Lecture outline:

- The **food** of Mr. R: **VARIABLES**
- The **muscles** of Mr. R: **LOOPING MECHANISMS**
- The **brain** of Mr. R: **BRANCHING MECHANISMS**
- The **hands** of Mr. R: **FUNCTIONS**

Using R in iPython notebooks

To use R in the notebook simply, you need to:

(1) have R installed on your computer

(2) have the Python module `rpy2` installed

(2) invoke the following magic command:

In [41]:
```
%load_ext rmagic
```

```
The rmagic extension is already loaded. To reload it, use:
  %reload_ext rmagic
```

Then type in the **magic command**

```
%%R
```

at the beginning of the cell:

In [42]:
```
%%R

x = c(1,2,3,4)
y = c('a','b','c','d')

xy = data.frame(x,y)
colnames(xy) = c('X', 'Y')
print(xy)
```

```
  X Y
1 1 a
2 2 b
3 3 c
4 4 d
```

The food of Mr. R: VARIABLES

Variable assignment and basic types

As in Python, there are **basic types** in R reprensenting the usual things:

- numbers
- strings
- Boolean

To **retrieve the type** of a variable `x` use the function:

```
typeof(x)
```

(This is the R-equivalent of the `type` function in Python.)

To **print a variable** `x`, use the R command

```
print(x)
```

In [43]:
```
%%R

x = 3
y = 'Hello'
z = TRUE

print(typeof(x))
print(typeof(y))
print(typeof(z))
```

```
[1] "double"
[1] "character"
[1] "logical"
```

Variable assignments have the same syntax as in Python:

```
variable = value
```

Although, R provides also the **arrow syntax** for that same purpose:

```
variable <- value
value -> variable
```

(We will use the Python-like syntax for variable assignments.)

In [44]:
```
%%R
#Python-like variable assignment syntax works in R
x = 'Hello'

#Arrow syntax

y <- 'Bonjour'

'Guten tag' -> z

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

```
[1] "Hello"
[1] "Bonjour"
[1] "Guten tag"
```

Operations on basic types: Boolean, Numbers, and Strings

Boolean: values and operations

In R, the class representing the Boolean type is called `logical`.

As usual, it has only **two possible values**:

```
TRUE and FALSE
```

which can be abbreviated to

```
T and F
```

> **Difference with Python:** In R, the Boolean values are all **upper-cases** (instead of `True` and `False`).

In [45]:
```
%%R
a = TRUE; b = FALSE
c = T; d = F

print(typeof(a)); print(typeof(b))
print(a); print(b)
print(c); print(typeof(d))
```

```
[1] "logical"
[1] "logical"
[1] TRUE
[1] FALSE
[1] TRUE
[1] "double"
```

Here is the R-syntax for the **usual operations on Booleans**:

In [46]:
```
%%R

a = TRUE; b = FALSE

print(a & b) # & = Python's 'and'
print(a | b) # | = Python's 'or'
print(!a)    # ! = Python's 'not'


print( a | (!b & a))
```

```
[1] FALSE
```

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

In [47]:

```
a = 2; b = 3

print(a == b)
print(a <= b)
print(a >= b)
print(a < b)
print(a > b)
```

```
False
True
False
True
False
```

Numbers: The `integer` and `double` types

As Python, R has two types to represent numbers:

- `integer` to represent **natural numbers**
- `double` to represent **real numbers**

In [48]:

```
%%R
a = 3
b = 4.5

print(typeof(a))
print(typeof(b))
```

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

> ## Differences with Python:
>
> - R interprets any number passed to the R interpreter as a real number
> - Python interprets a number as areal number only if it has a **period**.

To differenciate between this types, one should use the **conversion operators**:

```
as.integer(x)
as.double(x)
```

In [49]:

```
%%R

a = as.integer(2)
```

```
print(typeof(a))

b = as.double(a)
print(typeof(b))
```

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

Numbers: operations

In [50]:
```
%%R
a = 2; b = 3; c = 3.4

print(a+b)
print(a*b)
print(a^b)   # The power syntax is different than in Python: a**b
print(a/b)
print(b%%a)  # The modulo syntax is diffenrent than that of Python: b%a
print(c%/%b) # Integer division
```

```
[1] 5
[1] 6
[1] 8
[1] 0.6666667
[1] 1
[1] 1
```

Strings: The `character` type

As in Python, one **creates strings using quotes** (double or singles):

In [51]:
```
%%R
a = 'hello'

b = "hello"

c = "hello! I'am good!"

print(a); print(b); print(c)

print(typeof(a))
```

```
[1] "hello"
[1] "hello"
[1] "hello! I'am good!"
[1] "character"
```

Newlines and **escape characters** in general are used the **same way as in Python**.

> **Differences with Python:**
>
> The R `print(x)` function does **not** print the **escape characters**.

The R function

```
cat(x,y,z,etc.)
```

prints the escape characters.

It behaves very much like the Python (version 2.7) print function:

```
print x,y,z,etc.
```

In [52]:

```
%%R

x = 'One\nTwo\nThree\nFour\n'

print(x)
cat(x)
cat('I need', 3,'pairs of gloves for my', 4, 'hands')
```

```
[1] "One\nTwo\nThree\nFour\n"
One
Two
Three
Four
I need 3 pairs of gloves for my 4 hands
```

Strings: operations

R does not provide the same level of **syntactical conveniences** as Python for **string manipulations**:

$\longrightarrow$ **no special syntax** for string **formatting** in R

$\longrightarrow$ **no addition operator** for string **concatenation** in R

$\longrightarrow$ **no multiplication operator** for string **repetition** in R

$\longrightarrow$ **no bracket operator** for **substring access** in R

All string operations are done through plain old functions

String formatting: `sprintf("... %d ...", x, etc.)`

In Python, one way to format a string is to use the **place holder syntax**:

```
"...%d...%s...etc." % (digit, string, etc.)
```

In R, one uses the function

```
sprintf("...%d...%s...etc.", digit, string, etc.)
```

which **returns the formatted string**.

In [53]:
```
%%R

var  = sprintf("%-10s\t%-10s\t%-10s\n", 'Name', 'Age', 'Weigth')

obs1 = sprintf("%-10s\t%-10d\t%-10.2f\n", 'Benoit', 56, 300)

obs2 = sprintf("%-10s\t%-10d\t%-10.2f\n", 'Claude', 12, 400)

cat(var); cat(obs1); cat(obs2)
```

```
Name          Age           Weigth
Benoit        56            300.00
Claude        12            400.00
```

String concatenation: `paste(x, y, etc., sep=s)`

The R function

```
paste(x, y, etc., sep=s)
```

- **returns** the concatenations of the strings stored in `x`, `y`, etc.
- places the **separator** `s` passed to the argument `sep` in between those strings

In [54]:
```
%%R

x = paste('a', 'b', 'c', 'd', sep='::')
cat(x)
```

```
a::b::c::d
```

String slicing: `substr(x, start=i, stop=j)`

The R function

```
substr(x, start=i, stop=j)
```

- **returns** the **subtring** of the string `x` that
  - **starts** at character position i^{th}
  - **stops** at character position j^{th} (**INCLUDED!!!!**)

In [55]:
```
%%R

x = '123456789'
y = substr(x, 2,6)

cat(y)
```

```
23456
```

23456

> **Differences with Python:**
>
> (1) All **ranges** in R always **start a 1 instead of 0**
>
> (2) All **ranges** in R always **include the upper-bound**
>
> This is valid whenever any type of ranges are around

Example: In the string

```
x = "abcde"
```

- **the first character 'a' has index 1** (and not 0 as in Python)
- `substr(x, 1, 3)` returns 'abc' (while `x[1:3]` returns 'bc' in Python)

A deeper view on R basic data types and data structures

Python basic data types and data structures

Python is a **general purpose** language.

Its **basic types** and **basic data structures** are very standard among such programming languages:

You have

- **First**, the **basic types**: `int`, `float`, `str`, and `bool`

that represent **scalar quantities** (i.e. single elements) of **Booleans**, **numbers**, and **strings**.

- **Second**, the **basic data structures**: `list`, `dict`, `sets`

that represent **vectorial** quantities (i.e. collections) of the **basic types**.

In most general purpose languages:

Basic Data Structures = **structured collections** of **basic types**

Depeding on their structures, these collections can be:

- **homogeneous**: collections of **identical basic types**

$\longrightarrow$ **Numpy arrays** and **Pandas Series**

- **heterogeneous**: collections of **different basic types**

$\longrightarrow$ **Python lists**, **Python Dictionaries**, and **Pandas DataFrames**

- **labelled**: The collection elements carry **names** or **labels**

$\longrightarrow$ **Python Dictionaries**, **Pandas Series**, and **Pandas DataFrames**

- **vectorized**: Functions defined at the element level can be **applied** to the **collection as a whole**

$\longrightarrow$ **Numpy arrays**, **Pandas Series**, and **Pandas Dataframes**

A view on types and structures inspired by data tables

> - **R was created with STATISTICS IN MIND**.
> - **The main object of statistics is that of a DATA TABLE**.
> - **The BASICS DATA TYPES and DATA STRUCTURES in R are reflecting this purpose!**

Actually, R doesn't really have **SEPARATE**

- basic **scalar** data types
- basic **vectorial** data structures

R has only

2 BASICS **VECTORIZED LABELLED DATA STRUCTURES**

- (1) **VECTORS** $\longrightarrow$ corresponding to data table COLUMNS (hence: HOMOGENEOUS)
- (1) **LISTS** $\longrightarrow$ corresponding to data table ROWS (hence: HETEROGENEOUS)

... **AND NO BASIC SCALAR DATA TYPES!!!!**

The "basic scalar types" that we just saw are in reallity ...

... VECTORS WITH ONLY ONE ELEMENT!!!!

There are 3 basic data types in R separated in 3 MODES:

- (1) Numerical Vectors: elements are numbers $\rightarrow$ `numeric mode`
- (2) Logical Vectors: elements are Booleans $\rightarrow$ `logical mode`

- (3) Character Vectors (elements are strings) $\rightarrow$ `character mode`

Remarks:

- In R, the **basic data types** are already **vectorized** and **labelled**!
- In statistics, **mode** = (data table) **column type**
- The **mode** of a variable is **rougher** than its **type**:

R distinguishes between two types in `numeric` mode:

$\longrightarrow$ `int` for **integers**

$\longrightarrow$ `doubles` for **doubles**

In [56]:
```
%%R

print(typeof(3))
print(mode(3))
```

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

In [57]:
```
%%R

a = as.integer(3)

print(typeof(a))
print(mode(a))
```

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

Similar data structures in Python and in R

R vectors $\simeq$ Pandas Series

From a data type perspective:

R $\simeq$ what we would obtain if were allowed to program in Python only with

$\longrightarrow$ quantitative Pandas Series instead of numbers

$\longrightarrow$ logical Pandas Series instead of Booleans

$\longrightarrow$ categorical Pandas Series instead of strings

R vectors: basic manipulations

Vector creation

R vectors are created using the special **concatenate** function

```
c(a=x1, b=x2, c=x3, d=x4, etc)
```

that returns a R vector with

- element values `x1, x2, x3, x4` etc.
- **labelled by the passed argument names**: `a, b, c, d,` etc.

The **element values** in a R vector should all be of **the same type** (i.e. numbers, Booleans, or strings).

Type of R vector = type of its elements

Remark: As for Pandas Series, the **labels** (i.e. **parameter names**) may be omitted.

In [58]:

```
%%R

x = c(a=12, b=34, c=45, d=34)


y = c('Hello', 'Bonjour', 'Guten Tag')


z = c(T, F, T, F, F)


cat("x = \n");print(x); print(typeof(x)); cat('\n\n')
cat("y = \n");print(y); print(typeof(y)); cat('\n\n')
cat("z = \n");print(z); print(typeof(z)); cat('\n\n')
```

```
x =
 a  b  c  d
12 34 45 34
[1] "double"



y =
[1] "Hello"     "Bonjour"   "Guten Tag"
[1] "character"



z =
         Bob Julien  Julie           Bob Julien  Julie    Bob Julien  Jul
ie
     1     62     39     84      1     62     39     84     62     39
84
[1] "double"
```

We may want to create an empty vector, which we will populate later on.

For that, we need to invoke the **vector class constructor** explictely:

```
x = vector(lenght, mode)
```

where

- `lenght` is the **vector length**
- `mode` is the **vector mode**:

  'numeric' 'logical' 'character'

In [59]:
```
%%R

x = vector(length=3, mode='character')

cat('The mode of the vector x is', mode(x),'its length is', length(x),'\n'
)

x[1] = 'elephant'
x[2] = 'raccoon'
x[3] = 'monkey'

print(x)
```

```
The mode of the vector x is character its length is 3
[1] "elephant" "raccoon"  "monkey"
```

Element access

Element indexing and **element retrieval** in R vectorized basic types (i.e vectors and lists) are very similar to that of Python:

On a **vector** `x`, the **BRACKET OPERATOR**

```
x[range]
```

gives us access to the elements specified by the `range`, which can be:

- a **single index** from 1 to `length(x)` (retrieving the corresponding element)
- a **vector of indices** (retrieving the corresponding sublist)
- one can replace indices by **element names** if provided

> ## Differences with Python:
>
> - Indices always start at 1 (instead of 0)
> - The slice notation n:m actually creates the integer vector $(n, n+1, \ldots, m-1, m)$

In [60]:

```
%%R

scores = c(Mark=88, John=24, Lucie=54, Bob=100)

a = scores['Mark']
print(a)

b = scores[1]
print(b)

c = scores[1:3]
print(c)

d = scores[-2]
print(d)
```

```
Mark
  88
Mark
  88
 Mark  John Lucie
   88    24    54
 Mark Lucie   Bob
   88    54   100
```

The muscles of Mr. R: LOOPING MECHANISMS

`for` looping mechanism

As in Python, we have `for` loops.

The main difference is that

code blocks are indicated by curly brackets instead of special indentation

The are also other minor syntactical differences, as you will see below.

The fact that

```
x = n:m
```

creates a integer vector `x` on which a for loop can iterate is very practical.

There is also the function

```
seq(from=a, to=b, by=c)
```

that creates integer vectors, very useful to loop over, and a function

```
rep(x, n)
```

that returns a the vector x repeated n times.

In [61]:
```
%%R

DNA   = rep(c('A','C','T'),4)
RANGE = 1:10
SEQ   = seq(0,100,20)

print(DNA)
print(RANGE)
print(SEQ)
```

```
 [1] "A" "C" "T" "A" "C" "T" "A" "C" "T" "A" "C" "T"
 [1]  1  2  3  4  5  6  7  8  9 10
[1]   0  20  40  60  80 100
```

In [62]:
```
%%R

for(x in 1:10){
    print(x^2)
}
```

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

Since basic types are vectors, one can loop on any 'numeric', 'logical', or 'character' types, even with a single element:

In [63]:
```
%%R
# Try the loop in with different a by uncommenting some lines below
a = 3
#a = c(3, 4, 5)
#a = 'Hello'
#a = c('Hello', 'Bonjour', 'Gutent Tag')

for( x in a) print(x) # We don't need curly braces with just one command
```

```
[1] 3
```

One can also retrieve the vector element names using the function:

```
names(x)
```

Then we can iterate over this names.

In [64]:
```
%%R
scores = c(Mark=88, John=24, Lucie=54, Bob=100)

for (student in names(scores)) cat(student, 'got', scores[student], '\n')
```

```
Mark got 88
John got 24
Lucie got 54
Bob got 100
```

Vectorized loops

When possible

Loops should be implemented the vectorized way!!!

since

All the operations for basic types are vectorized!!!

This works exactly the same way as for Numpy arrays:

Vectorized numerical operations

In [65]:
```
%%R

## VECTORIZED OPERATION ON NUMERIC TYPES

x = c(1,3,2,4)
y = c(4,1,4,2)

print(x+y)
print(x*y)
print(x/y)
print(x^y)
print(x%%y)
print(x%/%y)
```

```
[1] 5 4 6 6
[1] 4 3 8 8
[1] 0.25 3.00 0.50 2.00
[1]  1  3 16 16
[1] 1 0 2 0
[1] 0 3 0 2
```

In [66]:
```
%%R
# OTHER BASIC MATH and STAT OPERATIONS ON THE NUMERICAL TYPE
x = c(2, 1, 54, 21, 56, 7, 1, 4)
```

```
mean(x)
median(x)
sd(x)
quantile(x,0.2)

sum(x)
prod(x)
cumsum(x)
cumprod(x)

sqrt(x)
cos(x)
sin(x)
```

For instance, to normalize a sequence of numbers:

In [67]:
```
%%R
numbers = c(2, 1, 54, 21, 56, 7, 1, 4)
```

one could use a for loops as follows:

In [68]:
```
%%R
m = mean(numbers)
s = sd(numbers)
normalized = vector(length=length(x), mode=mode(x))

for (i in 1:length(numbers)){
    normalized[i] = (numbers[i] - m)/s
}

print(normalized)
```

```
[1] -0.6884905 -0.7308591  1.5146790  0.1165138  1.5994163 -0.4766472 -0.7
308591
[8] -0.6037532
```

The following vectorized version is much preferred:

In [69]:
```
%%R

normalized = (numbers-mean(numbers))/sd(numbers)

print(normalized)
```

```
[1] -0.6884905 -0.7308591  1.5146790  0.1165138  1.5994163 -0.4766472 -0.7
308591
[8] -0.6037532
```

The brain of Mr. R: BRANCHING MECHANISMS

`If` looping mechanisms

They function exactly as in Python, except for

- the curly brace to define the code blocks
- the round parenthesis surrounding the Boolean condition

In [70]:
```
%%R

condition = F

if(condition){
    print('If the boolean variable "condition" is True, this statement is executed.')
} else {
    print('Otherwise, this statement here is executed')
}
```

```
[1] "If the boolean variable \"condition\" is True, this statement is executed."
```

In [71]:
```
%%R
# The else part may be omitted in case there is nothing to do when "condition" is False
condition = T

if(condition){
    print('Great! Condition was True')
}
```

```
[1] "Great! Condition was True"
```

In [72]:
```
%%R

# try with number = 0, 1, 2, 3, 4
# the block of code corresponding to the first matching condition is executed;
# the remaining conditions are then skipped

number = 0.5

if (number < 1){
    cat('number is smaller than', 1)
} else if (number < 2){
    cat('number"is smaller than', 2)
} else if (number < 3){
    cat('number is smaller than', 3)
} else{
    print('number is big!')
}
```

```
number is smaller than 1
```

Vectorized branching mechanisms

When possible:

Branching should be implemented the vectorized way!!!

since:

All the operations for basic types are vectorized!!!

This works exactly the same way as Numpy arrays:

Vectorized boolean operations

In [73]:
```
%%R

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

print(a & b) # & = Python's 'and'
print(a | b) # | = Python's 'or'
print(!a)    # ! = Python's 'not'
print( a | (!b & a))
```

```
          Bob Julien  Julie    Bob Julien  Julie                  Bob Juli
en
  TRUE   TRUE   TRUE   TRUE   TRUE   TRUE   TRUE   TRUE   TRUE   TRUE   TR
UE
 Julie
  TRUE
          Bob Julien  Julie    Bob Julien  Julie                  Bob Juli
en
  TRUE   TRUE   TRUE   TRUE   TRUE   TRUE   TRUE   TRUE   TRUE   TRUE   TR
UE
 Julie
  TRUE
          Bob Julien  Julie    Bob Julien  Julie                  Bob Juli
en
 FALSE  FALSE  FALSE  FALSE  FALSE  FALSE  FALSE  FALSE  FALSE  FALSE  FAL
SE
 Julie
 FALSE
          Bob Julien  Julie    Bob Julien  Julie                  Bob Juli
en
  TRUE   TRUE   TRUE   TRUE   TRUE   TRUE   TRUE   TRUE   TRUE   TRUE   TR
UE
 Julie
  TRUE
```

In [74]:
```
%%R
a = c(1, 2, 3, 4, 5)
b = c(9, 8, 7, 6, 5)
```

```
print(a == b)
print(a <= b)
print(a >= b)
print(a < b)
print(a > b)
```

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

As for Numpy arrays, one can retrieve elements from an R vector by logical indexing:

In [75]:
```
%%R

dat = c(1, 2, 3, 4, 5, 6)
ind = c(T, F, T, T, F, F)

print(dat[ind])
```

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

In [76]:
```
%%R

dat = c(1, 2, 3, 4, 5, 6)
ind = dat < 4

print(ind)
```

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

Putting everything together:

In [77]:
```
%%R

filtered_data = dat[dat < 4]
print(filtered_data)
```

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

Problem: extracting the ouliers from a sequence of data points using

(1) conventional for loop and if statement

(2) vectorized logical indexing

In [78]:
```
%%R
# DATA POINTS
```

```
x = c(1,2, 89, 50, 44, 53, 60, 45, 62, 53, 37, 48, 70, 100, 55)

# FIRST AND THIRD QUARTILES
Q1 = quantile(x, 0.25)
Q3 = quantile(x, 0.75)

# OUTLIER LOWER AND UPPER CUTOFFS
L = Q1 - 1.5*(Q3 - Q1)
U = Q3 + 1.5*(Q3 - Q1)
```

(1) using conventional if and for

In [79]:
```
%%R
upper_outliers = c()
lower_outliers = c()

for(a in x){
    if (a > U) upper_outliers = c(upper_outliers, a)
    if (a < L) lower_outliers = c(lower_outliers, a)
}

cat('Upper Outliers:', upper_outliers, '\n')
cat('Lower Outliers:', lower_outliers, '\n')
```

```
Upper Outliers: 89 100
Lower Outliers: 1 2
```

(2) vectorized version

In [80]:
```
%%R

upper_outliers = x[x > U]
lower_outliers = x[x < L]

cat('Upper Outliers:', upper_outliers, '\n')
cat('Lower Outliers:', lower_outliers, '\n')
```

```
Upper Outliers: 89 100
Lower Outliers: 1 2
```

Useful functions for vectorized computations

Let `x` be a logical vector. The functions

```
any(x)
```

returns `TRUE` if **ONE** of the elements in `x` is `TRUE` and `FALSE` otherwise.

```
all(x)
```

returns `TRUE` if **ALL** the elements in `x` are `TRUE` and `FALSE` otherwise.

In [81]:
```
%%R
x = c(T, F, T)

print(any(x))
print(all(x))
```

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

The function

```
z = ifelse(cond, x, y)
```

- takes a logical vector `cond`
- returns a vector from `x` and `y` as follows:

```
z[i] = x[i] if cond[i] == TRUE

z[i] = y[i] if cond[i] == FALSE
```

Suppose, we have two series of observations, and we want to keep to the highest value for each observation.

We can do that with a classical for/if statement, or use vectorization with `ifelse`:

In [82]:
```
%%R
obs1 = c(12, 34, 55, 21, 54, 22, 78 ,65, 34)
obs2 = c(24, 14, 85, 12, 99, 10, 1 ,9, 100)
```

In [83]:
```
%%R
# CLASSICAL FOR/IF

max_obs = c()
for (i in 1:length(obs1)){
    if (obs1[i] >= obs2[i]) max_obs = c(max_obs, obs1[i])
    else max_obs = c(max_obs, obs2[i])
}
print(max_obs)
```

```
[1]  24  34  85  21  99  22  78  65 100
```

In [84]:
```
%%R
# VECTORIZED VERSION

max_obs = ifelse(obs1 > obs2, obs1, obs2)

cat('\n','obs1:', obs1,'\n','obs2: ', obs2,'\n','maxo: ', max_obs)
```

```
 obs1: 12 34 55 21 54 22 78 65 34
```

```
obs2:  24 14 85 12 99 10 1 9 100
maxo:  24 34 85 21 99 22 78 65 100
```

The hands of Mr. R: FUNCTIONS

R way of defining functions ressembles much **Python inline** function definitions:

In [85]:
```python
f = lambda x, y : x + 2*y

f(3, 2)
```

Out[85]:
```
7
```

The Python keyword `lambda` creates a function whose

- input variables are the variables defined before the colon
- output is the evaluation of the statement after the colon

The function is then stored into the variable (here: `f`), which becomes the function name.

In R,

- The keyword `lambda` is replaced by the keyword `function`
- The **function output** is preceeded by the keyword `return`

> ### Difference with Python:
>
> If the keyword `return` is omitted in a R function, the **function output** will coincide with the **last statement output** in the function body.
>
> In R, returning nothing corresponds to returning the object `NULL` (corresponding to the object `None` in Python).

Other than that, it's very much the same business:

In [86]:
```r
%%R

print_and_return_nothing = function(string='Hello!'){
    print(string)
    return(NULL)
    }

a = print_and_return_nothing()

print(a)
```

```
[1] "Hello!"
```

```
NULL
```

In [87]:

```
%%R

# the code in the previous cell above is the same as the following one tha
t returns None

print_and_return_nothing = function(string='Hello!'){
print(string)
return(NULL)
}

a = print_and_return_nothing()

print(a)
```

```
[1] "Hello!"
NULL
```

In [88]:

```
%%R
dont_print_but_return_something = function(string='Hello!') string

xxx = dont_print_but_return_something()

print(xxx)
```

```
[1] "Hello!"
```