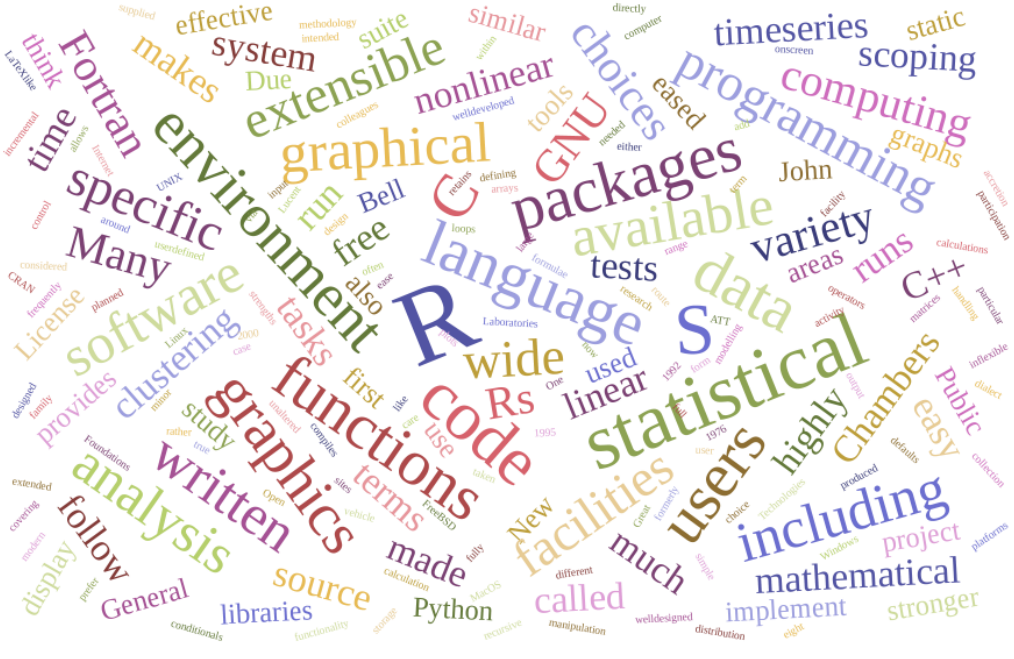
What is R?

R is an open-source language and environment for statistical computing and graphics. It includes:

- A data handling and storage facility
- A suite of operators for calculations on matrices
- A large collection of intermediate tools for data analysis
- Graphical facilities for data analysis and display
- A simple and effective programming language



Who uses R?

Traditionally academics and research, now rapidly expanding into the enterprise market.

Worldwide, Apr 2020 compared to a year ago:

Rank	Change	Language	Share	Trend
1		Python	30.61 %	+3.9 %
2		Java	18.45 %	-1.9 %
3		Javascript	7.91 %	-0.4 %
4		C#	7.27 %	-0.0 %
5		PHP	6.07 %	-1.1 %
6		C/C++	5.76 %	-0.2 %
7		R	3.8 %	-0.2 %

Credit to http://pypl.github.io/PYPL.html

Why should you learn R?

Pros:

- Arguably the most common language for data scientists
- Created with statistics and data in mind
- Excellent for learning data science
- Open source
- Wide range of high-quality packages

Cons:

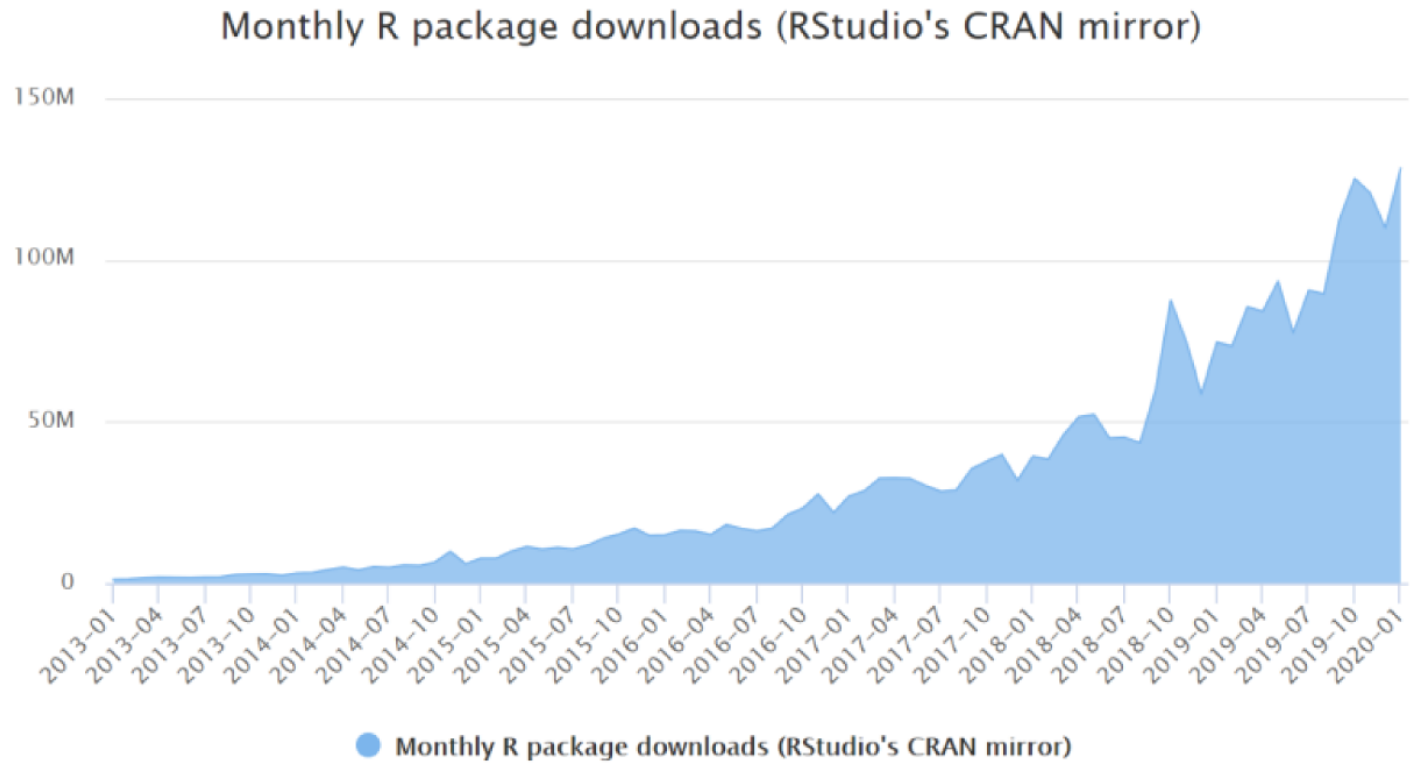
- Performance
- Domain specificity
- A few unusual features compared to other languages

Some alternatives to R:

- Python: general-purpose programming, with data science libraries
- SAS: commercial and expensive, slower development

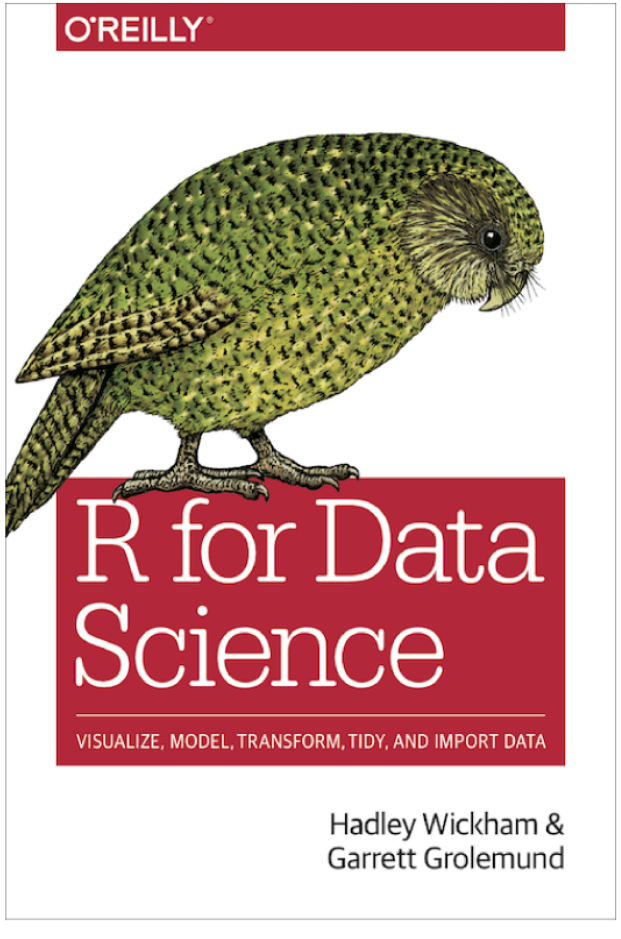
What makes R good?

As of April 2020, 15,515 packages on CRAN, 1823 on Bioconductor and many others on GitHub.



Credit to http://blog.revolutionanalytics.com

Main reference



Available here: http://r4ds.had.co.nz/

Setting up your R environment

Installing R

R is open-source and cross-platform (Linux, Mac, Windows).

You can download it from to the R Project website.

Download the latest version for your OS and follow the instructions.

Each year: one new version of R, and 2-3 minor releases.

It’s a good idea to update regularly.

Running R code

Interpreter mode:

- open a terminal
- launch R by typing “R”
- type R commands interactively in the command line

Scripting mode:

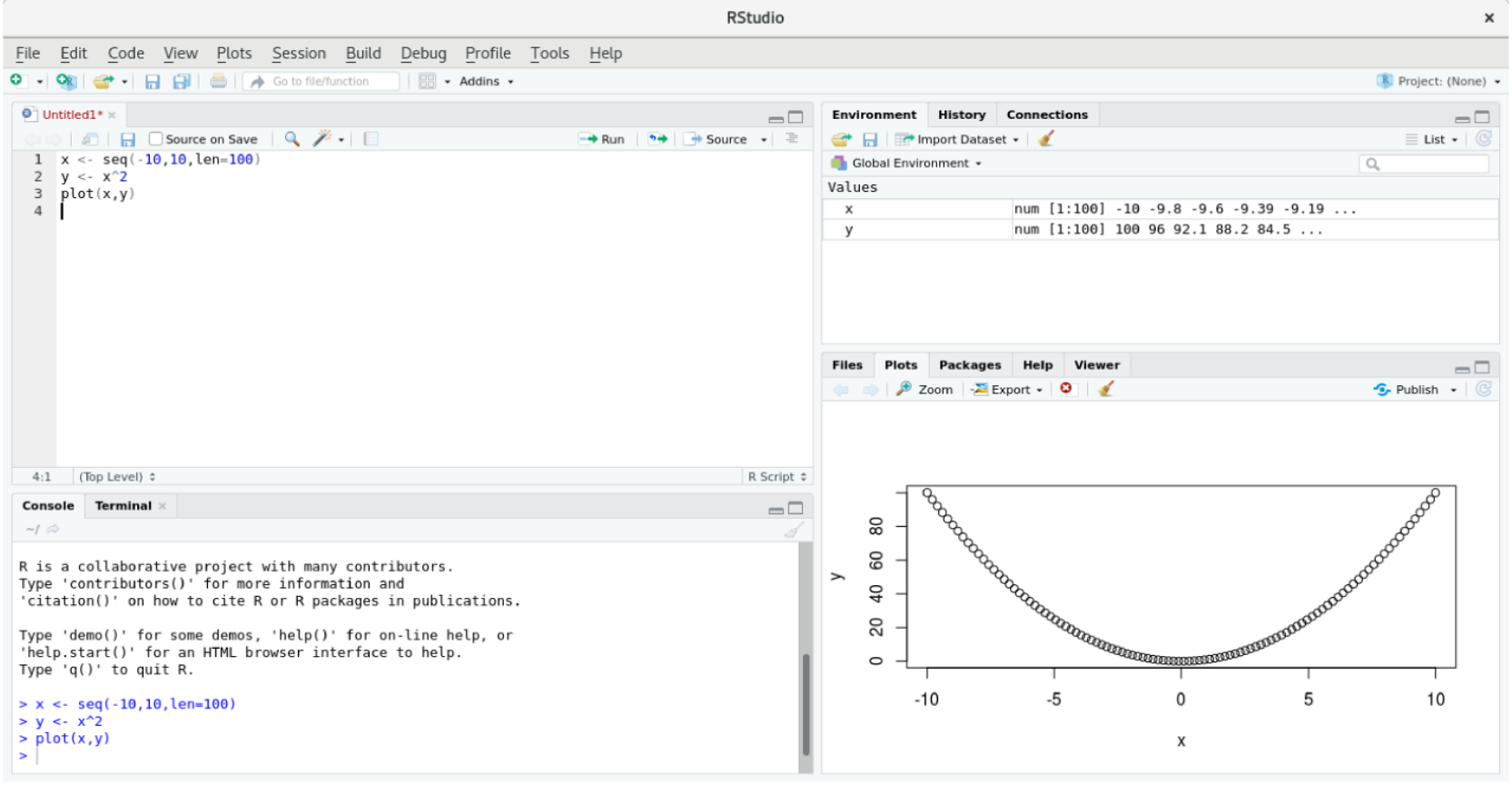
- write a text file containing all commands to be executed
- save it as an R script file (e.g. “script.R”)
- execute the code from terminal with “Rscript script.R”

RStudio is a development environment that offers both, and much more.

Installing RStudio

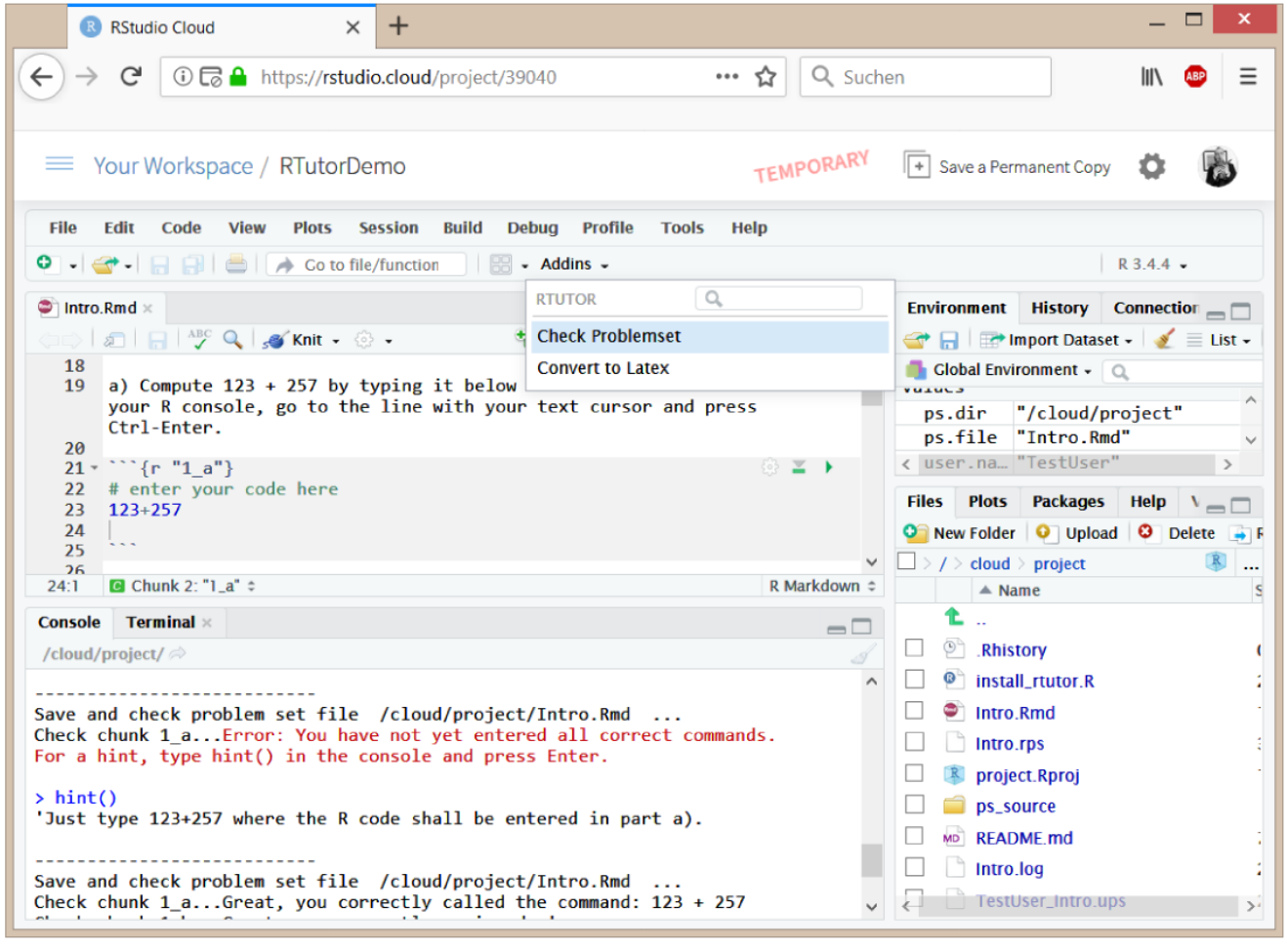
RStudio is open-source and cross-platform (Linux, Mac, Windows).

Download and install the latest version for your OS from the official website.



Rstudio Cloud

Rstudio Cloud allows you to use R within Rstudio without any installation.



Credit to https://www.r-bloggers.com/

R packages

R packages are a collection of R functions, complied code and sample data.

They are stored under a directory called library in the R environment.

Some packages are installed by default and are automatically loaded.

Additional packages are available from:

- CRAN: the first and largest R repository
- Bioconductor: packages for the analysis of biological data
- GitHub: packages under development

In R, you can list the installed packages by typing

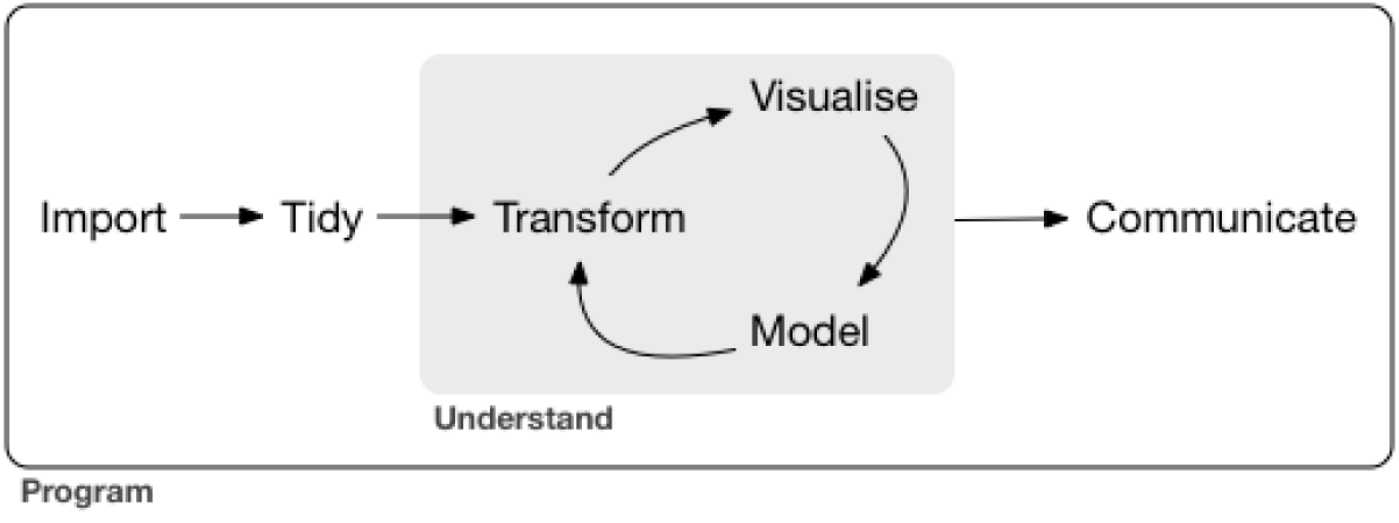
```
installed.packages()
```

The `tidyverse`

The `tidyverse` is a coherent system of R packages for data manipulation, exploration and visualization.

Intended to make data scientists more productive:

- guide through workflows
- facilitate communication,
- result in reproducible products.



Credit: http://r4ds.had.co.nz/

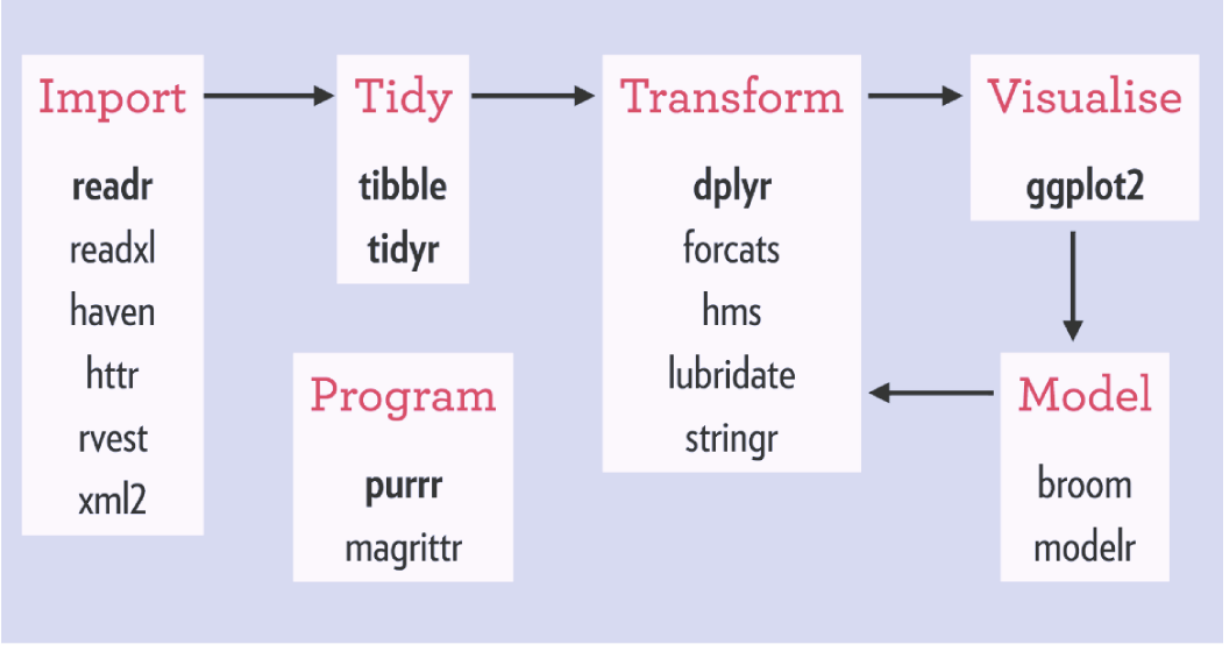
Installing the `tidyverse`

Install `tidyverse` from CRAN:

```
install.packages("tidyverse")
```

Once installed, load it with

```
library(tidyverse)
```



Credit: https://rviews.rstudio.com/

Basics of coding in R

R as a calculator

You can use R as a calculator

```
1.2+2
```

```
## [1] 3.2
```

```
(15-10)^2 / 5
```

```
## [1] 5
```

```
2 * sin(pi/2)
```

```
## [1] 2
```

```
exp(1)
```

```
## [1] 2.718282
```

You can create new objects with <-

```
a <- 3*2
(5+a) / 2
```

```
## [1] 5.5
```

In general, assignments are:

```
object_name <- value
```

To inspect an object, type its name

```
a
```

```
## [1] 6
```

Calling functions

R has a large collection of built-in functions that are called like this:

```
function_name(arg1 = val1, arg2 = val2, ...)
```

Some arguments are set by default, others must be specified.

```
seq(1, 10)
```

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

```
seq(1, 10, by = 2)
```

```
## [1] 1 3 5 7 9
```

To learn more about a function, look it up in the help pages:

```
?seq
```

Naming objects

Object names must start with a letter, and can only contain:

- letters
- numbers
- the character `_`
- the character `.`

```
a <- 0
first.variable <- 1
First.Variable <- 2
variable_2 <- 1 + first.variable
very_long_name.3 <- 4
```

Some words are reserved in R and cannot be used as object names:

- `Inf` and `-Inf` which respectively stand for positive and negative infinity.
  R will return this value if results are numbers that are too big.
- `NULL` denotes a null object.
- `NA` represents a missing value (“Not Available”).
- `NaN` means “Not a Number”.
  R will return this value if a computation is undefined.

Data types

There are 6 atomic classes:

- Logical: TRUE/FALSE (or T/F)
- Numeric: 12.4, 30, 2, 1009, 3.141593
- Integer: 2L, 34L, -21L, 0L
- Complex: 3 + 2i, -10 - 4i
- Character: “a”, “23.5”, “good”, “Hello world!”, “TRUE”
- Raw: used to hold raw bytes.

Objects can have different structures based on atomic class and dimensions:

Dimensions	Homogeneous	Heterogeneous
1d	vector	list (generic vector)
2d	matrix	data.frame
nd	array	

R also supports more complicated objects built upon these.

Vectors

Vectors are the simplest data structure: contiguous cells containing data.

```
# Create vectors with "combine"
x1 <- c(1,2,3,4,5)
x2 <- c("apple", "pear", "orange")
# Look at contents of x1
x1
```

```
## [1] 1 2 3 4 5
```

```
# Non-character values are coerced
# Including in () prints content
( x3 <- c("apple", 1.1, TRUE) )
```

```
## [1] "apple" "1.1"   "TRUE"
```

```
# Generate numerical sequence
( x3 <- seq(1, 5) )
```

```
## [1] 1 2 3 4 5
```

```
# Indexing by position
x1[2]
```

```
## [1] 2
```

```
# Negative indexing indicates
# complement positions
x1[-2]
```

```
## [1] 1 3 4 5
```

```
# Logical indexing
x1[c(T,F,F,F,T)]
```

```
## [1] 1 5
```

Vector arithmetics

Element-wise operations between vectors of same length.

```r
# Create two vectors
v1 <- c(12,9,7,3,8,16)
v2 <- c(2,3,7,10,0,8)
# Vector addition
(vec.sum <- v1+v2)
```

```
## [1] 14 12 14 13  8 24
```

```r
# Vector subtraction
(vec.difference <- v1-v2)
```

```
## [1] 10  6  0 -7  8  8
```

```r
# Vector multiplication
(vec.product <- v1*v2)
```

```
## [1]  24  27  49  30   0 128
```

```r
# Vector division
(vec.ratio <- v1/v2)
```

```
## [1] 6.0 3.0 1.0 0.3 Inf 2.0
```

```r
# Vector concatenation
vec.concat <- c(v1, v2)
# Size of vector
length(vec.concat)
```

```
## [1] 12
```

Recycling

Recycling is an automatic lengthening of vectors in certain settings.

```
# Element-wise multiplication
v1 <- c(1,2,3,4,5)
v1 * 2
```

```
## [1]  2  4  6  8 10
```

R repeats the shorter vector until the length of the longer vector is reached.

```
# Element-wise multiplication
v1 + c(1,2)
```

```
## Warning in v1 + c(1, 2): longer object length is not a multiple of shorter
## object length
```

```
## [1] 2 4 4 6 6
```

Logical operations

```
# Comparisons (==,!=,>,>=,<,<=)
1 == 2
```

```
## [1] FALSE
```

```
# Check whether number is even
# (%% is the modulus)
(5 %% 2) == 0
```

```
## [1] FALSE
```

```
# Logical indexing
x <- seq(1,10)
x[(x%%2) == 0]
```

```
## [1]  2  4  6  8 10
```

```
# Element-wise operations
c(1,2,3) > c(3,2,1)
```

```
## [1] FALSE FALSE  TRUE
```

```
# Check whether numbers are even,
# one by one
(seq(1,4) %% 2) == 0
```

```
## [1] FALSE  TRUE FALSE  TRUE
```

```
# Logical indexing
x <- seq(1,10)
x[x>=5]
```

```
## [1]  5  6  7  8  9 10
```

Matrices

2D vectors. A matrix can be created with the function

```
matrix(data, nrow, ncol, byrow, dimnames)
```

where:

- `data`: input vector with elements of the matrix
- `nrow`: number of rows to be created
- `ncol`: number of columns to be created (default: `NULL`)
- `byrow`: whether matrix is filled by columns (default: `FALSE`)
- `dimnames`: list of length 2, gives row and column names (default: `NULL`)

```
(M <- matrix(seq(1,9), nrow = 3))
```

```
##      [,1] [,2] [,3]
## [1,]    1    4    7
## [2,]    2    5    8
## [3,]    3    6    9
```

Accessing matrix elements

```
M <- matrix(seq(1,9), nrow = 3)
```

```
M[2,3] # 2nd row, 3rd column
```

```
## [1] 8
```

```
M[2,] # The 2nd row
```

```
## [1] 2 5 8
```

```
M[,3] # The 3rd column
```

```
## [1] 7 8 9
```

```
M[c(2,3),] # 3rd and 2nd row
```

```
##      [,1] [,2] [,3]
## [1,]    2    5    8
## [2,]    3    6    9
```

```
M[,-2] # 1st and 2nd column
```

```
##      [,1] [,2]
## [1,]    1    7
## [2,]    2    8
## [3,]    3    9
```

Matrix computations

```
# Create two 2x3 matrices.
(A <- matrix(seq(1,6), nrow=2))
```

```
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6
```

```
(B <- matrix(seq(-2,3), nrow=2))
```

```
##      [,1] [,2] [,3]
## [1,]   -2    0    2
## [2,]   -1    1    3
```

```
A*B # Element-wise product
```

```
##      [,1] [,2] [,3]
## [1,]   -2    0   10
## [2,]   -2    4   18
```

```
t(A) # Matrix transpose
```

```
##      [,1] [,2]
## [1,]    1    2
## [2,]    3    4
## [3,]    5    6
```

Matrix algebra

True matrix multiplication $A \times B$ with $A\ (m \times n)$ and $B\ (n \times p)$:

$$(A \times B)_{ij} = \sum_{k=1}^{n} A_{ik} B_{kj}$$

```
A %*% t(B) # 2x2
```

```
##      [,1] [,2]
## [1,]    8   17
## [2,]    8   20
```

```
t(A) %*% B # 3x3
```

```
##      [,1] [,2] [,3]
## [1,]   -4    2    8
## [2,]  -10    4   18
## [3,]  -16    6   28
```

More on matrix algebra here.

Lists

Contain elements of different types, e.g., numbers, vectors and other lists.

```r
# Unnamed list
v = c("Jan","Feb","Mar")
M = matrix(c(1,2,3,4),nrow=2)
( u.list <- list(v, TRUE, M) )
```

```
## [[1]]
## [1] "Jan" "Feb" "Mar"
##
## [[2]]
## [1] TRUE
##
## [[3]]
##      [,1] [,2]
## [1,]    1    3
## [2,]    2    4
```

```r
# Access first element
u.list[[1]]
```

```r
# Named list
(n.list <- list(first="Jane",
                last="Doe",
                yearOfBirth=1990))
```

```
## $first
## [1] "Jane"
##
## $last
## [1] "Doe"
##
## $yearOfBirth
## [1] 1990
```

```r
# Access "yearOfBirth" element
n.list$yearOfBirth
```

```
## [1] 1990
```

Data frames

A data frame is a table or a 2D rectangular array-like structure.

- Columns can store data different data types e.g. logical, numeric, character
- Each column must contain the same number of data items
- The column names should be non-empty
- The row names should be unique

The elements can be accessed by indexing (just like matrices).

```
##    name salary start_date
## E1 Rick 623.30 2012-01-01
## E2  Dan 515.20 2013-09-23
## E3 Alex 611.00 2014-11-15
## E4 Ryan 729.00 2014-05-11
## E5 John 843.25 2015-03-27
```

Columns can also be accessed by name

```
employees$name
```

```
## [1] "Rick" "Dan"  "Alex" "Ryan" "John"
```

Creating data frames

```r
# Create a data frame
employees <- data.frame(
  row.names = c("E1","E2","E3","E4","E5"),
  name = c("Rick","Dan","Alex","Ryan","John"),
  salary = c(623.3,515.2,611.0,729.0,843.25),
  start_date = as.Date(c("2012-01-01","2013-09-23","2014-11-15", "2014-05-11","2015-03-
        27"))
  )
employees # Print the data frame
```

```
##    name salary start_date
## E1 Rick 623.30 2012-01-01
## E2  Dan 515.20 2013-09-23
## E3 Alex 611.00 2014-11-15
## E4 Ryan 729.00 2014-05-11
## E5 John 843.25 2015-03-27
```

```r
str(employees) # See the structure of the data frame
```

Summarizing data frames

If the data frame is large we do not want to print all of it.

```
head(employees, 2) # Print first 2 rows of the employees data frame
```

```
##    name salary start_date
## E1 Rick  623.3 2012-01-01
## E2  Dan  515.2 2013-09-23
```

```
summary(employees) # Print a summary
```

```
##    name       salary        start_date
##  Alex:1   Min.   :515.2   Min.   :2012-01-01
##  Dan :1   1st Qu.:611.0   1st Qu.:2013-09-23
##  John:1   Median :623.3   Median :2014-05-11
##  Rick:1   Mean   :664.4   Mean   :2014-01-14
##  Ryan:1   3rd Qu.:729.0   3rd Qu.:2014-11-15
##           Max.   :843.2   Max.   :2015-03-27
```

Factors

Factors are used to categorize the data and store it as levels.

Useful for variables that can take a limited number of unique values.

```r
days = c("Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun")
is.factor(days) # Check whether these are factors
```

```
## [1] FALSE
```

```r
class(days) # Indeed these are strings of characters
```

```
## [1] "character"
```

```r
( days <- factor(days) ) # Convert to factors
```

```
## [1] Mon Tue Wed Thu Fri Sat Sun
## Levels: Fri Mon Sat Sun Thu Tue Wed
```

Factor ordering

If not specified, R will order character type by alphabetical order.

```
( days.sample <- sample(days, 3) ) # Randomly pick 3 days
```

```
## [1] Fri Sun Sat
## Levels: Fri Mon Sat Sun Thu Tue Wed
```

```
# Create factor with given levels
( days.sample <- factor(days.sample, levels=days) )
```

```
## [1] Fri Sun Sat
## Levels: Mon Tue Wed Thu Fri Sat Sun
```

```
# Create factor with ordered levels
( days.sample <- factor(days.sample, levels=days, ordered=TRUE) )
```

Dates

R makes it easy to work with dates.

```
# Define a sequence of dates
x <- seq(from=as.Date("2018-01-01"),to=as.Date("2018-05-31"), by=1)
table(months(x)) # Convert to months and count occurrences
```

```
##
##    April February  January    March      May
##       30       28       31       31       31
```

```
# Number of days until the end of the year.
as.Date('2018-12-31' ) - Sys.Date()
```

```
## Time difference of -463 days
```

Type `?strptime` for a list of possible date formats.

Random numbers

You can generate vectors of random numbers from different distributions.

To make your results reproducible, provide a seed for the generator.

```r
set.seed(123)
```

```r
runif(5, min = 0, max = 1)    # Uniform distribution
```

```
## [1] 0.2875775 0.7883051 0.4089769 0.8830174 0.9404673
```

```r
rnorm(5, mean = 0, sd = 1)    # Normal distribution
```

```
## [1] -1.6895557  1.2394959 -0.1089660 -0.1172420  0.1830826
```

Random sampling

You can generate a random sample from the elements of a vector using the function `sample`.

```
v <- 1:10  # Population vector  (equivalent to: v <- seq(1,10))
```

```
sample(v, 5)    # Sampling without replacement
```

```
## [1] 5 3 9 1 4
```

```
sample(v, 10, replace=TRUE)    # Sampling with replacement
```

```
##  [1]  9  9  3  8 10  7 10  9  3  4
```

Tables

The contents of a discrete vector can be easily summarized in a table.

```r
v <- 1:10  # Population vector
x <- sample(v, 1000, replace=TRUE) # Random sample
```

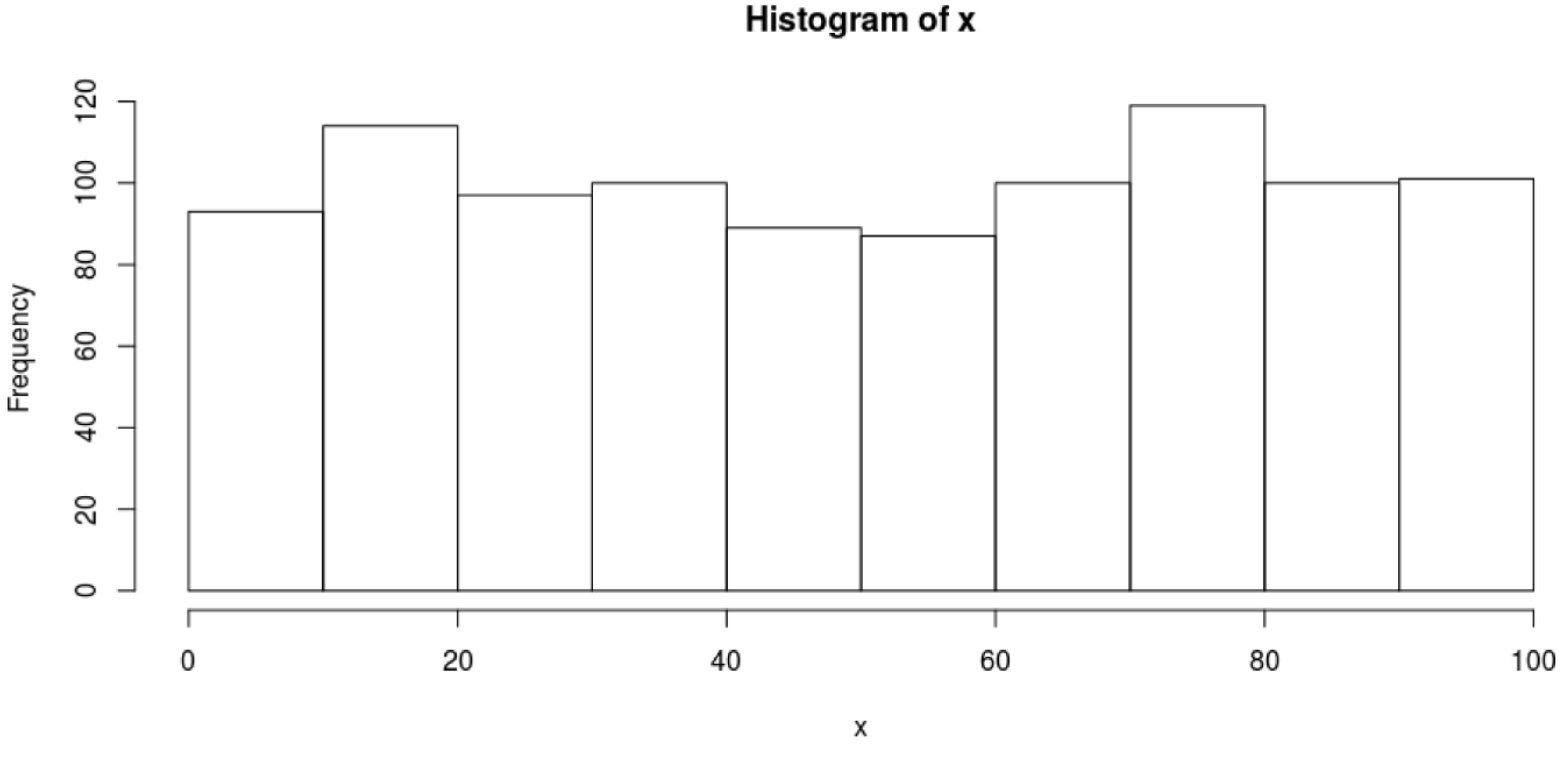
```r
table(x)
```

```
## x
##   1   2   3   4   5   6   7   8   9  10
##  92  90 105  86  94  93 122 108 103 107
```

Histograms

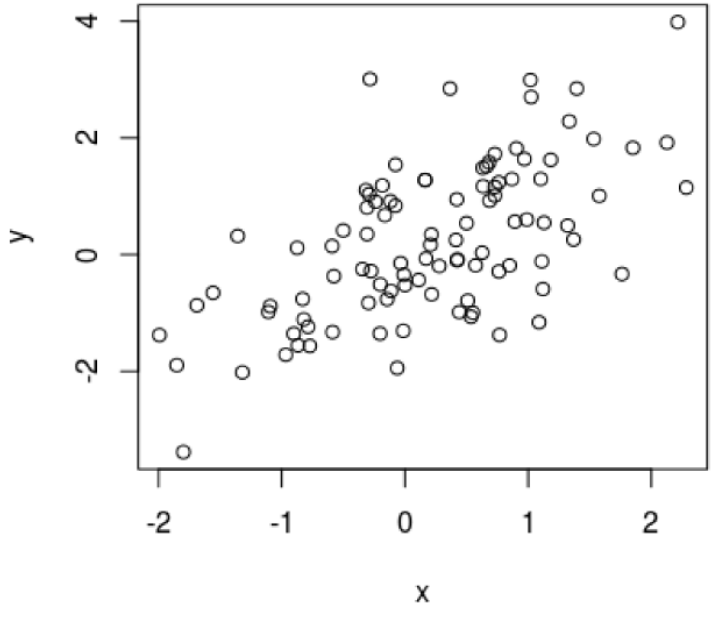
The contents of a discrete or continuous vector can be easily summarized in a histogram.

```
x <- sample(1:100, 1000, replace=TRUE)
hist(x)
```



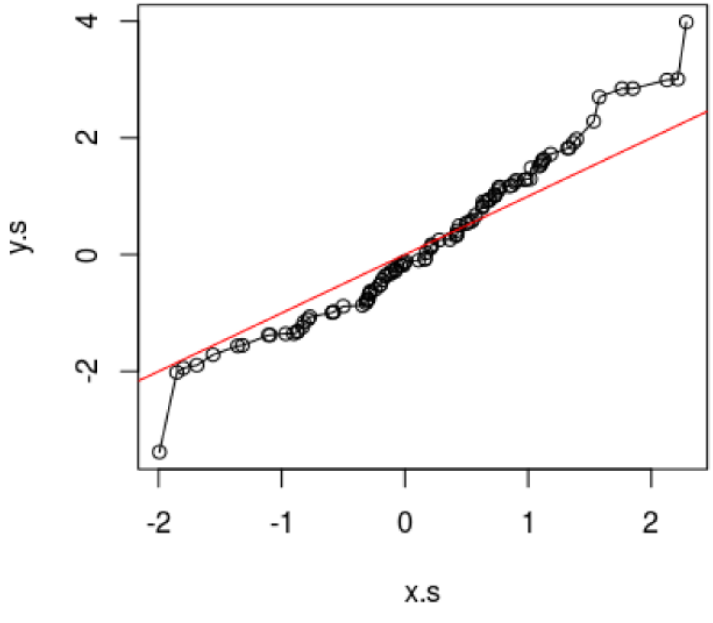
Basic plotting

```r
x <- rnorm(100, mean = 0, sd = 1)
y <- x + rnorm(100, mean = 0, sd = 1)
plot(x,y)
```



Basic plotting (II)

```
x.s <- sort(x)
y.s <- sort(y)
plot(x.s, y.s); lines(x.s, y.s); abline(a=0, b=1, col="red")
```



Next time we will learn how to make fancier plots.