




# Primer on R

---

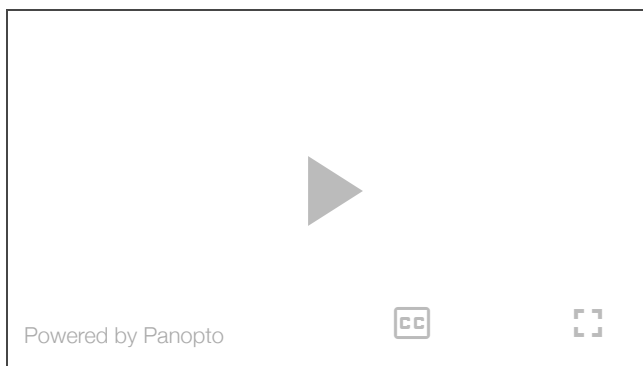
## Getting R and R Studio

Before we start with the basics of R make sure you have the latest version of R. To install R as a desktop app, go to the [R Project website](http://www.r-project.org)  [\\_.\(http://www.r-project.org\)](http://www.r-project.org)\_. For development, an Integrated Development Environment (IDE) is recommended. R Studio is the leading IDE for R and can be obtained from [\\_.\(http://www.rstudio.com\)](http://www.rstudio.com)\_. Rather than installing R and R Studio on your desktop, you can also use a cloud version at [RStudio.cloud](http://rstudio.cloud)  [\\_.\(http://rstudio.cloud\)](http://rstudio.cloud)\_.

## 32-bit vs 64-bit R

On Windows you have a choice between a 32-bit and a 64-bit version of R. We recommend you use the 64-bit version and that you allow R to auto-detect the rendering engine. The 64-bit version allows for larger objects and allows your program to use more than 2GB of memory (32-bit addressing limits you to a max of 4GB but Windows reserves 2GB for the operating systems). The 32-bit version of R is provided for backwards compatibility with older packages and is only used when a package does not have a 64-bit version. Using the [cloud version](http://rstudio.cloud)  [\\_.\(http://rstudio.cloud\)](http://rstudio.cloud) eliminates those choices.

## Writing R on the Console vs R Notebooks



While ad-hoc code can be quickly entered on the R Console, it is recommended that all work in R be done in R Notebooks -- a style of R Markdown files. This video tutorial explains the difference between the two and how to create and work with R Notebooks and Markdown.

## Basic R

Data is generally stored in R as a *dataframe*. A dataframe is similar to a spreadsheet in that it has columns and rows. Each row holds a data record (also called an observation, a case, a row, or an object). All values in a column must be of the same type. Dataframes are created by:

- Reading data from an external file
- Retrieving data from a URL
- Creating an object directly from the command line
- Instantiating an object from within a program

## Expressions

R can be directly used to solve simple or complex mathematical expressions.

```
12*21
```

```
## [1] 252
```

```
# [1] in the above answer indicates the index of your results.  
# R always shows the result with index for each row.
```

```
((2^3)*5)-1
```

```
## [1] 39
```

```
# sqrt and exp are built-in functions in R for finding Square root and exponential respectively.  
sqrt(4)* exp(2)
```

```
## [1] 14.77811
```

## Variables & Identifiers

Holding a value in a variable is done through *assignment*. Once you assign a value to a variable, the variable becomes an R object. There are two ways to do an assignment, with a '=' and with a '<-' . The latter is the preferred way in R.

Note that variables do not need to be explicitly defined. The first time a variable is assigned a value defines the variable and its type. The type is based on the value that is assigned. Unlike other programming languages such as C++, C#, or Java, R is not strongly typed: the type of a variable can change when a value of a different type is assigned. A variable can be used in an expression. Its value can be inspected by just using the variable by itself.

The value of a variable can be displayed either by using the variable by itself or using the `print()` function.

```
# assignment with '=' of a number  
x = 12  
# inspect (print/display) the value  
x
```

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

```
# assignment a new value and change its type to "text"  
x = "Hello"  
x
```

```
## [1] "Hello"
```

```
# assignment with '<-'  
x <- 12  
print(x)
```

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

The rules for naming an identifier (variable, function, or package name) for an object are as follows:

- identifiers are case-sensitive and cannot contain spaces or special characters such as #, %, \$, @, \*, &, ^, !, ~
- an identifier must start with a letter, but may contain any combination of letters and digits thereafter
- special characters dot (.) and underscore (\_) are allowed

The dot (.) is a regular character in R and that can be confusing as other language (e.g., Java) use dot to designate property or method access, e.g, in Java `x.val` means that you are accessing the `val` property of the object `x`.

Some examples of legal variable names are: `df`, `df2`, `df.txns`, `df_all2017`. These are some illegal variable names: `2df` (cannot start with a digit), `rs$all` (cannot contain a \$; the \$ is used to access columns in a dataframe), `rs#` (only . and \_ are allowed in addition to digits and letters).

It is considered good programming practice to give identifiers a sensible name that hints as to what is stored in the variable rather than using random name like `x`, `val`, or `i33`. Identifiers should be named consistently. Many programmers use one of two styles:

- underscores, e.g., `interest_rate`
- camelCase, e.g., `squareRoot`, `graphData`, `currentWorkingDirectory`

Note that R is case sensitive which means that R treats the identifiers `AP` and `ap` as different objects. As a side note, files may also be case sensitive but that depends on the operating system. MacOS and Linux are case sensitive, while Windows is case aware but not case sensitive. For example, on MacOS and Linux there is a difference between “AirPassengers.txt” and “airpassengers.txt” while on Windows there is not. SQL is also not case sensitive. It is a best practice to assume case sensitivity.

## Coercion

Types can be converted using `as.xxxx` functions, e.g., `as.numeric()`, `as.date()` or `as.string()`. Conversions are often necessary when data is read from CSV or XML files or databases as the data is always text (strings).

```
x <- 3.14  
v <- as.integer(x)    # removes fraction (no rounding)  
  
s <- "3.14"           # numbers in quotes are text: s + 1 would be an error
```

```
w <- as.numeric(s)    # converts text to a number
w <- w + 2.71
```

## Rounding

```
x <- 23 / 77
print(x)
```

```
## [1] 0.2987013
```

```
w <- round(x,3)        # round to three decimals
print(w)
```

```
## [1] 0.299
```

```
w <- floor(x)          # round down to nearest integer
print(w)
```

```
## [1] 0
```

```
w <- ceiling(x)        # round up
print(w)
```

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

```
w <- trunc(x)          # remove decimals
print(w)
```

```
## [1] 0
```

## Functions

R has numerous built-in functions. Many more are found in the hundreds of packages (libraries of functions) available for download. R functions are invoked (or called) by using their name with parenthesis after. The parenthesis distinguish a function from a data object.

### Function Documentation

Details of any built-in functions can be accessed by adding a question mark (?) in front of the function. More information can generally be found through a web search.

```
?sum
```

## Built-in Datasets

R has numerous built-in datasets that are useful for testing and learning R. They should not be used for any actual analysis. Some examples are `mtcars`, `sunspots`. Most built-in datasets are dataframe objects. Using the `head()` and `tail()` functions is recommended so that only the first few or the last few rows of a dataframe are displayed when inspecting a dataframe.

```
mtcars          # displays the full data frame
```

```
head(mtcars)    # displays only first 6 rows
```

```
##
## Mazda RX4          21.0  6  160 110 3.90 2.620 16.46 0 1  4  4
## Mazda RX4 Wag      21.0  6  160 110 3.90 2.875 17.02 0 1  4  4
## Datsun 710          22.8  4  108  93 3.85 2.320 18.61 1 1  4  1
## Hornet 4 Drive      21.4  6  258 110 3.08 3.215 19.44 1 0  3  1
## Hornet Sportabout   18.7  8  360 175 3.15 3.440 17.02 0 0  3  2
## Valiant             18.1  6  225 105 2.76 3.460 20.22 1 0  3  1
```

```
head(mtcars,2)  # displays only first 2 rows
```

```
##
## Mazda RX4          21  6  160 110  3.9 2.620 16.46 0 1  4  4
## Mazda RX4 Wag      21  6  160 110  3.9 2.875 17.02 0 1  4  4
```

## Vectors

Vectors, a collection of numbers or text, are another fundamental object type in R. Vectors can be generated as sequences or collections.

```
v <- c(2,9,3,11)      # vector of four numbers
sum(v)                 # add the numbers in the vector
```

```
## [1] 25
```

```
v <- 10:30             # vectors of numbers from 10 to 30
v <- seq(10,100,by = 5) # vector of multiples of 5

# vector of text (string) objects
v <- c("Northeastern","MIT","Cornell")

l <- length(v)         # length (number elements)
```

Vectors can be concatenated, *i.e.*, put together.

```
v <- c(1,3,5,7,11)
w <- c(1,2,3,5,8)

q <- c(v,w)
q
```

```
## [1] 1 3 5 7 11 1 2 3 5 8
```

```
q <- round(q,2)           # round all values
```

Vectors of the same length can also be used in algebraic operators, such as addition, subtraction, etc. For example, adding two vectors adds each of the elements in the same position, *i.e.*,  $v = \{v_1, v_2, \dots, v_n\} + w = \{w_1, w_2, \dots, w_n\}$  results in  $\{v_1 + w_1, v_2 + w_2, \dots, v_n + w_n\}$ . Vectors of unequal length only add the number of elements of the smallest vector.

```
v <- c(1,3,5,7,11)
w <- c(1,2,3,5,8,13)
v + w
```

```
## Warning in v + w: longer object length is not a multiple of shorter object
## length
```

```
## [1]  2  5  8 12 19 14
```

```
r <- v / w
```

```
## Warning in v/w: longer object length is not a multiple of shorter object
## length
```

## Common Statistical Functions

### Random Numbers and Samples

```
# vector of 4 random numbers
runif(4)
```

```
## [1] 0.1713571 0.5989702 0.5068705 0.4158434
```

```
# vector of 3 random numbers from 0 to 100
runif(3, min=0, max=100)
```

```
## [1] 44.15431 13.66275 94.51060
```

```
# vector of 3 random integers from 0 to 100
# use max=101 because it will never actually equal 101
floor(runif(3, min=0, max=101))
```

```
## [1] 60  1 37
```

```
# same result but using a different approach
sample(1:100, 3, replace=TRUE)
```

```
## [1] 49 39 42
```

```
# random sample WITHOUT replacement:  
sample(1:100, 3, replace=FALSE)
```

```
## [1] 6 8 46
```

```
# sample of random numbers from normal distribution  
rnorm(4)
```

```
## [1] -1.2372379 0.5873847 1.4301181 -0.3308952
```

```
# use different mean and standard deviation parameters  
rnorm(10, mean=50, sd=10)
```

```
## [1] 39.65975 64.06015 66.80990 52.25034 53.50414 62.75961 42.17646  
## [8] 46.39955 47.64585 55.52759
```

## Histograms

Histograms (or frequency plots) are commonly used to visually inspect the distribution of a set of numbers.

```
# visualize distribution using a histogram of the numbers  
x <- rnorm(500, mean=50, sd=10)  
  
hist(x)
```

## Descriptive Statistics

```
# create a vector of 100 random numbers  
v <- runif(100, min=0, max=1000)  
  
mean(v)          # mean (average)
```

```
## [1] 547.2007
```

```
mean(v, trim=0.1) # 10% trimmed mean
```

```
## [1] 557.1284
```

```
median(v)         # median (middle in ordered list)
```

```
## [1] 622.8363
```

```
sd(v)             # standard deviation
```

```
## [1] 303.8632
```

```
var(v)          # variance (sd squared)
```

```
## [1] 92332.86
```

```
range(v)        # min and max values
```

```
## [1] 17.9904 992.3217
```

```
IQR(v)          # interquartile range
```

```
## [1] 538.6237
```

## Missing Values

Missing values are indicated in R with the `NA` object. Many functions can automatically ignore or remove missing values.

```
# create a vector
x <- c(11,7,3,4.7,18,2,NA,54,-31,8,-6.2,NA)

x.mean <- mean(x)
print(x.mean)
```

```
## [1] NA
```

```
# mean with NA dropped
x.noNA.mean <- mean(x,na.rm = TRUE)
print(x.noNA.mean)
```

```
## [1] 7.05
```

## Flow Control

Often code needs to be executed conditionally or repeatedly. For such situations, R, like most programming languages, offers flow control statements, including `if`, `switch`, and `for`.

### Conditional Execution: *if*

An `if` statement allows for conditional execution of one or more statements. Whether the code is run depends on whether the *boolean\_expression* evaluates to *TRUE*.

```
if (boolean_expression) {
  # statement(s) will execute if the boolean expression is true.
}
```

### Boolean Expressions



The *boolean\_expression* consists of one or more logical statements. Multiple statements are connected with logical operators AND (&), OR (|) and NOT (!). The logical operators are summarized in the table below.

Table of logical operators

Operator	Description
<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to
==	exactly equal to
!=	not equal to
!x	not x
x	y
x & y	x AND y
isTRUE(x)	test if X is TRUE

In boolean algebra,  $\wedge$  means AND  $\vee$  means OR, and  $\neg$  means NOT. AND is only *true* if both operands are true. OR is true if either or both operands are *true*.

## DeMorgan's Laws

DeMorgan's Laws can be used to simplify certain boolean expressions.

$$\neg(p \wedge q) \longleftrightarrow (\neg p \vee \neg q)$$

$$\neg(p \vee q) \longleftrightarrow (\neg p \wedge \neg q)$$

together with double negation elimination

$$\neg\neg p \longleftrightarrow p$$

may be used to convert between conjunctive and disjunctive boolean expressions, but notice you'll often need negation too.

# Packages

## Built-in and Package Functions

Many functions have parameters which allows you to pass data to the function and influence what it does. To use a function in a package requires that the package is loaded into the current session (after having been installed).

```
# load the installed Rcurl package into the current session
library(Rcurl)
```

```
# use a function from that package
omegahatExists = RCurl::url.exists("http://www.omegahat.net")

# the package name is optional if there's no ambiguity in the function name
```

The package name (`RCurl::`) is not necessary if there is no ambiguity, *i.e.*, there aren't two functions name `url.exists` in two different packages both of which are loaded into the current session.

## Installing Packages

To ensure that packages are automatically installed, you can use the followign code. That way your code becomes portable.

```
if("RCurl" %in% rownames(installed.packages()) == FALSE) {
  install.packages("RCurl")
}

library("RCurl")
```

In the above code the function `installed.packages()` returns a list of the names of all installed packages. The operator `%in%` is a set operator that checks if “*RCurl*” is one of the returned names. If it is, the boolean expression evaluates to **TRUE**, otherwise **FALSE**. If it is false, then it means the package is not installed and the optional code that installs the package is executed. The way, the loading of the package with `library("RCurl")` cannot fail.

## User-Defined Functions

User defined functions are an important part of programming and likewise in R. They allow code to be reused and to be better roganized. Here is an example of the definition of the function `fraction`. It is clearly not very useful but it demonstrates the syntax for defining functions.

```
fraction <- function(x,y) {
  result <-x/y
  print (result)
}
fraction(3,2)
```

```
## [1] 1.5
```

The body (or code) of the function is between two curly braces (`{` and `}`). The name of the function must follow the same rules as all identifiers. This function is actually a *procedure* because it carries out some action but does not return a result. The example below shows a function that returns a result that calculates the mode<sup>1</sup>.

```
#
# Function: calcMode()
# Parameters:
#   v -- a vector of numeric values
#
calcMode <- function(v)
```

```
{
  uniq.v <- unique(v)
  uniq.v[which.max(tabulate(match(v, uniq.v)))]
}

# create numeric vector of integers
v <- c(1,3,2,3,2,2,3,4,1,5,5,3,2,3)

# calculate the mode using our function
v.mode <- calcMode(v)
print(v.mode)
```

```
## [1] 3
```

```
# create the vector of strings
s <- c("a","it","the","it","it", "the","a","a")

# determine the mode using our function
s.mode <- calcMode(s)
print(s.mode)
```

```
## [1] "a"
```

It does not matter whether the opening `{` is on the same line as the `function` keyword or not. Different programmers have different styles but you should have a consistent style for yourself and your organization. Note that the function `calcMode` uses the *camelCase* naming convention.

Functions in R do not have a return type; the return type is determined by the value that is returned. This is a form of polymorphism and can be helpful in working with different data types consistently. Similarly, the parameters (function arguments) do not have a defined type. While that is useful it can lead to programming issues if the wrong type of argument is passed.

The return value of an R function is the last value computed. Often programmers will assign the return value to the name of the function to make it clearer.

```
calcMode <- function(v)
{
  uniq.v <- unique(v)
  calcMode <- uniq.v[which.max(tabulate(match(v, uniq.v)))]
}
```

Here is another useful function: one that checks whether a package is installed – and, if not, installs it. Note how the function checks the input argument to ensure it is text rather than some other object, like a vector, a dataframe, or a number.

```
is.installed <- function(x)
{
  if (!require(x,character.only = TRUE))
  {
    install.packages(x,dep=TRUE)
    if(!require(x,character.only = TRUE)) stop("package not installed")
  }
}
```

# Dataframes

The dataframe is the primary data object in which data is stored for analysis.

## Accessing Cells, Rows, and Columns

Data in a data frame is a set of rows organized into columns. A column is a vector of values of the same type. Columns can have optional headers. Access to a single cell is for data frame `df` is `df[row,column]`. To access an entire row (a single data object) you use `df[row,]`. For an entire column you either use the columns position or its name: `df[,column]` or `df$columnName`

```
mtcars[1,2]      # same but different syntax
mtcars[1]        # column 1 as a data frame
mtcars[1,]       # all of row 1 as a vector
mtcars[c(1,4)]   # columns 1 and 4 as a new dataframe

mtcars[,2]       # all of column 2
mtcars[5:7,]     # rows 5 to 7 as a new dataframe
mtcars$cyl       # column named "cyl"
mtcars$cyl[2]    # 2nd row in the column "cyl"

mtcars$cyl[3:9]  # rows 3 to 9 for column "cyl" as a vector
```

Below are some examples of analyzing data in a data frame.

```
mean(mtcars$cyl)      # mean of a column
median(mtcars$cyl)    # median of a column
sd(mtcars$cyl)        # standard deviation
min(mtcars$cyl)       # min value
max(mtcars$cyl)       # max value
range(mtcars$cyl)     # range

head(df)              # show the first 6 rows only
tail(df)              # show the last 6 rows only

n <- nrow(mtcars)     # number of rows

mtcars[nrow(mtcars),] # last row only
```

## Adding Columns

New columns can be added easily.

```
df <- mtcars          # for convenience
df$mpc = 0            # new column "mpc" set to 0
df$mpc = NA           # new column set to "NA"
```

## Querying Dataframe

```
df = mtcars           # for convenience
nrow(df)              # number of rows
```

```
## [1] 32
```

```
ncol(df)          # number of columns
```

```
## [1] 11
```

```
dim(df)           # dimensions: number of rows and columns
```

```
## [1] 32 11
```

The `any` function returns ***TRUE*** or ***FALSE*** depending on whether any column (or row) in the dataframe satisfies a boolean expression.

```
# is there any car with mpg > 25
any(df$mpg > 25)
```

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

```
any(df$mpg > 25 & df$cyl < 4)
```

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

To find which rows in a dataframe satisfy some boolean expression, use the `which` function. Boolean expressions can be constructed using compound conditions: AND (&), OR (|), and NOT (!).

```
# which cars have an mpg > 21
which(df$mpg > 21)
```

```
## [1] 3 4 8 9 18 19 20 21 26 27 28 32
```

```
# list cars which have an mpg > 30
df[which(df$mpg > 30),]
```

```
##           mpg cyl disp  hp drat   wt  qsec vs am gear carb
## Fiat 128    32.4   4  78.7  66 4.08 2.200 19.47 1  1   4     1
## Honda Civic 30.4   4  75.7  52 4.93 1.615 18.52 1  1   4     2
## Toyota Corolla 33.9  4  71.1  65 4.22 1.835 19.90 1  1   4     1
## Lotus Europa 30.4   4  95.1 113 3.77 1.513 16.90 1  1   5     2
```

```
# list all cars which have an mpg > 30 and more than 100 hp
df[which(df$mpg > 30 & df$hp > 100),]
```

```
##           mpg cyl disp  hp drat   wt  qsec vs am gear carb
## Lotus Europa 30.4   4  95.1 113 3.77 1.513 16.9  1  1   5     2
```

```
# list all information for a specific car by column name
df['Fiat X1-9',]
```

```
##           mpg cyl disp hp drat   wt  qsec vs am gear carb
## Fiat X1-9 27.3   4   79 66 4.08 1.935 18.9  1  1    4    1
```

```
# how many cars have 8 cylinders?
rs = which(df$cyl == 8)
n = length(rs)

# what is the average mpg of all cars with 8 cylinders?
rs = which(df$cyl == 8)
mean(df$mpg[rs])
```

```
## [1] 15.1
```

Rows can have names like column. The row name looks like a column but isn't. Most dataframes do not have row names.

```
# list names of cars which have an mpg > 21 - car name is a row name not a column
rownames(df)[which(df$mpg > 21)]
```

```
## [1] "Datsun 710"      "Hornet 4 Drive" "Merc 240D"      "Merc 230"
## [5] "Fiat 128"        "Honda Civic"    "Toyota Corolla" "Toyota Corona"
## [9] "Fiat X1-9"       "Porsche 914-2"  "Lotus Europa"   "Volvo 142E"
```

```
# a bit simpler in two lines
rs = which(df$mpg > 21)
rownames(df)[rs]
```

```
## [1] "Datsun 710"      "Hornet 4 Drive" "Merc 240D"      "Merc 230"
## [5] "Fiat 128"        "Honda Civic"    "Toyota Corolla" "Toyota Corona"
## [9] "Fiat X1-9"       "Porsche 914-2"  "Lotus Europa"   "Volvo 142E"
```

```
# names of cars that have 8 cylinders
rs = which(df$cyl == 8)
rownames(df)[rs]
```

```
## [1] "Hornet Sportabout" "Duster 360"      "Merc 450SE"
## [4] "Merc 450SL"        "Merc 450SLC"     "Cadillac Fleetwood"
## [7] "Lincoln Continental" "Chrysler Imperial" "Dodge Challenger"
## [10] "AMC Javelin"       "Camaro Z28"      "Pontiac Firebird"
## [13] "Ford Pantera L"    "Maserati Bora"
```

The queries can also be used to identify and locate missing values in a dataframe.

```
# are there any missing values in any cell?
any(is.na(airquality))
```

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

```
# which rows have missing values
rs = which(is.na(airquality$Solar.R))
airquality[rs,]
```

```
##      Ozone Solar.R Wind Temp Month Day
## 5      NA      NA 14.3   56     5    5
## 6      28      NA 14.9   66     5    6
## 11     7      NA  6.9   74     5   11
## 27     NA      NA  8.0   57     5   27
## 96     78      NA  6.9   86     8    4
## 97     35      NA  7.4   85     8    5
## 98     66      NA  4.6   87     8    6
```

```
# remove rows with missing values
air_complete <- na.omit(airquality)
```

## Loading Data from Files

R supports most file format, including CSV, plain text, Excel, Google Sheets, XML, JSON, SPSS, Matlab, etc. Reading CSV can be done in Base R while reading other file types requires the use of specific packages.

## Reading Data from CSV



Data is most often in CSV (comma separated values) files. These plain text files are organized as rows and each row has the column values separated by commas. If the values are text and can contain commas, then the values are often enclosed in double-quotes ("). Some files use a separator other than comma, e.g., semicolon. For that you use the `read.table()` function with the *delim* parameter

rather than the `read.csv()` function. The result of reading a CSV file is a dataframe.

R attempts to automatically coerce data into an appropriate format but often that may fail and therefore you may need to convert the data yourself to text, numbers, etc. A common issue is the conversion of text columns into “factors”. Factors are R’s way to encoding categorical variables – often required for statistical analysis. However, text columns should often be left as text, so you need to specify the *stringsAsFactors=FALSE* parameter when calling `read.csv()` or `read.table()`.

R assumes that the first row in a CSV contains header labels. If there is no header, then specify *headers=FALSE*.

```
# read the file from a folder (relative path)
df.salaries = read.csv("../datasets/salaries.csv", stringsAsFactors = FALSE)
head(df.salaries, 3)
```

```
##   Name Salary
## 1   Jon  50000
## 2  Mary  50000
## 3  Jane  45000
```

```
# R assumes first row in CSV is header row
avgSal = mean(df.salaries$Salary)
avgSal
```

```
## [1] 41875
```

```
#
```

There is an equivalent `write-csv()` function for export the contents of a dataframe to a CSV file.

## Reading Data from XML

Data in an XML store can be read into R using one of several packages, including the *XML* package. Be sure to install the package and then load it before using any XML functions. An XML store can be internal text, a local file, or a URL.

```
library(XML)

# a simple XML document as internal text
txt = "<doc> <el> aa </el> </doc>"
res = xmlParse(txt, asText=TRUE)
```

```
library(XML)
library(RCurl)
```

```
## Loading required package: bitops
```

```
# URL to XML file - or local file
xml.url <- "https://www.w3schools.com/xml/plant_catalog.xml"
xData <- getURL(xml.url)
```

## Retrieve Elements via XPath

While XML can be processed by navigating the internal DOM tree, using XPath is generally preferable as it is less susceptible to changes in the XML structure.

Note that all values in an XML store are text by default and therefore any numeric data has to be explicitly converted. In addition, any additional characters (such as currency symbols or commas) need to be removed.

```
library(XML)
library(RCurl)

# URL to XML file - or local file
xml.url <- "https://www.w3schools.com/xml/plant_catalog.xml"
```



```
xData <- getURL(xml.url)

xmlDoc = xmlParse(xData)    # parse the document into an internal tree
r = xmlRoot(xmlDoc)         # retrieve the root of the tree
xmlSize(r)                  # number of child elements of root
```

```
## [1] 36
```

```
prices = xpathSApply(xmlDoc,'//PRICE',xmlValue)

# remove the $ before the price
prices.vals = substring(prices,2)

# convert to a number
prices.vals <- as.numeric(prices.vals)

# do some calculation
usd2euro = 1.091
prices.euros <- prices.vals * usd2euro

print(round(prices.euros,2))
```

```
## [1] 2.66 10.22 7.43 10.80 7.03 9.85 4.85 4.35 3.52 3.25 3.05
## [12] 6.10 7.19 4.25 3.49 9.86 7.57 10.45 9.67 9.99 5.01 7.81
## [23] 10.69 2.80 10.19 3.03 7.70 7.16 8.52 9.34 10.10 4.76 8.61
## [34] 9.38 6.14 3.29
```

## Navigate DOM Tree

To example below shows how the tree can also be navigated positionally.

```
library(XML)
library(RCurl)

# URL to XML file - or local file
xml.url <- "https://www.w3schools.com/xml/plant_catalog.xml"
xData <- getURL(xml.url)

xmlTree = xmlTreeParse(xData)    # parse tree into internal object
r = xmlRoot(xmlTree)             # get root of the XML tree
xmlName(r)                       # name of root element
```

```
## [1] "CATALOG"
```

```
xmlSize(r)                      # number of child elements of root
```

```
## [1] 36
```

```
# access 2nd (or nth) child node
r[[2]]
```

```
## <PLANT>
## <COMMON>Columbine</COMMON>
## <BOTANICAL>Aquilegia canadensis</BOTANICAL>
## <ZONE>3</ZONE>
## <LIGHT>Mostly Shady</LIGHT>
```

```
## <PRICE>$9.37</PRICE>
## <AVAILABILITY>030699</AVAILABILITY>
## </PLANT>
```

```
# access child element off a child element, i.e. navigate tree
r[[2]][[5]]
```

```
## <PRICE>$9.37</PRICE>
```

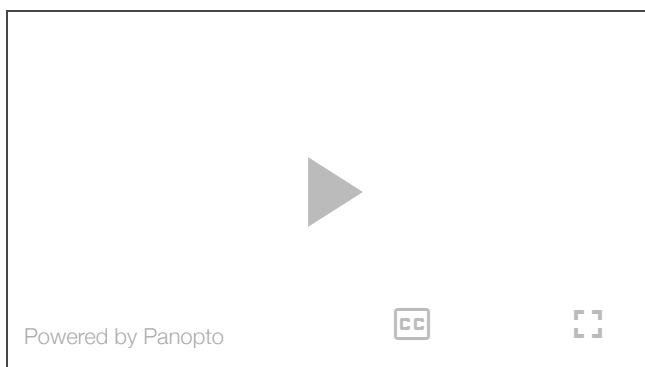
```
# get the value between tags for some child
p = xmlValue(r[[2]][[5]])
print(p)
```

```
## [1] "$9.37"
```

```
# all data is text, so parse and convert
p.n = substring(p,2)
p.val = as.numeric(p.n)
print(p.val)
```

```
## [1] 9.37
```

## Querying Relational Databases



Data is often in databases, most commonly relational databases such as Oracle, MySQL, Microsoft SQL Server, SQLite, etc. Each database vendor has a unique way of connecting to their database. In R, you will need the **RSQLite** package installed.

To read data from a database into R, follows these steps:

1. open connection to database
2. build SQL query
3. execute SQL query by sending to database
4. capture result in dataframe

Connecting to a database is done in a database-specific way and each database is different. Packages specific to the database need to be loaded (of course, after installation). The code below assumes that the package *RSQLite* for connecting to SQLite databases is installed but not loaded.

To connect to a database you need to know where the database is located. For most client/server databases like MySQL you need to know the server's IP address on which the database runs. For SQLite you need the database file path (as SQLite does run not on an actual remote server).

To run a query (retrieve data) you most commonly use the `dbGetQuery` function. To perform an INSERT, UPDATE, DELETE, CREATE TABLE, DROP TABLE, ALTER TABLE you need to use

`dbSendQuery`

```
library(RSQLite)

# connect to the SQLite database in the specified file
db.conn <- dbConnect(SQLite(), dbname="../datasets/projectdb.sqlitedb")

# construct a SQL query
sqlCmd = "SELECT * FROM projects"

# send the SQL query to the database
rs = dbGetQuery(db.conn, sqlCmd)

# print part of the result table
head(rs,3)
```

```
##      pid      pname budget pmgr
## 1      1      TOGAF  25000   100
## 2      2    AirDrop  45000   103
## 3      3  WebQueue  55500   100
```

```
# once no further access to the data is needed, disconnect
dbDisconnect(db.conn)
```

## Worked Examples

For the built-in dataset sunspots, carry out the following queries:

1. How many years were fewer than five sunspots observed?
2. In which years were fewer than five sunspots observed?
3. What were the average number of sunspots between 1960 and 1969 (inclusive)?
4. Which year had the most sunspots?
5. What were the total number of sunspots in each year?
6. What were the average number of sunspots for August?
7. What is the standard deviation of sunspots?
8. Which years were an outliers, i.e., number of sunspots with more than 3 z-scores?

## Solutions

How many years were fewer than five sunspots observed?

```
sp <- sunspots
i <- 1
y <- 1
x <- seq(from = 1, to = length(sp), by = 12)
for (v in x) { y[i] <- sum(sp[v:(v+11)]); i <- i + 1 }
length(which(y < 5))
```

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

In which years were fewer than five sunspots observed?

```
1749+(which(y == max(y))-1)
```

```
## [1] 1957
```

What were the average number of sunspots between 1960 and 1969 (inclusive)?

```
x <- x + 7
mean(sp[x])
```

```
## [1] 52.06511
```

```
# or
for (v in x) { y[i] <- sum(sp[(v+7)]); i <- i + 1 }
```

Which years were an outliers, i.e., number of sunspots with more than 3 z-scores?

```
m <- mean(y)
sdev <- sd(y)

1749 + which(abs((m - y)/ sdev) > 3)
```

```
## numeric(0)
```

- 
1. Oddly enough, R does not have a standard in-built function to calculate mode. The built-in function `mode` returns the data type of an object. ↩