

Introduction to R

Marco Chiapello

28/03/2017

Center for Proteomics
University of Cambridge
mc983@cam.ac.uk

Introduction

Basic concepts in R

- Vector

- R packages

- Data structures

RULES

1. **EVERY** time you do not understand... **RAISE YOUR HAND**
2. There are **NOT** stupid questions
3. Feel free to **interrupt me** every time you need

Introduction


The R Project for Statistical Computing

- R is a free software environment for statistical computing and graphics
- Open source and cross platform (UNIX platforms, Windows and MacOS)
- Extensive graphics capabilities
- Diverse range of add-on packages
- Active community of developers
- Thorough documentation

What R is?

The R Project for Statistical Computing

You can find R here: <https://www.r-project.org>



[Home]

Download

[CRAN](#)

R Project

[About R](#)
[Logo](#)
[Contributors](#)
[What's New?](#)
[Mailing Lists](#)
[Bug Tracking](#)
[Development Site](#)
[Conferences](#)
[Search](#)

R Foundation

[Foundation](#)
[Board](#)
[Members](#)
[Donors](#)
[Donate](#)

Documentation

[Manuals](#)
[FAQs](#)
[The R Journal](#)
[Books](#)
[Certification](#)
[Other](#)

The R Project for Statistical Computing

Getting Started

R is a free software environment for statistical computing and graphics. It compiles and runs on a wide variety of UNIX platforms, Windows and MacOS. To [download R](#), please choose your preferred [CRAN mirror](#).

If you have questions about R like how to download and install the software, or what the license terms are, please read our [answers to frequently asked questions](#) before you send an email.

News

- **R version 3.3.1 (Bug in Your Hair) prerelease versions** will appear starting Saturday 2016-06-11. Final release is scheduled for Tuesday 2016-06-21.
- **R version 3.3.0 (Supposedly Educational)** has been released on 2016-05-03.
- **R version 3.2.5 (Very, Very Secure Dishes)** has been released on 2016-04-14. This is a rebadging of the quick-fix release 3.2.4-revised.
- **Notice XQuartz users (Mac OS X)** A security issue has been detected with the Sparkle update mechanism used by XQuartz. Avoid updating over insecure channels.
- **The R Logo** is available for download in high-resolution PNG or SVG formats.
- **useR! 2016**, will take place at Stanford University, CA, USA, June 27 - June 30, 2016.
- **The R Journal Volume 7/2** is available.
- **R version 3.2.3 (Wooden Christmas-Tree)** has been released on 2015-12-10.
- **R version 3.1.3 (Smooth Sidewalk)** has been released on 2015-03-09.

What R is?

The R Project for Statistical Computing

- R version 3.3.3 (released 2017-03-06)
- Currently, the CRAN (Comprehensive R Archive Network) package repository features 10241 available packages
 - https://cran.r-project.org/web/packages/available_packages_by_name.html
- Currently, the Bioconductor repository features 1296 available packages
 - <http://www.bioconductor.org>
- Executed using command line, or a graphical user interface (GUI)
- On this course, we use the RStudio GUI
 - www.rstudio.com

What R is?

Examples of R project

- Plot https://figshare.com/articles/Slovenia_s_Gross_Domestic_Expenditure_on_R_amp_D_By_Sector_and_Field_2007_2013_/1564745
- Report <https://www.bioconductor.org/packages/release/bioc/vignettes/MSnbase/inst/doc/MSnbase-demo.pdf>
- Flexdashboard https://beta.rstudioconnect.com/llaire/htmlwidgets-ggplotly-geoms/htmlwidgets-ggplotly-geoms.html#geom_point
- Application <http://shiny.rstudio.com/gallery/retirement-simulation.html>

Getting started

- R is a program which, once installed on your system, can be launched and is immediately ready to take input directly from the user

R can be launched in 2 ways:

1. From command line

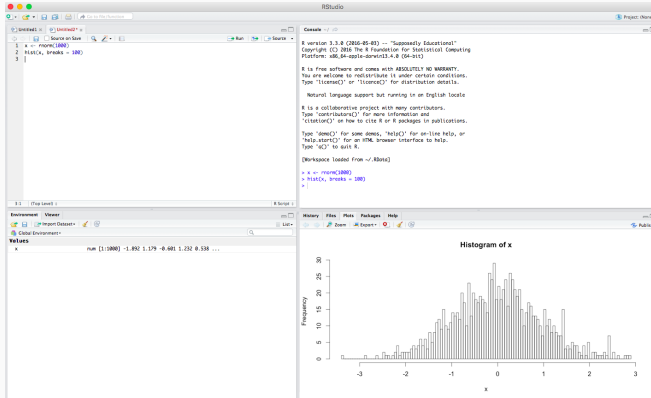
- To start R you need to enter the console (also called terminal or shell)
- To start R, at the prompt simply type: `R`

2. Using RStudio

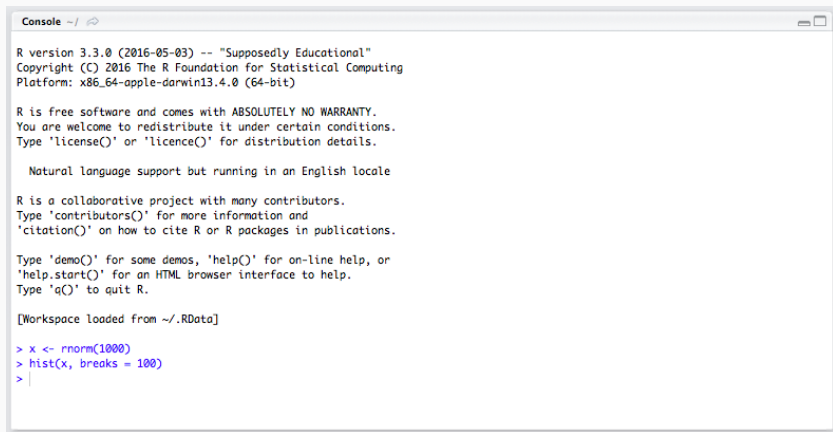
- To launch RStudio, find the RStudio icon and double-click




RStudio presentation

Since we will use RStudio in this course, let's have a look of the program



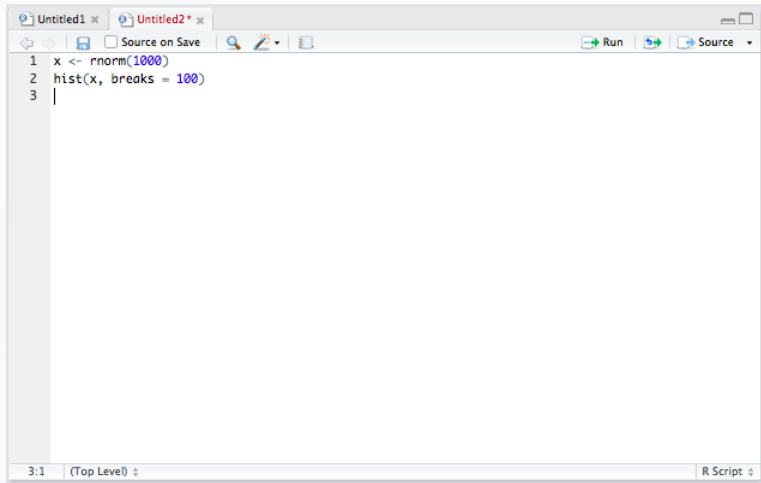
R console



```
Console ~/     
  
R version 3.3.0 (2016-05-03) -- "Supposedly Educational"  
Copyright (C) 2016 The R Foundation for Statistical Computing  
Platform: x86_64-apple-darwin13.4.0 (64-bit)  
  
R is free software and comes with ABSOLUTELY NO WARRANTY.  
You are welcome to redistribute it under certain conditions.  
Type 'license()' or 'licence()' for distribution details.  
  
Natural language support but running in an English locale  
  
R is a collaborative project with many contributors.  
Type 'contributors()' for more information and  
'citation()' on how to cite R or R packages in publications.  
  
Type 'demo()' for some demos, 'help()' for on-line help, or  
'help.start()' for an HTML browser interface to help.  
Type 'q()' to quit R.  
  
[Workspace loaded from ~/.RData]  
  
> x <- rnorm(1000)  
> hist(x, breaks = 100)  
> |
```

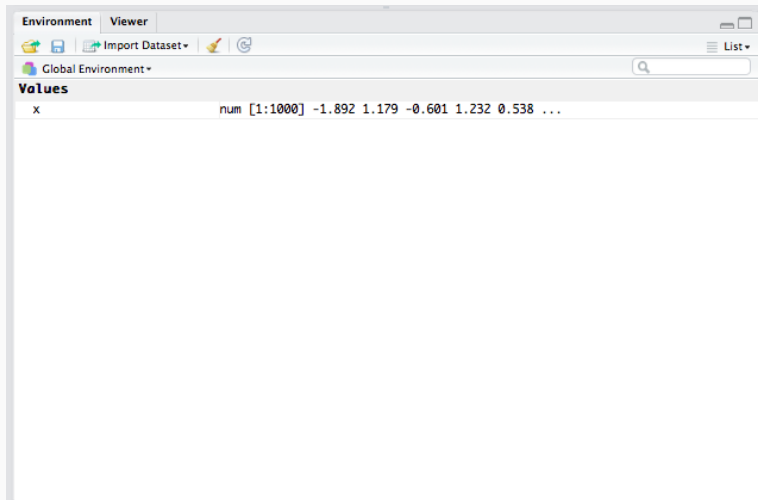
It is the place where you can interactively run R commands

Source editor for R scripts



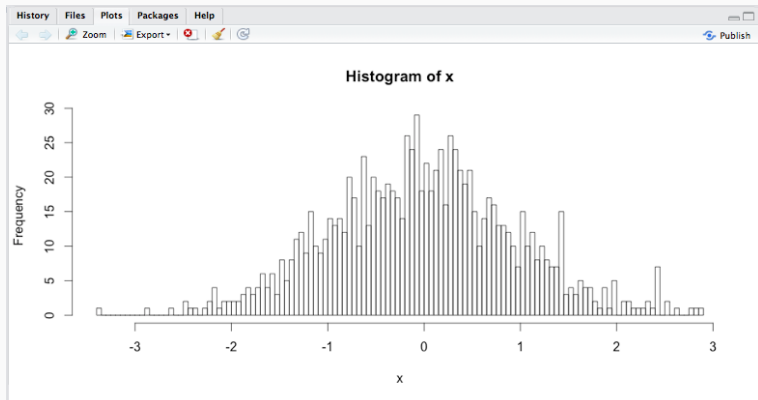
It is the place where you can write your scripts

Workspace



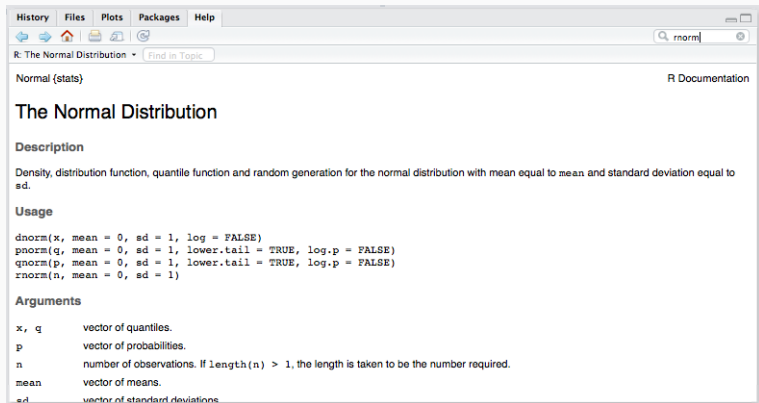
It is the place where you can view object in the global environment

Plot panel



It is the place where you can view your plots

R help

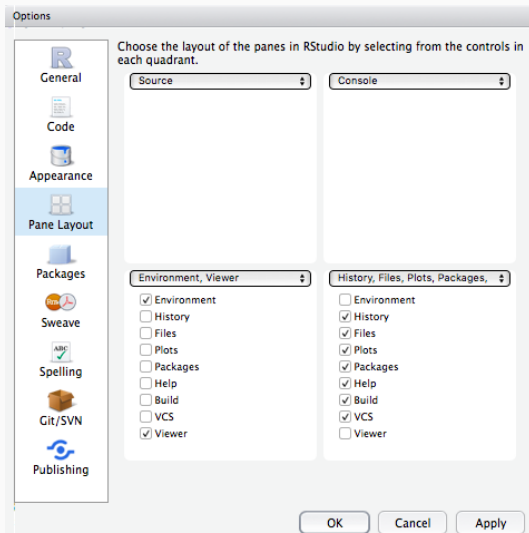


It is the place where you can find help

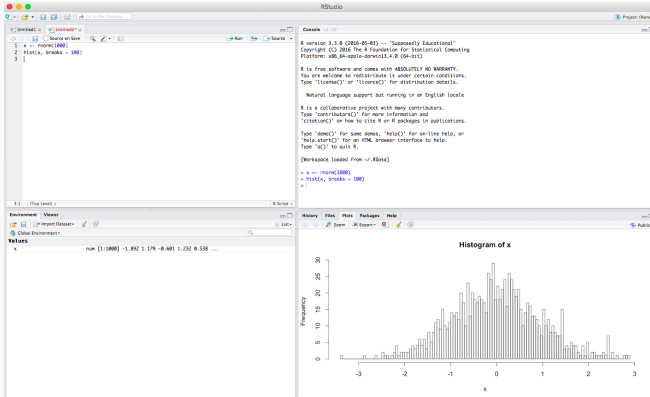
RStudio presentation

The GUI is divided into 4 main sub-windows

These sub-windows are customizable



First challenge: Set up you RStudio as mine



Basic concepts in R

Based on https://github.com/lgatto/TeachingMaterial/tree/master/_basicr

The R syntax

```
1  ### This function checks that all the plot IDs (listed in the 'plots.csv' file)
2  ### occur in the survey file ('surveys.csv'). If all the plots are found, the
3  ### function shows a message and returns 'TRUE', otherwise the function emits a
4  ### warning, and returns 'FALSE'
5  check_plots <- function(survey_file="data/surveys.csv",
6  ~      plot_file="data/plots.csv") {
7
8      ## load files
9      srvy <- read.csv(file=survey_file, stringsAsFactors=FALSE)
10     plts <- read.csv(file=plot_file, stringsAsFactors=FALSE)
11
12     ## Get unique plot_id
13     unique_srvy_plots <- unique(srvy$plot_id)
14
15     ~ if (all(unique_srvy_plots %in% plts$plot_id)) {
16         message("Everything looks good.")
17         return(TRUE)
18     ~ } else {
19         warning("Something is wrong.")
20         return(FALSE)
21     }
22 }
23
24 check_plots()
25
26 surveys <- read.csv(file="data/surveys.csv", stringsAsFactors=FALSE)
27 plots <- read.csv(file="data/plots.csv", stringsAsFactors=FALSE)
28
29 nrow(surveys)
30 ncol(surveys)
31
32 dim(plots)
33 |
34 unique(surveys$species)
```

The R syntax

Element to notice:

- functions
- the assignment operator `<-`
- the `=` for arguments
- the comments `#` and how they are used to document function and its content
- the `$` operator
- Point to indentation and consistency in spacing to improve clarity

Functions and their arguments

- Functions are scripts that automate something complicated or convenient or both.
- Many functions are predefined, or can be made available by importing R packages.
- A function is always followed by brackets that contain the arguments.

The R syntax

Function examples:

```
> round(3.14159)
```

```
[1] 3
```

```
> args(round)
```

```
function (x, digits = 0)
```

```
NULL
```

```
> round(3.14159, digits=2)
```

```
[1] 3.14
```

```
> round(3.14159, 2)
```

```
[1] 3.14
```

```
> round(digits=2, x=3.14159)
```

```
[1] 3.14
```

Seeking help

1. I'm not sure how to use a function
 - `?barplot`

Seeking help

1. I'm not sure how to use a function
 - `?barplot`
2. I get an error message that I don't understand
 - Googling the error message
 - Check Stack Overflow
 - Ask your friendly colleagues
 - Ask on R-help mailing list

Seeking help

1. I'm not sure how to use a function
 - `?barplot`
2. I get an error message that I don't understand
 - Googling the error message
 - Check Stack Overflow
 - Ask your friendly colleagues
 - Ask on R-help mailing list
3. How I should ask help
 - Make it as easy as possible to pinpoint where the issue might be
 - Use the correct words to describe your problem
 - Try to generalize what you are doing
 - Always include the output of `sessionInfo()`
 - [Write reproducible example](<http://adv-r.had.co.nz/Reproducibility.html>)

Numbers

The command line can be used as a calculator

```
> 5 + 7
```

```
[1] 12
```

```
> 5 - 7
```

```
[1] -2
```

```
> 5 * 7
```

```
[1] 35
```

```
> 5 / 7
```

```
[1] 0.7142857
```

Note: The number in the square brackets is an indicator of the position in the output

You can solve simple or complex calculations

```
> (((20/5)^2)-((5+1/3+4/5-10)*(2-34))-20)  
[1] -127.7333
```

But, of course, R is not a calculator

Variables

A **variable** is a letter or word which takes (or contains) a value.

We use the assignment 'operator', `<-`

- We can assign a number to a variable

```
> x <- 5
```

```
> x
```

```
[1] 5
```

- We can assign the result of an operation to a variable

```
> y <- 5 + 7
```

```
> y
```

```
[1] 12
```

Variables

- We can assign use the variables to perform calculation
-

```
> x + y  
[1] 17
```

- We can assign the change the content of the variable
-

```
> x  
[1] 5  
> x <- x - y  
> x  
[1] -7
```

Function

Functions in R perform operations on arguments (the input(s) to the function).

Arguments are always contained in parentheses, i.e. curved brackets `()`, separated by commas.

```
> sum(3, 4, 5, 6)
```

```
[1] 18
```

```
> max(3, 4, 5, 6)
```

```
[1] 6
```

```
> min(3, 4, 5, 6)
```

```
[1] 3
```

Function extention

R contains a lot of pre-builtin functions, but through the so called *packages* is possible extend the R functionalities enormously. Alternately, you can write your own function

```
> summ <- function(a,b){ a + b }  
> summ(1,2)  
[1] 3
```

Vector

The basic data structure in R is a **vector**, an ordered collection of values. R even treats single values as 1-element vectors.

The simplest way to create a **vector** in R is by using the `c()` operator:

```
> c(1,2,3,50)
[1]  1  2  3 50
```

The simplest way to create a **sequence of numbers** is by using the ':' operator:

```
> 1:10  
[1] 1 2 3 4 5 6 7 8 9 10
```

That gave us every integer between (and including) 1 and 10.

What happens if we do 15:1? Give it a try to find out.

What happens if we do 15:1? Give it a try to find out.

```
> 15:1
```

```
[1] 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1
```

It counted backwards in increments of 1!

Remember that if you have questions about a particular R function, you can access its documentation with a question mark followed by the function name:

`?functionnamehere` (1)

However, in the case of an operator like the colon used above, you must enclose the symbol in backticks like this:

`?' : '` (2)

Often, we'll desire more control over a sequence we're creating than what the ':' operator gives us. The `seq()` function serves this purpose.

Try it: Remember what we said about the function arguments

Often, we'll desire more control over a sequence we're creating than what the ':' operator gives us. The `seq()` function serves this purpose.

Try it: Remember what we said about the function arguments

```
> seq(1,10)
[1] 1 2 3 4 5 6 7 8 9 10
```

This gives us the same output as `1:10`. However, let's say that instead we want a vector of numbers ranging from 0 to 4, incremented by 0.5. `seq(0, 4, by=0.5)` does just that.

Try it out.

This gives us the same output as `1:10`. However, let's say that instead we want a vector of numbers ranging from 0 to 4, incremented by 0.5. `seq(0, 4, by=0.5)` does just that.

Try it out.

```
> seq(0, 4, by = 0.5)
[1] 0.0 0.5 1.0 1.5 2.0 2.5 3.0 3.5 4.0
```

Or maybe we don't care what the increment is and we just want a sequence of 10 numbers between 5 and 10. `seq(5, 10, length=10)` does the trick. Give it a shot now and store the result in a new variable called *mySeq*.

Try it out.

Or maybe we don't care what the increment is and we just want a sequence of 10 numbers between 5 and 10. `seq(5, 10, length=10)` does the trick. Give it a shot now and store the result in a new variable called *mySeq*.

Try it out.

```
> mySeq <- seq(5, 10, length=10)
> round(mySeq,1)
[1] 5.0 5.6 6.1 6.7 7.2 7.8 8.3 8.9 9.4 10.0
```

To confirm that `mySeq` has length 10, we can use the `length()` function.

Try it now

To confirm that `mySeq` has length 10, we can use the `length()` function.

Try it now

```
> length(mySeq)
[1] 10
```

Vector

- Let's pretend we don't know the length of mySeq, but we want to generate a sequence of integers from 1 to N, where N represents the length of the mySeq vector.
- We want a new vector (1, 2, 3, ...) that is the same length as mySeq.
- There are several ways we could do this.
- One possibility is to combine the ':' operator and the length() function.

Give that a try

Vector

- Let's pretend we don't know the length of `mySeq`, but we want to generate a sequence of integers from 1 to N, where N represents the length of the `mySeq` vector.
- We want a new vector `(1, 2, 3, ...)` that is the same length as `mySeq`.
- There are several ways we could do this.
- One possibility is to combine the `:` operator and the `length()` function.

Give that a try

```
> 1:length(mySeq)
[1]  1  2  3  4  5  6  7  8  9 10
```

Another option is to use `seq(along.with = mySeq)`. Give that a try.

```
> seq(along.with = mySeq)
[1] 1 2 3 4 5 6 7 8 9 10
```

R has a separate built-in function for this purpose

```
> seq_along(mySeq)
[1] 1 2 3 4 5 6 7 8 9 10
```

Vector

- There are often **several approaches** to solving the same problem in R
- Simple approaches that involve **less typing** are generally best
- It is also important for your code to be **readable**, so that you and others can figure out what's going on without too much hassle

```
> # Create a sequence of 10 numbers  
> seq_along(mySeq)  
[1] 1 2 3 4 5 6 7 8 9 10
```

The comments in R begin with **hash**. You should have about 1/3 of your code commented.

One more function related to creating sequences of numbers is `rep()`, which stands for 'replicate'.

If we're interested in creating a vector that contains 1 and 0 five times, we can use `rep(c(1,0), times = 5)`.

Try it out

One more function related to creating sequences of numbers is `rep()`, which stands for 'replicate'.

If we're interested in creating a vector that contains 1 and 0 five times, we can use `rep(c(1,0), times = 5)`.

Try it out

```
> # Create a sequence of 1 and 0  
> rep(c(1,0), times = 5)  
[1] 1 0 1 0 1 0 1 0 1 0
```

If we want our vector to contain 5 ones and then 5 zeros, we can do this with the 'each' argument instead of 'times' argument.

Try it out

If we want our vector to contain 5 ones and then 5 zeros, we can do this with the 'each' argument instead of 'times' argument.

Try it out

```
> # Create a sequence of 1 and 0  
> rep(c(1,0), each = 5)  
[1] 1 1 1 1 1 0 0 0 0 0
```

Vector

- We'll see, now, how to **extract** elements from a vector (subset)
- The square brackets `[]` indicate position within the vector
 - R even treats single values as 1-element vectors
 - The vector in R starts from position 1
- We can extract individual elements by using the `[]` notation

Try `mySeq[1:3]`

Vector

- We'll see, now, how to **extract** elements from a vector (subset)
- The square brackets `[]` indicate position within the vector
 - R even treats single values as 1-element vectors
 - The vector in R starts from position 1
- We can extract individual elements by using the `[]` notation

Try `mySeq[1:3]`

```
> mySeq[1:3]  
[1] 5.000000 5.555556 6.111111
```

If we want to 3th, 5th and 10th elements of the vector mySeq.
Try it out

If we want to 3th, 5th and 10th elements of the vector `mySeq`.

Try it out

```
> round(mySeq, 1)
[1] 5.0 5.6 6.1 6.7 7.2 7.8 8.3 8.9 9.4 10.0
> round(mySeq[c(3,5,10)], 1)
[1] 6.1 7.2 10.0
```

If we want all the elements bigger than 7.

Try it out

If we want all the elements bigger than 7.

Try it out

```
> round(mySeq, 1)
[1] 5.0 5.6 6.1 6.7 7.2 7.8 8.3 8.9 9.4 10.0
> round(mySeq[mySeq > 7], 1)
[1] 7.2 7.8 8.3 8.9 9.4 10.0
```

Vector

If we ask you to produce a vector of 1000 even numbers (from 2 to 2000), extract the 345th and the 987th elements and sum them, would you know how to do it?

Try it out

Vector

If we ask you to produce a vector of 1000 even numbers (from 2 to 2000), extract the 345th and the 987th elements and sum them, would you know how to do it?

Try it out

```
> a <- seq(2,2000,by=2)
> length(a)
[1] 1000
> a[345] + a[987]
[1] 2664
> # Short version
> sum(seq(2,2000,by=2)[c(345,987)])
[1] 2664
```

When applying all standard arithmetic operations to vectors,
application is element-wise.

```
> x <- 1:10  
> y <- x * 2  
> y  
[1] 2 4 6 8 10 12 14 16 18 20
```

Vector

Adding two vectors

```
> z <- x^2  
> y + z  
[1] 3 8 15 24 35 48 63 80 99 120
```

If vectors are not the same length, the shorter one will be recycled

```
> x  
[1] 1 2 3 4 5 6 7 8 9 10  
> x + 1:2  
[1] 2 4 4 6 6 8 8 10 10 12
```

Vector

- All the vectors we have seen so far have contained numbers, but we can also store strings

```
> gene.names <- c("Pax6", "Beta-actin", "FoxP2", "Hox9")  
> gene.names  
[1] "Pax6"          "Beta-actin"    "FoxP2"         "Hox9"
```

- We can name elements of vectors using the *names* function

```
> gene.expression <- c(0, 3.2, 1.2, -2)  
> gene.expression  
[1] 0.0 3.2 1.2 -2.0  
> names(gene.expression) <- gene.names  
> gene.expression  
      Pax6 Beta-actin      FoxP2      Hox9  
      0.0      3.2      1.2      -2.0
```

Exercise: genes and genomes

- Let's try some **vector arithmetic**. Here are the genome lengths and number of protein coding genes for several model organisms:

Species	Genome size (Mb)	Protein coding genes
<i>Homo sapiens</i>	3,102	20,774
<i>Mus musculus</i>	2,731	23,139
<i>Drosophila melanogaster</i>	169	13,937
<i>Caenorhabditis elegans</i>	100	20,532
<i>Saccharomyces cerevisiae</i>	12	6,692

- Create **genome.size** and **coding.genes** vectors to hold the data in each column using the `c` function
- Create a **species.name** vector and use this vector to name the values in the other two vectors.

Exercise: genes and genomes

- Let's assume a **coding gene has an average length of 1.5 kilobases** (1.5 kilobases is 0.0015 Megabases)
- On average, how many base pairs of each genome is made of coding genes?
- **Create a new vector to record this called `coding.bases`**
- **What percentage of each genome is made up of protein coding genes?**
- Use your **`coding.bases`** and **`genome.size`** vectors to calculate this
- **How many times more bases are used for coding in the human genome compared to the yeast genome?**
- **How many times more bases are in the human genome in total compared to the yeast genome?**
- Look up indices of your vectors to find out.

Exercise: genes and genomes

- Creating vectors:

```
> genome.size<-c(3102,2731,169,100,12)
> coding.genes<-c(20774,23139,13937,20532,6692)
> species.name<-c("H. sapiens","M. musculus","D. melanogaster","C.
+ cerevisiae")
> names(genome.size)<-species.name
> names(coding.genes)<-species.name
```

Exercise: genes and genomes

- Creating vectors:

```
> genome.size<-c(3102,2731,169,100,12)
> coding.genes<-c(20774,23139,13937,20532,6692)
> species.name<-c("H. sapiens","M. musculus","D. melanogaster","C.
+ cerevisiae")
> names(genome.size)<-species.name
> names(coding.genes)<-species.name
```

- To calculate the number of coding bases, we need to use the same scale as we used for genome size: 1.5 kilobases is 0.0015 Megabases

```
> coding.bases<-coding.genes*0.0015
> coding.bases
```

H. sapiens	M. musculus	D. melanogaster	C. elegans	S.
31.1610	34.7085	20.9055	30.7980	

Exercise: genes and genomes

- To calculate the percentage of coding bases in each genome:

```
> coding.pc<-coding.bases/genome.size*100
```

```
> coding.pc
```

H. sapiens	M. musculus	D. melanogaster	C. elegans	S. cerevisiae
1.004545	1.270908	12.370118	30.798000	1.000000

Exercise: genes and genomes

- To calculate the percentage of coding bases in each genome:

```
> coding.pc<-coding.bases/genome.size*100
```

```
> coding.pc
```

H. sapiens	M. musculus	D. melanogaster	C. elegans	S. cerevisiae
1.004545	1.270908	12.370118	30.798000	3.104304

- To compare human to yeast:

```
> coding.bases[1]/coding.bases[5]
```

```
H. sapiens
```

```
3.104304
```

```
> genome.size[1]/genome.size[5]
```

```
H. sapiens
```

```
258.5
```

Exercise: genes and genomes

- Note that if a new vector is created using a named vector, the names are usually carried across to the new vector. Sometimes this is what we want (as for **coding.pc**) but sometimes it is not (when we are comparing human to yeast). We can remove names by setting them to the special NULL value:

```
> names(coding.pc) <- NULL  
> coding.pc  
[1] 1.004545 1.270908 12.370118 30.798000 83.650000
```

R packages

R packages

- R comes ready loaded with various libraries of functions called **packages**. e.g. the function **sum()** is in the base package and **sd()**, which calculates the standard deviation of a vector, is in the stats package

R packages

- R comes ready loaded with various libraries of functions called **packages**. e.g. the function **sum()** is in the base package and **sd()**, which calculates the standard deviation of a vector, is in the stats package
- There are 1000s of additional packages provided by third parties, and the packages can be found in numerous server locations on the web called repositories

R packages

- R comes ready loaded with various libraries of functions called **packages**. e.g. the function **sum()** is in the base package and **sd()**, which calculates the standard deviation of a vector, is in the stats package
- There are 1000s of additional packages provided by third parties, and the packages can be found in numerous server locations on the web called repositories
- The two repositories you will come across the most are
 - The Comprehensive R Archive Network (CRAN)
 - Bioconductor

R packages

- R comes ready loaded with various libraries of functions called **packages**. e.g. the function **sum()** is in the base package and **sd()**, which calculates the standard deviation of a vector, is in the stats package
- There are 1000s of additional packages provided by third parties, and the packages can be found in numerous server locations on the web called repositories
- The two repositories you will come across the most are
 - The Comprehensive R Archive Network (CRAN)
 - Bioconductor
- CRAN is mirrored in many locations. Set your local mirror in RStudio using Tools > Options, and choose a CRAN mirror

R packages

- R comes ready loaded with various libraries of functions called **packages**. e.g. the function **sum()** is in the base package and **sd()**, which calculates the standard deviation of a vector, is in the stats package
- There are 1000s of additional packages provided by third parties, and the packages can be found in numerous server locations on the web called repositories
- The two repositories you will come across the most are
 - The Comprehensive R Archive Network (CRAN)
 - Bioconductor
- CRAN is mirrored in many locations. Set your local mirror in RStudio using Tools > Options, and choose a CRAN mirror
- Bioconductor packages are then loaded with the biocLite() function
 - `source("http://bioconductor.org/biocLite.R")`
 - `biocLite("PackageName")`

Exercise

- Matrix is a CRAN extras package
 - Use **install.packages()** function. . .
 - or in RStudio go to Tools > Install Packages. . . and type the package name
- aCGH is a BioConductor package (www.bioconductor.org)

- R needs to be told to use the new functions from the installed packages
- Use **library(...)** function to load the newly installed features
- **library("Matrix");** loads matrix functions

PS: `library()`: Lists all the packages you've got installed locally

PS2: `.libPaths()`: to know where the libraries are installed

DATA STRUCTURES

R is designed to handle experimental data

- In this lesson, we'll cover **matrices** and **data frames**. Both represent 'rectangular' data types, meaning that they are used to store tabular data, with rows and columns
- The main difference, as you'll see, is that **matrices** can only contain a single class of data, while **data frames** can consist of many different classes of data

Data frame

- A data frame is a **set of observations** of a set of variables in other words, the outcome of an experiment.

[Wickham, H. (2014). Tidy Data. Journal of Statistical Software, 59(10), 1 - 23]

Data frame

- A data frame is a **set of observations** of a set of variables in other words, the outcome of an experiment.

[Wickham, H. (2014). Tidy Data. Journal of Statistical Software, 59(10), 1 - 23]

- Tidy datasets provide a standardized way to link the structure of a dataset (**its physical layout**) with its semantics (**its meaning**)

	treatmenta	treatmentb
John Smith	—	2
Jane Doe	16	11
Mary Johnson	3	1

Table 1: Typical presentation dataset.

	John Smith	Jane Doe	Mary Johnson
treatmenta	—	16	3
treatmentb	2	11	1

Table 2: The same data as in Table 1 but structured differently.

Data frame

- A dataset is a collection of **values**

Data frame

- A dataset is a collection of **values**
- Every value belongs to a **variable** and an **observation**

Data frame

- A dataset is a collection of **values**
- Every value belongs to a **variable** and an **observation**
- A variable contains all values that **measure the same underlying attribute** (like height, temperature, duration) across units

Data frame

- A dataset is a collection of **values**
- Every value belongs to a **variable** and an **observation**
- A variable contains all values that **measure the same underlying attribute** (like height, temperature, duration) across units
- An observation contains all values **measured on the same unit** (like a person, or a day, or a race) across attributes

	treatmenta	treatmentb
John Smith	—	2
Jane Doe	16	11
Mary Johnson	3	1

Table 1: Typical presentation dataset.

	John Smith	Jane Doe	Mary Johnson
treatmenta	—	16	3
treatmentb	2	11	1

Table 2: The same data as in Table 1 but structured differently.

Data frame

- Table 3 reorganizes Table 1 to make the values, variables and observations more clear
- The dataset contains 18 values representing three variables and six observations; the variables are:
 1. **person**, with three possible values (John Smith, Mary Johnson, and Jane Doe)
 2. **Treatment**, with two possible values (a and b)
 3. **Result**, with five or six values depending on how you think of the missing value (NA, 16, 3, 2, 11, 1)

person	treatment	result
John Smith	a	—
Jane Doe	a	16
Mary Johnson	a	3
John Smith	b	2
Jane Doe	b	11
Mary Johnson	b	1

Table 3: The same data as in Table 1 but with variables in columns and observations in rows.

Data frame

- A dataset is messy or tidy depending on how rows, columns and tables are matched up with observations, variables and types
- In tidy data:

- A dataset is messy or tidy depending on how rows, columns and tables are matched up with observations, variables and types
- In tidy data:
 1. Each variable forms a column

- A dataset is messy or tidy depending on how rows, columns and tables are matched up with observations, variables and types
- In tidy data:
 1. Each variable forms a column
 2. Each observation forms a row

- A dataset is messy or tidy depending on how rows, columns and tables are matched up with observations, variables and types
- In tidy data:
 1. Each variable forms a column
 2. Each observation forms a row
 3. Each type of observational unit forms a table

- A dataset is messy or tidy depending on how rows, columns and tables are matched up with observations, variables and types
- In tidy data:
 1. Each variable forms a column
 2. Each observation forms a row
 3. Each type of observational unit forms a table
- Three common problems with messy datasets are:

Column headers are values, not variable names

religion	<\$10k	\$10-20k	\$20-30k	\$30-40k	\$40-50k	\$50-75k
Agnostic	27	34	60	81	76	137
Atheist	12	27	37	52	35	70
Buddhist	27	21	30	34	33	58
Catholic	418	617	732	670	638	1116
Don't know/refused	15	14	15	11	10	35
Evangelical Prot	575	869	1064	982	881	1486
Hindu	1	9	7	9	11	34
Historically Black Prot	228	244	236	238	197	223
Jehovah's Witness	20	27	24	24	21	30
Jewish	19	19	25	25	30	95

Table 4: The first ten rows of data on income and religion from the Pew Forum. Three columns, \$75-100k, \$100-150k and >150k, have been omitted.

Data frame

Column headers are values, not variable names

religion	income	freq
Agnostic	<\$10k	27
Agnostic	\$10–20k	34
Agnostic	\$20–30k	60
Agnostic	\$30–40k	81
Agnostic	\$40–50k	76
Agnostic	\$50–75k	137
Agnostic	\$75–100k	122
Agnostic	\$100–150k	109
Agnostic	>150k	84
Agnostic	Don't know/refused	96

Multiple variables stored in one column

country	year	column	cases
AD	2000	m014	0
AD	2000	m1524	0
AD	2000	m2534	1
AD	2000	m3544	0
AD	2000	m4554	0
AD	2000	m5564	0
AD	2000	m65	0
AE	2000	m014	2
AE	2000	m1524	4
AE	2000	m2534	4
AE	2000	m3544	6
AE	2000	m4554	5
AE	2000	m5564	12
AE	2000	m65	10
AE	2000	f014	3

(a) Molten data

country	year	sex	age	cases
AD	2000	m	0-14	0
AD	2000	m	15-24	0
AD	2000	m	25-34	1
AD	2000	m	35-44	0
AD	2000	m	45-54	0
AD	2000	m	55-64	0
AD	2000	m	65+	0
AE	2000	m	0-14	2
AE	2000	m	15-24	4
AE	2000	m	25-34	4
AE	2000	m	35-44	6
AE	2000	m	45-54	5
AE	2000	m	55-64	12
AE	2000	m	65+	10
AE	2000	f	0-14	3

(b) Tidy data

- For example, we might want to analyse information about a set of patients. To start with, let's say we have ten patients and for each one we know their **name, sex, age, weight and whether they give consent** for their data to be made public

Data frame

- For example, we might want to analyse information about a set of patients. To start with, let's say we have ten patients and for each one we know their **name, sex, age, weight and whether they give consent** for their data to be made public
- We are going to create a data frame called 'patients', which will have ten rows (observations) and seven columns (variables).
- **The columns must all be equal lengths**

The patients data frame

```
> patients
  firstName secondName paste.firstName..secondName. sex age weight consent
1      Adam      Jones      Adam Jones    Male  50   70.8     TRUE
2       Eve     Parker     Eve Parker Female  21   67.9     TRUE
3      John     Evans    John Evans    Male  35   75.3    FALSE
4      Mary     Davis    Mary Davis Female  45   61.9     TRUE
5     Peter     Baker    Peter Baker    Male  28   72.4    FALSE
6      Paul   Daniels    Paul Daniels    Male  31   69.9    FALSE
7   Joanna Edwards Joanna Edwards Female  42   63.5    FALSE
8   Matthew     Smith    Matthew Smith    Male  33   71.5     TRUE
9     David   Roberts    David Roberts    Male  57   73.2    FALSE
10    Sally    Wilson    Sally Wilson Female  62   64.8     TRUE
```

Data frame

- **Each column is a vector**, like previous vectors we have seen, for example:

```
> age<-c(50, 21, 35, 45, 28, 31, 42, 33, 57, 62)
```

```
> weight<-c(70.8, 67.9, 75.3, 61.9, 72.4, 69.9, 63.5, 71.5, 73.2, 64.8)
```

Data frame

- **Each column is a vector**, like previous vectors we have seen, for example:

```
> age<-c(50, 21, 35, 45, 28, 31, 42, 33, 57, 62)
> weight<-c(70.8, 67.9, 75.3, 61.9, 72.4, 69.9, 63.5, 71.5, 73.2, 64.8)
```

- We can define the names using character vectors:

```
> firstName<-c('Adam', 'Eve', 'John', 'Mary', 'Peter', 'Paul',
+              'Joanna','Matthew', 'David', 'Sally')
> secondName<-c('Jones', 'Parker', 'Evans', 'Davis', 'Baker',
+               'Daniels', 'Edwards', 'Smith', 'Roberts', 'Wilson')
```

Data frame

- **Each column is a vector**, like previous vectors we have seen, for example:

```
> age<-c(50, 21, 35, 45, 28, 31, 42, 33, 57, 62)
> weight<-c(70.8, 67.9, 75.3, 61.9, 72.4, 69.9, 63.5, 71.5, 73.2, 64.8)
```

- We can define the names using character vectors:

```
> firstName<-c('Adam', 'Eve', 'John', 'Mary', 'Peter', 'Paul',
+              'Joanna', 'Matthew', 'David', 'Sally')
> secondName<-c('Jones', 'Parker', 'Evans', 'Davis', 'Baker',
+               'Daniels', 'Edwards', 'Smith', 'Roberts', 'Wilson')
```

- We also have a new type of vector, the logical vector, which only contains the values TRUE and FALSE:

```
> consent<-c(TRUE, TRUE, FALSE, TRUE, FALSE, FALSE, FALSE, TRUE, FALSE, TRUE)
```

Data structures

- **Vectors can only contain one type of data**; we cannot mix numbers, characters and logical values in the same vector
- If we try this, R will convert everything to characters:

```
> c(20, 'a string', TRUE)
[1] "20"          "a string" "TRUE"
```

- We can see the type of a particular vector using the mode function

```
> mode(firstName)
[1] "character"
> mode(age)
[1] "numeric"
> mode(weight)
[1] "numeric"
> mode(consent)
[1] "logical"
```

Data structures

- Character vectors are fine for some variables, like names
 - But sometimes we have categorical data and we want R to recognize this
 - **A factor is R's data structure for categorical data**
-

```
> sex
[1] "Male"    "Female"  "Male"    "Female"  "Male"    "Male"    "Female"  "Male"
[9] "Male"    "Female"
> factor(sex)
[1] Male    Female Male    Female Male    Male    Female Male    Male    Female
Levels: Female Male
```

- R has converted the strings of the sex character vector into two levels, which are the categories in the data
- Note the values of this factor are not character strings, but levels
- We can use this factor to compare data for males and females

Creating a data frame

- We can construct a data frame from other objects

```
> patients<-data.frame(firstName, secondName, paste(firstName,secondName),  
+                      sex, age, weight, consent)  
> patients
```

	firstName	secondName	paste.firstName..secondName.	sex	age	weight	consent
1	Adam	Jones	Adam Jones	Male	50	70.8	TRUE
2	Eve	Parker	Eve Parker	Female	21	67.9	TRUE
3	John	Evans	John Evans	Male	35	75.3	FALSE
4	Mary	Davis	Mary Davis	Female	45	61.9	TRUE
5	Peter	Baker	Peter Baker	Male	28	72.4	FALSE
6	Paul	Daniels	Paul Daniels	Male	31	69.9	FALSE
7	Joanna	Edwards	Joanna Edwards	Female	42	63.5	FALSE
8	Matthew	Smith	Matthew Smith	Male	33	71.5	TRUE
9	David	Roberts	David Roberts	Male	57	73.2	FALSE
10	Sally	Wilson	Sally Wilson	Female	62	64.8	TRUE

- The paste function joins character vectors together
- We can access particular variables using the dollar operator

```
> patients$age  
[1] 50 21 35 45 28 31 42 33 57 62
```

Naming data frame variables

- R has inferred the names of our data frame variables from the names of the vectors or the commands (eg the paste command)
- We can name the variables after we have created a data frame using the **names** function, and we can use the same function to see the names

```
> names(patients)<-c('First_Name', 'Second_Name', 'Full_Name', 'Sex',  
+                   'Age', 'Weight', 'Consent')  
> names(patients)  
[1] "First_Name" "Second_Name" "Full_Name"   "Sex"         "Age"  
[6] "Weight"     "Consent"
```

- Or we can name the variables when we define the data frame

```
> patients<-data.frame(First_Name=firstName, Second_Name=secondName,  
+                       Full_Name=paste(firstName,secondName), Sex=sex,  
+                       Age=age, Weight=weight, Consent=consent)  
> names(patients)  
[1] "First_Name" "Second_Name" "Full_Name"   "Sex"         "Age"  
[6] "Weight"     "Consent"
```

Inspecting data.frame Objects

- Size:
 - `dim(patients)` - returns a vector with the number of rows in the first element, and the number of columns as the second element (the dimensions of the object)
 - `nrow(patients)` - returns the number of rows
 - `ncol(patients)` - returns the number of columns
- Content:
 - `head(patients)` - shows the first 6 rows
 - `nrow(patients)` - - shows the last 6 rows
- Names:
 - `names(patients)` - returns the column names (synonym of `colnames()` for data.frame objects)
 - `rownames(patients)` - returns the row names
- Summary:
 - `str(patients)` - structure of the object and information about the class, length and content of each column
 - `summary(patients)` - summary statistics for each column

Indexing data frames

- You can index multidimensional data structures like:

object [rows , columns]

Indexing data frames

- You can index multidimensional data structures like:

object [rows , columns]

Different ways of specifying these coordinates lead to results with different classes:

- `patients[1, 1]` # first element in the first column of the data frame (as a vector)
- `patients[1, 6]` # first element in the 6th column (as a vector)
- `patients[1:3, 7]` # first three elements in the 7th column (as a vector)
- `patients[c(1,3), 7]` # first and third elements in the 7th column (as a vector)

Indexing data frames

Different ways of specifying these coordinates lead to results with different classes:

- `patients[, 1]` # first column in the data frame (as a vector)
- `patients[3,]` # the 3rd element for all columns (as a data.frame)

Indexing data frames

Different ways of specifying these coordinates lead to results with different classes:

- `patients[, 1]` # first column in the data frame (as a vector)
- `patients[3,]` # the 3rd element for all columns (as a data.frame)
- `patients[1]`

Indexing data frames

Different ways of specifying these coordinates lead to results with different classes:

- `patients[, 1]` # first column in the data frame (as a vector)
- `patients[3,]` # the 3rd element for all columns (as a data.frame)
- `patients[1]` # first column in the data frame (as a data.frame)

Question: Is it possible subset a row as vector?

Indexing data frames and matrices

- There is a simpler way to refer to variables by name in a **data frame**
 - Namely separating the **data frame name** from the **name of the variable** with a dollar sign (**\$**)
-

```
> patients$Age
[1] 50 21 35 45 28 31 42 33 57 62
> patients$Age < 30
[1] FALSE  TRUE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE
```

Advanced indexing

- As values in R are really vectors, so indices are actually vectors, and can be numeric or logical
-

```
> s <- letters[1:5]
> s
[1] "a" "b" "c" "d" "e"
> s[c(TRUE, FALSE, TRUE, FALSE, FALSE)]
[1] "a" "c"
> a<-1:5
> a<3
[1] TRUE TRUE FALSE FALSE FALSE
> s[a<3]
[1] "a" "b"
> s[a>1 & a<3]
[1] "b"
> s[a==2]
[1] "b"
```

Operators

- arithmetic

\wedge , / , * , - , +

- comparison

\neq , $==$, $>$, $<=$, $>$, $<$

- logical

! , & , |

Exercise

- Create a data.frame using the following vectors
 - `n = 1, 2, 3, 4, 5, 6, 7, 8, 9, 10`
 - `sex = male, female, male, female, male, female, male, female, male, female`
 - `age = 23, 22, 21, 22, 24, 30, 23, 29, 19, 29`
 - `weight = 72, 90, 120, 80, 75, 65, 91, 58, 78, 50`
 - `height = 171, 185, 210, 170, 189, 150, 168, 165, 188, 143`

Exercise

- Which is the average age? [use the function **mean**]
- Which is the maximum female weight? [Subset the data frame and use the function **max**]
- Which is the minimum male weight? [Subset the data frame and use the function **min**]
- How many are the male over 180cm? [Subset the data frame]
- Which is their average age?
- Who is the younger and lighter female? [this has to be done in 2 steps; use the function **order**]
- Who is the person with the minimum BMI [weight/height²; this also has to be done in 2 steps; use the function **which.min**]

NOTE:

1. To know how to use a function use '?'; e.g. ?mean
2. In R there are several ways to achieve the same result

Exercise

- Create a data.frame using the following vectors

```
> n <- 1:10
> sex <- rep(c('male', 'female'), 5)
> age <- c(23, 22, 21, 22, 24, 30, 23, 29, 19, 29)
> weight <- c(72, 90, 120, 80, 75, 65, 91, 58, 78, 50)
> height <- c(171, 185, 210, 170, 189, 150, 168, 165, 188, 143)
> df <- data.frame(n = n, sex = sex, age = age, weight = weight, height = height)
> df
```

	n	sex	age	weight	height
1	1	male	23	72	171
2	2	female	22	90	185
3	3	male	21	120	210
4	4	female	22	80	170
5	5	male	24	75	189
6	6	female	30	65	150
7	7	male	23	91	168
8	8	female	29	58	165
9	9	male	19	78	188
10	10	female	29	50	143

Exercise

- Which is the average age?

Exercise

- Which is the average age?

```
> mean(df$age)
[1] 24.2
```

Exercise

- Which is the average age?

```
> mean(df$age)
[1] 24.2
```

- Which is the maximum female weight?

Exercise

- Which is the average age?

```
> mean(df$age)
[1] 24.2
```

- Which is the maximum female weight?

```
> ind <- df$sex == 'female'
> ind
[1] FALSE TRUE FALSE TRUE FALSE TRUE FALSE TRUE
> tmp <- df[ind,]
> tmp
   n    sex age weight height
2  2 female  22     90    185
4  4 female  22     80    170
6  6 female  30     65    150
8  8 female  29     58    165
10 10 female  29     50    143
> max(tmp$weight)
[1] 90
```

Exercise

- Which is the average age?

```
> mean(df$age)
[1] 24.2
```

- Which is the maximum female weight?

```
> ind <- df$sex == 'female'
> ind
[1] FALSE TRUE FALSE TRUE FALSE TRUE FALSE TRUE
> tmp <- df[ind,]
> tmp
   n    sex age weight height
2  2 female  22     90    185
4  4 female  22     80    170
6  6 female  30     65    150
8  8 female  29     58    165
10 10 female  29     50    143
> max(tmp$weight)
[1] 90
> max(df[df$sex == 'female',]$weight)
[1] 90
> max(df[df$sex == 'female',4])
[1] 90
> max(subset(df, sex == 'female')$weight)
[1] 90
```

Exercise

- Which is the minimum male weight?

Exercise

- Which is the minimum male weight?

```
> min(df[df$sex == 'male',]$weight)
[1] 72
```

Exercise

- Which is the minimum male weight?

```
> min(df[df$sex == 'male',]$weight)
[1] 72
```

- How many are the male over 180cm?

Exercise

- Which is the minimum male weight?

```
> min(df[df$sex == 'male',]$weight)
[1] 72
```

- How many are the male over 180cm?

```
> df[(df$sex == 'male' & df$height > 180),]
  n sex age weight height
3 3 male 21   120    210
5 5 male 24    75    189
9 9 male 19    78    188
```

Exercise

- Which is the minimum male weight?

```
> min(df[df$sex == 'male',]$weight)
[1] 72
```

- How many are the male over 180cm?

```
> df[(df$sex == 'male' & df$height > 180),]
  n sex age weight height
3 3 male 21   120    210
5 5 male 24    75    189
9 9 male 19    78    188
```

- Which is their average age?

Exercise

- Which is the minimum male weight?

```
> min(df[df$sex == 'male',]$weight)
[1] 72
```

- How many are the male over 180cm?

```
> df[(df$sex == 'male' & df$height > 180),]
  n sex age weight height
3 3 male 21   120    210
5 5 male 24    75    189
9 9 male 19    78    188
```

- Which is their average age?

```
> mean(df[(df$sex == 'male' & df$height > 180),]$age)
[1] 21.33333
```

Exercise

- Who is the younger and lighter female?

Exercise

- Who is the younger and lighter female?

```
> df1 <- df[df$sex == 'female',]  
> df1[order(df1$age, df1$weight),][1,]  
  n    sex age weight height  
4 4 female 22     80     170
```

Exercise

- Who is the younger and lighter female?

```
> df1 <- df[df$sex == 'female',]  
> df1[order(df1$age,df1$weight),][1,]  
  n    sex age weight height  
4 4 female 22     80    170
```

- Who is the person with the minimum BMI $[\text{weight}/\text{height}^2]$

Exercise

- Who is the younger and lighter female?

```
> df1 <- df[df$sex == 'female',]  
> df1[order(df1$age,df1$weight),][1,]  
  n    sex age weight height  
4 4 female 22    80    170
```

- Who is the person with the minimum BMI $[\text{weight}/\text{height}^2]$

```
> df2 <- df$weight / df$height^2  
> df2  
[1] 0.002462296 0.002629657 0.002721088 0.002768166 0.002099605 0.002888889  
[7] 0.003224206 0.002130395 0.002206881 0.002445107  
> which.min(df2)  
[1] 5  
> df[which.min(df2),]  
  n    sex age weight height  
5 5 male  24    75    189
```
