Introduction to R

Emanuele Guidotti

Contents

Quick Start

In this introduction to R, the reader will master the basics of this beautiful open source language with hands-on experience. With over 2 million users worldwide R is rapidly becoming the leading programming language in statistics and data science. Every year, the number of R users grows by 40% and an increasing number of organizations are using it in their day-to-day activities.

Installation

Download and install **R** at this link

Download and install **Rstudio** (free version) at this link

Comments

All text after the sign `#` within the same line is considered a comment.

```
# this is a comment
this is NOT a comment
```

Variable Assignment

Values can be assigned to variables with the operators `<-`, `=` or `->`.

```
# assign 1 to variable x
x <- 1
# or
x = 1
# or
1 -> x
```

Functions

R functions are invoked by their name, then followed by the parenthesis, and zero or more arguments.

```
# summing 1+2+3+4+5
sum(1,2,3,4,5)
```

Extension Packages

Additional functionality beyond those offered by the core R library are available with R packages. In order to install an additional package, the `install.packages` function can be invoked.

```
# install the "xts" package
install.packages('xts')
```

There are two ways to invoke functions from add-on packages: using the package namespace or loading the package.

```
# using the namespace.
# Invoke the function as package_name::function_name
xts::is.xts(1)

# loading the package with the 'require' function.
# This makes its functions available without using namespaces
require(xts)
is.xts(1)
```

Help

R provides extensive documentation. Enter `?function_name` to access the documentation of a function.

```
# examples
?sum
?mean
?rnorm
```

Basic Data Types

There are several basic R data types that are of frequent occurrence in routine R calculations.

Numeric

Decimal values are called numerics in R. It is the default computational data type. If a decimal value is assigned to a variable `x` as follows, `x` will be of **numeric** type.

```
x <- 10.5 # assign a decimal value
class(x)  # class of x
```

```
## [1] "numeric"
```

Furthermore, even if an integer is assigned to a variable `x`, it is still being saved as a numeric value.

```
x <- 10        # assign an integer value
is.integer(x)  # is integer?
```

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

Integer

In order to create an integer variable in R, the `as.integer` function can be invoked.

```
x <- as.integer(10) # assign an integer data type
is.integer(x)       # is integer?
```

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

Integers can also be declared by appending an `L` suffix.

```
x <- 10L       # assign an integer data type
is.integer(x)  # is integer?
```

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

Logical

A logical value is often created via comparison between variables.

```
x <- 2 > 1    # is 2 greater than 1?
x
```

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

Standard logical operations are `&` (and), `|` (or), and `!` (negation).

```
u <- TRUE
v <- FALSE
```

```
u & v
```

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

```
u | v
```

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

```
!u
```

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

Character

A character object is used to represent string values in R. Two character values can be concatenated with the `paste` function.

```r
address <- 'example'
domain <- 'gmail.com'
paste(address, domain, sep = '@')
```

```
## [1] "example@gmail.com"
```

However, it is often more convenient to create a readable string with the `sprintf` function, which has a C language syntax.

```r
sprintf("%s has %d dollars", "Sam", 100)
```

```
## [1] "Sam has 100 dollars"
```

And to replace the first occurrence of the word "little" by another word "big" in the string, the `sub` function can be applied.

```r
sub("little", "big", "Mary has a little lamb.")
```

```
## [1] "Mary has a big lamb."
```

More functions for string manipulation can be found in the R documentation.

```r
?sub
```

Basic Data Structures

Vector

The basic data structure in R is the vector. They are usually created with the `c()` function, short for combine:

```r
c(1,2,3)
```

```
## [1] 1 2 3
```

Named Vector

```r
# Declaring a named vector
c('first' = 1, 'second' = 2, 'third' = 3)
```

```
##  first second  third
##      1      2      3
```

```r
# Generating a named vector
x <- c(1,2,3)                       # vector
n <- c('first','second','third')    # vector of names
names(x) <- n                       # assign names
x
```

```
##  first second  third
##      1      2      3
```

Matrix

A matrix is a collection of data elements arranged in a two-dimensional rectangular layout. They are usually created with the `matrix()` function:

```r
matrix(data = c(1,2,3,4,5,6), # the data elements
       ncol = 3,              # number of columns
       nrow = 2,              # number of rows
       byrow = TRUE)          # fill matrix by rows
```

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

Named Matrix

```
# Declaring a named matrix
matrix(data = c(1,2,3,4,5,6), # the data elements
       ncol = 3,              # number of columns
       nrow = 2,              # number of rows
       byrow = TRUE,          # fill matrix by rows
       dimnames = list(       # list containing names
         c('r1','r2'),           # rownames
         c('c1','c2','c3')       # colnames
       ))
```

```
##    c1 c2 c3
## r1  1  2  3
## r2  4  5  6
```

```
# Generating a named matrix
M <- matrix(data = c(1,2,3,4,5,6), # the data elements
            ncol = 3,              # number of columns
            nrow = 2,              # number of rows
            byrow = TRUE)          # fill matrix by rows
rn <- c('r1','r2')        # vector of rownames
cn <- c('c1','c2','c3')   # vector of colnames
rownames(M) <- rn         # assign rownames
colnames(M) <- cn         # assign colnames
M
```

```
##    c1 c2 c3
## r1  1  2  3
## r2  4  5  6
```

Data Frame

A data frame is used for storing data tables. It is a list of vectors of equal length. They are usually created with the `data.frame()` function. Beware `data.frame()`'s default behaviour which turns strings into factors (a factor is a vector that can contain only predefined values, and is used to store categorical data). Use `stringsAsFactors = FALSE` to suppress this behaviour:

```
v1 <- c(10,20,30)                                  # numeric vector
v2 <- c('a','b','c')                               # character vector
v3 <- c(TRUE,TRUE,FALSE)                           # logical vector
data.frame(v1, v2, v3, stringsAsFactors = FALSE) # data.frame
```

```
##   v1 v2    v3
## 1 10  a  TRUE
## 2 20  b  TRUE
## 3 30  c FALSE
```

Named Data Frame

```
# Declaring a named data.frame
v1 <- c(10,20,30)                          # numeric vector
v2 <- c('a','b','c')                       # character vector
v3 <- c(TRUE,TRUE,FALSE)                   # logical vector
data.frame('c1' = v1,                      # column named 'c1'
           'c2' = v2,                      # column named 'c2'
           'c3' = v3,                      # column named 'c3'
           row.names = c('r1', 'r2', 'r3'), # vector of rownames
           stringsAsFactors = FALSE)       # suppress character conversion
```

```
##    c1 c2    c3
## r1 10  a  TRUE
## r2 20  b  TRUE
## r3 30  c FALSE
```

```
# Generating a named data.frame
v1 <- c(10,20,30)                                    # numeric vector
v2 <- c('a','b','c')                                 # character vector
v3 <- c(TRUE,TRUE,FALSE)                             # logical vector
rn <- c('r1','r2','r3')                              # vector of rownames
cn <- c('c1','c2','c3')                              # vector of colnames
df <- data.frame(v1, v2, v3,stringsAsFactors = FALSE)  # data.frame
rownames(df) <- rn                                   # assign rownames
colnames(df) <- cn                                   # assign colnames
df
```

```
##    c1 c2    c3
## r1 10  a  TRUE
## r2 20  b  TRUE
## r3 30  c FALSE
```

List

A list is the most generic structure containing other objects. They are usually created with the `list()` function:

```
list(matrix(100),
     data.frame(1,2,3),
     c('a','b','c','d'))
```

```
## [[1]]
##      [,1]
## [1,]  100
##
## [[2]]
##   X1 X2 X3
## 1  1  2  3
##
## [[3]]
## [1] "a" "b" "c" "d"
```

Named List

```r
# Declaring a named list
list('matrix' = matrix(100),           # matrix
     'data.frame' = data.frame(1,2,3), # data.frame
     'vector' = c('a','b','c','d'))    # vector
```

```
## $matrix
##      [,1]
## [1,]  100
##
## $data.frame
##   X1 X2 X3
## 1  1  2  3
##
## $vector
## [1] "a" "b" "c" "d"
```

```r
# Generating a named list
M <- matrix(100)                       # matrix
df <- data.frame(1,2,3)                # data.frame
v <- c('a','b','c','d')                # vector
n <- c('matrix','data.frame','vector') # vector of names
l <- list(M, df, v)                    # list
names(l) <- n                          # assign names
l
```

```
## $matrix
##      [,1]
## [1,]  100
##
## $data.frame
##   X1 X2 X3
## 1  1  2  3
##
## $vector
## [1] "a" "b" "c" "d"
```

Basic Operations

Subsetting

Vector

Values in a `vector` are retrieved by using the single square bracket `[]` operator.

```r
s = c("aaa"="a", "bbb"="b", "ccc"="c", "ddd"="d", "eee"="e")
s # print the full vector
```

```
## aaa bbb ccc ddd eee
## "a" "b" "c" "d" "e"
```

```r
# retrieve the 3rd element
s[3]
```

```
## ccc
## "c"
```

```r
# drop the 3rd element
s[-3]
```

```
## aaa bbb ddd eee
## "a" "b" "d" "e"
```

```r
# out-of-range returns NA
s[10]
```

```
## <NA>
##   NA
```

```r
# retrieve the 2nd, 3rd, 5th and 5th element
i <- c(2,3,5,5)
s[i]
```

```
## bbb ccc eee eee
## "b" "c" "e" "e"
```

```r
# drop the 1st and 3rd element
i <- c(1,3)
s[-i]
```

```
## bbb ddd eee
## "b" "d" "e"
```

```r
# retrieve the elements named 'ddd' and 'bbb'
i <- c('ddd','bbb')
s[i]
```

```
## ddd bbb
## "d" "b"
```

```r
# retrieve the 3rd element using a logical vector
i <- c(FALSE,FALSE,TRUE,FALSE,FALSE)
s[i]
```

```
## ccc
## "c"
```

```r
# the logical vector will be recycled if it is shorter than the vector to subset
i <- c(FALSE,TRUE)  # ->  c(FALSE,TRUE,FALSE,TRUE,FALSE)
s[i]
```

```
## bbb ddd
## "b" "d"
```

```r
# select elements greater than 'b'
i <- s > 'b'
s[i]
```

```
## ccc ddd eee
## "c" "d" "e"
```

Matrix

Values in a `matrix` are retrieved by using the single square bracket `[]` operator.

```r
M <- matrix(1:12, nrow = 3, ncol = 4, byrow = TRUE)
rownames(M) <- c('r1','r2','r3')
```

```r
colnames(M) <- c('c1','c2','c3','c4')
M # print the full matrix
```

```
##    c1 c2 c3 c4
## r1  1  2  3  4
## r2  5  6  7  8
## r3  9 10 11 12
```

```r
# retrieve the element in 2nd row, 3rd column
M[2,3]
```

```
## [1] 7
```

```r
# retrieve the 1st row
M[1,]
```

```
## c1 c2 c3 c4
##  1  2  3  4
```

```r
# retrieve the 1st column
M[,1]
```

```
## r1 r2 r3
##  1  5  9
```

```r
# retrieve the 2nd and 3rd row
i <- c(2,3)
M[i,]
```

```
##    c1 c2 c3 c4
## r2  5  6  7  8
## r3  9 10 11 12
```

```r
# drop the 1st and 3rd column
i <- c(1,3)
M[,-i]
```

```
##    c2 c4
## r1  2  4
## r2  6  8
## r3 10 12
```

```r
# retrieve the elements in 1st and 3rd row, 2nd and 4th column
M[c(1,3),c(2,4)]
```

```
##    c2 c4
## r1  2  4
## r3 10 12
```

```r
# retrieve the rows named 'r1' and 'r3'
i <- c('r1','r3')
M[i,]
```

```
##    c1 c2 c3 c4
## r1  1  2  3  4
## r3  9 10 11 12
```

```r
# retrieve the columns named 'c2' and 'c4'
i <- c('c2','c4')
M[,i]
```

```
##    c2 c4
## r1  2  4
## r2  6  8
## r3 10 12
```

```
# retrieve the 3rd row of the columns named 'c2' and 'c4'
i <- c('c2','c4')
M[3,i]
```

```
## c2 c4
## 10 12
```

```
# retrieve the 1st row using a logical vector
i <- c(TRUE,FALSE,FALSE)
M[i,]
```

```
## c1 c2 c3 c4
##  1  2  3  4
```

```
# the logical vector will be recycled if it is shorter than the number of rows/columns to subset
i <- c(TRUE,FALSE)  # ->  c(TRUE,FALSE,TRUE)
M[i,]
```

```
##    c1 c2 c3 c4
## r1  1  2  3  4
## r3  9 10 11 12
```

```
# select the column named 'c4' where 'c3' is less than twice 'c1'
i <- M[,'c3'] < 2*M[,'c1']
M[i,'c4']
```

```
## r2 r3
##  8 12
```

Data Frame

Values in a `data.frame` are retrieved by using the single square bracket `[]` operator as done in a `matrix` object (see above). Here, also the `$` or `[[]]` operators can be used to retrieve columns.

```
df <- data.frame('age' = c(48,18,51), 'sex' = c('M','F','M'))
df # print full data.frame
```

```
##   age sex
## 1  48   M
## 2  18   F
## 3  51   M
```

```
# retrieve the "age" column
df$age              # equivalent to df[["age"]] or df[,"age"]
```

```
## [1] 48 18 51
```

```
# retrieve the age of males ("M")
i <- df$sex == "M"   # equivalent to df[["sex"]]=="M" or df[,"sex"]=="M"
df$age[i]            # equivalent to df[["age"]][i] or df[i,"age"]
```

```
## [1] 48 51
```

List

A list is subsetted using the single square bracket `[]` operator.

```
l <- list(
  'data' = data.frame('age' = c(48,18,51), 'sex' = c('M','F','M')),
  'letters' = c('a','b','c'),
  'extra' = c(1:5)
)
l # print full list
```

```
## $data
##   age sex
## 1  48   M
## 2  18   F
## 3  51   M
##
## $letters
## [1] "a" "b" "c"
##
## $extra
## [1] 1 2 3 4 5
```

```
# select the 1st and 3rd elements
i <- c(1,3)
l[i]
```

```
## $data
##   age sex
## 1  48   M
## 2  18   F
## 3  51   M
##
## $extra
## [1] 1 2 3 4 5
```

```
# select the elements named "extra" and "letters"
i <- c("extra","letters")
l[i]
```

```
## $extra
## [1] 1 2 3 4 5
##
## $letters
## [1] "a" "b" "c"
```

```
# drop the "extra" element
l["extra"] <- NULL
l
```

```
## $data
##   age sex
## 1  48   M
## 2  18   F
## 3  51   M
##
## $letters
## [1] "a" "b" "c"
```

Objects in a `list` are retrieved by using the operator `[[]]` or `$`.

```
# extract the 2nd object
l[[2]]
```

```
## [1] "a" "b" "c"
```

```
# extract the "data" object
l$data   # equivalent to l[["data"]]
```

```
##   age sex
## 1  48   M
## 2  18   F
## 3  51   M
```

Arithmetics

Arithmetic operations of **vectors** and **matrices** are performed element-by-element, **data.frames** are treated as **matrices** when containing only numeric elements. If two vectors are of unequal length, the shorter one will be recycled in order to match the longer vector. For example, the following vectors `u` and `v` have different lengths, and their sum is computed by recycling values of the shorter vector `u`.

```
u <- c(10, 20, 30)
v <- c(1, 2, 3, 4, 5, 6, 7, 8, 9)
M <- matrix(c(1, 2, 3, 4, 5, 6, 7, 8, 9), ncol = 3, nrow = 3, byrow = TRUE)

# vector + vector
u + v
```

```
## [1] 11 22 33 14 25 36 17 28 39
```

```
# vector + 1
u + 1
```

```
## [1] 11 21 31
```

```
# vector * 2
u * 2
```

```
## [1] 20 40 60
```

```
# matrix + 1
M + 1
```

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

```
# matrix + vector
M + u
```

```
##      [,1] [,2] [,3]
## [1,]   11   12   13
## [2,]   24   25   26
## [3,]   37   38   39
```

```
# matrix + matrix
M + M
```

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

```
## [2,]    8   10   12
## [3,]   14   16   18
```

```
# matrix * vector
M * u
```

```
##      [,1] [,2] [,3]
## [1,]   10   20   30
## [2,]   80  100  120
## [3,]  210  240  270
```

```
# matrix product (rows x columns)
M %*% u
```

```
##      [,1]
## [1,]  140
## [2,]  320
## [3,]  500
```

Extra

Custom Functions

Abstracting code into many small functions is key for writing nice R code. Functions are defined by code with a specific format:

```
function.name <- function(arg1, arg2, arg3=NULL, ...) {
  # code here...
  return(...)
}
```

where

- `function.name`: the name of the function
- `arg1`, `arg2`, `arg3`, `...`: input values
- `arg3=NULL`: default value. If `arg3` is not provided when calling the function, `NULL` will be used instead
- `return()`: the output value

Define a function to compute the sum of the first `n` integer numbers.

```
sum.int <- function(n){
  s <- sum(1:n)
  return(s)
}
```

Compute the sum of the first 10 integers

```
sum.int(10)
```

```
## [1] 55
```

Define a function to compute the `p` norm of a vector `x`. By default, compute the Euclidean norm (`p = 2`).

```
norm <- function(x, p = 2){
  d <- sum(x^p)^(1/p)
  return(d)
}
```

Compute the Euclidean norm of the vector `c(1,1)`

```r
norm(x = c(1,1))  # equivalento to norm(x = c(1,1), p = 2)
```

```
## [1] 1.414214
```

Compute the 3-norm of the vector `c(1,1)`

```r
norm(x = c(1,1), p = 3)
```

```
## [1] 1.259921
```

Compute the ∞-norm of the vector `c(1,1)`

```r
norm(x = c(1,1), p = Inf)
```

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

Time Series

Time Index

A time series is a series of data points indexed in time order. In R, all data types for which an order is defined can be used to index a time series. If the operator `<` is defined for a data type, then the data type can be used to index a time series.

Date

```r
today <- Sys.Date()      # current Date
yesterday <- today - 1   # subtract 1 day
yesterday < today        # the order is defined for Date
```

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

POSIXct

```r
now <- Sys.time()        # current time
ago <- now - 3600        # subtract 3600 seconds
ago <  now               # the order is defined for POSIXct
```

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

Character

```r
'a' < 'b'                # the order is defined for character
```

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

Numeric

```r
1 < 2                    # the order is defined for numeric
```

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

Complex

```r
2+0i < 1+3i              # the order is NOT defined for complex
```

```
## Error in 2 + (0+0i) < 1 + (0+3i): invalid comparison with complex values
```

The 'zoo' Package

The `zoo` package consists of the methods for totally ordered indexed observations. All indexes discussed above can be used. The package aims at performing calculations containing irregular time series of numeric

vectors, matrices and factors. The package is an infrastructure that tries to do all basic things well, but it doesn't provide modeling functionality.

```
# install the package
install.packages('zoo')
```

```
# load the package
require(zoo)
```

The below set of exercises shows some of zoo concepts.

Declaration

```
# create a unidimensional zoo object indexed by default
zoo(x = c(100,123,43,343,22))
```

```
##   1   2   3   4   5
## 100 123  43 343  22
```

```
# create a unidimensional zoo object indexed by numeric
x <- c(100, 123, 43, 343, 22)
i <- c(0, 0.2, 0.4, 0.5, 1)
zoo(x = x, order.by = i)
```

```
##   0 0.2 0.4 0.5   1
## 100 123  43 343  22
```

```
# create a unidimensional zoo object indexed by character
x <- c(100, 123, 43, 343, 22)
i <- c('z', 'b', 'd', 'c', 'a')
zoo(x = x, order.by = i)
```

```
##   a   b   c   d   z
##  22 123 343  43 100
```

```
# create a multidimensional zoo object indexed by Date
x <- data.frame('price' = c(100,99.3,100.2), 'volume' = c(9.9,1.3,3.6))
i <- as.Date(c('2018/01/01', '2018/02/23', '2018/05/01'), format = "%Y/%m/%d")
zoo(x = x, order.by = i)
```

```
##            price volume
## 2018-01-01 100.0    9.9
## 2018-02-23  99.3    1.3
## 2018-05-01 100.2    3.6
```

```
# create a multidimensional zoo object indexed by POSIXct
x <- data.frame('price' = c(100,99.3,100.2), 'volume' = c(9.9,1.3,3.6))
i <- as.POSIXct(c('20180101 120631', '20180223 085145', '20180501 182309'), format = "%Y%m%d %H%M%S")
zoo(x = x, order.by = i)
```

```
##                     price volume
## 2018-01-01 12:06:31 100.0    9.9
## 2018-02-23 08:51:45  99.3    1.3
## 2018-05-01 18:23:09 100.2    3.6
```

Manipulation

```
# assign colnames
x <- data.frame(c(100,99.3,100.2), c(9.9,1.3,3.6))
z <- zoo(x = x)
```

```r
colnames(z) <- c('p','v')
z
```

```
##       p   v
## 1 100.0 9.9
## 2  99.3 1.3
## 3 100.2 3.6
```

```r
# assign indexes
index(z) <- as.Date(c('2018/01/01', '2018/02/23', '2018/05/01'), format = "%Y/%m/%d")
z
```

```
##                p   v
## 2018-01-01 100.0 9.9
## 2018-02-23  99.3 1.3
## 2018-05-01 100.2 3.6
```

```r
# starting index
start(z)
```

```
## [1] "2018-01-01"
```

```r
# ending index
end(z)
```

```
## [1] "2018-05-01"
```

```r
# select specific indexes
i <- as.Date(c('2018-01-01', '2018-05-01'))
z[i]
```

```
##                p   v
## 2018-01-01 100.0 9.9
## 2018-05-01 100.2 3.6
```

```r
# select specific columns
z$p                # equivalent to z[,'p']
```

```
## 2018-01-01 2018-02-23 2018-05-01
##      100.0       99.3      100.2
```

```r
# change the 2nd observation 'p' value
z$p[2] <- 105      # equivalent to z[2,'p'] <- 105
z
```

```
##                p   v
## 2018-01-01 100.0 9.9
## 2018-02-23 105.0 1.3
## 2018-05-01 100.2 3.6
```

```r
# subset the series
window(z, start = '2018-01-01', end = '2018-03-1')
```

```
##              p   v
## 2018-01-01 100 9.9
## 2018-02-23 105 1.3
```

```r
# increments
diff(z)
```

```
##               p    v
```

```
## 2018-02-23  5.0 -8.6
## 2018-05-01 -4.8  2.3
```

```
# lag the series
lag(z, k = 1)      # shift the time base back
```

```
##                p   v
## 2018-01-01 105.0 1.3
## 2018-02-23 100.2 3.6
```

```
# lag the series
lag(z, k = -1)     # shift the time base forward
```

```
##              p   v
## 2018-02-23 100 9.9
## 2018-05-01 105 1.3
```

```
# merge series
z.next   <- lag(z, k =  1)
z.prev   <- lag(z, k = -1)
z.merged <- merge(z, z.next, z.prev)
z.merged
```

```
##              p.z v.z p.z.next v.z.next p.z.prev v.z.prev
## 2018-01-01 100.0 9.9    105.0      1.3       NA       NA
## 2018-02-23 105.0 1.3    100.2      3.6      100      9.9
## 2018-05-01 100.2 3.6       NA       NA      105      1.3
```

```
# handle missing data. Approx with previous non-NA value
na.locf(z.merged)
```

```
##              p.z v.z p.z.next v.z.next p.z.prev v.z.prev
## 2018-01-01 100.0 9.9    105.0      1.3       NA       NA
## 2018-02-23 105.0 1.3    100.2      3.6      100      9.9
## 2018-05-01 100.2 3.6    100.2      3.6      105      1.3
```

```
# handle missing data. Approx with next non-NA value
na.locf(z.merged, fromLast = TRUE)
```

```
##              p.z v.z p.z.next v.z.next p.z.prev v.z.prev
## 2018-01-01 100.0 9.9    105.0      1.3      100      9.9
## 2018-02-23 105.0 1.3    100.2      3.6      100      9.9
## 2018-05-01 100.2 3.6       NA       NA      105      1.3
```

```
# handle missing data. Drop NA
z.merged[complete.cases(z.merged),]
```

```
##            p.z v.z p.z.next v.z.next p.z.prev v.z.prev
## 2018-02-23 105 1.3    100.2      3.6      100      9.9
```

Arithmetic operations are performed element-by-element on matching indexes of the two zoo obejcts. If the operation involves a zoo and a vector object, then the operation is performed on the whole zoo object.

```
x <- matrix(101:112, nrow = 3, ncol = 4, byrow = TRUE)
z <- zoo(x)
# add 1 to the whole series
z + 1
```

```
##
## 1 102 103 104 105
```

```
## 2 106 107 108 109
## 3 110 111 112 113
```

```
# multiply the first observation by 0, the second one by 1 and the third one by 2
z * c(0,1,2)
```

```
##
## 1   0   0   0   0
## 2 105 106 107 108
## 3 218 220 222 224
```

```
# compute the increments
z - lag(z, -1)  # equivalent to diff(z)
```

```
##
## 2 4 4 4 4
## 3 4 4 4 4
```

```
# compute the percentage increments
z / lag(z, -1) - 1
```

```
##
## 2 0.03960396 0.03921569 0.03883495 0.03846154
## 3 0.03809524 0.03773585 0.03738318 0.03703704
```

```
# compute the rolling mean on a 2-observation window
rollapply(z, width = 2, FUN = mean)
```

```
##
## 1 103 104 105 106
## 2 107 108 109 110
```

The 'xts' Package

The `xts` package provides an extensible time series class, enabling uniform handling of many R time series classes by extending `zoo`. An `xts` object can be indexed by the `Date`, `POSIXct`, `chron`, `yearmon`, `yearqtr`, `DateTime` data types but not by `numeric` or `character`.

```
# install the package
install.packages('xts')
```

```
# load the package
require(xts)
```

The methods seen for `zoo` objects can be applied to `xts`. The below set of exercises shows some of additional xts specific concepts.

```
# create an xts object
dates <- seq(as.Date("2017-05-01"), length=1000, by="day")          # generate a sequence of dates
data  <- c(price = cumprod(1+rnorm(1000, mean = 0.001, sd = 0.01))) # generate some random data
x <- xts(x = data, order.by = dates)                                # create the xts object
colnames(x) <- 'price'                                              # assign colnames
head(x)                                                             # print the first observations
```

```
##                price
## 2017-05-01 0.9953952
## 2017-05-02 0.9940995
## 2017-05-03 1.0105887
## 2017-05-04 1.0123118
## 2017-05-05 1.0146329
```

```
## 2017-05-06 1.0330492
```

```
# change format of time display
indexFormat(x) <- "%Y/%m/%d"
head(x)
```

```
##                price
## 2017/05/01 0.9953952
## 2017/05/02 0.9940995
## 2017/05/03 1.0105887
## 2017/05/04 1.0123118
## 2017/05/05 1.0146329
## 2017/05/06 1.0330492
```

```
# estimate frequency of observations
periodicity(x)
```

```
## Daily periodicity from 2017-05-01 to 2020-01-25
```

```
# first observation
first(x)
```

```
##                price
## 2017/05/01 0.9953952
```

```
# last observation
last(x)
```

```
##               price
## 2020/01/25 3.039245
```

```
# first 3 days of the last week of data
first(last(x, '1 week'), '3 days')
```

```
##               price
## 2020/01/20 3.059311
## 2020/01/21 3.059618
## 2020/01/22 3.095431
```

```
# convert to OHLC
# valid periods are "seconds", "minutes", "hours", "days", "weeks", "months", "quarters","years"
x.ohlc <- to.period(x, period = 'quarters')
head(x.ohlc)
```

```
##               x.Open   x.High     x.Low  x.Close
## 2017/06/30 0.9953952 1.106982 0.9940995 1.106982
## 2017/09/30 1.1025282 1.217479 1.0792620 1.137135
## 2017/12/31 1.1268060 1.268922 1.1245544 1.233901
## 2018/03/31 1.2586082 1.574023 1.2307955 1.574023
## 2018/06/30 1.5432948 1.632296 1.5026029 1.574151
## 2018/09/30 1.6134651 1.940108 1.5884681 1.865520
```

```
# calculate the yearly mean
ep <- endpoints(x.ohlc, on = "years")
period.apply(x.ohlc , INDEX = ep, FUN = mean)
```

```
##              x.Open   x.High    x.Low  x.Close
## 2017/12/31 1.074910 1.197794 1.065972 1.159339
## 2018/12/31 1.565839 1.813814 1.542075 1.747158
## 2019/12/31 2.227582 2.608671 2.177939 2.480005
```

```
## 2020/01/25 2.954804 3.095431 2.932865 3.039245
```

Code Download

Download the full code to generate this document and reproduce the examples. The file is in R Markdown, format for making dynamic documents with R. An R Markdown document is written in markdown, an easy-to-write plain text format, and contains chunks of embedded R code.
Download: https://storage.googleapis.com/emanueleguidotti/R/introduction-to-R.Rmd