Lecture outline:

- Data frame basics
- Loading data into data frames
- Basic statistical operations

In [1]:
```
%load_ext rmagic
```

Data frame basics

Constructing and manipulating data frames

The R `data.frame` class

The central class in R is the class

```
data.frame
```

that represents data tables in a similar way than that of Pandas DataFrame.

Remark: The period in the class name `data.frame` has no programming meaning; it is a mere convention among R programmers, used to group related variables, functions, methods, and classes. (We already saw that convention with the naming of class methods.)

As we saw in the last lecture, **classes in R are just "enhanced lists"** with

- a **class name** stored in the **list hidden "class string"**, which is set by using the **`class(x)` function**
- a **class constructor**, which is a **regular function** used to contruct **class instances** (or **objects**)
- a collection of **methods** and **generic functions** acting on the class instances

Since data frames are R classes, one can think of them as enhanced lists

- with the **hidden class string** changed to `'data.frame'`,
- with a **contructor** named `data.frame`,
- with a **collection of methods** and a collections of associated **generic functions**.

The **underlying list** of a data frame contains the **vectors** representing the **data table columns** (i.e. the **statistical variables**).

One passes these vectors to the data frame constructor to populate its underlying list:

In [2]:
```
%%R
```

```
X = c(a=1, b=2, c=3)
Y = c(a=3, b=9, c=1)
Z = c(a='u', b='v', c='w')

df1 = data.frame(X, Y, Z)

print(df1)
```

```
  X Y Z
a 1 3 u
b 2 9 v
c 3 1 w
```

Data frame transposition

One can always transpose a data frame using the **transpose generic function** `t(x)`:

In [12]:
```
%%R

print(t(df1))
```

```
  a   b   c
X "1" "2" "3"
Y "3" "9" "1"
Z "u" "v" "w"
```

Remark: In a data frame, the **values in a given column** should all be of the **same type**. **After transposition**, some column values may end up beeing of different types. In this case, all the **column values will be converted (casted, or coerced) to a matching type** (in the example above: character vectors, since it is possible to cast a number into a string ('1' instead of 1), but the opposite operation does not make sense.

Adding new observations to an existing data frame

Suppose we have two data frames, representing **two samples** of the **same population** for **the same variables**. For instance, the data frame `df1` above and the following new one:

In [13]:
```
%%R

X = c(d=12, e=12, f=63)
Y = c(d=33, e=96, f=0)
Z = c(d='k', e='l', f='m')

df2 = data.frame(X, Y, Z)

print(df2)
```

```
   X  Y Z
d 12 33 k
```

```
e 12 96 l
f 63  0 m
```

We can combine these two sets of observations into a single data frame using the generic **row binding function**

```
rbind(df1, df2)
```

that will return a data frame with the combined sets of observations:

In [14]:
```
%%R

df3 = rbind(df1, df2)

print(df3)
```

```
   X  Y Z
a  1  3 u
b  2  9 v
c  3  1 w
d 12 33 k
e 12 96 l
f 63  0 m
```

Adding new variables to an existing data frame

Suppose we have two data frames, representing the **same sample** but containing values for **different population variables**. For instance, the data frame `df3` above and the following new one:

In [16]:
```
%%R

df4 = data.frame(sex=c(a='M', b='F', c='M', d='F', e='M', f='F'))

print(df4)
```

```
  sex
a   M
b   F
c   M
d   F
e   M
f   F
```

We add additional variables to a data frame by using the generic **column bind function**

```
cbind(df1, df2)
```

that returns a data frame with additional column representing the new variables:

In [17]:
```
%%R

df5 = cbind(df3, df4)

print(df5)
```

```
   X  Y Z sex
a  1  3 u   M
b  2  9 v   F
c  3  1 w   M
d 12 33 k   F
e 12 96 l   M
f 63  0 m   F
```

Bracket operator to retrieve columns

One accesses the **columns of a data frame**, the very same way one accesses the **elements of a list**. So you know how to do that from the previous lecture.

Since the **data frame columns are vectors**, one can do **vectorized arithmetic** with them. This allows us to produce new variables by adding, multipliying, taking square roots, etc. from data frame columns.

Example: a grade data frame (as always)

Formal setting: universe and variables

Let's have a look at our favorite grade example. So we have a population (or universe) of three students in a certain class:

$$\Omega = \Big\{\text{Bob, Aline, Agnes}\Big\}$$

Remark: The element in the set Ω are meant to represent actual students, with all their characteristics/features/variables, and not only the student names.

Since the students have taken three exams in this class (the first and second midterm, denoted by $M1$ and $M2$, and the final exam denoted by F), we have to three **quantitative variables** associated with these exams: that is, we have three mathematical functions:

$$M1,\ M2,\ F:\Omega\to[0,100]$$

For instance, the grade of student $s\in\Omega$ at the final exam is $F(s)$, which is a number between 0 and 100.

Data frame instanciation from vectors

Let's now construct three R vectors representing the values of our variables $M1, M2$, and F:

In [18]:
```
%%R
```

In [18]: %%R

```
M1 = c(Bob=67, Aline=88, Agnes=99)
M2 = c(Bob=82, Aline=91, Agnes=100)
F  = c(Bob=3 , Aline=38, Agnes=97)
```

and construct an R data frame with them:

In [19]:

```
%%R

grades = data.frame(M1, M2, F)
```

R data frames have a method `print.data.frame` method implemented, so we can invoke the **generic function** `print` on data frame instances:

In [20]:

```
%%R

print(grades)
```

```
      M1  M2  F
Bob   67  82  3
Aline 88  91 38
Agnes 99 100 97
```

Maliciously changing the class string of a data frame...

One sees that the `print` function is not the same as the `print` function for R lists: it's much better.

By curiosity, let's change the class string of our data.frame object `grades`, and see if it changes the behavior of the generic function `print`:

In [21]:

```
%%R

class(grades) = 'list'
print(grades)
```

```
$M1
[1] 67 88 99

$M2
[1]  82  91 100

$F
[1]  3 38 97

attr(,"row.names")
[1] "Bob"   "Aline" "Agnes"
```

We see that the generic function `print` now understand that our object is no longer a data frame, but simply a list. As a result, it prints the underlying list the old fashion way.

frame, but simply a list. As a result, it prints the underlying list the old fashion way.

Let's restore the status our fallen data frame `grades`:

In [13]:
```
%%R

class(grades) = 'data.frame'
print(grades)
```

```
      M1  M2  F
Bob   67  82  3
Aline 88  91 38
Agnes 99 100 97
```

Accessing and computing with columns

Accessing data frame columns amount to accessing the elements of the underlying list, which we know how to do. We have two syntax for that:

- the **dollar sign operator** syntax
- the **bracket operator** syntax

The return value of these operators is the corresponding data frame column: i.e. a R vector. Thus we can perform arithmetical vectorized operation, in the very same way as we did with Pandas Series. For instance, let's compute the total class grade with the following crazy and mean formula:

$$TG = \sqrt{M_1 \cos^2(F) + M_2 \sin^2(F)}$$

In [14]:
```
%%R

grades['TG'] = sqrt(cos(grades$F)^2*grades$M1 + sin(grades$F)^2*grades$M2)
```

In [15]:
```
%%R

print(grades)
```

```
      M1  M2  F       TG
Bob   67  82  3 8.203580
Aline 88  91 38 9.394866
Agnes 99 100 97 9.957113
```

Accessing rows and single variable values

R data frames enjoy a similar **double entry bracket syntax** as Numpy arrays.

- The **data frame rows** are indexed **from 1** to the **total number of rows** (i.e. the **number of individuals** in our sample**)

- The **data frame columns** are indexed **from 1** to the **total number of columns** (i.e. the **number of variables** we are considering).

Now, given a data frame `data` the bracket operator

```
data[i, j]
```

will return the value of variable j (i.e. in the j^{th} column) for individual i (i.e. in the i^{th} row).

In [25]:
```
%%R

print(grades[1,2])
```

```
Error in grades[1, 2] : incorrect number of dimensions
```

The notation

```
data[,j]
```

will return the j^{th} column as a **vector**, while

```
data[i,]
```

will return the i^{th} row as a **one row data frame**.

In [35]:
```
%%R

class(grades) = 'data.frame'

a = grades[1, ]

print(class(a)); print(a)
```

```
[1] "data.frame"
    M1 M2 F
Bob 67 82 3
```

In [18]:
```
%%R
a = grades[, 2]
print(class(a)); print(a)
```

```
[1] "numeric"
[1]  82  91 100
```

Loading data into a data frame

As for Pandas data frames, we have functions that allow us to load data tables located either on the **internet** at a given **`url`** or on the **local file system** at a given **path**.

The functions

The functions

```
read.table(address, row.names, col.names, header, sep)
```

```
read.csv(address, row.names, col.names, header)
```

will return a data frame using the **data table** located at the value passed to the argument `address`.

The **column names** will be infered from the first line of the data table if the parameter `header` is set to `TRUE` (the default is `FALSE`). They can also be specified by passing to the argument `col.name` a character vector.

The **row names** can be specified by passing to the argument `row.names` either

- a **character vector**
- the **column index** to use for the row names

The difference between these two functions is that

- `read.table` expects a **blanck character** (i.e. spaces, tabs, etc.) as separator by default, although the **separator** can be specified by setting the argument `sep`

- `read.csv` expects a **comma separated value (csv)** data table

Have a look [here (http://stat.ethz.ch/R-manual/R-devel/library/utils/html/read.table.html)](http://stat.ethz.ch/R-manual/R-devel/library/utils/html/read.table.html) for a description of all the possible arguments that can be passed to these functions, as well as for their default values.

Examples: loading tabular data

Comma separated values

Consider the comma separated file located at:

In [36]:
```
csv_url = 'http://www.stat.berkeley.edu/classes/s133/data/world.txt'
!curl $csv_url 2>/dev/null | head -4
```

```
country,gdp,income,literacy,military
Albania,4500,4937,98.7,56500000
Algeria,5900,6799,69.8,2.48e+09
Angola,1900,2457,66.8,183580000
```

This variable `csv_url` is defined in a Python cell, and thus not accessible by default to R cells. To have access to it in R cells, one can use the `-i` (i for "input") option of the magic command `%%R`:

In [37]:
```
%%R -i csv_url
countries = read.csv(csv_url, header=TRUE, row.names=1)
```

To display the first or last entries of a data frame, one has the generic functions:

```
head(x)
tail(x)
```

In [38]:
```
%%R
print(head(countries))
```

```
            gdp income literacy   military
Albania    4500   4937     98.7 5.6500e+07
Algeria    5900   6799     69.8 2.4800e+09
Angola     1900   2457     66.8 1.8358e+08
Argentina 11200  12468     97.2 4.3000e+09
Armenia    3900   3806     99.4 1.3500e+08
Australia 28900  29893     99.9 1.6650e+10
```

Custom separator

In [39]:
```
custom_url = 'http://www.stat.berkeley.edu/classes/s133/data/movies.txt'
!curl $custom_url 2>/dev/null | head -4
```

```
rank|name|box|date
1|Avatar|$759.563|December 18, 2009
2|Titanic|$600.788|December 19, 1997
3|The Dark Knight|$533.184|July 18, 2008
```

In [40]:
```
%%R -i custom_url

movies = read.table(custom_url, header=TRUE, sep='|')
```

In [41]:
```
%%R

print(head(movies))
```

```
  rank                                name      box              date
1    1                              Avatar $759.563 December 18, 2009
2    2                             Titanic $600.788 December 19, 1997
3    3                     The Dark Knight $533.184     July 18, 2008
4    4 Star Wars: Episode IV - A New Hope $460.998      May 25, 1977
5    5                             Shrek 2 $437.212      May 19, 2004
6    6          E.T. the Extra-Terrestrial $434.975     June 11, 1982
```

Blanck-space separated tables

In [43]:
```
table_url = 'http://www.stat.berkeley.edu/classes/s133/data/pop.txt'
!curl $table_url 2>/dev/null | head -4
```

```
State                   pop2004    pop2003
Alabama                 4530182    4500752

Alaska                   655435     648818
```

```
Arizona                    5743834    5580811
```

In [47]:
```
%%R -i table_url

states = read.table(table_url, header=TRUE, , row.names=1)
print(head(states))
```

```
            pop2004  pop2003
Alabama     4530182  4500752
Alaska       655435   648818
Arizona     5743834  5580811
Arkansas    2752629  2725714
California 35893799 35484453
Colorado    4601403  4550688
```

Basic statistical operations

Summary statistics

One gets **summary statistics** for the variables stored in a data frame, very much the same way as we did with Pandas. The difference is that we use R data frame methods through the generic function:

```
summary(data)
```

The argument `data` can be

- a **whole data frame**, in which case `summary` will return a table containing the summary statistics for all data frame variables
- a **numeric vector**, such as a data frame column

Let's print the summary statistics of our data frame containing the population of all states in 2004 (first column) and in 2003 (second column):

In [48]:
```
%%R
stats_all = summary(states)

summary_2004 = summary(states$pop2004)
```

In [49]:
```
%%R

print(stats_all)
```

```
    pop2004            pop2003
 Min.   :  506529   Min.   :  501242
 1st Qu.: 2375472   1st Qu.: 2323889
 Median : 4522976   Median : 4498543
 Mean   : 6123315   Mean   : 6060413
 3rd Qu.: 6282303   3rd Qu.: 6255088
 Max.   :35893799   Max.   :35484453
```

In [29]:
```
%%R

print(summary_2004)
```

```
   Min.  1st Qu.   Median     Mean  3rd Qu.     Max. 
 506500  2375000  4523000  6123000  6282000 35890000 
```

R basic statistical functions

R has a number of useful numerical built-in functions.

Since the basic type representing numbers in R is vectorized, all the numeric functions take **numeric vectors as input**.

Therefore, all these **built-in functions can take data frame columns** representing **quantitative variables** as their **input**.

Explore by yourself the following list of functions:

Statistics:

```
mean(x, trim=0,na.rm=FALSE)
sd(x)
var(x)
mad(x)
median(x)      m
quantile(x, probs)
range(x)
sum(x)
diff(x, lag=1)
min(x)
max(x)
scale(x, center=TRUE, scale=TRUE)
```

Some other useful math functions:

```
abs(x)
sqrt(x)
ceiling(x)
floor(x)
trunc(x)
round(x, digits=n)
signif(x, digits=n)
cos(x), sin(x), tan(x)
acos(x), cosh(x), acosh(x), etc.
log(x)
log10(x)
exp(x)
```