Lecture 2: Atomic Data Types/Homogeneous vectors

Wim R.M. Cardoen

Last updated: 10/05/2022 @ 09:34:04

Contents

R can be summarized in three principles (John M. Chambers, 2016)

- `Everything that exists in R is an` **`object`**`.`
- `Everything that happens in R is a` **`function`** `call.`
- **`Interfaces`** `to other languages are a part of R.`

1 R Objects

- An object in R is (internally) represented as a pair: (**symbol**, **value**).
- A **symbol** is assigned a **value** by the use of an arrow pointing to the left (**<-**).
- There are **less favored** ways:
  - A simple equality sign (**=**).
  - Using the **assign()** function.

1.1 Examples

- Clean up the global environment i.e. remove all objects from the current R environment. `Recommended!`

```
rm(list=ls())
ls()
```

```
character(0)
```

- **preferred** way to assign variables

```
x <- 5.0
x
```

```
[1] 5
```

- alternative 1: mainly used to assign default function arguments

```
y = 5.0
y
```

```
[1] 5
```

```
mysamplevariance <- function(x, av=0){

  n <- length(x)
  if(n>1){
      return(1.0/(n-1)*sum((x-av)^2))
  }
  else{
      stop("ERROR:: Dividing by zero (n==1) || (n==0) ")
  }
}

x <- rnorm(10)
mysamplevariance(x)
```

```
[1] 0.9205981
```

```
mysamplevariance(x,mean(x))
```

```
[1] 0.9044171
```

```
var(x)
```

```
[1] 0.9044171
```

- alternative 2: even less used

```
assign("z", 5.0)
z
```

```
[1] 5
```

- functions are objects

```
f <- mean
f
```

```
function (x, ...)
UseMethod("mean")
<bytecode: 0x55698ce83f70>
<environment: namespace:base>
```

```
val <- f(1:10)
val
```

```
[1] 5.5
```

`"Nothing exists except atoms and empty space; everything else is opinion".  (Democritos)`

2 Atomic Data Types

2.1 The core/atomic data types

- R has the following 6 **atomic** data types:
  - logical (i.e. boolean)
  - integer
  - double
  - character (i.e. string)
  - complex
  - raw (i.e. byte)

The latter 2 types (i.e. complex and especially raw) are less common.

The **typeof()** function determines the **INTERNAL** storage/type of an R object.

2.1.1 Examples

- boolean/logical values: either **TRUE** or **FALSE**

```
x1 <- TRUE
x1
```

```
[1] TRUE
```

```
typeof(x1)
```

```
[1] "logical"
```

- integer values ($\in \mathbb{Z}$):

```
x2 <- 3L
x2
```

```
[1] 3
```

```
typeof(x2)
```

```
[1] "integer"
```

- double (precision) values:

```
x3 <- 3.14
x3
```

```
[1] 3.14
```

```
typeof(x3)
```

```
[1] "double"
```

- character values/strings

```r
x4 <- "Hello world"
x4
```

```
[1] "Hello world"
```

```r
typeof(x4)
```

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

- complex values ($\in \mathbb{C}$):

```r
x5 <- 2.0 + 3i
x5
```

```
[1] 2+3i
```

```r
typeof(x5)
```

```
[1] "complex"
```

2.2 Operations on atomic data types

- **logical** operators: **==**, **!=**, **&&**, **||**, **!**
- **numerical** operators: **+**, **-**, *****, **/**, **^**, ****** (same as the caret), but also:
  - integer division: **%/%**
  - modulo operation: **%%**
  - **Note**: matrix multiplication will be performed using **%*%**
- **character/string** manipulation:
  - **nchar()**:
  - **paste()**:
  - **cat()**:
  - **sprintf()**:
  - **substr()**:
  - **strsplit()**:
  - **Note**: Specialized R libraries were developed to manipulate strings e.g. *stringr*
- explicit **cast**/conversion: https://data-flair.training/blogs/r-string-manipulation/
  - **as.{logical, integer, double, complex, character}()**
- explicit **test** of the type of a variable:
  - **is.{logical, integer, double, complex, character}()**

2.2.1 Examples

- Logical operators:

```
x <-3
y <-7
(x<=3) &&(y==7)
```

```
[1] TRUE
```

```
!(y<7)
```

```
[1] TRUE
```

- Mathematical operations

```
2**4
```

```
[1] 16
```

```
7%%4
```

```
[1] 3
```

```
7/4
```

```
[1] 1.75
```

```
7%/%4
```

```
[1] 1
```

- String operations

```r
s <- "Hello"
nchar(s)
```

```
[1] 5
```

```r
news <- paste(s,"World")
news
```

```
[1] "Hello World"
```

```r
sprintf("My new string:%20s\n", news)
```

```
[1] "My new string:          Hello World\n"
```

```r
city <- "Witwatersrand"
substr(city,4,8)
```

```
[1] "water"
```

- Conversion and testing of types

```r
s <- "Hello World"
is.character(s)
```

```
[1] TRUE
```

```r
s1 <- "-500"
is.character(s1)
```

```
[1] TRUE
```

```r
s2 <- as.double(s1)
is.character(s2)
```

```
[1] FALSE
```

```r
is.double(s2)
```

```
[1] TRUE
```

```r
s3 <- as.complex(s2)
s3
```

```
[1] -500+0i
```

```r
sqrt(s3)
```

```
[1] 0+22.36068i
```

2.3 Exercises

-
  - Calculate $\log_2(10)$ using `R`'s **log()** function
  - Perform the inverse operation and check that you get 10 back
- Let $z = 3 + 4i$
  - Use `R`'s **Re()**, **Im()** functions to extract the real and imaginary parts of z.
  - Calculate the modulus of z using `R`'s **Mod()** function and check whether you the same answer using $\sqrt{\Re(z)^2 + \Im(z)^2}$.
  - Calculate the argument of z using `R`'s **Arg()** function and check whether you have the same answer using $\arctan\left(\frac{\Im(z)}{\Re(z)}\right)$.

3 Atomic vectors

- An **atomic** vector is a data structure containing elements of **only one atomic** data type.
  Therefore, an atomic vector is **homogeneous**.
- Atomic vectors are stored in a **linear** fashion.
- R does **NOT** have scalars:
  - An atomic vector of **length 1** plays the role of a scalar.
  - Vectors of **length 0** also exist (and they have some use!).
- A **list** is a vector not necessarily of the atomic type.
  A list is also known as a **recursive/generic** vector (*vide infra*).

3.1 Creation of atomic vectors

Atomic vectors can be created in a multiple ways:

- Use of the **vector()** function.
- Use of the **c()** function (**c** stands for concatenate).
- Use of the column operator **:**
- Use of the **seq()** and **rep()** functions.

The length of a vector can be retrieved using the **length()** function.

3.1.1 Examples

- use of the **vector()** function:

```
x <- vector()  # Empty vector (Default:'logical')
x
```

```
logical(0)
```

```
length(x)
```

```
[1] 0
```

```
typeof(x)
```

```
[1] "logical"
```

```
x <- vector(mode="complex", length=4)
x
```

```
[1] 0+0i 0+0i 0+0i 0+0i
```

```
length(x)
```

```
[1] 4
```

```
x
```

```
[1] 0+0i 0+0i 0+0i 0+0i
```

```
x[1] <- 4
x
```

```
[1] 4+0i 0+0i 0+0i 0+0i
```

- use of the **c()** function:

```
x1 <- c(3, 2, 5.2, 7)
x1
```

```
[1] 3.0 2.0 5.2 7.0
```

```
x2 <- c(8, 12, 13)
x2
```

```
[1]  8 12 13
```

```
x3 <- c(x2, x1)
x3
```

```
[1]  8.0 12.0 13.0  3.0  2.0  5.2  7.0
```

```
x4 <- c(FALSE,TRUE,FALSE)
x4
```

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

```
x5 <- c("Hello", "Salt", "Lake", "City")
x5
```

```
[1] "Hello" "Salt"  "Lake"  "City"
```

- use of the column operator:

```
y1 <- 1:10
y1
```

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

```
y2 <- 5:-5
y2
```

```
 [1]  5  4  3  2  1  0 -1 -2 -3 -4 -5
```

```
y3 <- 2.3:10
y3
```

```
[1] 2.3 3.3 4.3 5.3 6.3 7.3 8.3 9.3
```

```
y4 <- 2.0*7:1
y4
```

```
[1] 14 12 10  8  6  4  2
```

```
y5 <- 1:7-1
y5
```

```
[1] 0 1 2 3 4 5 6
```

- **seq()** and **rep()** functions

```
z1 <- seq(from=1, to=15, by=3)
z1
```

```
[1]  1  4  7 10 13
```

```r
z2 <- seq(from=-2,to=5,length=4)
z2
```

```
[1] -2.0000000  0.3333333  2.6666667  5.0000000
```

```r
z3 <- rep(c(3,2,4), time=2)
z3
```

```
[1] 3 2 4 3 2 4
```

```r
z4 <- rep(c(3,2,4), each=3)
z4
```

```
[1] 3 3 3 2 2 2 4 4 4
```

```r
z5 <- rep(c(1,7), each=2, time=3)
z5
```

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

```r
length(z5)
```

```
[1] 12
```

3.2 Operations on vectors: element-wise

- All operations on vectors in R happen **element by element** (cfr. *NumPy*).
- **Vector Recycling**:

  If 2 vectors of **different** lengths are involved in an operation, the **shortest vector** will be repeated until all elements of the longest vector are matched.
  A message will be sent to the stdout.

3.2.1 Examples

```r
x <- -3:3
x
```

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

```r
y <- 1:7
y
```

```
[1] 1 2 3 4 5 6 7
```

```r
xy <- x*y
xy
```

```
[1] -3 -4 -3  0  5 12 21
```

```r
xpy <- x^y
xpy
```

```
[1]   -3    4   -1    0    1   64 2187
```

```
x <- 0:10
y <- 1:2
length(x)
```

```
[1] 11
```

```
length(y)
```

```
[1] 2
```

```
x
```

```
 [1]  0  1  2  3  4  5  6  7  8  9 10
```

```
y
```

```
[1] 1 2
```

```
x+y
```

```
Warning in x + y: longer object length is not a multiple of shorter object
length
```

```
 [1]  1  3  3  5  5  7  7  9  9 11 11
```

3.3 Retrieving elements of vectors

- Indexing: starts at **1** (**not 0** like C/C++, Python, Java, ...) see also: Edsger Dijkstra: Why numbering should start at zero
- Use of vector with indices to extract values.
- Advanced features:
  - use of boolean values to extract values.
  - the membership operator: **%in%**.
  - the deselect/omit operator: **-**
  - **which()**: returns the indices for which the condition is true.
  - **any()**/**all()** functions.
    - **any()** : **TRUE** if at least 1 value is true
    - **all()** : **TRUE** if all values are true

3.3.1 Examples

- Use of a simple index:

```
x <- seq(2,100,by=15)
x[4]
```

```
[1] 47
```

```
x[1]
```

```
[1] 2
```

- Select several indices at once using vectors:

```r
x
```

```
[1]  2 17 32 47 62 77 92
```

```r
x[3:5]
```

```
[1] 32 47 62
```

```r
x[c(1,3,5,7)]
```

```
[1]  2 32 62 92
```

```r
x[seq(1,7,by=2)]
```

```
[1]  2 32 62 92
```

- Extraction via booleans (i.e. retain only those values that are equal to **TRUE**):

```r
x
```

```
[1]  2 17 32 47 62 77 92
```

```r
x>45
```

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

```r
x[x>45]
```

```
[1] 47 62 77 92
```

- Use of the **%in%** operator:

```r
x
```

```
[1]  2 17 32 47 62 77 92
```

```r
10 %in% x
```

```
[1] FALSE
```

```r
62 %in% x
```

```
[1] TRUE
```

```r
c(32,33,43) %in% x
```

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

```r
!(c(32,33,43) %in% x)
```

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

- Negate/filter out the elements with **negative** indices:

```r
x
```

```
[1]  2 17 32 47 62 77 92
```

```
x[-c(2,4,6)]
```

```
[1]  2 32 62 92
```

```
z <- x[-1] - x[-length(x)]
z
```

```
[1] 15 15 15 15 15 15
```

- The **which()** function returns **only those indices** of which the condition/expression is **true**.

```
# Sample 10 numbers from N(0,1)
vecnum <- rnorm(n=10)
vecnum
```

```
 [1] -0.8493688  1.4227308 -0.2265752  0.1218204  0.2978450 -1.1198118
 [7]  0.9833643  1.7506498 -0.2007401 -0.6080138
```

```
which(vecnum>1.0)
```

```
[1] 2 8
```

- Use of the **any()**/**all()** functions.

```
y <- seq(0,100,by=10)
x
```

```
[1]  2 17 32 47 62 77 92
```

```
y
```

```
 [1]   0  10  20  30  40  50  60  70  80  90 100
```

```
any(x<y)
```

```
Warning in x < y: longer object length is not a multiple of shorter object
length
```

```
[1] TRUE
```

```
all(x[6:7]>y[2:3])
```

```
[1] TRUE
```

3.4 Hash tables

A **hash table** is a data structure which implements an associative array or dictionary. It is an abstract data which maps data to keys.

- There are several ways to create one:
  - Map names to an existing vector
  - Add names when creating the vector
- To remove the map, map the names to NULL

3.4.1 Examples

- Creation of 2 independent vectors

```
capitals <- c("Albany", "Providence", "Hartford", "Boston", "Montpelier", "Concord", "Augusta")
states <- c("NY", "RI", "CT", "MA", "VT", "NH", "ME")
capitals
```

```
[1] "Albany"     "Providence" "Hartford"   "Boston"     "Montpelier"
[6] "Concord"    "Augusta"
```

```
states
```

```
[1] "NY" "RI" "CT" "MA" "VT" "NH" "ME"
```

```
capitals[3]
```

```
[1] "Hartford"
```

- Create the hashtable/dictionary

```
# Method 1
names(capitals) <- states
capitals
```

```
          NY           RI           CT           MA           VT           NH 
    "Albany" "Providence"   "Hartford"     "Boston" "Montpelier"    "Concord" 
          ME 
   "Augusta" 
```

```
capitals["MA"]
```

```
      MA 
"Boston" 
```

```
names(capitals)
```

```
[1] "NY" "RI" "CT" "MA" "VT" "NH" "ME"
```

```
# Method 2
phonecode <- c("801"="SLC", "206"="Seattle", "307"="Wyoming")
phonecode
```

```
      801       206       307 
    "SLC" "Seattle" "Wyoming" 
```

```
phonecode["801"]
```

```
  801 
"SLC" 
```

- Dissociate the 2 vectors

```r
names(capitals) <- NULL
capitals
```

```
[1] "Albany"     "Providence" "Hartford"   "Boston"     "Montpelier"
[6] "Concord"    "Augusta"
```

3.5 NA (Not Available values)

- **NA**: stands for 'Not Available'/Missing values
- has length of 1.
- **is.na()**: test all elements of a vector for NA values.
- some functions e.g. **mean()** return NA when an instance of NA is present.

3.5.1 Examples

- Check of the NA availability

```r
x <- c(NA, 1, 2, NA)
is.na(x)
```

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

- Functions on a vector containing NA

```r
mean(x)
```

```
[1] NA
```

```r
mean(x, na.rm=TRUE)
```

```
[1] 1.5
```

3.6 Alia

Still to be developed!

- boolean: Vector operators vs. unique value
- && vs. &.
- || vs. |.
- xor()

3.7 Exercises

- Use the **seq()** function to generate the following sequence:
  6 13 20 27 34 41 48
- R has the its own inversion function, **rev()**, e.g,:

```
x <- seq(from=2,to=33,by=3)
x
```

```
 [1]  2  5  8 11 14 17 20 23 26 29 32
```

```
y <- rev(x)
y
```

```
 [1] 32 29 26 23 20 17 14 11  8  5  2
```

  Invert the vector x without invoking the **rev()** function.

- Create the following R vector using **only** the **seq()** and **rep()** functions:
  -8 -8 -8 -8 0 8 8 8 16 16 16 16 16
- Create the following vector (do **not** use **c()**!):
  -512 -216 -64 -8 0 8 64 216 512 1000
- Generate a random vector of integers using the following code:

```
x <- sample(x=0:1000,size=100, replace=TRUE)
```

  - Invoke the above code to generate the vector `x`
  - Find if there are any integers in the vector `x` which can be divided by 4 and 6
  - Find those numbers and their corresponding indices in the vector `x`.

- The `Taylor series` for $\ln(1+x)$ is converging when $|x| < 1$ and is given by:

$$\ln(1+x) = x - \frac{x^2}{2} + \frac{x^3}{3} - \frac{x^4}{4} + \frac{x^5}{5} - \frac{x^6}{6} + \dots$$

  Calculate the sum of the first 5, 10, 15 terms in the above expression to approximate $\ln(1.2)$. Compare with `R`'s value i.e.: **log(1.2)**.

- The `logarithmic` return in finance is defined as:

$$R_t = \ln\left(\frac{P_t}{P_{t-1}}\right)$$

  - Generate a financial time series using the following `R` code:

```
thecasino <- abs(rcauchy(1000))+1.E-6
```

  - Calculate the `logarithmic return` for the financial time series `thecasino`.
    Your adjusted time series will be 1 element shorter in length than the original one.
    Compare your result with **diff(log(thecasino))**.

- `Monte-Carlo` approximation of π

  Let `S1` be the square spanned by the following 4 vertices: $\{(0,0),(0,1),(1,0),(1,1)\}$.
  Let `S2` be the first quadrant of the unit-circle $\mathcal{C}: x^2 + y^2 = 1$.

  The ratio ρ defined as:

$$\rho := \frac{\text{Area S2}}{\text{Area S1}} = \frac{\#\text{Points in S2}}{\#\text{Points in S1}}$$

allows us to estimate $\frac{\pi}{4}$ numerically.

Therefore:

- Sample 100000 independent x-coordinates from `Unif`.
- Sample 100000 independent y-coordinates from `Unif`.
- Calculate an approximate value for π using the Monte-Carlo approach.

Note: The uniform distribution $[0, 1)$ (`Unif`) can be sampled using **runif()**.

- A family has installed a device to monitor their daily energy consumption (in kWh).
When a measurement fails or is unavailable NA is recorded.

You can invoke the following code to generate the measurements generated by the device.

```
dailyusage <- 30.0 + runif(365, min=0, max=5.0)
dailyusage[sample(1:365, sample(1:50,1), replace=FALSE)] <- NA
```

- How many measurements failed?
- What is the average daily energy consumption (based on the non-failed) measurements?

```
"It is my experience that proofs involving matrices can be
shortened by 50% if one throws the matrices out" (Emil Artin)
```

4 Matrices & Arrays

Matrices and arrays are **homogeneous atomic vectors** with
an **extra** attribute: dimension

By default, the elements are stored in a **column-major** fashion. (cfr. **Fortran**).
However, we can store the elements in **row-major** order (cfr. **C**) as well.

4.1 Creation of matrices

Matrices can be created in different ways:

- use of the **matrix()** function
- use of **rbind()**/**cbind()**
- set the **attributes()** of a vector
- special functions like e.g. **diag()**

4.1.1 Examples

- use of the **matrix()** function:

The **matrix()** function creates a matrix based on a vector.
By default, the elements are stored in a **column-major** fashion.

The use of the flag `byrow=TRUE` will store the data in a **row-major** fashion.

```r
A <- matrix(data=1:10, nrow=2)   # Column-major (like Fortran)
A
```

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

```r
B <- matrix(data=c(2,3,893,0.17), nrow=2, ncol=2)
B
```

```
     [,1]   [,2]
[1,]    2 893.00
[2,]    3   0.17
```

```r
C <- matrix(data=1:10, nrow=2, byrow=TRUE)   # Row-major (like C, C++)
C
```

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

- use of the **rbind()**/**cbind()** functions:

  - **rbind()**: Bind several vectors (as rows) into a matrix.
  - **cbind()**: Bind several vectors (as columns) into a matrix.

```
A <- rbind(1:10,11:20)
A
```

```
     [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
[1,]    1    2    3    4    5    6    7    8    9    10
[2,]   11   12   13   14   15   16   17   18   19    20
```

```
typeof(A)
```

```
[1] "integer"
```

```
class(A)
```

```
[1] "matrix" "array"
```

```
B <- cbind(1:5,6:10,11:15)
B
```

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

```
class(B)
```

```
[1] "matrix" "array"
```

- modifying the **dim** attribute.

The **fundamental** difference between an `R` vector and matrix is the presence (in the case of matrices) of a non `NULL` **dim** attribute.

We change easily convert a vector into a matrix by setting the dimensions of the vector:

- through the **dim()** function.
- through the **attr()** function.

The inverse can be done as well by setting the **dim** attribute of matrix to `NULL`.

4.1.2 Examples

```
A <- 1:10
typeof(A)
```

```
[1] "integer"
```

```
class(A)
```

```
[1] "integer"
```

```
dim(A)
```

```
NULL
```

```r
# Matrix
B <- matrix(1:10,nrow=2,ncol=5,byrow=TRUE)
typeof(B)
```

```
[1] "integer"
```

```r
class(B)
```

```
[1] "matrix" "array"
```

```r
dim(B)
```

```
[1] 2 5
```

```r
# Vector
A <- 1:10
A
```

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

```r
dim(A)
```

```
NULL
```

```r
typeof(A)
```

```
[1] "integer"
```

```r
class(A)
```

```
[1] "integer"
```

```r
# OPTION I: Using the dim function transform a vector into a matrix
dim(A) <- c(2,5)
A
```

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

```r
dim(A)
```

```
[1] 2 5
```

```r
typeof(A)
```

```
[1] "integer"
```

```r
class(A)
```

```
[1] "matrix" "array"
```

```r
# Converting the matrix back to a vector
dim(A) <- NULL
dim(A)
```

```
NULL
```

```r
typeof(A)
```

```
[1] "integer"
```

```r
class(A)
```

```
[1] "integer"
```

```r
# Option II: More general way
# Convert vector into a matrix
A <- 1:8
A
```

```
[1] 1 2 3 4 5 6 7 8
```

```r
class(A)
```

```
[1] "integer"
```

```r
attr(A,'dim') <- c(2,4)
A
```

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

```r
class(A)
```

```
[1] "matrix" "array"
```

```r
# Convert matrix into a vector.
attr(A, 'dim') <- NULL
A
```

```
[1] 1 2 3 4 5 6 7 8
```

```r
class(A)
```

```
[1] "integer"
```

4.2 Operations on matrices

- Operations like ***,/, +** happpen element-wise.
- There are also more specialized functions:
  - the mean over rows and columns (**rowMeans()**, **colMeans()**)
  - linear algebra functions (**%*%**, **t()**, ...)

4.2.1 Examples

- Operations (by **default: element-by-element**):

```
A <- matrix(1:10, nrow=2)
B <- matrix( seq(10, 100, by=10), nrow=2)
A
```

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

```
B
```

```
     [,1] [,2] [,3] [,4] [,5]
[1,]   10   30   50   70   90
[2,]   20   40   60   80  100
```

```
A*B
```

```
     [,1] [,2] [,3] [,4] [,5]
[1,]   10   90  250  490  810
[2,]   40  160  360  640 1000
```

```
C <- matrix(rep(2,10), nrow=2)
C**A
```

```
     [,1] [,2] [,3] [,4] [,5]
[1,]    2    8   32  128  512
[2,]    4   16   64  256 1024
```

- Calculate **row and column means** :

```
# Means of rows and columns
A
```

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

```
rowMeans(A)
```

```
[1] 5 6
```

```
colMeans(A)
```

```
[1] 1.5 3.5 5.5 7.5 9.5
```

- **Matrix multiplication** (%*%) :

```
A <- matrix(1:6, nrow=2)
A
```

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

```r
B <- matrix(seq(10,120,by=10), nrow=3)
B
```

```
     [,1] [,2] [,3] [,4]
[1,]   10   40   70  100
[2,]   20   50   80  110
[3,]   30   60   90  120
```

```r
C <- A%*%B
C
```

```
     [,1] [,2] [,3] [,4]
[1,]  220  490  760 1030
[2,]  280  640 1000 1360
```

```r
dim(C)
```

```
[1] 2 4
```

- **Linear algebra** routines

Some of the more common ones in R:

- **solve()** : **invert** a square matrix
- **diag()**
  - **extracts** the diagonal of a matrix when a matrix is provided.
  - **creates** a diagonal matrix when a vector is provided.
- **eigen()** : calculates the **eigenvalues** and **eigenvectors** of a matrix
- **det()** : calculates the **determinant** of a matrix.
- **t()**: calculates the **transpose**[1] of a matrix.

```r
# Invert matrix A
A <- matrix(c(1, 3, 2, 4), ncol = 2, byrow = T)
Ainv <- solve(A)
Ainv %*% A
```

```
     [,1] [,2]
[1,]    1    0
[2,]    0    1
```

```r
# Create a diagonal matrix
C <- diag(c(1,4,7))
C
```

```
     [,1] [,2] [,3]
[1,]    1    0    0
[2,]    0    4    0
[3,]    0    0    7
```

```r
# Extract the diagonal elements
D <- matrix(1:8,nrow=4)
D
```

[1] Can also be used for dataframes (see later)

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

```r
diag(D)
```

```
[1] 1 6
```

```r
# Calculate eigenvalues and eigenvectors of A
r <- eigen(A)
r
```

```
eigen() decomposition
$values
[1]  5.3722813 -0.3722813

$vectors
           [,1]       [,2]
[1,] -0.5657675 -0.9093767
[2,] -0.8245648  0.4159736
```

```r
# Eigenvalues
r$values
```

```
[1]  5.3722813 -0.3722813
```

```r
# Matrix with eigenvectors
r$vectors
```

```
           [,1]       [,2]
[1,] -0.5657675 -0.9093767
[2,] -0.8245648  0.4159736
```

```r
# Diagonal Matrix (Similarity Transformation)
solve(r$vectors) %*% A %*% r$vectors
```

```
              [,1]       [,2]
[1,]  5.372281e+00  0.0000000
[2,] -3.330669e-16 -0.3722813
```

Note that under the hood `R` calls `BLAS` and `LAPACK`.

```r
# Find the version used of BLAS and LAPACK
La_library()
```

```
[1] "/usr/lib/x86_64-linux-gnu/lapack/liblapack.so.3.7.1"
```

```r
extSoftVersion()["BLAS"]
```

```
                                              BLAS
"/usr/lib/x86_64-linux-gnu/blas/libblas.so.3.7.1"
```

4.3 Retrieving elements/subsetting

Matrices (and arrays) can be subsetted in different ways:

- use an **index** for each dimension, where the dimensions are comma-separated
  - If an **index** for a dimension is **omitted**:
    consider all dimensions (may lead to reduction of the dimension)
  - **but** you can use **drop=FALSE** to prevent dimensionality reduction.
- use another **vector** (can be either linear or a vector for each dimension)
- by using another **matrix**.

4.3.1 Examples

- Use of **indices**:

```
A <- matrix(1:30, nrow=6, ncol=5)
A
```

```
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    7   13   19   25
[2,]    2    8   14   20   26
[3,]    3    9   15   21   27
[4,]    4   10   16   22   28
[5,]    5   11   17   23   29
[6,]    6   12   18   24   30
```

```
A[3,4]
```

```
[1] 21
```

```
A[6,2]
```

```
[1] 12
```

```
x1 <- A[2,]
x1
```

```
[1]  2  8 14 20 26
```

```
dim(x1)
```

```
NULL
```

```
x2 <- A[,3]
x2
```

```
[1] 13 14 15 16 17 18
```

```
dim(x2)
```

```
NULL
```

The flag **drop=FALSE** can be used to prevent dimensionality reduction

```
y1 <- A[2,,drop=FALSE]
y1
```

```
     [,1] [,2] [,3] [,4] [,5]
[1,]    2    8   14   20   26
```

```r
dim(y1)
```

```
[1] 1 5
```

```r
y2 <- A[,3,drop=FALSE]
y2
```

```
     [,1]
[1,]   13
[2,]   14
[3,]   15
[4,]   16
[5,]   17
[6,]   18
```

```r
dim(y2)
```

```
[1] 6 1
```

- Use of **vector(s)**:

```r
A
```

```
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    7   13   19   25
[2,]    2    8   14   20   26
[3,]    3    9   15   21   27
[4,]    4   10   16   22   28
[5,]    5   11   17   23   29
[6,]    6   12   18   24   30
```

```r
x1 <- A[2:4,]
x1
```

```
     [,1] [,2] [,3] [,4] [,5]
[1,]    2    8   14   20   26
[2,]    3    9   15   21   27
[3,]    4   10   16   22   28
```

```r
dim(x1)
```

```
[1] 3 5
```

```r
x2 <- A[,1:3]
x2
```

```
     [,1] [,2] [,3]
[1,]    1    7   13
[2,]    2    8   14
[3,]    3    9   15
```

```
[4,]    4   10   16
[5,]    5   11   17
[6,]    6   12   18
```

```r
dim(x2)
```

```
[1] 6 3
```

```r
# Using a vector for EACH dimension
A[c(1,3),c(2,4)]
```

```
     [,1] [,2]
[1,]    7   19
[2,]    9   21
```

```r
# Using 1 vector => Linear index
A[c(1,3,8,10)]
```

```
[1]  1  3  8 10
```

```r
A[c(TRUE,FALSE,TRUE,TRUE,FALSE,TRUE),c(2,3)]
```

```
     [,1] [,2]
[1,]    7   13
[2,]    9   15
[3,]   10   16
[4,]   12   18
```

```r
A[c(TRUE,FALSE,TRUE,TRUE,FALSE,TRUE),]
```

```
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    7   13   19   25
[2,]    3    9   15   21   27
[3,]    4   10   16   22   28
[4,]    6   12   18   24   30
```

```r
# Use of a linear index
A[c(TRUE,FALSE,TRUE,TRUE,FALSE,TRUE)]
```

```
 [1]  1  3  4  6  7  9 10 12 13 15 16 18 19 21 22 24 25 27 28 30
```

- Use of a **matrix**:

```r
A
```

```
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    7   13   19   25
[2,]    2    8   14   20   26
```

```
[3,]    3    9   15   21   27
[4,]    4   10   16   22   28
[5,]    5   11   17   23   29
[6,]    6   12   18   24   30
```

```
mysubset <- matrix(c( 2, 1,
                      3, 5,
                      4, 2,
                      6, 5), ncol=2, byrow=TRUE )
A[mysubset]
```

```
[1]  2 27 10 30
```

4.4 Hash tables/dictionaries

We can also use hashes for matrices. We can select one or both dimensions. To create hashes, for: - rows: use **rownames** - columns: use **colnames**

To remove the hash, use the **NULL** (like for vectors).

4.4.1 Examples

```
A1 <- c(0       ,       5471.52, 5091.57,       5392.82,
        5416.45,       4584.33, 4904.83,       3851.73)
A2 <- c(5471.52,       0, 1315.28,      927.35,
        1505.11,        944.40, 1157.42,       1945.42)
A3 <- c(5091.57,     1315.28,             0, 2166.00,
        2724.01,       1571.76,        293.52, 1240.77)
A4 <- c(5392.82,      927.35, 2166.00,             0,
        577.85,  973.23, 1947.28,       2422.32)
A5 <- c(5416.45,     1505.11, 2724.01,        577.85,
        0, 1366.63, 2490.97,       2838.62)
A6 <- c(4584.33,      944.40, 1571.76,        973.23,
        1366.63,             0, 1290.15,       1474.26)
A7 <- c(4904.83,     1157.42,  293.52,       1947.28,
        2490.97,       1290.15,             0, 1064.41)
A8 <- c(3851.73,     1945.42, 1240.77,       2422.32,
        2838.62,       1474.26, 1064.41,       0)
```

```
dist <- rbind(A1,A2,A3,A4,A5,A6,A7,A8)
dist
```

```
      [,1]    [,2]    [,3]    [,4]    [,5]    [,6]    [,7]    [,8]
A1    0.00 5471.52 5091.57 5392.82 5416.45 4584.33 4904.83 3851.73
A2 5471.52    0.00 1315.28  927.35 1505.11  944.40 1157.42 1945.42
A3 5091.57 1315.28    0.00 2166.00 2724.01 1571.76  293.52 1240.77
A4 5392.82  927.35 2166.00    0.00  577.85  973.23 1947.28 2422.32
A5 5416.45 1505.11 2724.01  577.85    0.00 1366.63 2490.97 2838.62
A6 4584.33  944.40 1571.76  973.23 1366.63    0.00 1290.15 1474.26
```

```
A7 4904.83 1157.42  293.52 1947.28 2490.97 1290.15    0.00 1064.41
A8 3851.73 1945.42 1240.77 2422.32 2838.62 1474.26 1064.41    0.00
```

```r
# Adding hashes to both rows and columns
cities <- c("Anchorage","Atlanta","Austin","Baltimore","Boston", "Chicago", "Dallas","Denver")
rownames(dist) <- cities
colnames(dist) <- cities
dist
```

```
          Anchorage Atlanta  Austin Baltimore  Boston Chicago  Dallas  Denver
Anchorage      0.00 5471.52 5091.57   5392.82 5416.45 4584.33 4904.83 3851.73
Atlanta     5471.52    0.00 1315.28    927.35 1505.11  944.40 1157.42 1945.42
Austin      5091.57 1315.28    0.00   2166.00 2724.01 1571.76  293.52 1240.77
Baltimore   5392.82  927.35 2166.00      0.00  577.85  973.23 1947.28 2422.32
Boston      5416.45 1505.11 2724.01    577.85    0.00 1366.63 2490.97 2838.62
Chicago     4584.33  944.40 1571.76    973.23 1366.63    0.00 1290.15 1474.26
Dallas      4904.83 1157.42  293.52   1947.28 2490.97 1290.15    0.00 1064.41
Denver      3851.73 1945.42 1240.77   2422.32 2838.62 1474.26 1064.41    0.00
```

```r
dist["Chicago", "Denver"]
```

```
[1] 1474.26
```

```r
dist["Austin", "Boston"]
```

```
[1] 2724.01
```

4.5 Arrays

Say something about arrays.

4.6 Exercises

- Do some slicing
- Calculate the colSums/rowSums with and without colSums
- outer, lower and upper tridiag
- Linear regression $Y = X\text{Beta}+\text{epsilon}$ Create a model $Y = 0.2 + 1.5\ X + 2.5\ X\hat{}2 + \text{epsilon}(0, \text{sig}=2.)$ Choose: X from 1 to 10 Create matrix X

5 Other types

- Attributes
- Special types:
  - Factors
  - Date
  - Time
- NA, NaN, NULL

Other topics on Data structures

- List
- Dataframe & Tibble
- IO (read.csv, read.file)
- Names
- Subsetting, [[]] vs. []

Conditionals & Loops

- if, else, else if switch and elseif
- for
- while
- repeat
- return()

Environments

- search(), attach, detach
- library

Functions

- lexical scoping
- simple functions
- args(), formals()
- default arg, ...
- lazy evaluation
- closure
- anonymous functions
- make your own operators
- loop functions: {l,s,m}apply, split

Capita selecta

- profiling, debugging