Introduction to DataFrames in Julia

- a Tutorial -

by Eduard Zapp

Julia Ver. 0.38, July 2015

1. Introduction

Julia

There are many fine statistical software packages out there and Julia is certainly among the finest. Every statistical tool was made to address certain problems in mind. Trouble arises, when these applications were led to their limits. So Julia is the right choice when it comes to push these limits.

Julia can be seen as an universal instrument that covers vast areas of interest. Simple from the start, but yet resourceful enough to crunch big numbers, a glue language that builds upon what other computer languages have found useful. Suitable for statistical calculations by its very nature, it supports a vast area of functions.

While Julia has the reputation of blazing speed, in practical programming you will experience a somewhat sluggish execution. This is due to Just-in-time compilation of the code, where Julia statements are translated into maschine code first, before the real things are performed. This overhead has to be added to execution time.

In opposite to Python, Julia does not care about line space and dentation. It closes blocks with an “end” and starts indexing with “1” instead of “0”. Although it is possible to construct object-like entities, the main unit is the “function” and the function-oriented programming paradigm. There are many similarities between Python and Julia, but Julia is based on totally different principles and should be treated this way.

Let's presume that you have installed Julia already. You have started
- REPL (the programming environment of Julia, running in a terminal) or
- IJulia (the Julia-Version of Ipython, see *http://jupyter.org/* and *https://github.com/JuliaLang/IJulia.jl*).

It is advisable to regard this tutorial in context. Therefore please turn to following texts for basic and further studies :

www.julialang.org
  - The central hub to all of what Julia concerns

http://docs.julialang.org/en/release-0.3/
  - Mind the "v:release-0.3" to download the PDF-version of the manual

https://readthedocs.org/projects/julia/downloads/
  - The Julia-manual in different formats

http://www.quant-econ.net/
  - Source of an excellent reader on Julia (and the QuantEcon package of course)

(By the way, in Ubuntu, Julia is installed most easily via the “Ubuntu Software-Center”.)

Statistics

Statistics is about bending data to reveal what keep them together. Data with inherent relations are collected, squeezed and then transformed by statistical methods to purify and to bring these very relations into the open. Thus the process of empirical work can summarized as follows :

1- Form your theory. Map your concept. State your thesis
2- Collect the necessary and available data
3- Transform, edit and process the data, purify and select, so that they become suitable for your purpose
4- Choose the statistical methods wisely, be aware of their premises, meaning and limitations
5- Interpret the results and translate them into understandable statements
6- Display and express the results
7- Publish and present your findings

DataFrames
DataFrames are constructs to hold data in memory in a structured manner to make them accessible to computation. More or less DataFrames come in handy regarding Step # 3 from above.
And this is what they basically look like. We will use the data below again and again for further demonstrations.

	ID	Person	Wage	Children
1	1	Adam	17.5	3
2	2	Betty	13.7	6
3	3	Chris	19.3	2
4	4	Daisy	15.0	3
5	5	Eduard	7.95	2
6	6	Foo	3.14	0
7	7	Gandalf	21.7	0

A DataFrame is a table of values, organized in rows and columns that reminds a little bit of an Excel-sheet. Datasets in rows, variables in columns. Every variable may contain a different data type. Every column may be addressed directly by the name in its headline (for instance **Children**).
DataFrames are held in RAM, thus giving access to complex data operations in runtime.

In Julia, there are many similar concepts around :

- **Vectors**. Example a = [3 6 2 3 2 0 0] # a row vector
  a = [3, 6, 2, 3, 2, 0, 0] # a column vector

- **Arrays**. Example a = [3 6 2 ; # also called Matrix.
  3 2 0 ; # Matrices may have more than 2 dimensions.
  0 0 0]

- **DataArrays**. The little brother of DataFrames. They are a little bit more robust than Arrays, because they allow to operate with NAs (that stands for Not Available). It is possible to omit data, where the values are missing.

- **Dictionaries**. Key-value-combination that can be accessed by keys.
Example a = [Adam=> 3, Betty => 6, Chris=> 2]

- **SQL**. Example "SELECT Person, Wage FROM df WHERE Wage > 15”
The workhorse in the world of data bases. The Structured Query Language forms questions to search large tables that refer to each other. The data are mostly stored on disk to fulfill tasks of security, availability and multiuser requirements. Being the de-facto standard, it serves the duties appropriately if the data are well organized and well defined.

- **NoSQL** . In case of unstructured data, there are NoSQLs provided. These Databases are not founded on strict tables, but consist of loose collections of data of any kind and combination, like Json. A NonSQL data format, the son of dictionaries; and YAML. Like Json a human readable data format.

- **HDF5**. A “big data” format to handle millions of scientific entries like in meteorology, astronomy or particle physics. There are packages in the Julia repository to deal with these formats.

- **Spreadsheets**. It seems almost a natural idea to use spreadsheets to do statistics, when it is so easy to shuffle together the data. This is fine for small data without any complicated links like travel expense accounting. As soon as it gets complex and loops involved, turn to serious programming. Some economists and their reputations have taken damage by not taking this advice into account.

Now DataFrames combine several attributes of the formats mentioned above. They bundle complex queries, great speed and the embedding into algorithms. DataFrames weld together data with code, what make them so fertile for statistical computing.

Here we have sources for DataFrames:

https://github.com/JuliaStats/DataFrames.jl
- Repository of the DataFrames package

http://dataframesjl.readthedocs.org/en/latest/
- Manual on the DataFrame package

https://en.wikibooks.org/wiki/Introducing_Julia/DataFrames
- A wiki-book on Julia

https://github.com/JuliaStats/DataArrays.jl
- Repository of DataArrays, a minor version of DataFrames

http://pandas.pydata.org/
- Home of Pandas, the origin of DataFrames in Python

http://pandas.pydata.org/pandas-docs/stable/
- The manual of Pandas with many chapters devoted to DataFrames

To begin with, fire up REPL and install once the DataFrames package with

```
> Pkg.add("DataFrames")
```

The “>” means a single command in the terminal window. Do not type it in. It does not belong to the code. The “#” tags the start of the code annotations.

Every script using DataFrames has to start with

```
> using DataFrames
```

Do not forget to load the DataFrame package into memory. Once loaded, it stays in RAM until REPL is shut down.

2. Creating DataFrames

… in several steps

First, setup a DataFrame.

```
> df = DataFrame()
```

It is "DataFrame()" as a function, but "using DataFrames" with an s.
Due to a "tradition" in tutorials about DataFrames, there is always a df. But everything else should be fine too, like “people =DataFrame()”

Now create the columns that state the variables.

```
> df[:ID] = 1:7
> df[:Person] = ["Adam", "Betty", "Chris", "Daisy", "Eduard", "Foo", "Gandalf"]
> df[:Wage] = [17.5, 13.7, 19.3, 15.0, 7.95, 3.14, 21.7]
> df[:Children] = [3,6,2,3,2,0,0]
```

The elements in df[:Person] must be quoted to mark them as strings, because Julia is such a "strongly typed" language.

```
> show(df)  # necessary or not, try it. Changes the look of data.

7x4 DataFrame
| Row | ID | Person     | Wage | Children |
|-----|----|-----------|------|----------|
| 1   | 1  | "Adam"    | 17.5 | 3        |
| 2   | 2  | "Betty"   | 13.7 | 6        |
| 3   | 3  | "Chris"   | 19.3 | 2        |
| 4   | 4  | "Daisy"   | 15.0 | 3        |
| 5   | 5  | "Eduard"  | 7.95 | 2        |
| 6   | 6  | "Foo"     | 3.14 | 0        |
| 7   | 7  | "Gandalf" | 21.7 | 0        |
```

… in one line

```
> df = DataFrame()
```

The lines above can be condensed in one single line.

```
> df = DataFrame(ID   = 1:7,
                 Person =
["Adam","Betty","Chris","Daisy","Eduard","Foo","Gandalf"],
                 Wage = [17.5, 13.7, 19.3, 15.0, 7.95, 3.14, 21.7],
                 Children = [3,6,2,3,2,0,0] )
```

Mind the difference between round & squared brackets.

	ID	Person	Wage	Children
1	1	Adam	17.5	3
2	2	Betty	13.7	6
3	3	Chris	19.3	2
4	4	Daisy	15.0	3
5	5	Eduard	7.95	2
6	6	Foo	3.14	0
7	7	Gandalf	21.7	0

… by reading from csv

The Comma Delimted Values (csv) is the bread-and-butter format to store tables with data sets. It has the transparency, that is needed for a human to keep the overview.

At first we have to create a csv on hard disk. Type in an editor the following :

```
ID,Person,Wage,Children
1,Adam,17.5,3
2,Betty,13.7,6
3,Chris,19.3,2
4,Daisy,15.0,3
5,Eduard,7.95,2
6,Foo,3.14,0
7,Gandalf,21.7,0
```

...and save it as "csv2df_test.csv".

Alternatively, in Excel/OpenOffice export an csv-File.

Yes, type everything in to internalize it. In some cases, Copy-and-paste may transfer layout characters that disturbs the data. As with everything : Hack the code in manually to understand, and to make mistakes that will lead you to a better understanding !

Beware : An unseen "empty space" after a number may change it to a string !

```
> a = readcsv("/home/user/Desktop/csv2df_test.csv")
```

This is a Julia-command that loads the csv into an Array.
The import-function automatically recognizes strings, but you better check everything nevertheless afterwards. Do not forget to specify the correct path of your system.

```
8x4 Array{Any,2}:
  "ID"   " Person"       " Wage"   " Children"
 1.0    " Adam"     17.5       3.0
 2.0    " Betty"    13.7       6.0
 3.0    " Chris"    19.3       2.0
 4.0    " Daisy"    15.0       3.0
 5.0    " Eduard"    7.95      2.0
 6.0    " Foo"       3.14      0.0
 7.0    " Gandalf"  21.7       0.0
```

The next expression does the same as above, but specifies the separators.

```
> b = readdlm("/home/user/Desktop/csv2df_test.csv", ',')

8x4 Array{Any,2}:
  "ID"   " Person"       " Wage"   " Children"
 1.0    " Adam"     17.5       3.0
 2.0    " Betty"    13.7       6.0
 3.0    " Chris"    19.3       2.0
 4.0    " Daisy"    15.0       3.0
 5.0    " Eduard"    7.95      2.0
 6.0    " Foo"       3.14      0.0
 7.0    " Gandalf"  21.7       0.0
```

Here we have a DataFrame-command. Don't forget "using Dataframes" in the head of your code.

```
> c = readtable("/home/user/Desktop/csv2df_test.csv")
> show(c)       # maybe skipped.

7x4 DataFrame

| Row | ID | Person      | Wage | Children |
|-----|----|-----------|------|----------|
| 1   | 1  | "Adam"    | 17.5 | 3        |
| 2   | 2  | "Betty"   | 13.7 | 6        |
| 3   | 3  | "Chris"   | 19.3 | 2        |
| 4   | 4  | "Daisy"   | 15.0 | 3        |
| 5   | 5  | "Eduard"  | 7.95 | 2        |
| 6   | 6  | "Foo"     | 3.14 | 0        |
| 7   | 7  | "Gandalf" | 21.7 | 0        |
```

Having composed a DataFrame in memory, you write it to disk by

```
> writetable("/home/user/Desktop/csv2df_test.csv_NEU", c)
```

It stores DataFrame c (not Array c), into DataFrame "csv2df_test_NEU.csv" at “/home/user/Desktop/” or where ever you want it, in this case linux-style.

3. Transformation

Transformation means changing the structure and content of the data table in general.
By “Selection”, only parts of it are separated.
In a “query” you get values by some defined conditions.

Joining Tables
First create a new DataFrame-vector for joining.

```
> old = DataFrame(Person = df[:Person], Age = [23,44,37,66,27,23,55])
```

Now join. This is just a simple operation, there are many more options.

```
> df_plus = join(df, old, on = :Person)
```

	ID	Person	Wage	Children	Age
1	1	Adam	17.5	3	23
2	2	Betty	13.7	6	44
3	3	Chris	19.3	2	37
4	4	Daisy	15.0	3	66
5	5	Eduard	7.95	2	27
6	6	Foo	3.14	0	23
7	7	Gandalf	21.7	0	55

Arrays
Create a simple vector.

```
> d = [1:10]   # numbers from 1 to 10
```

Let us double every single value to introduce an important concept : The Point.
"." refers to rowwise operation ! This tiny dot may make or break your code, so be careful.

```
> d = d .* 2

10-element Array{Int64,1}:  # To remember : This is an Array, not a DataFrame.
  2
  4
  6
  8
 10
 12
 14
 16
 18
 20
```

Expand the function.

```
 > d = (d .* 2) .+ 100   # Again, mind the Point.
```

```
10-element Array{Int64,1}:
 104
 108
 112
 116
 120
 124
 128
 132
 136
 140
```

This works with functions too.

```
> d = d .-  mean(d)
```

```
10-element Array{Float64,1}:
 -18.0
 -14.0
 -10.0
  -6.0
  -2.0
   2.0
   6.0
  10.0
  14.0
  18.0
```

Now for the **DataFrames.**

The point arithmetic works in Arrays as well as in DataArrays and DataFrames.

```
> x = df[:Wage] .- mean(df[:Wage])
```

x denotes a DataArray, that holds deviation from the mean of wages in absolute terms.

```
7-element DataArray{Float64,1}:
   3.45857
  -0.341429
   5.25857
   0.958571
  -6.09143
 -10.9014
   7.65857
```

```
> x1 = x ./ mean(df[:Wage])     # x1 takes x to divide it by the mean to get the
                               #  deviation in %
7-element DataArray{Float64,1}:
  0.246312
 -0.0243158
  0.374504
  0.0682674
 -0.433818
 -0.776376
  0.545427
```

We change the DataArray into a DataFrame, if necessary.

This gives us the possibility to address the column by name.

```
> x2 = DataFrame(Wage_normalized = x)
```

	Wage_normalized
1	3.458571428571428
2	-0.3414285714285725
3	5.258571428571429
4	0.9585714285714282
5	-6.091428571428572
6	-10.901428571428571
7	7.6585714285714275

Now turn absolute into relative deviation from mean.

```
> x1 = (df[:Wage] .- mean(df[:Wage])) ./ mean(df[:Wage])
```

To be on the safe side, use brackets to set priorities in validation.

```
7-element DataArray{Float64,1}:
  0.246312
 -0.0243158
  0.374504
  0.0682674
 -0.433818
 -0.776376
  0.545427

> x1 = round(((df[:Wage] .- mean(df[:Wage])) ./ mean(df[:Wage]).*100),2)

# Now rounded in %.

# mean() could be precalculated once to save computer time.
```

```
> for i in x1         # go through x1

    println( i," %")

end

24.63 %
-2.43 %
37.45 %
6.83 %
-43.38 %
-77.64 %
54.54 %
```

Be aware of what data belongs to what collection type !

Julia may be tolerant in most cases, but DataFrame-routines do not work on Arrays, unless there is a similar procedure.

Learn the data type by “typeof()”.

```
> println(typeof(x3))

DataFrame

> println(typeof(x1))

DataArray{Float64,1}
```

4. Selection

From a DataFrame, select the desired values by naming the location in the table.
This reads : From DataFrame df, take [first row, and show me all columns/variables]

```
> df[1,:]
```

	ID	Person	Wage	Children
1	1	Adam	17.5	3

Now show row 1, column 3.

```
> df[1,3]

17.5
```

Pick it by the variable “Person” of the column

```
> df[:Person]

7-element DataArray{ASCIIString,1}:
 "Adam"
 "Betty"
 "Chris"
 "Daisy"
 "Eduard"
 "Foo"
 "Gandalf"
```

Now df[row 2 to 3,[column 2 and column 3]]

```
> df[2:3 , [:Person,:Children]]
```

	Person	Children
1	Betty	6
2	Chris	2

Show from df [(row 1 to 3, and 6),colum ID, Children and Person]

```
> df[[[1:3,6]] , [:ID,:Children,:Person]]
```

	ID	Children	Person
1	1	3	Adam
2	2	6	Betty
3	3	2	Chris
4	6	0	Foo

For a shortcut, “head()” shows the first six rows.

```
> head(df)
```

	ID	Person	Wage	Children
1	1	Adam	17.5	3
2	2	Betty	13.7	6
3	3	Chris	19.3	2
4	4	Daisy	15.0	3
5	5	Eduard	7.95	2
6	6	Foo	3.14	0

```
> tail(df) # The last six rows
```

	ID	Person	Wage	Children
1	2	Betty	13.7	6
2	3	Chris	19.3	2
3	4	Daisy	15.0	3
4	5	Eduard	7.95	2
5	6	Foo	3.14	0
6	7	Gandalf	21.7	0

5. Query Data

.. single condition

Again, this is the DataFrame, we like to query.

```
> df
```

	ID	Person	Wage	Children
1	1	Adam	17.5	3
2	2	Betty	13.7	6
3	3	Chris	19.3	2
4	4	Daisy	15.0	3
5	5	Eduard	7.95	2
6	6	Foo	3.14	0
7	7	Gandalf	21.7	0

Who has to feed *no* kids at all ?

```
> df[ df[:Children] .== 0, :]

# Mind the difference between comparititive "==" and declarative "=" !
```

	ID	Person	Wage	Children
1	6	Foo	3.14	0
2	7	Gandalf	21.7	0

Show all values above 15 over all columns.

```
> df[ df[:Wage] .> 15 ,: ]
```

Daisy is excluded, because she earns *exactly* 15, no more.

	ID	Person	Wage	Children
1	1	Adam	17.5	3
2	3	Chris	19.3	2
3	7	Gandalf	21.7	0

Show all entries of "Wage" equal *or* less than 15 over all columns.

```
> df[df[:Wage] .<= 15 , : ]
```

Now you get a DataFrame with Daisy.

	ID	Person	Wage	Children
1	2	Betty	13.7	6
2	4	Daisy	15.0	3
3	5	Eduard	7.95	2
4	6	Foo	3.14	0

Give me just the names of those who earn *more* than 15.

```
> df[ df[:Wage] .> 15 , :Person ]
```

Again you get an one-dimensional DataArray = Vector

```
3-element DataArray{ASCIIString,1}:
 "Adam"
 "Chris"
 "Gandalf"
```

Show all entries of "Wage" above its average over all columns
Yes, we had that before.

```
> df[ df[:Wage] .> mean(df[:Wage]), : ]
```

	ID	Person	Wage	Children
1	1	Adam	17.5	3
2	3	Chris	19.3	2
3	4	Daisy	15.0	3
4	7	Gandalf	21.7	0

Let a be columns ID & Wage, with 2 or less Children.

```
> a = df[df[:Children] .<= 2, [:ID , :Wage]]
```

	ID	Wage
1	3	19.3
2	5	7.95
3	6	3.14
4	7	21.7

Names of people with Wage *between* 10 and 20.

```
> df[ 10 .< df[:Wage] .< 20, :Person ]
```

```
4-element DataArray{ASCIIString,1}:
 "Adam"
 "Betty"
 "Chris"
 "Daisy"
```

Compound conditions

Who has low income (<15) *and* many kids(>=3) ?

This time we try a combined query.

First we create a subset of many children :

```
> d = df[df[:Children] .>= 3, : ]
```

	ID	Person	Wage	Children
1	1	Adam	17.5	3
2	2	Betty	13.7	6
3	4	Daisy	15.0	3

Second from these we select a subset of low income :

```
> d[d[:Wage] .< 15,:]
```

	ID	Person	Wage	Children
1	2	Betty	13.7	6

Encore, who has many kids (> 3) *and* low income (< 15) ?

But this time as an one-liner :

```
> df[(df[:Children] .>= 3) & (df[:Wage] .< 15), :]
```

	ID	Person	Wage	Children
1	2	Betty	13.7	6

Who has many kids(> 3) *or* low income (< 15) ?

It is “or” this time, not “and”.

```
> df[(df[:Children] .> 3) | (df[:Wage] .< 15), : ]
```

	ID	Person	Wage	Children
1	2	Betty	13.7	6
2	5	Eduard	7.95	2
3	6	Foo	3.14	0

Who has *not* 3 kids ?

```
> df[(df[:Children] .!= 3), : ]
```

	ID	Person	Wage	Children
1	2	Betty	13.7	6
2	3	Chris	19.3	2
3	5	Eduard	7.95	2
4	6	Foo	3.14	0
5	7	Gandalf	21.7	0

Here are some (not all) logical relations :

== ...equals...
& ...and...
!= ...not...
| ...or...

6. Dates

Introduction

The Package "Dates" make dates accessible to computation. It turns numbers or strings to a format according to calender and clock. So "dates" or "Dates" are points in time, but "Date()" refers to the special type. One of the charms of with DataFrames lies in the possibilities to work with dates.

In Julia 0.3 the Dates-Package must once be installed separately :

```
> Pkg.add("Dates")
```

In later versions of Julia the routines are said to be included.

Sources :
http://docs.julialang.org/en/latest/manual/dates/
http://docs.julialang.org/en/latest/stdlib/dates/

https://github.com/JuliaLang/julia/blob/master/test/dates/io.jl
https://github.com/quinnj/Dates.jl

Initialize the Package (until Version 0.4 will be published) by :

```
> using Dates
```

Format

The conversion of numbers into dates takes the form Date(year, month, day).

So the 1th January 2000 looks like this :

```
> Date(2000,1,1)

2000-01-01
```

Remember, the Package is "Dates", but the routine spells "Date", singular. A single character in the wrong place stops the correct evaluation.

This creates a new Date-data type:

```
> a = Date(2000,1,1)
> typeof(a)

Date (constructor with 24 methods)
```

As we had seen, a common data format in Dates is “year-month-day”, like 2000-01-01.
So, let's try :

```
> Date(2000-01-01)

1998-01-01
```

What went wrong ?

This is equal to Date(2000 minus 1 minus 1). And (2000-2) is 1998.

To make it digestible by Date(), input with quotes as strings.

This is the correct syntax :

```
> Date("2000-01-01")

2000-01-01
```

Dates without quotation marks are Date(). Dates with quotation marks are strings !

To speed things up, declare DateFormat() and proceed it to Date().

```
> dformat = Dates.DateFormat("y-m-d")

> dt = Date("2000-01-01",dformat)

2000-01-01
```

Though year-month-day seems the natural format of Date(), dates may take another form when declared. We proceed as we have learned above.

```
> dt = Date("01.01.2000")

ArgumentError("Delimiter mismatch. Couldn't find first delimiter, \"-\", in date string")
while loading In[63], in expression starting on line 1

 in parse at /home/user/.julia/v0.3/Dates/src/dates/io.jl:116
 in Date at /home/user/.julia/v0.3/Dates/src/dates/io.jl:165
```

Does not work. So what to do ?

```
> dformat = Dates.DateFormat("d.m.y")

> dt = Date("01.01.2000",dformat)

2000-01-01
```

There we are.

Duration

Dates() gives us the option to work with periods and durations.

First, we compose a couple of days in a sequence.

```
> same_days = [Date(2000,1,1):Date(2000,1,12)]

12-element Array{Date,1}:
 2000-01-01
 2000-01-02
 2000-01-03
 2000-01-04
 2000-01-05
 2000-01-06
 2000-01-07
 2000-01-08
 2000-01-09
 2000-01-10
 2000-01-11
 2000-01-12
```

How many days are between January 1th 2001 and 2000 ? Obviously 365, the days in a year.

```
Date(2001,1,1) - Date(2000,1,1)

366 days
```

Oops. How come ? A regular year has 365 days. Because 2000 was a leap year, it had actually 366.

So Dates goes beyond simply counting.

Beware and check the results. In every language. Always.

```
> Date("2001-01-01") - Date("2000-01-01")

366 days
```

This is fine too, just in another notation.

What time is it ?

```
> println( now() )         # This is in DateTime() format.

2015-05-17T18:12:17

> println( Date(now()) )  # This is in Date() format.

2015-05-17                # You have your own real-time.
```

Conditions

Let us add some dates into a DataFrame and bring together, what we have learned so far.

```
> b_day = Date(["1965-03-04","1996-12-11","1987-01-27","1958-05-16","1973-08-
08","1983-03-08","1971-09-28"])

> bday = DataFrame(Person = df[:Person], Birthday = b_day)

> df_dates = join(df,bday, on = :Person)
```

	ID	Person	Wage	Children	Birthday
1	1	Adam	17.5	3	1965-03-04
2	2	Betty	13.7	6	1996-12-11
3	3	Chris	19.3	2	1987-01-27
4	4	Daisy	15.0	3	1958-05-16
5	5	Eduard	7.95	2	1973-08-08
6	6	Foo	3.14	0	1983-03-08
7	7	Gandalf	21.7	0	1971-09-28

```
> println( typeof(df_dates[1,5]) )
Date

> println(df_dates[1,5])

1965-03-04
```

How many days are between March 3th 1965 and today ?

```
> Date(now()) - Date("1965-3-4")   # Clearly, my now() is not identical with yours.

18336 days
```

```
> println("This is ",round((int(Date(now()) - df_dates[1,5])/356)-1, 2), " years.")

This is 51.51 years.
```

Print out a list of birthdays.

```
> df_dates[:Birthday]

7-element DataArray{Date,1}:
 1965-03-04
 1996-12-11
 1987-01-27
 1958-05-16
 1973-08-08
 1983-03-08
 1971-09-28
```

Show all columns, of people who were born *after* the 60's.

```
> df_dates[ df_dates[:Birthday] .> Date(1970,01,01) ,: ]
```

	ID	Person	Wage	Children	Birthday
1	2	Betty	13.7	6	1996-12-11
2	3	Chris	19.3	2	1987-01-27
3	5	Eduard	7.95	2	1973-08-08
4	6	Foo	3.14	0	1983-03-08
5	7	Gandalf	21.7	0	1971-09-28

Show *just* names and wages of people who were born *after* the 60's.

```
> df_dates[df_dates[:Birthday] .> Date(1970,01,01) ,[:Person, :Wage]]
```

	Person	Wage
1	Betty	13.7
2	Chris	19.3
3	Eduard	7.95
4	Foo	3.14
5	Gandalf	21.7

Which of the people were born *in* the 80s ?

```
> df_dates[ Date(1980,1,1) .<= df_dates[:Birthday] .< Date(1990,1,1), : ]
```

	ID	Person	Wage	Children	Birthday
1	3	Chris	19.3	2	1987-01-27
2	6	Foo	3.14	0	1983-03-08

By the way, how old is Chris ?

First, pick the right date.

```
> df_dates[df_dates[:Person].== "Chris",:Birthday]

1-element DataArray{Date,1}:
 1987-01-27
```

You may do it even the other way round : df_dates["Chris" .== df_dates[:Person],:Birthday]

Next, address the first and only element in the resulting DataArray, then change it into a string.

```
> string(df_dates[df_dates[:Person].== "Chris",:Birthday][1])

"1987-01-27"
```

Then insert the string into Date().

```
> Date(string(df_dates[df_dates[:Person].== "Chris",:Birthday][1]))

1987-01-27
```

Substract it from now(), divide it by days in a year, print. Ready.

```
> println("Chris is ",round((int(Date(now()) -
Date(string(df_dates[df_dates[:Person].== "Chris",:Birthday][1])))/356), 0), "
years old.")

Chris is 29.0 years old.
```

Next we try a nested condition.

Who of the people, born in the 80's, has children ?

First create a subset 1 according to the first condition, aka the bunch of 80's-people.

Mind the point-notation.

```
> subset1 = df_dates[ Date(1980,1,1) .<= df_dates[:Birthday] .< Date(1990,1,1),: ]
```

	ID	Person	Wage	Children	Birthday
1	3	Chris	19.3	2	1987-01-27
2	6	Foo	3.14	0	1983-03-08

Now query subset 1.

```
> subset2 = subset1[ subset1[:Children] .> 1, : ]
```

It is Chris, again.

	ID	Person	Wage	Children	Birthday
1	3	Chris	19.3	2	1987-01-27

Oh, we have left out so many things. Julia is in development, yet it seems remarkably versatile and handy for numerical calculations. And in the end this is what computing is about.

- End of Tutorial -