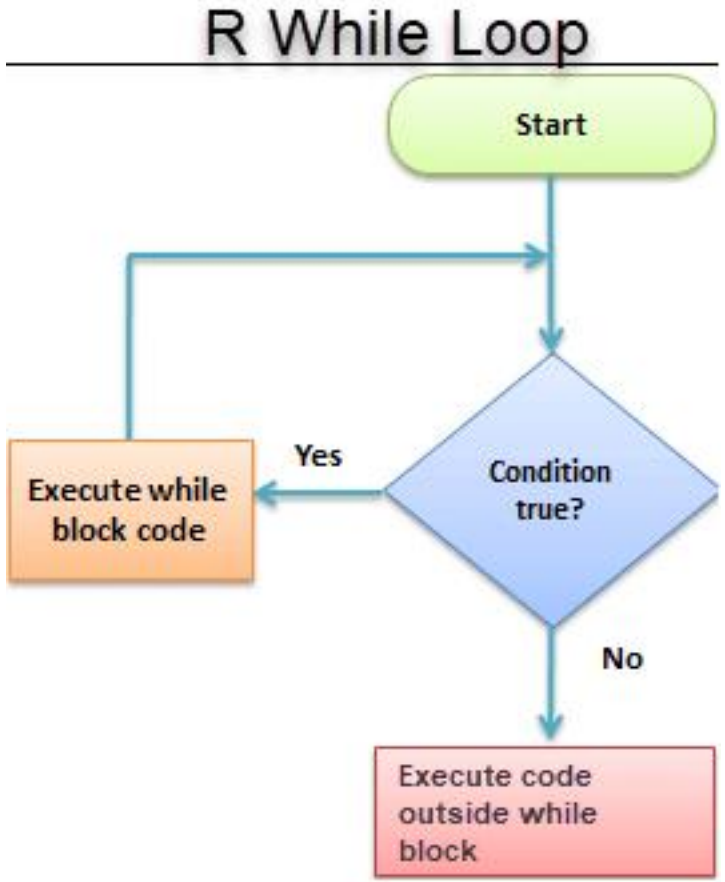
```
## [1] "Row 1 and column 1 have values of 10"
## [1] "Row 1 and column 2 have values of 16"
## [1] "Row 2 and column 1 have values of 11"
## [1] "Row 2 and column 2 have values of 17"
## [1] "Row 3 and column 1 have values of 12"
## [1] "Row 3 and column 2 have values of 18"
## [1] "Row 4 and column 1 have values of 13"
## [1] "Row 4 and column 2 have values of 19"
## [1] "Row 5 and column 1 have values of 14"
## [1] "Row 5 and column 2 have values of 20"
## [1] "Row 6 and column 1 have values of 15"
## [1] "Row 6 and column 2 have values of 10"
```

# while Loops

While loops begin by testing a condition. If it is true, then they execute the loop body. Once the loop body is executed, the condition is tested again, and so forth, until the condition is false, after which the loop exits.

```
> count <- 0
> while(count < 10) {
+       print(count)
+       count <- count + 1
+ }
[1] 0
[1] 1
[1] 2
[1] 3
[1] 4
[1] 5
[1] 6
[1] 7
[1] 8
[1] 9
```

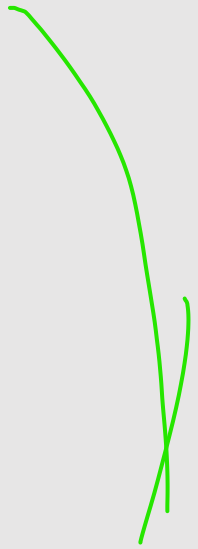
While loops can potentially result in infinite loops if not written properly. Use with care!



# while Loops

Sometimes there will be more than one condition in the test.

```
> z <- 5
> set.seed(1)
>
> while(z >= 3 && z <= 10) {
+       coin <- rbinom(1, 1, 0.5)
+
+       if(coin == 1) {  ## random walk
+             z <- z + 1
+       } else {
+             z <- z - 1
+       }
+ }
> print(z)
[1] 2
```



Conditions are always evaluated from left to right. For example, in the above code, if **z** were less than 3, the second test would not have been evaluated.

# repeat Loops

repeat initiates an infinite loop right from the start. These are not commonly used in statistical or data analysis applications but they do have their uses. The only way to exit a repeat loop is to call break.

One possible paradigm might be in an iterative algorithm where you may be searching for a solution and you don't want to stop until you're close enough to the solution. In this kind of situation, you often don't know in advance how many iterations it's going to take to get

```
x0 <- 1
tol <- 1e-8

repeat {
    x1 <- computeEstimate()

    if(abs(x1 - x0) < tol) {  ## Close enough?
        break
    } else {
        x0 <- x1
    }
}
```

Note that the above code will not run if the computeEstimate() function is not defined (I just made it up for the purposes of this demonstration). The loop above is a bit dangerous because there's no guarantee it will stop. You could get in a situation where the values of x0 and x1 oscillate back and forth and never converge. Better to set a hard limit on the number of iterations by using a for loop and then report whether convergence was

# next, break

**next** is used to skip an iteration of a loop.

```
for(i in 1:100) {
      if(i <= 20) {
            ## Skip the first 20 iterations
            next
      }
      ## Do something here
}
```

**break** is used to exit a loop immediately, regardless of what iteration the loop may be on.

```
for(i in 1:100) {
      print(i)

      if(i > 20) {
            ## Stop loop after 20 iterations
            break
      }
}
```

# Example: Find the factorial of a number

*The factorial of a number is the product of all the integers from 1 to that number. For example, the factorial of 6 (denoted as 6!) is 1*2*3*4*5*6 = 720.*
*Factorial is not defined for negative numbers and the factorial of zero is one, 0! = 1.*
*This example finds the factorial of a number normally. However, you can find it using recursion as well.*

```
# take input from the user
num = as.integer(readline(prompt="Enter a number: "))
factorial = 1
# check is the number is negative, positive or zero
if(num < 0) {
        print("Sorry, factorial does not exist for negative
numbers") }
else if(num == 0) {
        print("The factorial of 0 is 1") }
else {
        for(i in 1:num) {
                factorial = factorial * i }
        print(paste("The factorial of", num ,"is",factorial)) }
```

```
> factorial(8)
[1] 40320
```

# R Notebooks

Seurat - RStudio

File Edit Code View Plots Session Build Debug Profile Tools Help

Untitled1*

Source on Save | Run | Source

```
1  library(tidyverse)
2  setwd("C:/Users/andrewss/Desktop/ggplot_data_files")
3
4  # Read some data
5  read_tsv("small_file.txt") -> small
6
7  # Do some summarisation
8  small %>%
9    group_by(Category) %>%
10   summarise(
11     count=n(),
12     length=mean(Length)
13   )
14
15 # Plot a graph
16 small %>%
17   ggplot(aes(x=Category, y=Length)) +
18   geom_bar(stat="summary", fun="mean", fill="grey")+
19   stat_summary(geom="errorbar", width=0.3, size=1, fun.data=mean_se)
20
```

14:1 (Top Level) R Script

Code

Comments

Console Terminal Jobs

C:/Users/andrewss/Desktop/ggplot_data_files/

```
+   )
`summarise()` ungrouping output (override with `.groups` argument)
# A tibble: 4 x 3
  Category count length
  <chr>    <int>  <dbl>
1 A           10   68.3
2 B           10   70.5
3 C           10   75.6
4 D           10   78.9
>
> # Plot a graph
> small %>%
+   ggplot(aes(x=Category, y=Length)) +
+   geom_bar(stat="summary", fun="mean", fill="grey")+
+   stat_summary(geom="errorbar", width=0.3, size=1, fun.data=mean_se)
>
```

Text output

Environment History Connections Tutorial

Import Dataset | List

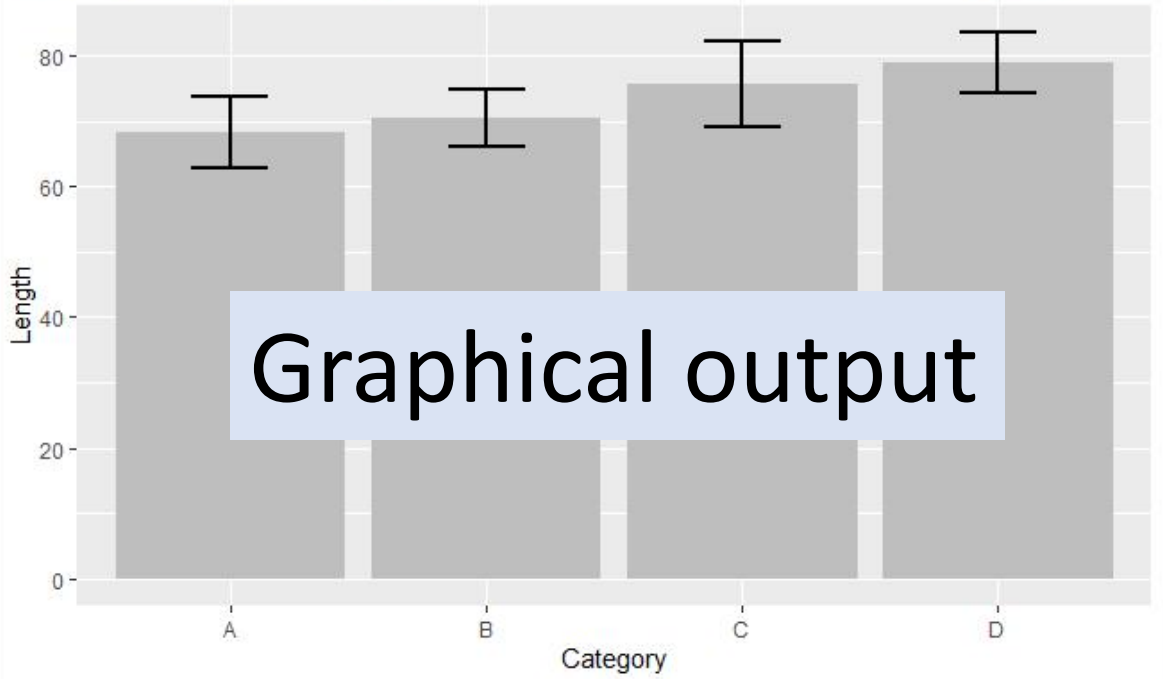
Global Environment

Data

| small | 40 obs. of 3 variables |
|---|---|

Files Plots Packages Help Viewer

Zoom | Export | Publish




Graphical output

# Problems with conventional scripts

- Only the code is generally distributed
  - Output not included – users have to run it again
- No collation of output
  - Can’t see which bit of code generated what output
  - No automated saving of results
- Limited commenting
  - Text comments, no formatting or structure

# R Notebooks

- Alternative document format to conventional scripts
- Collates into a single document
  - Code
  - Formatted commentary
  - Output (text and graphical)
- Exported to HTML, PDF or Word

# Code



```
---
title: "Example Notebook"
output:
  html_document:
    df_print: paged
    toc: true
    toc_float: true
---

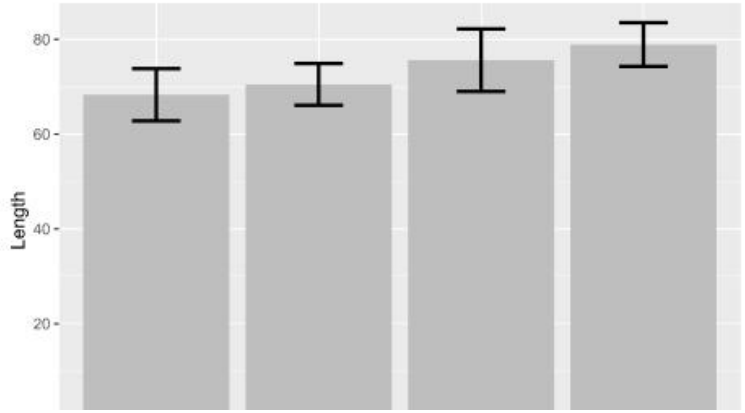
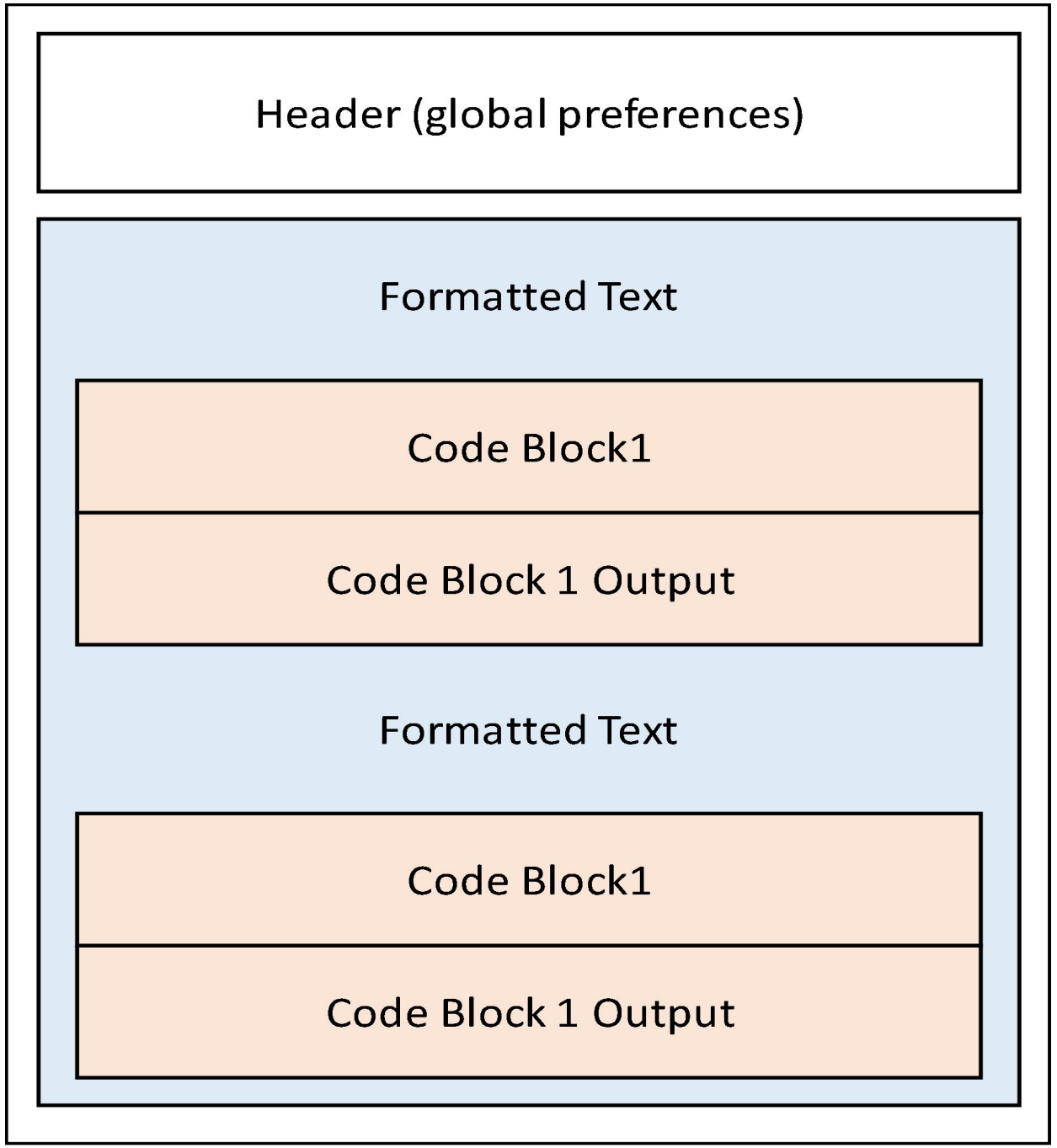
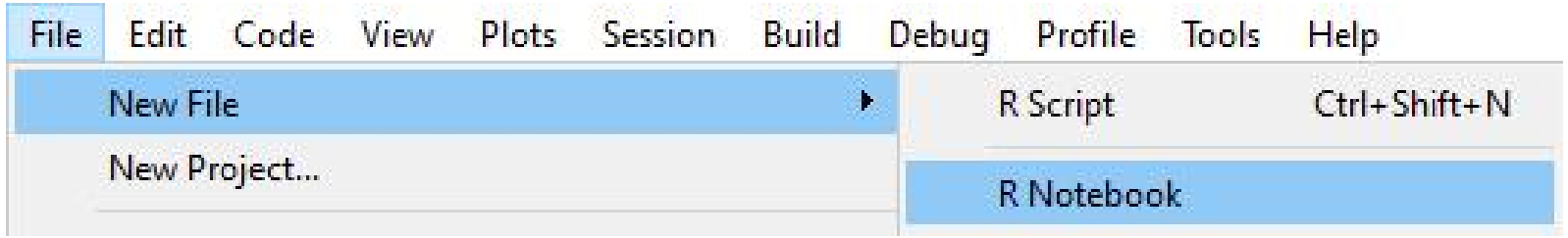
Introduction
============

This is an example of a notebook to show how they work.

```{r message=FALSE}
library(tidyverse)
```

Processing
==========

Read the data
-------------

```{r message=FALSE}
read_tsv("small_file.txt") -> small
head(small)
```
```

| Sample <chr> | Length <dbl> | Category <chr> |
|---|---|---|
| x_1 | 45 | A |
| x_2 | 82 | B |
| x_3 | 81 | C |
| x_4 | 56 | D |
| x_5 | 96 | A |

## Summarise

We're going to calculate the mean of the lengths per category

```
small %>%
  group_by(Category) %>%
  summarise(
    count=n(),
    length=mean(Length)
  )
```

| Category <chr> | count <int> | length <dbl> |
|---|---|---|
| A | 10 | 68.3 |
| B | 10 | 70.5 |
| C | 10 | 75.6 |
| D | 10 | 78.9 |

4 rows

## Plot

```
small %>%
  ggplot(aes(x=Category, y=Length)) +
  geom_bar(stat="summary", fun="mean", fill="grey")+
  stat_summary(geom="errorbar", width=0.3, size=1, fun.data=mean_se)
```




# Output

# Notebook Structure

- Single overall text document, split into sections
  - Header (mostly preferences)
  - Body
    - Commentary (default)
    - R Code
    - Output (graphical and text)



# Creating a Notebook in RStudio




- You may need to install some packages (Rstudio will prompt you if you do)
- Opens a default template which you can then edit

```
---
title: "R Notebook"
output: html_notebook
---

This is an [R Markdown](http://rmarkdown.rstudio.com) Notebook. When you execute code within the notebook, the results appear beneath the code.

Try executing this chunk by clicking the *Run* button within the chunk or by placing your cursor inside it and pressing *Ctrl+Shift+Enter*.

```{r}
plot(cars)
```

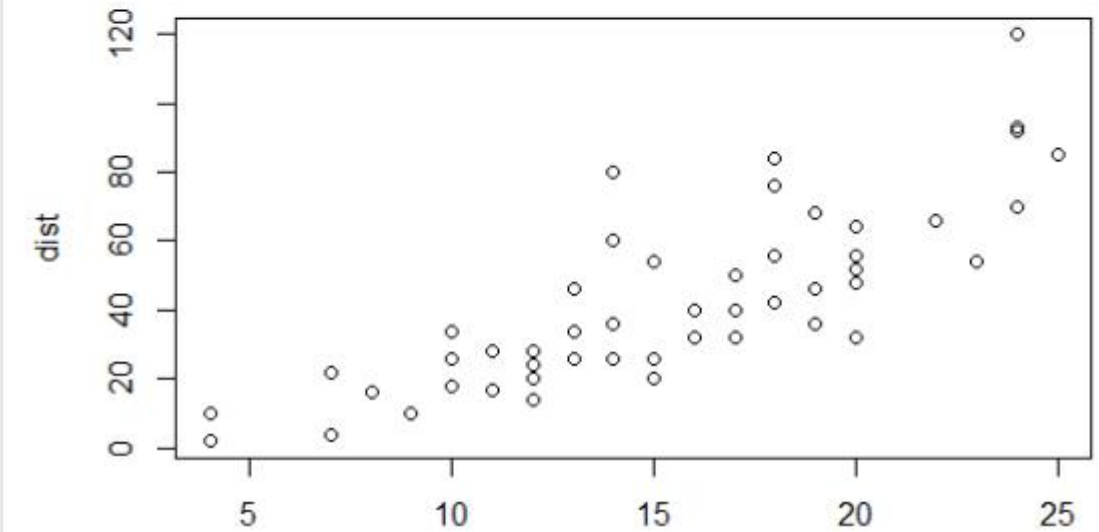
Add a new chunk by clicking the *Insert Chunk* button on the toolbar or by pressing *Ctrl+Alt+I*.

When you save the notebook, an HTML file containing the code and output will be saved alongside it (click the *Preview* button or press *Ctrl+Shift+K* to preview the HTML file).

The preview shows you a rendered HTML copy of the contents of the editor. Consequently, unlike *Knit*, *Preview* does not run any R code chunks. Instead, the output of the chunk when it was last run in the editor is displayed.
```

# Notebook sections

Header

Commentary

Code

```
---
title: "R Notebook"
output: html_notebook
---

This is an [R Markdown](http://rmarkdown.rstudio.com) Notebook. When you execute code within the notebook, the results appear beneath the code.

Try executing this chunk by clicking the *Run* button within the chunk or by placing your cursor inside it and pressing *Ctrl+Shift+Enter*.

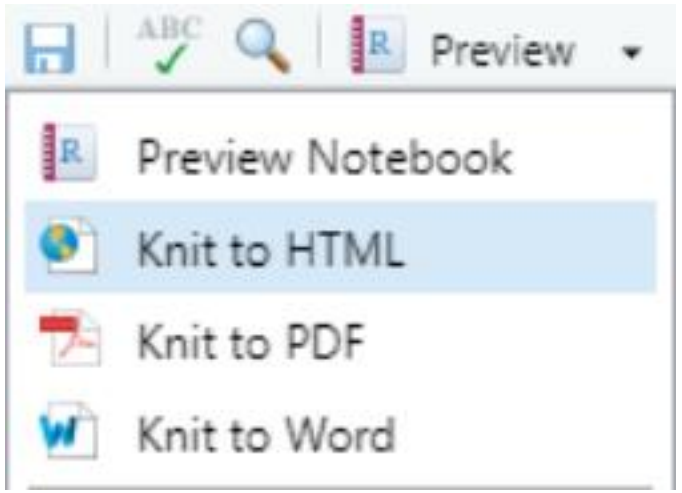
```{r}
plot(cars)
```
```




Sections are marked by special quotes

```
---
```
`---` for header

```
```{r}
```
```` for R code

Default for unquoted text is commentary

# Notebook workflow

- Create  new notebook document
- Save it straight away (use a .Rmd extension)
- Add commentary in Markdown format
- Add R sections using Insert > R
- Run code blocks to generate output

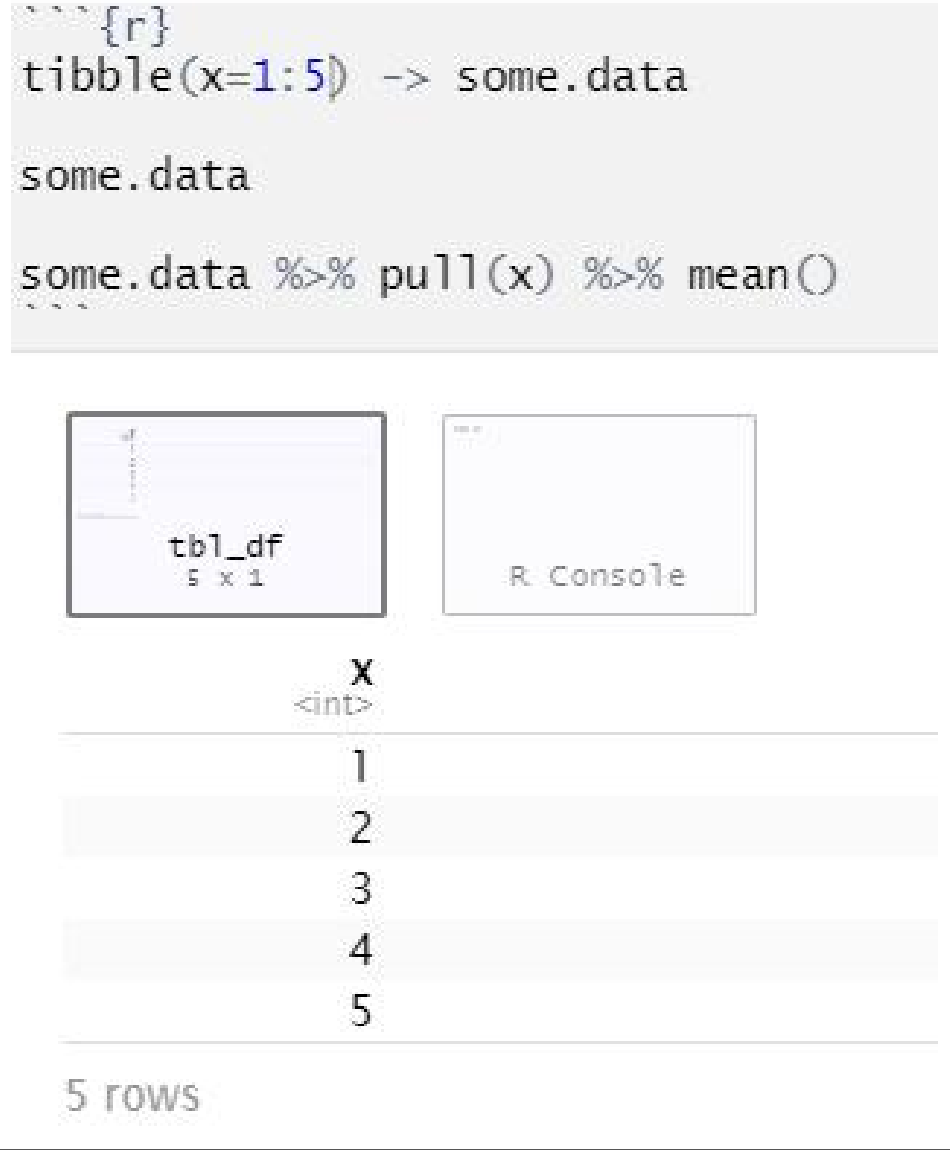

- Knit document to HTML / PDF / Word




Be careful not to delete any of the section markers added by ‘insert’ or the header

# Running R code in a notebook

- Control + Return runs one line
  - Output goes below
  - Output replaces any previous block output
- Control + Shift + Return runs the block
  - Multiple outputs put into clickable windows
  - Will be interspersed in compiled document
  - Can also press the ‘play’ button at top right



# Exercise

## Exercise 1: Simple Calculations

- Use R to calculate the following:
  - 31 * 78
  - 697 / 41
- Assign the value of 39 to `x`
- Assign the value of 22 to `y`
- Make `z` the value of `x - y`
- Display the value of `z` in the console
- Calculate the square root of 2345, and perform a log2 transformation on the result.

# Exercise

## Exercise 2: Working with Vectors

- Create a vector called `vec1` containing the numbers 2,5,8,12 and 16
- Use [lower]`:`[upper] notation to make a second vector called `vec2` containing the numbers 5 to 9
- Subtract `vec2` from `vec1` and look at the result
- Use `seq()` to make a vector of 100 values starting at 2 and increasing by 3 each time and store it in a new variable
- Extract the values at positions 5,10,15 and 20 in the vector of 100 values you made
- Extract the values at positions 10 to 30 in the vector of 100 values you made

# Exercise 3

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.

# Exercise

### Exercise 4: Reading in data from a file

Set your working directory to where the data files are stored. Make sure that the folder of data files has been unzipped. e.g.
`setwd("D:/Data_folder")`

**4a**

- Read the file 'small_file.txt' into a new data structure. This is a tab delimited file so you should use `read.delim()`. Remember to assign a name to the data that you read in using the assignment operator, e.g.

```
my.small.file <- read.delim("small_file.txt")
```

- View the data set to check that it has imported correctly.

**4b**

- Read the file 'Child_Variants.csv' into a new data structure. This is a comma separated file so you should use `read.csv()`. Again, remember to assign a name to the data when you import it.
- Use `head` and `View` to look at the data set to check that it has imported correctly.
- Calculate the `mean` of the column named `MutantReadPercent.` Think about how you are going to access a single column first (probably by using the `$` notation), then once you can access the data pass it to the `mean` function.

# Exercises 5

Using the dataset murder:

A. Create a histogram of the state populations.

B. Generate boxplots of the state populations by region.

## Exercise 6: basic plots

•Read in the file `'brain_bodyweight.txt'`. This is a tab delimited file so you can use `read.delim()`. The first column contains species names, not data, so use `row.names=1` to set these up correctly in your data frame.
•Log transform the data (base 2).
•Create a scatterplot with default parameters with the log transformed data.

# EX 7

*Multiplication Table:* **print the multiplication table of a number (entered by the user) from 1 to 10.**

*Hints:*
*Here, we ask the user for a number which is stored in num variable.*
*Then, the for loop is iterated 10 times from i equals to 1 to i equals to 10.*

# Exercise 8

Write a program to check if the input number is prime or not

*Hints*

*Here, we take an integer from the user and check whether it is prime or not. Numbers less than or equal to 1 are not prime numbers.*

*Hence, we only proceed if the num is greater than 1. We check if num is exactly divisible by any number from 2 to num – 1.*

*If we find a factor in that range, the number is not prime. Else the number is prime.*

*We can decrease the range of numbers where we look for factors.*

*In the above program, our search range is from 2 to num – 1.*

*We could have used the range, [2, num / 2] or [2, num ** 0.5]. The later range is based on the fact that a composite number must have a factor less than square root of that number. Otherwise the number is prime.*