Introduction to R

Marco Chiapello

June 19, 2016

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

Overview

Introduction

Basic concepts in R

- Vector

Introduction

What R is?

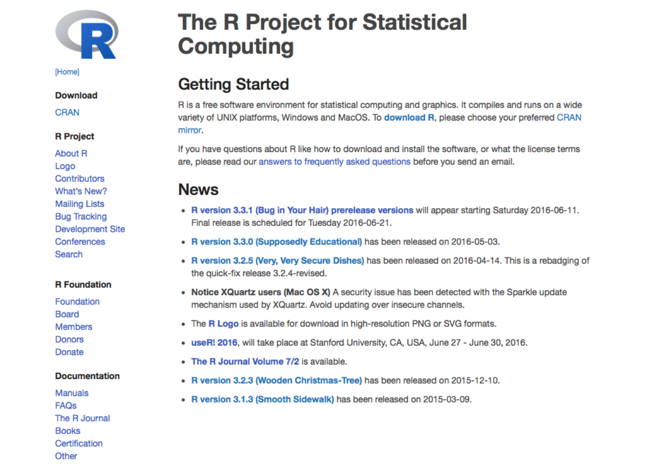
The R Project for Statistical Computing

- 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 3.3.1 (released 2016-06-21)
- Currently, the CRAN (Comprehensive R Archive Network) package repository features 8609 available packages
    - https://cran.r-project.org/web/packages/available_packages_by_name.html
- Currently, the Bioconductor repository features 1211 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

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
- There are two ways to launch R:
  1. From the command line (particularly useful if you're quite familiar with Linux)
  2. As an application called RStudio (very good for beginners)

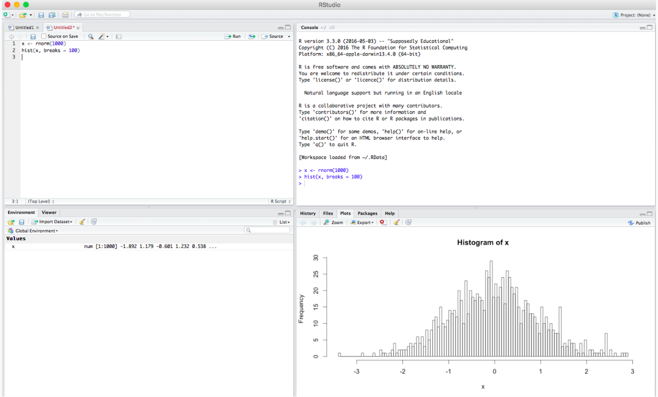
Launch R

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

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



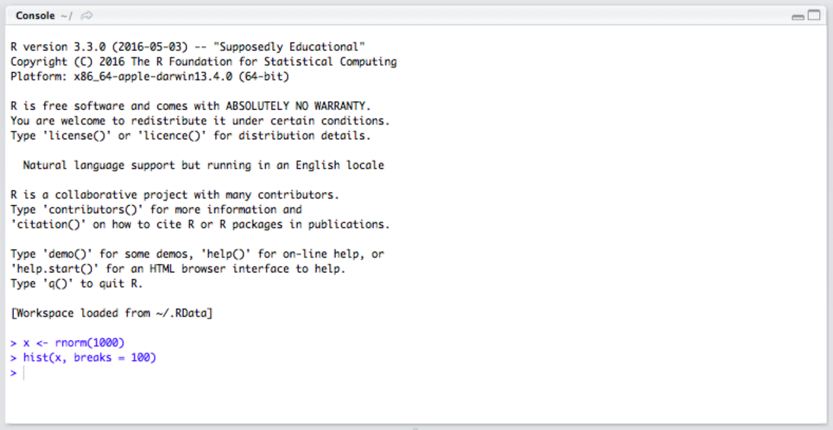
```
x <- rnorm(1000)
hist(x, breaks = 100)
```

```
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)
> 
```

```
Values
  x                    num [1:1000] -1.892 1.179 -0.601 1.232 0.538 ...
```

RStudio

R console

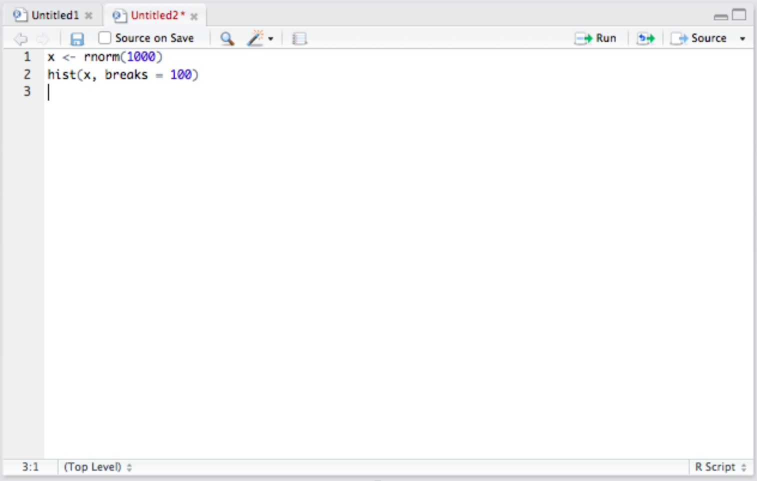


```
R version 3.3.0 (2016-05-03) -- "Supposedly Educational"
Copyright (C) 2016 The R Foundation for Statistical Computing
Platform: x86_64-apple-darwin13.4.0 (64-bit)

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

RStudio

Source editor for R scripts

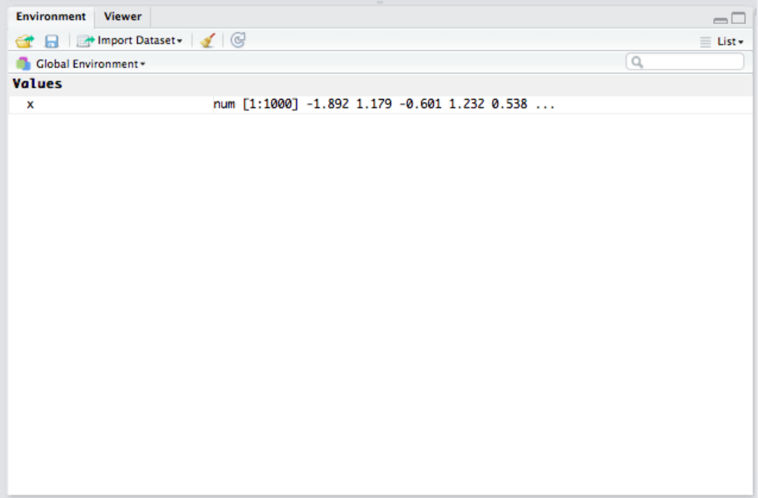


```
x <- rnorm(1000)
hist(x, breaks = 100)
```

It is the place where you can write your scripts

RStudio

Workspace

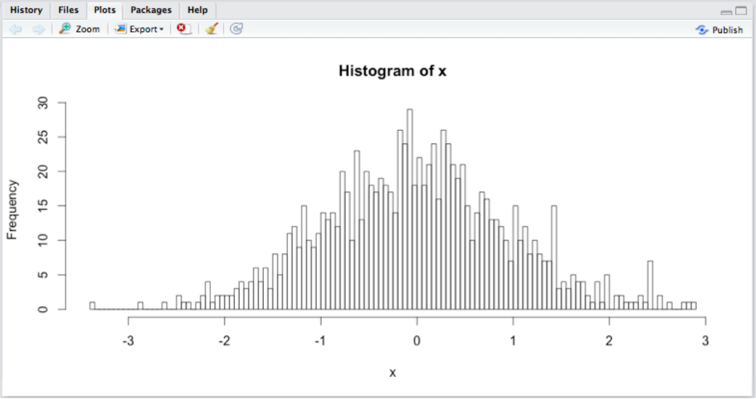


```
x                    num [1:1000] -1.892 1.179 -0.601 1.232 0.538 ...
```

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

RStudio

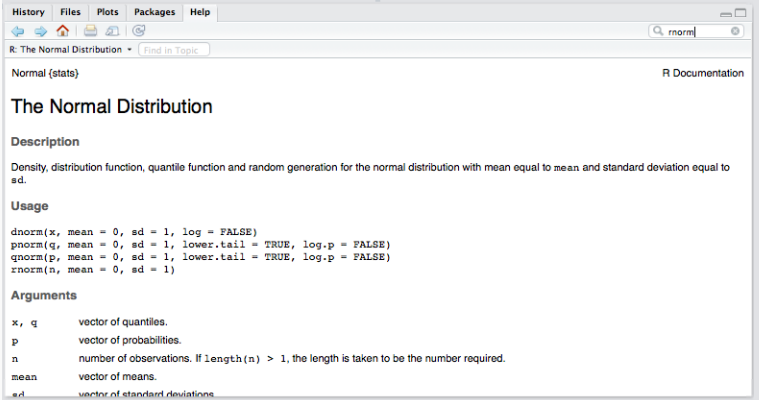
Plot pannel



It is the place where you can view your plots

RStudio

R help



Normal {stats} R Documentation

The Normal Distribution

Description

Density, distribution function, quantile function and random generation for the normal distribution with mean equal to `mean` and standard deviation equal to `sd`.

Usage

```
dnorm(x, mean = 0, sd = 1, log = FALSE)
pnorm(q, mean = 0, sd = 1, lower.tail = TRUE, log.p = FALSE)
qnorm(p, mean = 0, sd = 1, lower.tail = TRUE, log.p = FALSE)
rnorm(n, mean = 0, sd = 1)
```

Arguments

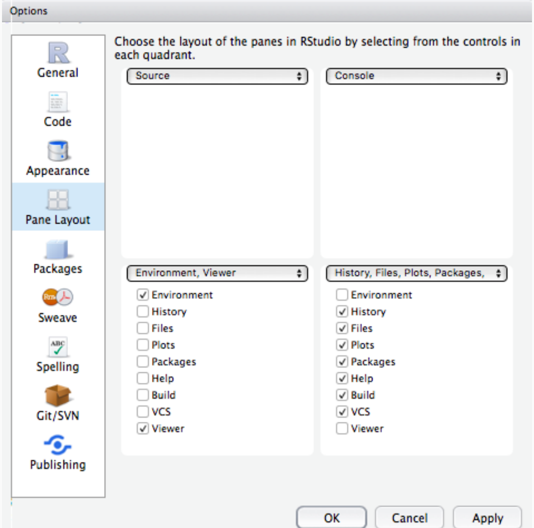
`x, q`	vector of quantiles.
`p`	vector of probabilities.
`n`	number of observations. If `length(n) > 1`, the length is taken to be the number required.
`mean`	vector of means.
`sd`	vector of standard deviations.

It is the place where you can find help

RStudio

The GUI is divided into 4 main sub-windows

These sub-windows are customizable



Choose the layout of the panes in RStudio by selecting from the controls in each quadrant.

Source

Console

Environment, Viewer

- ☑ Environment
- □ History
- □ Files
- □ Plots
- □ Packages
- □ Help
- □ Build
- □ VCS
- ☑ Viewer

History, Files, Plots, Packages,

- □ Environment
- ☑ History
- ☑ Files
- ☑ Plots
- ☑ Packages
- ☑ Help
- ☑ Build
- ☑ VCS
- □ Viewer

Basic concepts in R

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

Numbers

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. Alternatevely, you can write your own function

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

Vector

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
```

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

Vector

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.

Vector

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

Vector

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!

Vector

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:

$$?functionnamehere \tag{1}$$

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

$$?` : \tag{2}$$

Vector

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

Vector

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
```

Vector

This gives us the same output as 1:20. 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.

Vector

This gives us the same output as 1:20. 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
```

Vector

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.

Vector

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
```

Vector

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

Try it now

Vector

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
```

Vector

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.

Vector

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

Vector

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
```

Vector

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

Vector

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
```

Vector

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

Try it out

Vector

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
```

Vector

If we want all the elements bigger than 7.

Try it out

Vector

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
```

Vector

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
Homo sapiens	3,102	20,774
Mus musculus	2,731	23,139
Drosophila melanogaster	169	13,937
Caenorhabditis elegans	100	20,532
Saccharomyces cerevisiae	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.

Vector

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
```

Vector

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
```

Vector

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.
       1.004545        1.270908       12.370118       30.798000
```

Vector

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.
       1.004545        1.270908       12.370118       30.798000
```

- 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
```

Vector

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.bases[1]/coding.bases[5])
[1] "H. sapiens"
> names(genome.size[1]/genome.size[5])
[1] "H. sapiens"
```

Vector