Basic Concepts of the R Language

L. Torgo

ltorgo@dcc.fc.up.pt
Departamento de Ciência de Computadores
Faculdade de Ciências / Universidade do Porto

Oct, 2014

Basic interaction with the R console

- The most common form of interaction with R is through the command line at the console
  - User types a command
  - Presses the ENTER key
  - R "returns" the answer
- It is also possible to store a sequence of commands in a file (typically with the `.R` extension) and then ask R to execute all commands in the file

Basic interaction with the R console (2)

- We may also use the console as a simple calculator

```
1 + 3/5 * 6^2

## [1] 22.6
```


Basic interaction with the R console (3)

- We may also take advantage of the many functions available in R

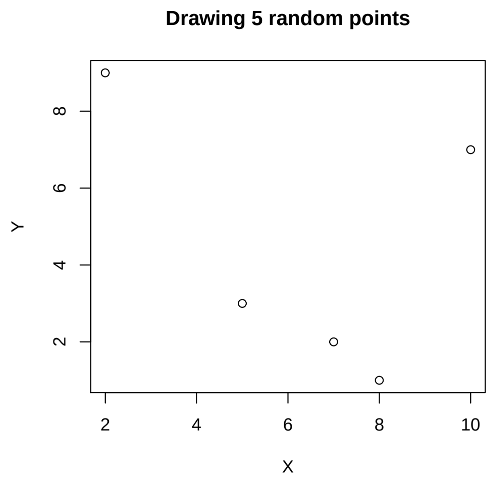
```
rnorm(5, mean = 30, sd = 10)

## [1] 28.100  4.092 29.904 10.611 23.599
```

```
# function composition example
mean(sample(1:1000, 30))

## [1] 530.3
```

Basic interaction with the R console (4)

- We may produce plots

```
plot(sample(1:10, 5), sample(1:10, 5),
    main = "Drawing 5 random points",
    xlab = "X", ylab = "Y")
```



The notion of Variable

- In R, data are stored in variables.
- A variable is a "place" with a **name** used to store information
  - Different types of **objects** (e.g. numbers, text, data tables, graphs, etc.).
- The assignment is the operation that allows us to store an object on a variable
- Later we may use the content stored in a variable using its name.

Basic data types

R objects may store a diverse type of information.

R basic data types

- *Numbers*: e.g. `5`, `6.3`, `10.344`, `-2.3`, `-7`
- *Strings* : e.g. `"hello"`, `"it is sunny"`, `"my name is Ana"`
  Note: one the of the most frequent errors - confusing *names* of variables with *text values* (i.e. strings)! `hello` is the name of a variable, whilst `"hello"` is a string.
- *Logical values*: `TRUE`, `FALSE`
  Note: R is case-sensitive!
  `TRUE` is a logical value; `true` is the name of a variable.

The assignment - 1

- The assignment operator "`<-`" allows to store some content on a variable

```
vat <- 0.2
```

- The above stores the number 0.2 on a variable named `vat`
- Afterwards we may use the value stored on the variable using its name

```
priceVAT <- 240 * (1 + vat)
```

- This new example stores the value 288 ($= 240 \times (1 + 0.2)$) on the variable `priceVAT`
- We may thus put expressions on the right-side of an assignment

The assignement - 2

What goes on in an assignment?

1. **Calculate** the result of the expression on the right-side of the assignment (e.g. a numerical expression, a function call, etc.)
2. **Store** the result of the calculation in the variable indicated on the left side

- In this context, what do you think it is the value of `x` after the following operations?

```
k <- 10
g <- k/2
x <- g * 2
```


Still the variables...

- We may check the value stored in a variable at any time by typing its name followed by hitting the ENTER key

```
x <- 23^3
x

## [1] 12167
```

- The `^` signal is the exponentiation operator
- The odd `[1]` will be explained soon...
- And now a common mistake!

```
x <- true

## Error: object 'true' not found
```

A last note on the assignment operation...

- It is important to be aware that the assignment is destructive
- If we assign some content to a variable and this variable was storing another content, this latter value is “lost”,

```
x <- 23
x

## [1] 23

x <- 4
x

## [1] 4
```


Functions

- In R almost all operations are carried out by functions
- A function is a mathematical notion that maps a set of arguments into a result
  - e.g. the function `sin` applied to 0.2 gives as result 0.1986693
- In terms of notation a function has a name and can have 0 or more arguments that are indicated within parentheses and separated by commas
  - e.g. `xpto(0.2, 0.3)` has the meaning of applying the function with the name `xpto` to the numbers 0.2 and 0.3

Functions (2)

- R uses exactly the same notation for functions.

```
sin(0.2)

## [1] 0.1987

sqrt(45)   # sqrt() calculates the square root

## [1] 6.708
```


Creating new functions

Any time we execute a set of operations frequently it may be wise to create a new function that runs them automatically.

- Suppose we convert two currencies frequently (e.g. Euro-Dollar). We may create a function that given a value in Euros and an exchange rate will return the value in Dollars,

```
euro2dollar <- function(p, tx) p * tx
euro2dollar(3465, 1.36)

## [1] 4712
```

- We may also specify that some of the function parameters have default values

```
euro2dollar <- function(p, tx = 1.34) p * tx
euro2dollar(100)

## [1] 134
```

Function Composition

- An important mathematical notion is that of function composition - $(f \circ g)(x) = f(g(x))$, that means to apply the function f to the result of applying the function g to x
- R is a functional language and we will use function composition extensively as a form of performing several complex operations without having to store every intermediate result

```
x <- 10
y <- sin(sqrt(x))
y

## [1] -0.02068
```


Function Composition - 2

```
x <- 10
y <- sin(sqrt(x))
y

## [1] -0.02068
```

- We could instead do (without function composition):

```
x <- 10
temp <- sqrt(x)
y <- sin(temp)
y

## [1] -0.02068
```

Vectors

- Vectors are a type of R objects that can store sets of values of the same base type
  - e.g. the prices of an article sold in several stores
- Everytime some set of data has something in common and are of the same type, it may make sense to store them as a vector
- A vector is another example of a content that we may store in a R variable

Vectors (2)

- Let us create a vector with the set of prices of a product across 5 different stores

```
prices <- c(32.4, 35.4, 30.2, 35, 31.99)
prices

## [1] 32.40 35.40 30.20 35.00 31.99
```

- Note that on the right side of the assignment we have a call to the function `c()` using as arguments a set of 5 prices
- The function `c()` creates a vector containing the values received as arguments

Vectors (3)

- The function `c()` allows us to associate names to the set members. In the above example we could associate the name of the store with each price,

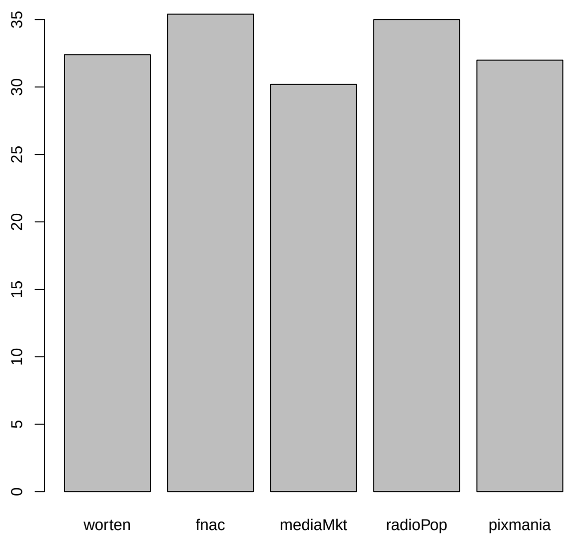
```
prices <- c(worten = 32.4, fnac = 35.4, mediaMkt = 30.2,
    radioPop = 35, pixmania = 31.99)
prices

##   worten     fnac mediaMkt radioPop pixmania
##    32.40    35.40    30.20    35.00    31.99
```

- This makes the vector meaning more clear and will also facilitate the access to the data as we will see.

Vectors (4)

- Besides being more clear, the use of names is also recommended as R will take advantage of these names in several situations.
- An example is in the creation of graphs with the data:

```
barplot(prices)
```



Basic Indexing

- When we have objects containing several values (e.g. vectors) we may want to access some of the values individually.
- That is the main **purpose of indexing**: access a subset of the values stored in a variable
- In mathematics we use indices. For instance, x_3 usually represents the 3rd element in a set of values x.
- In R the idea is similar:

```
prices <- c(worten=32.4,fnac=35.4,
            mediaMkt=30.2,radioPop=35,pixmania=31.99)
prices[3]

## mediaMkt
##     30.2
```

FACULDADE DE CIÊNCIAS UNIVERSIDADE DO PORTO

Basic Indexing (2)

- We may also use the vector position names to facilitate indexing

```
prices <- c(worten=32.4,fnac=35.4,
            mediaMkt=30.2,radioPop=35,pixmania=31.99)
prices["worten"]

## worten
##   32.4
```

- Please note that `worten` appears between quotation marks. This is essencial otherwise we would have an error! Why?
- Because without quotation marks R interprets `worten` as a variable name and tries to use its value. As it does not exists it complains,

```
prices[worten]

## Error:  object 'worten' not found
```

- Read and interpret error messages is one of the key competences we should practice.

FACULDADE DE CIÊNCIAS UNIVERSIDADE DO PORTO

Vectors of indices

- Using vectors as indices we may access more than one vector position at the same time

```
prices <- c(worten=32.4,fnac=35.4,
            mediaMkt=30.2,radioPop=35,pixmania=31.99)
prices[c(2,4)]

##      fnac radioPop
##      35.4     35.0
```

- We are thus accessing positions 2 and 4 of vector `prices`
- The same applies for vectors of names

```
prices[c("worten", "pixmania")]

##   worten pixmania
##    32.40    31.99
```


Vectors of indices (2)

- We may also use logical conditions to "query" the data!

```
prices[prices > 35]

## fnac
## 35.4
```

- The idea is that the result of the query are the values in the vector `prices` for which the logical condition is true
- Logical conditions can be as complex as we want using several logical operators available in R.
  What do you think the following instruction produces as result?

```
prices[prices > mean(prices)]

##      fnac radioPop
##      35.4     35.0
```

- Please note that this another example of function composition!

Vectorization of operations

- The great majority of R functions and operations can be applied to sets of values (e.g vectors)
- Suppose we want to know the prices after VAT in our vector `prices`

```
vat <- 0.23
(1 + vat) * prices

##    worten      fnac mediaMkt radioPop pixmania
##     39.85     43.54    37.15    43.05    39.35
```

- Notice that we have multiplied a number (1.2) by a set of numbers!
- The result is another set of numbers that are the result of the multiplication of each number by 1.2

Vectorization of operations (2)

- Although it does not make a lot of sense, notice this other example of vectorization,

```
sqrt(prices)

##    worten      fnac mediaMkt radioPop pixmania
##     5.692     5.950    5.495    5.916    5.656
```

- By applying the function `sqrt()` to a vector instead of a single number we get as result a vector with the same size, resulting from applying the function to each individual member of the given vector.

Vectorization of operations (3)

- We can do similar things with two sets of numbers
- Suppose you have the prices of the product on the same stores in another city,

```
prices2 <- c(worten=32.5,fnac=34.6,
             mediaMkt=32,radioPop=34.4,pixmania=32.1)
prices2

##   worten     fnac mediaMkt radioPop pixmania
##     32.5     34.6     32.0     34.4     32.1
```

- What are the average prices on each store over the two cities?

```
(prices + prices2)/2

##   worten     fnac mediaMkt radioPop pixmania
##    32.45    35.00    31.10    34.70    32.05
```

- Notice how we have summed two vectors!

Logical conditions involving vectors

- Logical conditions involving vectors are another example of vectorization

```
prices > 35

##   worten     fnac mediaMkt radioPop pixmania
##    FALSE     TRUE    FALSE    FALSE    FALSE
```

- `prices` is a set of 5 numbers. We are comparing these 5 numbers with one number (35). As before the result is a vector with the results of each comparison. Sometimes the condition is true, others it is false.
- Now we can fully understand what is going on on a statement like `prices[prices > 35]`. The result of this indexing expression is to return the positions where the condition is true, i.e. this is a vector of Boolean values as you may confirm above.

Hands On 1

A survey was carried out on several countries to find out the average price of a certain product, with the following resulting data:

Portugal	Spain	Italy	France	Germany	Greece	UK	Finland	Belgium	Austria
10.3	10.6	11.5	12.3	9.9	9.3	11.4	10.9	12.1	9.1

1. What is the adequate data structure to store these values?
2. Create a variable with this data, taking full advantage of R facilities in order to facilitate the access to the information.
3. Obtain another vector with the prices after VAT.
4. Which countries have prices above 10?
5. Which countries have prices above the average?
6. Which countries have prices between 10 and 11 euros?
7. How would you raise the prices by 10%?
8. How would you decrease by 2.5%, the prices of the countries with price above the average?

Hands On 2

Go to the site `http://www.xe.com` and create a vector with the information you obtain there concerning the exchange rate between some currencies. You may use the ones appearing at the opening page.

1. Create a function with 2 arguments: the first is a value in Euros and the second the name of other currency. The function should return the corresponding value in the specified currency.
2. What happens if we make a mistake when specifying the currency name? Try.
3. Try to apply the function to a vector of values provided in the first argument.

Matrices

- As vectors, matrices can be used to store **sets** of values of the same base type that are somehow related
- Contrary to vectors, matrices “spread” the values over two dimensions: rows and collumns
- Let us go back to the prices at the stores in two cities. It would make more sense to store them in a matrix, instead of two vectors
- Columns could correspond to stores and rows to cities

Matrices (2)

- Let us see how to create this matrix

```
prc <- matrix(c(32.40,35.40,30.20, 35.00, 31.99,
                32.50, 34.60, 32.00, 34.40, 32.01),
              nrow=2,ncol=5,byrow=TRUE)
prc

##      [,1] [,2] [,3] [,4]  [,5]
## [1,] 32.4 35.4 30.2 35.0 31.99
## [2,] 32.5 34.6 32.0 34.4 32.01
```

- The function `matrix()` can be used to create matrices
- We have at least to provide the values and the number of columns and rows

Matrices (3)

```
prc <- matrix(c(32.40,35.40,30.20, 35.00, 31.99,
                32.50, 34.60, 32.00, 34.40, 32.01),
              nrow=2,ncol=5,byrow=TRUE)
prc

##      [,1] [,2] [,3] [,4]  [,5]
## [1,] 32.4 35.4 30.2 35.0 31.99
## [2,] 32.5 34.6 32.0 34.4 32.01
```

- The parameter `nrow` indicates which is the number of rows while the parameter `ncol` provides the number of columns
- The parameter setting `byrow=TRUE` indicates that the values should be “spread” by row, instead of the default which is by column

Indexing matrices

- As with vectors but this time with two dimensions

```
prc

##      [,1] [,2] [,3] [,4]  [,5]
## [1,] 32.4 35.4 30.2 35.0 31.99
## [2,] 32.5 34.6 32.0 34.4 32.01

prc[2, 4]

## [1] 34.4
```

- We may also access a single column or row,

```
prc[1, ]

## [1] 32.40 35.40 30.20 35.00 31.99

prc[, 2]

## [1] 35.4 34.6
```

Giving names to Rows and Columns

- We may also give names to the two dimensions of matrices

```
colnames(prc) <- c("worten","fnac","mediaMkt","radioPop","pixmania")
rownames(prc) <- c("porto","lisboa")
prc

##        worten fnac mediaMkt radioPop pixmania
## porto    32.4 35.4     30.2     35.0    31.99
## lisboa   32.5 34.6     32.0     34.4    32.01
```

- The functions `colnames()` and `rownames()` may be used to get or set the names of the respective dimensions of the matrix
- Names can also be used in indexing

```
prc["lisboa", ]

##   worten     fnac mediaMkt radioPop pixmania
##    32.50    34.60    32.00    34.40    32.01

prc["porto", "pixmania"]

## [1] 31.99
```


Arrays

- Arrays are extensions of matrices to more than 2 dimensions
- We can create an array with the function `array()`

```
a <- array(1:18, dim = c(3, 2, 3))
a

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



```
## [2,]   14   17
```

Indexing Arrays

- Similar to matrices and vectors but now with multiple dimensions

```
a[1, 2, 1]

## [1] 4

a[1, , 2]

## [1]  7 10

a[, , 1]

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


Lists

- Lists are ordered collections of other objects, known as the *components*
- List components do not have to be of the same type or size, which turn lists into a highly flexible data structure.
- List can be created as follows:

```
lst <- list(id=12323,name="John Smith",
            grades=c(13.2,12.4,5.6))
lst

## $id
## [1] 12323
##
## $name
## [1] "John Smith"
##
## $grades
## [1] 13.2 12.4  5.6
```

Indexing Lists

- To access the content of a component of a list we may use its name,

```
lst$grades

## [1] 13.2 12.4  5.6
```

- We may access several components at the same time, resulting in a sub-list

```
lst[c("name", "grades")]

## $name
## [1] "John Smith"
##
## $grades
## [1] 13.2 12.4  5.6
```


Indexing Lists (2)

- We may also access the content of the components through their position, similarly to vector,

```
lst[[2]]

## [1] "John Smith"
```

- Please note the double square brakets! Single square brakets have different meaning in the context of lists,

```
lst[2]

## $name
## [1] "John Smith"
```

- As you see the result is a list (i.e. a sub-list of `lst`), while with double brakets the result is the actual content of the component, whilst with double square brackets we got the content of the component (in this case a string)

Data Frames

- Data frames are the R data structure used to store data tables
- As matrices they are bi-dimensional structures
- In a data frame each row represents a case (observation) of some phenomenon (e.g. a client, a product, a store, etc.)
- Each column represents some information that is provided about the entities (e.g. name, address, etc.)
- Contrary to matrices, data frames may store information of different data type

Create Data Frames

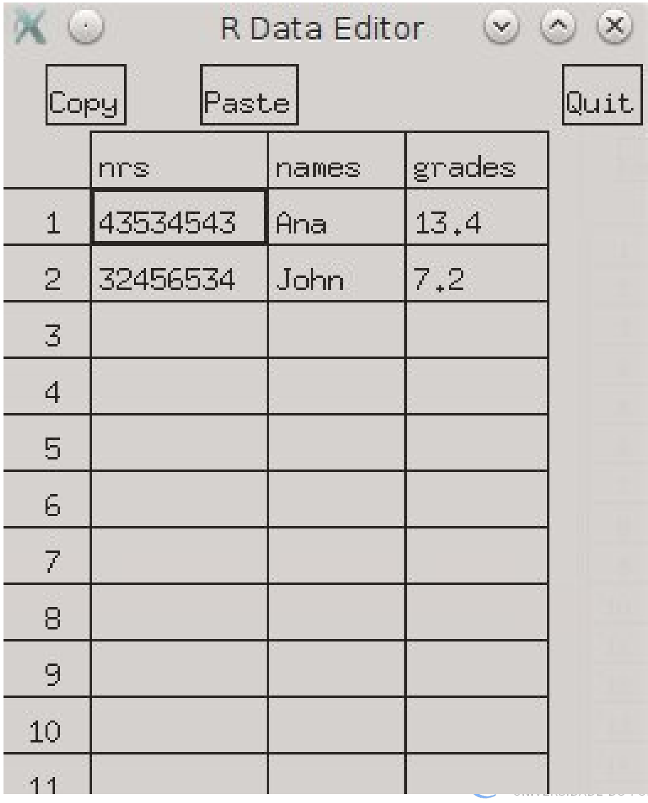
- Usually data sets are already stored in some infrastructure external to R (e.g. other software, a data base, a text file, the Web, etc.)
- Nevertheless, sometimes we may want to introduce the data ourselves
- We can do it in R as follows

```
stud <- data.frame(nrs=c("43534543","32456534"),
                   names=c("Ana","John"),
                   grades=c(13.4,7.2))
stud

##        nrs names grades
## 1 43534543   Ana   13.4
## 2 32456534  John    7.2
```

Create Data Frames (2)

- If we have too many data to introduce it is more practical to add new information using a spreadsheet like editor,

```r
stud <- edit(stud)
```



Querying the data

- Data frames are visualized as a data table

```r
stud

##        nrs names grades
## 1 43534543   Ana   13.4
## 2 32456534  John    7.2
```

- Data can be accessed in a similar way as in matrices

```r
stud[2,3]

## [1] 7.2

stud[1,"names"]

## [1] Ana
## Levels: Ana John
```

Querying the data (cont.)

- You can check sets of rows

```
stud[1:2,]

##        nrs names grades
## 1 43534543   Ana   13.4
## 2 32456534  John    7.2
```

- Or columns

```
stud[,c("names","grades")]

##   names grades
## 1   Ana   13.4
## 2  John    7.2
```

FACULDADE DE CIÊNCIAS UNIVERSIDADE DO PORTO

Querying the data (cont.)

- You may also include logical tests on the row selection

```
stud[stud$grades > 13,"names"]

## [1] Ana
## Levels: Ana John
```

- Or

```
stud[stud$grades <= 9.5, c("names","grades")]

##   names grades
## 2  John    7.2
```

FACULDADE DE CIÊNCIAS UNIVERSIDADE DO PORTO

Querying the data (cont.)

- Function `subset()` can be used to easily query the data set

```
subset(stud,grades > 13,names)

##   names
## 1   Ana

subset(stud,grades <= 9.5,c(nrs,names))

##          nrs names
## 2 32456534  John
```


Hands On Data Frames - Boston Housing

Load in the data set named “Boston” that comes with the package `MASS`. This data set describes the median house price in 506 different regions of Boston. You may load the data doing: `data(Boston,package='MASS')`. This should create a data frame named `Boston`. You may know more about this data set doing `help(Boston,package='MASS')`. With respect to this data answer the following questions:

1. What are the data on the regions with an median price higher than 45?
2. What are the values of `nox` and `tax` for the regions with an average number of rooms (`rm`) above 8?
3. Which regions have an average median price between 10 and 15?
4. What is the average criminality rate (`crim`) for the regions with a number of rooms above 6?

Handling Time Series in R

- R includes several data structures that can be used to store time series
- In this illustration we will use the infra-structured provided in package `xts`
  Note: this is an extra package that must be installed.
- The function `xts()` can be used to create a time series,

```
library(xts)
sp500 <- xts(c(1102.94,1104.49,1115.71,1118.31),
             as.Date(c("2010-02-25","2010-02-26",
                       "2010-03-01","2010-03-02")))
sp500

##              [,1]
## 2010-02-25 1102.94
## 2010-02-26 1104.49
## 2010-03-01 1115.71
## 2010-03-02 1118.31
```


Creating time series

- The function `xts` has 2 arguments: the time series values and the temporal tags of these values
- The second argument must contain dates
- The function `as.Date()` can be used to convert strings into dates
- If we supply a matrix on the first argument we will get a multivariate time series, with each column representing one of the variables

Indexing Time Series

- We may index the objects created by the function `xts()` as follows,

```
sp500[3]

##                [,1]
## 2010-03-01 1115.71
```

- However, it is far more interesting to make "temporal queries",

```
sp500["2010-03-02"]

##                [,1]
## 2010-03-02 1118.31

sp500["2010-03"]

##                [,1]
## 2010-03-01 1115.71
## 2010-03-02 1118.31
```


Indexing Time Series (2)

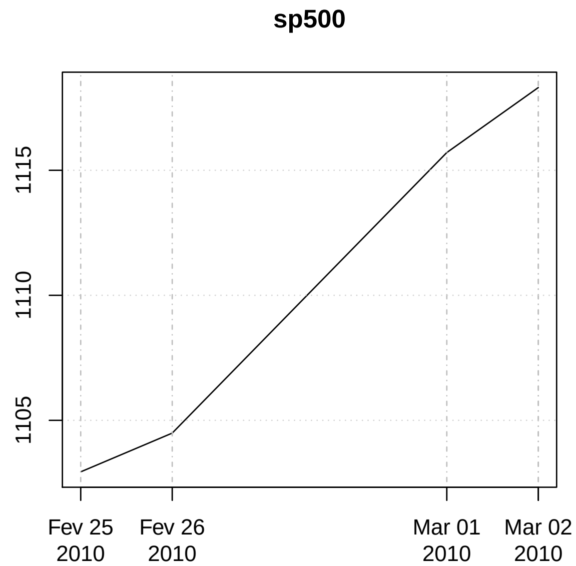
```
sp500["2010-02-26/"]

##                [,1]
## 2010-02-26 1104.49
## 2010-03-01 1115.71
## 2010-03-02 1118.31

sp500["2010-02-26/2010-03-01"]

##                [,1]
## 2010-02-26 1104.49
## 2010-03-01 1115.71
```

- The index is a string that may represent intervals using the symbol / or by omitting part of a date. You may also use `::` instead of `/`.

Temporal Plots

- The `plot()` function can be used to obtain a temporal plot of a time series
- R takes care of selecting the proper axes,

```
plot(sp500)
```



Hands On Time Series

Package **quantmod** (an extra package that you need to install) contains several facilities to handle financial time series. Among them, the function `getMetals` allows you to download the prices of metals from `oanda.com`. Explore the help page of the function to try to understand how it works, and the answer the following:

1. Obtain the prices of gold of the current year
2. Show the prices in January
3. Show the prices from February 10 till March 15
4. Obtain the prices of silver in the last 30 days
   Tip: explore the function `seq.Date()`
5. Plot the prices of silver in the last 7 days
   Tip: explore the function `last()` on package **xts**

The Package **dplyr**

- **dplyr** is a package that greatly facilitates manipulating data in R
- It has several interesting features like:
  - Implements the most basic data manipulation operations
  - Is able to handle several data sources (e.g. standard data frames, data bases, etc.)
  - Very fast

Data sources

- Data frame table
  - A wrapper for a local R data frame
  - Main advantage is printing

```
library(dplyr)
data(iris)
ir <- tbl_df(iris)
ir

## Source: local data frame [150 x 5]
##
##    Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1           5.1         3.5          1.4         0.2  setosa
## 2           4.9         3.0          1.4         0.2  setosa
## 3           4.7         3.2          1.3         0.2  setosa
## 4           4.6         3.1          1.5         0.2  setosa
## 5           5.0         3.6          1.4         0.2  setosa
## 6           5.4         3.9          1.7         0.4  setosa
## 7           4.6         3.4          1.4         0.3  setosa
## 8           5.0         3.4          1.5         0.2  setosa
## 9           4.4         2.9          1.4         0.2  setosa
## 10          4.9         3.1          1.5         0.1  setosa
## ..          ...         ...          ...         ...     ...
```

- Similar functions for other data sources (e.g. databases)

The basic operations

- **filter** - show only a subset of the rows
- **select** - show only a subset of the columns
- **arrange** - reorder the rows
- **mutate** - add new columns
- **summarise** - summarise the values of a column

The structure of the basic operations

- First argument is a data frame table
- Remaining arguments describe what to do with the data
- Return an object of the same type as the first argument (except summarise)
- Never change the object in the first argument

Filtering rows

`filter(data, cond1, cond2, ...)` corresponds to the rows of `data` that satisfy ALL indicated conditions.

```
filter(ir,Sepal.Length > 6,Sepal.Width > 3.5)

## Source: local data frame [3 x 5]
##
##   Sepal.Length Sepal.Width Petal.Length Petal.Width   Species
## 1          7.2         3.6          6.1         2.5 virginica
## 2          7.7         3.8          6.7         2.2 virginica
## 3          7.9         3.8          6.4         2.0 virginica
```

```
filter(ir,Sepal.Length > 7.7 | Sepal.Length < 4.4)

## Source: local data frame [2 x 5]
##
##   Sepal.Length Sepal.Width Petal.Length Petal.Width   Species
## 1          4.3         3.0          1.1         0.1    setosa
## 2          7.9         3.8          6.4         2.0 virginica
```


Ordering rows

`arrange(data, col1, col2, ...)` re-arranges the rows of `data` by ordering them by `col1`, then by `col2`, etc.

```
arrange(ir,Species,Petal.Width)

## Source: local data frame [150 x 5]
##
##    Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1           4.9         3.1          1.5         0.1  setosa
## 2           4.8         3.0          1.4         0.1  setosa
## 3           4.3         3.0          1.1         0.1  setosa
## 4           5.2         4.1          1.5         0.1  setosa
## 5           4.9         3.6          1.4         0.1  setosa
## 6           5.1         3.5          1.4         0.2  setosa
## 7           4.9         3.0          1.4         0.2  setosa
## 8           4.7         3.2          1.3         0.2  setosa
## 9           4.6         3.1          1.5         0.2  setosa
## 10          5.0         3.6          1.4         0.2  setosa
## ..          ...         ...          ...         ...     ...
```

Ordering rows - 2

```
arrange(ir,desc(Sepal.Width),Petal.Length)

## Source: local data frame [150 x 5]
##
##    Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1           5.7         4.4          1.5         0.4  setosa
## 2           5.5         4.2          1.4         0.2  setosa
## 3           5.2         4.1          1.5         0.1  setosa
## 4           5.8         4.0          1.2         0.2  setosa
## 5           5.4         3.9          1.3         0.4  setosa
## 6           5.4         3.9          1.7         0.4  setosa
## 7           5.1         3.8          1.5         0.3  setosa
## 8           5.1         3.8          1.6         0.2  setosa
## 9           5.7         3.8          1.7         0.3  setosa
## 10          5.1         3.8          1.9         0.4  setosa
## ..          ...         ...          ...         ...     ...
```


Selecting columns

`select(data, col1, col2, ...)` shows the values of columns `col1`, `col2`, etc. of `data`

```
select(ir,Sepal.Length,Species)

## Source: local data frame [150 x 2]
##
##    Sepal.Length Species
## 1           5.1  setosa
## 2           4.9  setosa
## 3           4.7  setosa
## 4           4.6  setosa
## 5           5.0  setosa
## 6           5.4  setosa
## 7           4.6  setosa
## 8           5.0  setosa
## 9           4.4  setosa
## 10          4.9  setosa
## ..          ...     ...
```

Selecting columns - 2

```
select(ir,-(Sepal.Length:Sepal.Width))

## Source: local data frame [150 x 3]
##
##    Petal.Length Petal.Width Species
## 1           1.4         0.2  setosa
## 2           1.4         0.2  setosa
## 3           1.3         0.2  setosa
## 4           1.5         0.2  setosa
## 5           1.4         0.2  setosa
## 6           1.7         0.4  setosa
## 7           1.4         0.3  setosa
## 8           1.5         0.2  setosa
## 9           1.4         0.2  setosa
## 10          1.5         0.1  setosa
## ..          ...         ...     ...
```


Selecting columns - 3

```
select(ir,starts_with("Sepal"))

## Source: local data frame [150 x 2]
##
##    Sepal.Length Sepal.Width
## 1           5.1         3.5
## 2           4.9         3.0
## 3           4.7         3.2
## 4           4.6         3.1
## 5           5.0         3.6
## 6           5.4         3.9
## 7           4.6         3.4
## 8           5.0         3.4
## 9           4.4         2.9
## 10          4.9         3.1
## ..          ...         ...
```

Adding new columns

`mutate(data, newcol1, newcol2, ...)` adds the new columns `newcol1`, `newcol2`, etc.

```
mutate(ir,sr=Sepal.Length/Sepal.Width,pr=Petal.Length/Petal.Width,rat=sr/pr)

## Source: local data frame [150 x 8]
##
##    Sepal.Length Sepal.Width Petal.Length Petal.Width Species    sr     pr
## 1           5.1         3.5          1.4         0.2  setosa 1.457  7.000
## 2           4.9         3.0          1.4         0.2  setosa 1.633  7.000
## 3           4.7         3.2          1.3         0.2  setosa 1.469  6.500
## 4           4.6         3.1          1.5         0.2  setosa 1.484  7.500
## 5           5.0         3.6          1.4         0.2  setosa 1.389  7.000
## 6           5.4         3.9          1.7         0.4  setosa 1.385  4.250
## 7           4.6         3.4          1.4         0.3  setosa 1.353  4.667
## 8           5.0         3.4          1.5         0.2  setosa 1.471  7.500
## 9           4.4         2.9          1.4         0.2  setosa 1.517  7.000
## 10          4.9         3.1          1.5         0.1  setosa 1.581 15.000
## ..          ...         ...          ...         ...     ...   ...    ...
## Variables not shown: rat (dbl)
```

NOTE: It does not change the original data!

FACULDADE DE CIÊNCIAS UNIVERSIDADE DO PORTO

Several Operations

```
select(filter(ir,Petal.Width > 2.3),Sepal.Length,Species)

## Source: local data frame [6 x 2]
##
##   Sepal.Length   Species
## 1          6.3 virginica
## 2          7.2 virginica
## 3          5.8 virginica
## 4          6.3 virginica
## 5          6.7 virginica
## 6          6.7 virginica
```

FACULDADE DE CIÊNCIAS UNIVERSIDADE DO PORTO

Several Operations (cont.)

Function composition can become hard to understand...

```
arrange(
    select(
        filter(
            mutate(ir,sr=Sepal.Length/Sepal.Width),
            sr > 1.6),
        Sepal.Length,Species),
    Species,desc(Sepal.Length))

## Source: local data frame [103 x 2]
##
##    Sepal.Length    Species
## 1           5.0     setosa
## 2           4.9     setosa
## 3           4.5     setosa
## 4           7.0 versicolor
## 5           6.9 versicolor
## 6           6.8 versicolor
## 7           6.7 versicolor
## 8           6.7 versicolor
## 9           6.7 versicolor
## 10          6.6 versicolor
## ..          ...        ...
```


The Chaining Operator as Alternative

```
mutate(ir,sr=Sepal.Length/Sepal.Width) %>% filter(sr > 1.6) %>%
    select(Sepal.Length,Species) %>% arrange(Species,desc(Sepal.Length))

## Source: local data frame [103 x 2]
##
##    Sepal.Length    Species
## 1           5.0     setosa
## 2           4.9     setosa
## 3           4.5     setosa
## 4           7.0 versicolor
## 5           6.9 versicolor
## 6           6.8 versicolor
## 7           6.7 versicolor
## 8           6.7 versicolor
## 9           6.7 versicolor
## 10          6.6 versicolor
## ..          ...        ...
```

Summarizing a set of rows

`summarise(data, sumF1, sumF2, ...)` summarises the rows in `data` using the provided functions

```
summarise(ir,avgPL= mean(Petal.Length),varSW = var(Sepal.Width))

## Source: local data frame [1 x 2]
##
##   avgPL varSW
## 1 3.758  0.19
```


Forming sub-groups of rows

`group_by(data, crit1, crit2, ...)` creates groups of rows of `data` according to the indicated criteria, applied one over the other (in case of draws)

```
sps <- group_by(ir,Species)
sps

## Source: local data frame [150 x 5]
## Groups: Species
##
##    Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1           5.1         3.5          1.4         0.2  setosa
## 2           4.9         3.0          1.4         0.2  setosa
## 3           4.7         3.2          1.3         0.2  setosa
## 4           4.6         3.1          1.5         0.2  setosa
## 5           5.0         3.6          1.4         0.2  setosa
## 6           5.4         3.9          1.7         0.4  setosa
## 7           4.6         3.4          1.4         0.3  setosa
## 8           5.0         3.4          1.5         0.2  setosa
## 9           4.4         2.9          1.4         0.2  setosa
## 10          4.9         3.1          1.5         0.1  setosa
## ..          ...         ...          ...         ...     ...
```

Summarization over groups

```
group_by(ir,Species) %>% summarise(mPL=mean(Petal.Length))

## Source: local data frame [3 x 2]
##
##      Species   mPL
## 1     setosa 1.462
## 2 versicolor 4.260
## 3  virginica 5.552
```


Hands On Data Manipulation with dplyr

Package **mlbench** (an extra package that you need to install) contains several data sets (from UCI repository). After loading the data set Zoo answer the following questions;

1. Create a data frame table with the data for easier manipulation
2. What is the average number of legs for the different types of animals?
3. Show the information on the airborne predators
4. For each combination of *hair* and *eggs* count how many animals exist