Review Final - Vectors, Lists, Functions, and Classes in R

In [1]:
```
%load_ext rmagic
%R rm()
```

In [2]:
```
%%R

NL = function() cat('\n\n')
```

Review Session Topics:

In this review section, we will focus on some of the key features of the R language: Vectorized computation, arrays, lists, and classes:

- **Vectorized programming in R**: vectors and functions
- **Object-oriented programming in R**: lists and classes

R is an array programming language (http://en.wikipedia.org/wiki/Array_programming) (or vectorized language) which means that its **basic data types** (i.e. numbers, Booleans, and strings) come in the form of **vectors**, which are **homogeneous containers** (or collections) of numbers, strings, or Boolean. This vectorized basic types are particular suited for data analysis: they are used to store values of **statistical variables** from a **population sample**. A vectors in R corresponds to the information stored in **a data table column**.

Parallely, R has a single **basic data structure**, called `list`, which is a **heterogenous container**. List can store collection of elements with different types, and can be thought of as representing the information stored in a **data table row**.

Both vectors and lists are **labelled containers**, which means that their elements can be assigned **labels** or **names**. In terms of statistics, vector labels can be regarded as **individual identifier** for the individuals in the sample, while list labels can be thought of as **variable names** in a data table (i.e. the **header labels**).

Programming in R would be very similar to programming in Python if we were allowed only to use dictionaries (corresponding to R lists) and Numpy arrays (corresponding to R vectors), in guise of data structures and data types.

There are two ways of programming with **classes and objects** in R. The less used one (**S4 classes**) ressembles much the object-oriented framework in Python, but we didn't see that in class. The most popular object-oriented framework in R (**S3 classes**) is very diffent from that of Python. It ressembles more a set of programming conventions than a strongly enforced framework provided by the language. S3 programming is quick and dirty, as one says in programming, and instanciating classes with wrong attributes is very easy...

Basically, **basic data types (vectors) and basic data structures (lists) can be promoted to classes by setting the hidden attribute "class"** to a custom name: the name of the class to which we are promoting the given vector or list to.

Class attributes are stored as named elements of the underlying list, using the **dollar sign operator** `X$attributeName`, corresponding to the **period operator** in Python: `X.attributeName`.

Methods are then normal functions that check the class name and act accordingly.

A data frames is a S3 class used to represent a data table. Data frame come up with a variety of methods to transform, analyse, and visualyse the data contained in the data frame.

Vectorized programming in R: vectors and functions

Ressources: Phil Spector Stat 133 lecture notes

- Vectors and Matrices (http://www.stat.berkeley.edu/users/spector/s133/R-2a.html)
- Functions (http://www.stat.berkeley.edu/users/spector/s133/R-6a.html)

Basic types in R are **vectors**. R does not have **scalar** types as in Python: a single string or a single number is always understood by R as a character or numeric vector **with one element**.

The elements of a vector are all of the same **mode** or **type**, which is queried upon using the functions `mode(x)` and `typeof(x)` (the latter function having a "finer grain" than the former, since it distinguishes between `integer` and `double`. Here are the different types in R:

- **numeric vectors:** elements are *numbers*
  - `mode(x)` ⟶ `numeric`
  - `typeof(x)` ⟶ `integer` or `double`
- **character vectors:** elements are *strings*
  - `mode(x), typeof(x)` ⟶ `character`
- **logical vectors:** elements are *Booleans*, i.e., either `TRUE` (short: `T`) or `FALSE` (short: `F`)
  - `mode(x), typeof(x)` ⟶ `logical`

Convertion between types

Convertions between basic types are possible using the **convertion function family**

```
as.logical(x)
as.integer(x)
as.double(x)
as.numeric(x)
as.character(x)
```

However, not all convertions are possible without loss of information (i.e. one can not or loose information by converting back to the original type). Here is the convertion chain with **no information loss** or errors:

`logical` $\longrightarrow$ `integer` $\longrightarrow$ `double` $\longrightarrow$ `character`

Forcing conversion by going in the opposite arrow direction may result in an error or in a loss of information:

- `integer` $\longrightarrow$ `logical` will only keep the information of whether the integer is zero or non-zero (see below)
- `double` $\longrightarrow$ `integer` results in the **decimal part is being lost**
- `character` $\longrightarrow$ `double` returns **missing value** `NA` provided the string is not a string resulting from the conversion of the lower types to `character` types (e.g. "34.5" or "0")

Conversion involving Booleans are a bit special:

- `T` $\longleftrightarrow$ 1 and `F` $\longleftrightarrow$ 0
- `T` $\longleftrightarrow$ "1" and `F` $\longleftrightarrow$ "0"
- `T` $\longleftrightarrow$ "TRUE" and `F` $\longleftrightarrow$ "FALSE"
- `T` $\longleftrightarrow$ "true" and `F` $\longleftrightarrow$ "false"
- non-zero number $\longleftrightarrow$ `T`

Remark: These convertions are important when creating data frame for text files in order to understand which **mode** will be assigned to the data frame columns.

In [3]:
```
%%R

a = c(T, F, T, F, F)

char = as.character(a)
numb = as.integer(a)

print(char)
print(numb)
```

```
[1] "TRUE"  "FALSE" "TRUE"  "FALSE" "FALSE"
[1] 1 0 1 0 0
```

In [3]:

In [3]:

Generating vectors

There are several way to generate vectors in R. In term of statistics, vectors corresponds to **statitical variables**. Simulating (or generating) variables is an important task in statistics.

Remark: Do not confuse the term *variable* as used in programming (which is just a **name** used to hold a **value**) and the term *variable* as used in statistics. Statistical variables correspond to vectors in R. Statistical variables, i.e. vectors, can be stored in R variables. However, R variables can also be used to store lists, data frames, or instances of classes.

Here is a list of function very useful to generate vectors (hence: statistical variables).

- The **concatenate** function (http://stat.ethz.ch/R-manual/R-devel/library/base/html/c.html)

In [4]:
```
%%R

x = c('a', 'b', 'c')
```

- The **replicate** function (http://stat.ethz.ch/R-manual/R-devel/library/base/html/rep.html)

In [5]:
```
%%R
x = rep(T, times=30)
x = rep(c(1,2,4), lenght=5)
```

- The **sequence generation** function (http://stat.ethz.ch/R-manual/R-devel/library/base/html/seq.html)

In [6]:
```
%%R
x = seq(1, 100, by=5)
```

- The **string concatenate** function (http://stat.ethz.ch/R-manual/R-devel/library/base/html/paste.html)

In [7]:
```
%%R
x = paste(c('X', 'Y', 'Z'), c(1,2,3), sep='')
```

- The **colon operator** (http://stat.ethz.ch/R-manual/R-devel/library/base/html/Colon.html)

In [8]:
```
%%R
x = 12:49
```

- The **random sample and permutation** function (http://stat.ethz.ch/R-manual/R-devel/library/base/html/sample.html)

In [9]:
```
%%R

x = sample(100)
y = sample(c('Bob', 'Luc', 'Martin'), 100, replace=T, prob=c(0.2, 0.5, 0.3
```

```
))

print(x)
print(y)
```

```
  [1]  52  29  21  83  39  18  40  84  23   5  74  46  65  11  57  56  86
 47
 [19]  19  27  41  78  48  97  62  44  54   8  33  68  87  79  80   7  76
 60
 [37]  22  93  59  14  91   1  20  34  64  45  70  85  82  36  95  88  10
 25
 [55]   6  17  13 100  50  51  31  24  77  72   3  75  96   9  38  37  55
 66
 [73]  94  81  35  28   2  89  58  16  73  98  69  53  99   4  32  30  42
 71
 [91]  12  61  43  15  67  90  26  92  63  49
  [1] "Martin" "Bob"    "Luc"    "Martin" "Luc"    "Bob"    "Bob"    "Luc"

  [9] "Bob"    "Martin" "Bob"    "Luc"    "Luc"    "Luc"    "Luc"    "Bob"

 [17] "Martin" "Bob"    "Luc"    "Luc"    "Martin" "Luc"    "Luc"    "Mart
in"
 [25] "Luc"    "Luc"    "Luc"    "Bob"    "Luc"    "Luc"    "Bob"    "Mart
in"
 [33] "Martin" "Luc"    "Bob"    "Luc"    "Bob"    "Martin" "Martin" "Luc"

 [41] "Martin" "Luc"    "Martin" "Martin" "Luc"    "Bob"    "Bob"    "Luc"

 [49] "Bob"    "Bob"    "Martin" "Bob"    "Luc"    "Luc"    "Luc"    "Luc"

 [57] "Luc"    "Luc"    "Martin" "Martin" "Luc"    "Luc"    "Martin" "Mart
in"
 [65] "Martin" "Martin" "Luc"    "Luc"    "Luc"    "Luc"    "Martin" "Luc"

 [73] "Luc"    "Luc"    "Luc"    "Luc"    "Bob"    "Bob"    "Luc"    "Luc"

 [81] "Martin" "Bob"    "Martin" "Luc"    "Martin" "Martin" "Luc"    "Luc"

 [89] "Martin" "Luc"    "Luc"    "Luc"    "Luc"    "Luc"    "Luc"    "Luc"

 [97] "Luc"    "Bob"    "Luc"    "Bob"
```

- The normal random sample function (http://stat.ethz.ch/R-manual/R-devel/library/stats/html/Normal.html) and the uniform random sample function (http://stat.ethz.ch/R-manual/R-devel/library/stats/html/Uniform.html)

In [10]:
```
%%R

x = rnorm(100, mean=0, sd=1)
x = runif(100, min=0, max=1)
```

Vectorized code

R is a **vectorized** programming language (or array programming language

(http://en.wikipedia.org/wiki/Array_programming)): i.e., operations involving **vectors** (1D array), **matrices** (2D array), or more generally n-dimensional **array** can be performed with **single** or **atomic instructions**, instead of **loops** accessing the array elements one at a time. For instance, the core of the Python language is not vectorized (one uses loops or list comprehension to iterate over collections), but the Numpy part of the language is vectorized.

Languages allowing for vectorized computations offer several advantages over standard "collection looping languages:"

- The code involving array computations is **clearer** and **shorter**, resulting into a gain of developping time and an easier maintenance
- When using computers with multiple processors, atomic vectorized instructions will be distributed in parallely over the processors, resulting into a **speed gain**.

The main three array programming languages in the spot lights in data science nowdays are **R**, **Octave** (a GNU version of Mathlab), and **Python with Numpy**. You already know how to use two of them, and learning Octave after this course will be very easy.

Here is an example of a **non-vectorized code**, where we add two vectors `x` and `y` by accessing the elments one at a time in a `for` loop, add them and store the result into the corresponding component of a previously intialized vector `z`:

In [11]:
```
%%R

x = sample(10)
y = sample(10)

z = rep(0, 10)
for(i in 1:length(x)){
    z[i] = x[i] + y[i]
}

print(x); print(y); print(z)
```

```
 [1]  6  3 10  4  5  8  1  9  7  2
 [1]  3  4  6 10  8  5  1  2  9  7
 [1]  9  7 16 14 13 13  2 11 16  9
```

Now we do the same with **vectorized code**, using the **vectorized addition operation:**

In [12]:
```
%%R

z = x + y
print(x); print(y); print(z)
```

```
 [1]  6  3 10  4  5  8  1  9  7  2
 [1]  3  4  6 10  8  5  1  2  9  7
 [1]  9  7 16 14 13 13  2 11 16  9
```

Another example of standard looping and branching code together with the equivalent

Another example of standard looping and branching code together with the equivalent vectorized code. Given a vector `x` of integers between 1 and 10, we want to keep only those integers that are above 5.

Make sure that you understand that the vectorized equivalent of branching statements in a loop is usually achieved using logical indexing.

In [13]:
```
%%R

x = sample(10)
z = c()
j = 1
for (i in 1:10){
    if(x[i] >= 5){
        z[j] = x[i]
        j = j + 1
    }
}

print(z)
```

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

In [14]:
```
%%R

#Equivalent vectorized code

z = x[x >= 5]
print(z)
```

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

To make a long story short, **writing vectorized code is nothing but using only vectorized opearations when dealing with vectors, or more generally with arrays**. You should be able to go back an forth between standard `for`, `if`, `else` code and corresponding vectorized code.

Using only the following functions and operations on the basic types will generate vectorized code:

- **Numeric fuctions and operations (http://www.statmethods.net/management/functions.html):**

```
  abs(x)
  sqrt(x)
  ceiling(x)
  floor(x)
  trunc(x)
  round(x, digits=n)
  cos(x), sin(x), tan(x)      also acos(x), cosh(x), acosh(x)
, etc.
  log(x)

  +   -   *   /   %% (modulo)   %/% (integer division)   ^
```

- **Boolean functions and operations:**

```
  any(x) (returns true if any of the elemenst in x is true,
otherwise false)
  all(x) (returns true if all the elements in x are true, ot
herwise false)
  & (and)   | (or)   ! (not)
  <   >   <=   >=   ==   !=
```

- **Character functions and operations (http://www.statmethods.net/management/functions.html):**

```
  paste(x, y, etc., sep=s)
  substr(x, start=i, stop=j)
  grep(pattern, x , ignore.case=FALSE, fixed=FALSE)
  sub(pattern, replacement, x, ignore.case =FALSE, fixed=FAL
SE)
  strsplit(x, split)
  toupper(x)
  tolower(x)
```

These operation are **vectorized** way: i.e., they are performed **component wise**.

Vector hidden attributes: names, class, and dim

R vectors have a number of **attributes** (different from the vector elements) that can be set using **special functions**. These attributes condition the way vectors are handled in computation and the way vectors are displayed while printed or evaluated. Attributes can be displayed using the function

```
attributes(x)
```

which returns a list of the attributes set for `x`.

By default, a vector has no attributes set, and the function `attributes` will return NULL upon a vector with no attribute set. Here is a list of the special functions that will set vector attributes:

The function `names` used as followed

```
names(x) = labels
```

will set the names attribute specifying the vector element labels using the strings in the character vector `labels`.

In [15]:
```
%%R

x = c(1,2,3,4,5,6,7,8)
print(x)

names(x) = c('A','B','C','D','E','F','G','H')
NL()
print(x)
```

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


A B C D E F G H
1 2 3 4 5 6 7 8
```

The function `class` used as followed

```
class(x) = className
```

will give the vector `x` the class name stored in the string `className`. We will come back to classes in the `list` section.

In [16]:
```
%%R

print(class(x))

class(x) = 'Cookie'

print(class(x))
```

```
[1] "numeric"
[1] "Cookie"
```

The function `dim` used as followed

```
dim(x) = dimVect
```

will set the dimension attribute of `x` to the integer vector `dimVect`. For instance, setting the dimension attribute to $(2, 3)$ will tell R to organize the elements contained in the vector `x` as an **array** (or here a **matrix**) with 2 rows and 3 colums.

In [17]:
```
%%R
```

```
dim(x) = c(2,4)
print(x)
```

```
     [,1] [,2] [,3] [,4]
[1,]    1    3    5    7
[2,]    2    4    6    8
attr(,"class")
[1] "Cookie"
```

If we set the dimenstion attribute to be $(2, 2, 2)$, this will tell R to interpret x has a 3D array of numbers, the first **axis** having 2 columns, the second and the third axis as well:

In [18]:
```
%%R

dim(x) = c(2,2,2)
print(x)
```

```
, , 1

     [,1] [,2]
[1,]    1    3
[2,]    2    4

, , 2

     [,1] [,2]
[1,]    5    7
[2,]    6    8

attr(,"class")
[1] "Cookie"
```

To create directly multidimentional arrays, without first creating a vector and setting its dimension attribute, one can use the function

```
x = matrix(aVect, ncol=k, nrow=l) (for 2D arrays or matrix)
x = array(aVect, dim=c(k,l,m,o,p))
```

All the vectorized operations we show above will be performed on the underlying "flat" vector, and the result will be interpreted back as an array of the corresponding dimension.

Conclusion: basic types in R are vectors with dimension: hence **arrays**!!!! That's why some people call this vector programming, **array programming**. This is completely similar to what we saw with Numpy arrays!

In [19]:
```
%%R

A = matrix(1:25, ncol=5)
print(A)
```

```
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    6   11   16   21
```

```
[2,]    2    7   12   17   22
[3,]    3    8   13   18   23
[4,]    4    9   14   19   24
[5,]    5   10   15   20   25
```

In [20]:
```
%%R

B = array(1:8, dim=c(2,2,2))
print(B)
```

```
, , 1

     [,1] [,2]
[1,]    1    3
[2,]    2    4

, , 2

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

Recycling or Broadcasting (Python term)

When vectorial operations involve vectors of **different size**, R will **cycle** through the elements of the smallest vector, as to make the two involved vectors of **equal length**. Then the vectorized operation will be performed on the equal length vectors that result:

In [21]:
```
%%R

a = c(2,3); b=c(1,1,1,1)
print(a+b)
print(a*b)
```

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

In [22]:
```
%%R

x = paste(c('X','Y'), c(1,2,3,4), '#', sep='')
print(x)
```

```
[1] "X1#" "Y2#" "X3#" "Y4#"
```

Applying function to vectors

Suppose we have a function that takes a scalar a (i.e. a vector of length 1) and return a vector y of a fixed length n:

$$f : a \longrightarrow \big(f_1(a), \ldots, f_m(a)\big)$$

At times, we'd like to apply the function f to each component of a given vector $x = (x_1, \ldots, x_n)$ and have the m vectors that result stored into a $m \times n$ matrix:

$$\begin{pmatrix} f_1(x_1) & f_1(x_2) & \cdots & f_1(x_n) \\ f_2(x_1) & f_2(x_2) & \cdots & f_2(x_n) \\ \vdots & & & \vdots \\ f_m(x_1) & f_m(x_2) & \cdots & f_m(x_n) \end{pmatrix}$$

Let's try to do that with the following function that, given a scalar, returns a vector of length 3:

In [23]:
```
%%R

f = function(a) return(c(a, a^2, a^3))
```

A non-vectorized way to do that would be the following:

In [24]:
```
%%R

x = 1:5

M = matrix(rep(0,15), ncol=5)

for (j in 1:5){
    for(i in 1:3){
        M[i, j] = f(x[j])[i]
    }
}

print(M)
```

```
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    2    3    4    5
[2,]    1    4    9   16   25
[3,]    1    8   27   64  125
```

The **vectorized way** of doing this is to use one of the **apply family functions (http://www.stat.berkeley.edu/users/spector/s133/R-6a.html)**

```
sapply(x, f)
```

that will do just that:

In [25]:
```
%%R

M = sapply(x, f)
print(M)
```

```
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    2    3    4    5
[2,]    1    4    9   16   25
[3,]    1    8   27   64  125
```

Applying function to matrices

Suppose now that we want to apply a given function f to the rows or columns of a matrix A and that the result be organized into a matrix whose columns contained the outputs of f when applied to the rows or columns of A.

For that one can use the following function from the **apply family**:

```
apply(A, axis, f)
```

where f will be applied sequencially to the **rows** of `A`, if `axis=1`, or to the **columns** of `A` if `axis=2`.

In [44]:
```
%%R

A = matrix(1:9, ncol=3)

row.sum = apply(A, 1, sum)
col.sum = apply(A, 2, sum)

print(A);NL(); print(row.sum); NL(); print(col.sum)
```

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


[1] 12 15 18


[1]  6 15 24
```

Subsetting vectors (and arrays)

You need to know how to retrieve (and modify if needed) **subsets** of vectors (and matrices), using the **bracket operator:**

```
y = x[ind] (copy of a subset of x into the variable y)
x[ind] = a (modification of x itself)
```

where `ind` is a vector of the same size of the vector x, which can be

- **Boolean vectors:** `x[ind]` is a handle on the elements of `x` whose indices correspond to elements in `ind` with value `TRUE`.

- **integer vectors:** `x[ind]` is a handle on the elements of `x` whose indices are in `ind`.
- **character vectors:** `x[ind]` is a handle on the elements of `x` whose labels (or names) are in `ind`.

Remark: Recall the two ways of naming to elements of a vector:

In [27]:
```
%%R

v = c('a'=1, 'b'=2)
print(v)
```

```
a b
1 2
```

In [28]:
```
%%R

w = c(3,4)
names(w) = c('Bob', 'Luc')

print(w)
```

```
Bob Luc
  3   4
```

Remark: If `ind` is an integer vector with non-negative entries, than `x[-ind]` will keep only the elements in `x` whose indices are not in `ind`.

In [29]:
```
%%R

x = 1:10
names(x) = LETTERS[1:10]


print(x)

y = x[-c(2,3,4)]

print(y)

x[-c(2,3,4)] = 444

print(x)
```

```
 A  B  C  D  E  F  G  H  I  J
 1  2  3  4  5  6  7  8  9 10
 A  E  F  G  H  I  J
 1  5  6  7  8  9 10
  A   B   C   D   E   F   G   H   I   J
444   2   3   4 444 444 444 444 444 444
```

Remark: Subsetting for matrices, arrays, and data frames works exaclty the same, except that

Remark: Subsetting for matrices, arrays, and data frames works exactly the same, except that we have now several axis to specify ranges to:

```
A[ind1, ind2]
```

In this case:

```
A[ind1, ] "means" only rows as specified by ind1 BUT ALL COLUMNS
A[, ind2] "means" only col as specified by ind2, BUT ALL ROWS
```

Object-oriented programming in R: List and Classes

In R, **vectors** are the **basic data types** (representing statistical variables or columns in a data table), while **lists** are the **basisc data structures** (representing variable values for sample individuals or data table rows).

Lists are **heterogeneous and labelled containers**. Its elements can have a **name** or **label** and they can be of any type, including lists.

Lists are the building block of (S3) object oriented programming in R.

Similarly to vectors, list can have **special attributes set** through **sepcial functions**:

```
names(x) = labels
class(x) = className
```

These functions works the same as for vectors.

List creation

As example, we create a list whose elements are named according the variable names:

- **first element** (with label 'employee'): a character vector with element
- **second element** (with label 'childrenAges'): a character vector with two elements
- **third element** (with label 'luckyMarix'): a 5×5 matrix

In [30]:
```
%%R

x = list(Employee='Bob Durant', childrenAges=c(1,3), luckyMatrix=matrix(1:
25,ncol=5))
print(x)
```

```
$Employee
[1] "Bob Durant"

$childrenAges
[1] 1 3

$luckyMatrix
     [,1] [,2] [,3] [,4] [,5]
```

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

Subsetting with the single bracket operator

One can retrieve a sublist y from a list x as for vectors using the **single bracket operator:**

```
y = x[ind]
```

where, again, the range `ind` can be

- a integer vector of indices
- a character vector of labels

Remark Since the single bracket operator return a **sublist**, the following subsetting

```
y = x[i]
y = x[label]
```

will return **a list with one element** and not the **element itself**. For instance, in our example

```
x[3]
```

is not a matrix, but a list with one element containing a matrix. This can lead to nasty errors, if not paid due care.

In [31]:
```
%%R

print(x[3])
print(class(x[3]))
```

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

[1] "list"
```

Double bracket operator and dollar sign

To retrieve the actual list elements, one needs to use the **double bracket operator** or the **dollar sign*:

```
y = x$elementLabel
y = x[['elementLabel']]
y = x[[i]]
```

In [32]:
```
%%R
print(x[[3]])
```

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

In [33]:
```
%%R

print(x$Employee)
```

```
[1] "Bob Durant"
```

In [34]:
```
%%R
print(x[['childrenAges']])
```

```
[1] 1 3
```

Applying a function to a list

Let us consider the function

In [35]:
```
%%R

f = function(x) return(x*x)
```

and the list

In [36]:
```
%%R

L = list(number=3, vector=1:10,matrix=matrix(1:9, ncol=3))
print(L)
print(names(L))
```

```
$number
[1] 3

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

$matrix
     [,1] [,2] [,3]
```

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

[1] "number" "vector" "matrix"
```

Suppose we want to apply the function f above to each of the elements of the list L so that the output is store back into a list.

A **non-vectorized way** to do that would be:

In [37]:
```
%%R

Lout = list(NULL, NULL, NULL)
names(Lout) = names(L)


for(name in names(L)){
       Lout[[name]] = f(L[[name]])
}

print(Lout)
```

```
$number
[1] 9

$vector
 [1]   1   4   9  16  25  36  49  64  81 100

$matrix
     [,1] [,2] [,3]
[1,]    1   16   49
[2,]    4   25   64
[3,]    9   36   81
```

The corresponding **vectorized code** makes us the the **apply family function**

```
lapply(L, f)
```

which return a list of whose elements are the output of the function f while iterating on the elements of L.

In [38]:
```
%%R

Lout = lapply(L, f)

print(Lout)
```

```
$number
[1] 9
```

```
$vector
 [1]   1   4   9  16  25  36  49  64  81 100

$matrix
     [,1] [,2] [,3]
[1,]    1   16   49
[2,]    4   25   64
[3,]    9   36   81
```

If one uses the **apply family function**

```
sapply(L, f)
```

the output list will be **simplified** into an array (i.e. vector, matrix, etc.) whenever it makes sense.

In [39]:
```
%%R

L = list(sample(5), sample(5), sample(5))
Lout = sapply(L, f)

print(Lout)
```

```
     [,1] [,2] [,3]
[1,]   16    9   16
[2,]    1   25   25
[3,]    9    4    1
[4,]    4   16    9
[5,]   25    1    4
```

Oject and classes in R using lists

In R, one create a class by simply setting the **class attribute** of a list.

To insure that **instances** of a given class are always intanciated the same way, it is recommended to write a **constructor** for your class that will take care of creating the right number of elements with the rigth names, and setting the class attribute to the correct value.

Access to the attributes is given using the **dollar sign operator**

```
myClass$attribute
```

using the fact that underlying our object is a mere list. This is the analog to the **period operator** in Python.

In [40]:
```
%%R

bankAccount = function(holderName, checking, saving)
{
    object = list(holderName=holderName, checking=checking, saving=saving)
```

```
    class(object) = 'bankAccount'
    return(object)
}
```

In [41]:
```
%%R

BobAccount = bankAccount(holderName='Bob Durant', checking=12, saving=0)

print(class(BobAccount))

saving = BobAccount$saving

print(saving)
```

```
[1] "bankAccount"
[1] 0
```

Methods are just usual function whose names should follow a **naming convention**

```
action.className(object, arg1, arg2, etc.)
```

Then **associated generic functions** are created according to the `action`

```
action = function(object, arg1, arg2, etc.)
{
     UseMethod('action',object)
}
```

Now if `x` is an object of `className`, invoking

```
action(x)
```

will trigger the function

```
UseMethod('action',object)
```

that will retrieve the string `className` from `object` and look up if a function with name

```
action.className
```

exists. If yes, `UseMethod` will invoke this function with the corresponding argument.

In [42]:
```
%%R

display.bankAccount = function(object)
{
    cat('Account holder', object$holderName, 'has', object$checking,
        'dollars in checking and', object$saving, 'dollars in saving.')
}

display = function(object)
{
    UseMethod('display', object)
```

```
}
```

In [43]:
```
%%R

display(BobAccount)
```

```
Account holder Bob Durant has 12 dollars in checking and 0 dollars in savi
ng.
```

In [43]:
```

```

In []:
```

```