SciencesPo Computational Economics
Spring 2017

Florian Oswald

April 27, 2017

1 Computing Basics

- It is important that we understand some basics about computers.
- Even though software (and computers) always get more and more sophisticated, there is still a considerable margin for "human error". This doesn't mean necessarily that there is something wrong, but certain ways of doing things may have severe performance implications.
- Whatever else happens, *you* write the code, and one way of writing code is different from another.

- In this session, we will look at some very basic computer structure, and see some common pitfalls in numerical analysis.
- We will take advantage to introduce the julia computing language.

1. Step Number 1: install julia!
2. Step Number 2: Why Julia?

1.1 Julia? Why Julia?

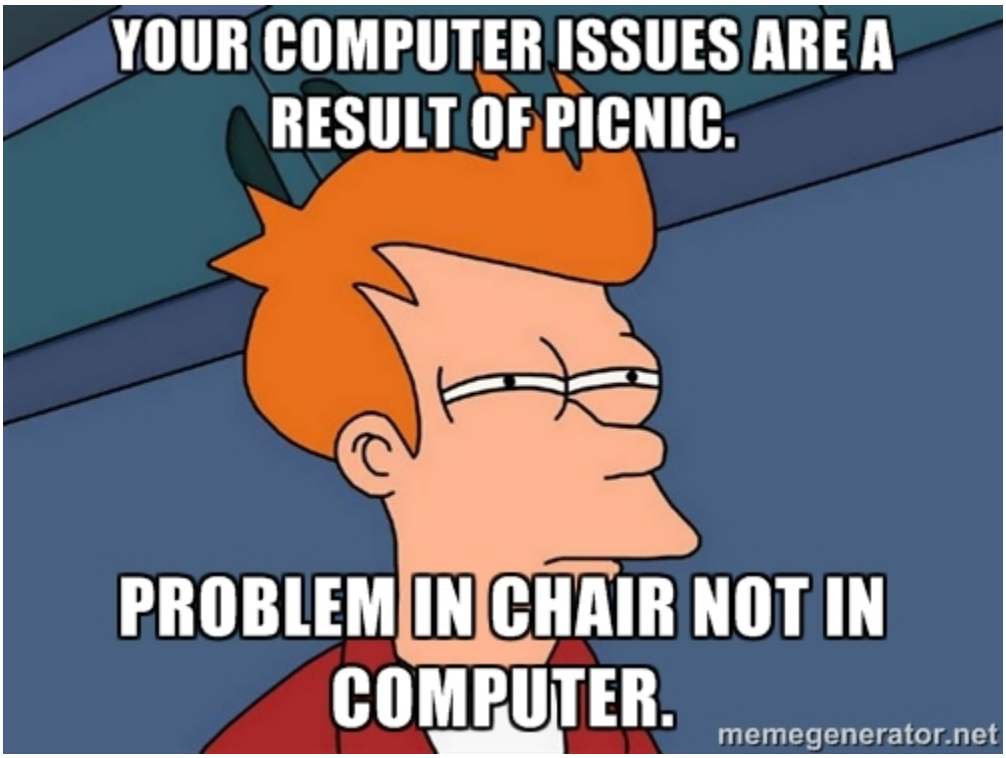
- The *best* software doesn't exist. All of the following statements depend on:

1. The problem at hand.
   - You are fine with Stata if you need to run a probit.
   - Languages have different comparative advantages with regards to different tasks.
2. Preferences of the analyst. Some people just *love* their software.

- That said, there are some general themes we should keep in mind when choosing a software.
- Stephen Johnson at MIT has a good pitch.

1.2 High versus Low Level Languages

- High-level languages for technical computing: Matlab, Python, R, ...
  - you get going immediately
  - very important for exploratory coding or data analysis
  - You don't want to worry about type declarations and compilers at the exploratory stage
- High-level languages are slow.



test

  - Traditional Solutions to this: Passing the high-speed threshold.
  - Using `Rcpp` or `Cython` etc is a bit like Stargate. You loose control the moment you pass the barrier to `C++` for a little bit. (Even though those are great solutions.) If the `C++` part of your code becomes large, testing this code becomes increasingly difficult.
  - You end up spending your time coding `C++`. But that has it's own drawbacks.

1.3 Julia is Fast

- Julia is fast.
  - But julia is also a high-level dynamic language. How come?
  - The JIT compiler.
  - The LLVM project.
- Julia is open source (and it's for free)
  - It's for free. Did I say that it's for free?
  - You will never again worry about licenses. Want to run 1000 instances of julia? Do it.
  - The entire standard library of julia is written in julia (and not in `C`, e.g., as is the case in R, matlab or python). It's easy to look and understand at how things work.
- Julia is a very modern language, combining the best features of many other languages.

1.4 Why not julia?

- Julia is still in version `0.5.0`. There may be language changes in future releases.

- There are way fewer packages for certain tasks than for, say, R.
- Relatively few people know it. Your supervisor almost surely doesn't know it.

1.5 Economists and Their Software

- [@jesus-computing] [1], in *A Comparison of Programming Languages in Economics*, compare some widely used languages on a close to identical piece of code.
- It can be quite contentious to talk about Software to Economists.
  - Religious War.
  - Look at the comments on this blog post regarding the paper.
  - There *are* switching costs from one language to another.
  - Network effects (Seniors handing down their software to juniors etc)
- Takeaway from that paper:
  - There are some very good alternatives to `fortran`
  - `fortran` is **not** faster than `C++`
  - It seems pointless to invest either money or time in `matlab`, given the many good options that are available for free.

1.6 The Fundamental Tradeoff

Developer Time (Your Time) is Much More Expensive than Computing Time

- It may well be that the runtime of a fortran program is one third of the time it takes to run the program in julia, or anything else for that matter.
- However, the time it takes to **develop** that program is very likely to be (much) longer in fortran.
- Particularly if you want to hold your program to the same quality standards.

Takeaway

- Given my personal experience with many of the above languagues, I think `julia` is a very good tool for economists with non-trivial computational tasks.
- This is why I am using it for demonstrations in this course.

1.7 A Second Fundamental Tradeoff

- Regardless of the software you use, there is one main problem with computation.
- It concerns **speed vs accuracy**.
- You may be able to do something very fast, but at very small accuracy (i.e. with a high numerical margin of error)
- On the other hand, you may be able to get a very accurate solution, but it may take you an irrealistic amount of time to get there.
- You have to face that tradeoff and decide for yourself what's best.

1.8 A Warning

In Donald Knuth's paper "Structured Programming With GoTo Statements", he wrote:

> "Programmers waste enormous amounts of time thinking about, or worrying about, the speed of noncritical parts of their programs, and these attempts at efficiency actually have a strong negative impact when debugging and maintenance are considered. We should forget about small efficiencies, say about 97% of the time: **premature optimization is the root of all evil**. Yet we should not pass up our opportunities in that critical 3%."

1.9 Computers

- At a high level, computers execute instructions.
- Most of the actual *computation* is performed on the Central Processing Unit (CPU). The CPU does
  - Addition
  - substraction
  - multiplication
  - division
  - maybe also (depends on Chipset)
    - exponentiation
    - logarithm
    - trogonometric operations
  - Everything else is a combination of those.
  - The speed of those operations varies a lot. Exponentiation is about 10 times slower than multiplication.

1.10 An Example

- Suppose we want to compute

$$u(c,l) = \frac{\left(c^{\alpha} l^{1-\alpha}\right)^{1-\gamma}}{1-\gamma}$$

- Let's take $c = 1.1, l = 0.8$ and compute this.

```
In [ ]: alpha = 0.7
        gamma = 2.1
        # straighforward approach
        u(c,l) = ((c^alpha * l^(1-alpha))^(1-gamma)) / (1-gamma)
```

1.11 more sophisticated approach

- Note that $x^{\alpha} = \exp(\alpha \log(x))$.
- (particularly non-integer) exponentiation is expensive. let's see if we can do better:

```
In [ ]: u2(c,l)      = exp( alpha*(1-gamma) * log(c) + (1-alpha)*(1-gamma) * log(l) )/(1-gamma)
        # benchmark
        # 1) same result?
```

```
using Base.Test
@test_approx_eq u(1.1,0.8) u2(1.1,0.8)
# 2) timing?
n = 1e7          # get a reasonable sample size
t1 = @elapsed for i in 1:n u(1.1,0.8) end
t2 = @elapsed for i in 1:n u2(1.1,0.8) end
println("u1 takes $(round(t1/t2,2)) times longer than u2")
```

2 Some Numerical Concepts and `Julia`

- Machine epsilon: The smallest number that your computer can represent, type `eps()`.
- Infinity: A number greater than all representable numbers on your computer. Obeys some arithmethmic rules
  - Overflow: If you perform an operation where the result is greater than the largest representable number.
  - Underflow: You take two (very small) representable numbers, but the result is smaller than `eps()`.
  - In Julia, you are wrapped around the end of your representable space:

    ```
    x = typemax(Int64)
    x + 1
    ```
- Integers and Floating Point Numbers.
- Single and Double Precision.
- In Julia, all of these are different *numeric primitive types (head over to julia manual for a second)*.
- Julia also supports *Arbitrary Precision Arithmetic*. Thus, overflow shouldn't become an issue anymore.
- See min and max for different types:

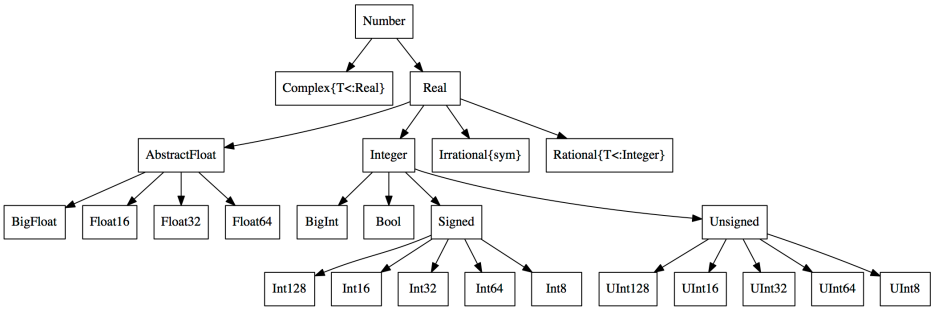
```
In [ ]: for T in [Int8,Int16,Int32,Int64,Int128,UInt8,UInt16,UInt32,
                         UInt64,UInt128,Float32,Float64]
                println("$(lpad(T,7)): [$(typemin(T)),$(typemax(T))]")
        end
```

2.1 Interacting with the `Julia REPL`

- REPL?
- different modes.
- incremental search with `CTRL r`
- documented in the manual

2.2 `Julia` Primer: What is a `module`?

- A separate `namespace`: variable names inside a module are not visible from outside, unless exported.
- If you use somebody else's code that contains the object `model`, and you have code that also defines `model`, you have a name conflict.
- Using modules avoids this conflict.



- Julia packages are provided as `modules`.
- You should adapt `modules` as the best workflow practice with julia.

3 `Julia` Primer: Types

- Types are at the core of what makes julia a great language.
- *Everything* in julia is represented as a datatype.
- Remember the different numeric *types* from before? Those are types.
- The manual, as usual, is very informative on this.
- From the wikibook on julia, here is a representation of the numeric type graph:

3.1 `Julia` Primer: Custom Types

- The great thing is that you can create you own types.
- Going with the example from the wikibook, we could have types `Jaguar` and `Pussycat` as being subtypes of `feline`:

```
abstract Feline
type Jaguar <: Feline
    weight::Float64
    sound::AbstractString
end
type Pussycat <: Feline
    weight::Float64
    sound::AbstractString
end
```

- This means both jaguar and pusscat are subtypes of `Feline`,
- They have common fields `weight` and `sound`.
- but there could be different functions. We call function specific to a type 'methods'

```
function do_your_thing(j::Jaguar)
    println(j.sound)
    println("I am going to throw my entire $(j.weight) kg at you, I'll catch, kill and eat you!"
```

```
    println("I am a $(typeof(j))!!!!")
end
function do_your_thing(c::Pussycat)
    println(c.sound)
    println("I should watch my line, $(c.weight) is enough for a $(typeof(c))")
    println(c.sound)
end
```

3.2 Julia Primer: Multiple Dispatch

- You have just learned `multiple dispatch`. The same function name dispatches to different functions, depending on the input argument type.
- Add all of the above code into the module `Example`
- Add Jaguar, Pussycat, and `do_your_thing` to `export` and save the file.

```
In [ ]: # types
        abstract Feline
        type Jaguar <: Feline
                weight::Float64
                sound::AbstractString
        end
        type Pussycat <: Feline
                weight::Float64
                sound::AbstractString
        end

        # methods
        hello(who::AbstractString) = "Hello, $who"
        domath(x::Number) = (x + 5)

        """
        makes a jaguar do their thing.
        """
        function do_your_thing(j::Jaguar)
                println(j.sound)
                println("I am going to throw my entire $(j.weight) kg at you, I'll catch, kill a
                println("I am a $(typeof(j))!!!!")
        end

        """
        makes a pussycat do their thing.
        """
        function do_your_thing(c::Pussycat)
                println(c.sound)
                println("I should watch my weight, $(c.weight) is enough for a $(typeof(c))")
                println(c.sound)
        end
```

```
In [ ]: j = Jaguar(130.1,"roaaarrrrrrr")
        c = Pussycat(9.8,"miauuu")
```

... and make them `do_their_thing`:

```
do_your_thing(j)
do_your_thing(c)
```

3.3 Julia Primer: Important performance lesson - Type Stability

- If you don't declare types, julia will try to infer them for you.
- DANGER: don't change types along the way.
  - julia optimizes your code for a specific type configuration.
  - it's not the same CPU operation to add two `Int`s and two `Float`s. The difference matters.
- Example

```
In [ ]: function t1(n)
            s  = 0  # typeof(s) = Int
            for i in 1:n
                s += s/i
            end
        end
        function t2(n)
            s  = 0.0   # typeof(s) = Float64
            for i in 1:n
                s += s/i
            end
        end
        @time t1(10000000)
        @time t2(10000000)
```

3.4 Julia Modules

- A module is a new workspace - a new *global scope*
- A module defines a separate *namespace*
- There is an illustrative example available at the julia manual, let's look at it.

3.4.1 An example Module

```
module MyModule
    #ăwhich other modules to use: imports
    using Lib
    using BigLib: thing1, thing2
    import Base.show
    importall OtherLib

    # what to export from this module
```

```
    export MyType, foo

    #ătype defs
    type MyType
        x
    end

    #ămethods
    bar(x) = 2x
    foo(a::MyType) = bar(a.x) + 1

    show(io::IO, a::MyType) = print(io, "MyType $(a.x)")
end
```

3.4.2 Modules and files

- you can easily have more files inside a module to organize your code.
- For example, you could `include` other files like this

```
module Foo

include("file1.jl")
include("file2.jl")

end
```

3.4.3 Working with Modules

- Look at the example at the manual!
- **Location of Modules**: Julia stores packages in a hidden folder `~/.julia` (system-dependent)
- You can develop your own modules in a different location if you want.
- Julia reads the file `~/.juliarc.jl` on each startup. Modify the `LOAD_PATH` variable:

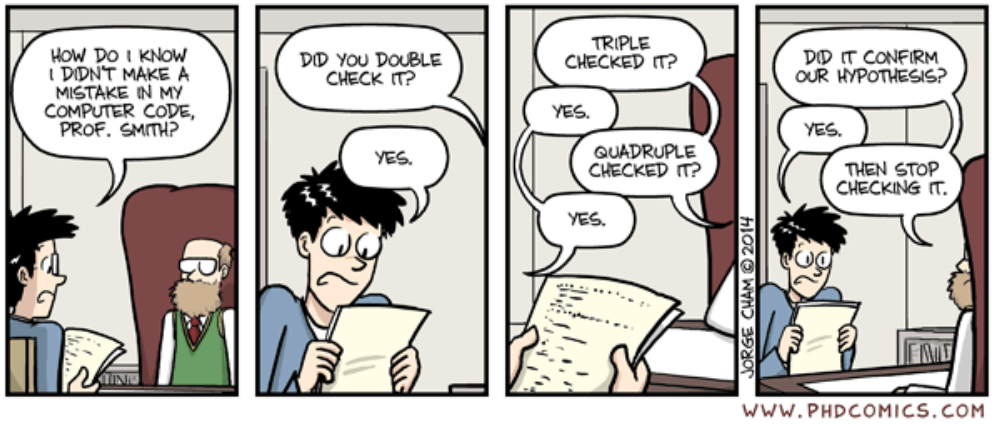
```
# add this to ~/.juliarc.jl
push!(LOAD_PATH, "/Path/To/My/Module/")
```

3.4.4 ScPoExample.jl

- I have created a fork of Example.jl at ScPoExample.jl for you.
- Use Github Desktop to clone this to your computer
- When done, open Juno, or your text editor to look at the code.

3.4.5 Extending ScPoExample.jl

1. add a function which creates a random vector array of size (2,3,3)
2. add a test that checks that the function works correctly



3.5 Unit Testing and Code Quality

3.6 What is Unit Testing? Why should you test you code?

- Bugs are very hard to find just by *looking* at your code.
- Bugs hide.
- From this very instructive MIT software construction class:

  Even with the best validation, it's very hard to achieve perfect quality in software. Here are some typical residual defect rates (bugs left over after the software has shipped) per kloc (one thousand lines of source code): * 1 - 10 defects/kloc: Typical industry software. * 0.1 - 1 defects/kloc: High-quality validation. The Java libraries might achieve this level of correctness. * 0.01 - 0.1 defects/kloc: The very best, safety-critical validation. NASA and companies like Praxis can achieve this level. This can be discouraging for large systems. For example, if you have shipped a million lines of typical industry source code (1 defect/kloc), it means you missed 1000 bugs!

3.7 Unit Testing in Science

- One widely-used way to prevent your code from having too many errors, is to continuously test it.
- This issue is widely neglected in Economics as well as other sciences.
  - If the resulting graph looks right, the code should be alright, shouldn't it?
  - Well, should it?
- It is regrettable that so little effort is put into verifying the proper functioning of scientific code.
- Referees in general don't have access to the computing code for paper that is submitted to a journal for publication.
- How should they be able to tell whether what they see in black on white on paper is the result of the actual computation that was proposed, rather than the result of chance (a.k.a. a bug)?
  - Increasingly papers do post the source code *after* publication.

  - The scientific method is based on the principle of **reproduciblity** of results.
    * Notice that having something reproducible is only a first step, since you can reproduce with your buggy code the same nice graph.
    * But from where we are right now, it's an important first step.
  - This is an issue that is detrimental to credibility of Economics, and Science, as a whole.
- Extensively testing your code will guard you against this.

3.8 Best Practice

- You want to be in **maximum control** over your code at all times:
  - You want to be **as sure as possible** that a certain piece of code is doing, what it actually meant to do.
  - This sounds trivial (and it is), yet very few people engage in unit testing.
- Things are slowly changing. See http://www.runmycode.org for example.
- **You** are the generation that is going to change this. Do it.
- Let's look at some real world Examples.

3.9 Ariane 5 blows up because of a bug

It took the European Space Agency 10 years and $$7 billion to produce Ariane 5, a giant rocket capable of hurling a pair of three-ton satellites into orbit with each launch and intended to give Europe overwhelming supremacy in the commercial space business. All it took to explode that rocket less than a minute into its maiden voyage last June, scattering fiery rubble across the mangrove swamps of French Guiana, was a small computer program trying to stuff a 64-bit number into a 16-bit space. This shutdown occurred 36.7 seconds after launch, when the guidance system's own computer tried to convert one piece of data -- the sideways velocity of the rocket -- from a 64-bit format to a 16-bit format. **The number was too big, and an overflow error resulted**. When the guidance system shut down, it passed control to an identical, redundant unit, which was there to provide backup in case of just such a failure. But the second unit had failed in the identical manner a few milliseconds before. And why not? It was running the same software.

3.10 NASA Mars Orbiter crashes because of a bug

For nine months, the Mars Climate Orbiter was speeding through space and speaking to NASA in **metric**. But the engineers on the ground were replying **in non-metric English**. It was a mathematical mismatch that was not caught until after the $$125-million spacecraft, a key part of NASA's Mars exploration program, was sent crashing too low and too fast into the Martian atmosphere. The craft has not been heard from since. Noel Henners of Lockheed Martin Astronautics, the prime contractor for the Mars craft, said at a news conference it was up to his company's engineers to assure the metric systems used in one computer program were compatible with the English system used in another program. The simple conversion check was not done, he said.

3.11 LA Airport Air Traffic Control shuts down because of a bug

(IEEE Spectrum) -- It was an air traffic controller's worst nightmare. Without warning, on Tuesday, 14 September, at about 5 p.m. Pacific daylight time, air traffic controllers lost voice contact with 400 airplanes they were tracking over the southwestern United States. Planes started to head toward one another, something that occurs routinely under careful control of the air traffic controllers, who keep airplanes safely apart. But now the controllers had no way to redirect the planes' courses. The controllers lost contact with the planes when the main voice communications system (VCS) shut down unexpectedly. To make matters worse, a backup system that was supposed to take over in such an event crashed within a minute after it was turned on. The outage disrupted about 800 flights across the country. Inside the control system unit (VCSU) is a countdown timer that ticks off time in milliseconds. The VCSU uses the timer as a pulse to send out periodic queries to the VSCS. It starts out at the highest possible number that the system's server and its software can handle — 232. It's a number just over 4 billion milliseconds. When the counter reaches zero, the system runs out of ticks and can no longer time itself. **So it shuts down**. *Counting down from 232 to zero in milliseconds takes just under 50 days*. The FAA procedure of having a technician reboot the VSCS every 30 days resets the timer to 232 almost three weeks before it runs out of digits.

3.12 Automated Testing

- You should try to minimize the effort of writing tests.
- Using an automated test suite is very helpful here.
- In Julia, we have got `Base.Test` in the Base package, and `FactCheck` as a package.
- Julia unit testing is described here

3.13 Automated Testing on Travis

- `https://travis-ci.org` is a continuous integration service.
- It runs your test on their machines and notifies you of the result.
- Every time you push a commit to github.
- If the repository is public on github, the service is for free.
- Many julia packages are testing on Travis.
- You should look out for the green badge.

3.14 Debugging Julia

There are at least 2 ways to chase down a bug.

1. Use logging facilities. This is writing `println` statements at various points in your code.
2. Use a debugger:
    - The Julia debugger is called Gallium.jl
    - This works well in Juno.

3.14.1 Logging.jl

- A simple logging package inspired by Python's logger class.

- Do this:

```
In [ ]: using Logging

        function log_test()
            debug("debug message")
            info("info message")
            warn("warning message")
            err("error message")
            critical("critical message")
        end

        Logging.configure(level=DEBUG)
        log_test()
```

We can also use Macros that can be turned off completely, resulting an almost zero function call overhead:

```
In [ ]: @Logging.configure(level=DEBUG)

        function macro_log_test()
            @debug("debug message")
            @info("info message")
            @warn("warning message")
            @err("error message")
            @critical("critical message")
        end

        macro_log_test()

        # now we are done debugging
        println("turning logging off completely")
        @Logging.configure(level=OFF)

        macro_log_test()
```

3.14.2 Debugging with Gallium

- Let's switch to Juno for a second.
- The ScPoExample.jl repo contains a branch called `dbg`. do

```
cd ScPoExample.jl
git checkout dbg
```

or select branch in github desktop.

- Then open Juno and load that code.

3.15 Julia and Data

- DataFrames.jl
- DataFramesMeta.jl
- Queries.jl
- StructuredQueries.jl - not there yet

3.15.1 DataFrames

- A dataframe is a tabular dataset: a spreadsheet
- columns can be of different data type. very convenient, very hard to optimize.
- Importantly, working with data requires one special data type. **missing data**.
  - What should be the features of such a data type?
- DataFrames still supports the `NA` value.
- This is going to disappear in future versions in favour of the new `Nullable` type.

```
In [ ]: using RDatasets    # popular Datasets from R
        using DataFrames
        iris = dataset("datasets", "iris")
        head(iris)  # get the first 6 rows
```

```
In [ ]: println(iris[:SepalLength][1:6])  # get a column

        show(iris[2,:])    # get a row

        describe(iris);    # get a description
```

3.15.2 Working with DataFrames

- RTFM
- We can sort, join (i.e. merge), split-apply-combine and reshape dataframes. We can do most things one can do in base R with a `data.frame`.
- Mention `pooled` data:

```
pdv = @pdata(["Group A", "Group A", "Group A",
              "Group B", "Group B", "Group B"])
```

```
In [ ]: # subsetting a dataframe can become cumbersome
        d = DataFrame(A = rand(10),B=rand([1,2],10),C=["word $i" for i in 1:10])
         d[(d[:A].>0.1) & (d[:B].==2),:]
        using DataFramesMeta
        @where(d,(:A .>0.1) & (:B.==2))
```

enter

3.15.3 DataFramesMeta.jl

- This makes this much easier.
- It makes heavy use of macros.
- The same thing from before is now

```
using DataFramesMeta
@where(d,(:A .>0.1) & (:B.==2))
```

- We can access column names as *symbols* inside an expression.

3.15.4 Chaining operations

- Very often we have to do a chain of operations on a set of data.
- Readibility is a big concern here.
- Here we can use the `@linq` macro together with a pipe operator.
- This is inspired by `LINK` (language integrated query) from microsoft .NET

```
In [ ]: using DataFramesMeta
        df = DataFrame(a = 1:20,b = rand([2,5,10],20), x = rand(20))
        x_thread = @linq df |>
            transform(y = 10 * :x) |>
            where(:a .> 2) |>
            by(:b, meanX = mean(:x), meanY = mean(:y)) |>
            orderby(:meanX) |>
            select(:meanX, :meanY, var = :b)
```

an alternative

3.15.5 Query.jl

- Much in the same spirit. But can query almost any data source, not only dataframes.
  - Data Sources: DataFrames, Dicts, Arrays, TypedTables, DataStreams,...
  - Data Sinks: DataFrames, Dicts, Csv
- It is not as convenient to summarise data, however.
- It is one of the best documented packages I know.
- Here is an example.

```
In [3]: using Query
        using DataFrames

        df = DataFrame(name=["John", "Sally", "Kirk"], age=[23., 42., 59.], children=[3,5,2])

        x = @from i in df begin
            @where i.age>50
            @select {i.name, i.children}
            @collect DataFrame
        end
        println(df)
        println(x)
```

```
3×3 DataFrames.DataFrame
 Row  name     age   children

 1    "John"   23.0  3
 2    "Sally"  42.0  5
 3    "Kirk"   59.0  2
1×2 DataFrames.DataFrame
 Row  name    children

 1    "Kirk"  2
```

3.16 Plotting with Julia

- There are many different plotting packages for julia. Gadfly.jl, PyPlot.jl, Winston.jl, Vega.jl, ...
- We will use the Plots.jl package with talks to them all.
- The aim is to provide a unique interface to the user, who then has the freedom to choose their preferred backend.
  - http://mth229.github.io/graphing.html has a nice intro to plotting.
  - See quant-econ.net website for good intro to PyPlot plotting.

```
In [ ]: using Plots
        plotlyjs()
        f(x) = exp(-x^2/2)
        plot(f, -3, 3) # plot f over [-3,3]
```

```
In [ ]: # We can add to the current plot with the `plot!` function:
        f(x) = cos(x)
        g(x) = 1 - x^2/2
        plot(f, -pi/2, pi/2)
        plot!(g, -pi/2, pi/2)   # try without the a and b, it may work.
```

4 References

- Judd Chapter 1 [2]

References

[1] S Boraugan Aruoba and Jesus Fernandez-Villaverde. A comparison of programming languages in economics. Technical report, National Bureau of Economic Research, 2014.

[2] Kenneth L. Judd. *Numerical methods in economics*. The MIT Press, 1998.