



# Scientific Computing and (Big) Data Analysis with Julia

Prof.Dr.Mehmet Hakan Satman  
mhsatman@istanbul.edu.tr

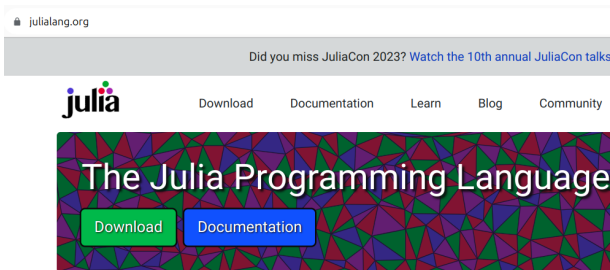
Istanbul University

2023.12.01



# Julia Programming Language

julialang.org



The screenshot shows the homepage of the Julia Programming Language website. At the top, there is a navigation bar with the Julia logo on the left and links for "Download", "Documentation", "Learn", "Blog", and "Community" on the right. A banner below the navigation bar features a colorful geometric pattern and the text "The Julia Programming Language". Below this banner are two buttons: a green "Download" button and a blue "Documentation" button. Above the banner, a grey bar contains the text "Did you miss JuliaCon 2023? Watch the 10th annual JuliaCon talks".

## Julia in a Nutshell

### Fast

Julia was designed for [high performance](#). Julia

### Dynamic

Julia is [dynamically typed](#), feels like a scripting



# Julia Programming Language

High Performance Computing & Dynamic

- Julia was designed for high performance. Julia programs automatically compile to efficient native code via LLVM, and support multiple platforms (Windows, MacOS, Linux, etc.).
- Julia is dynamically typed, feels like a scripting language, and has good support for interactive use, but can also optionally be separately compiled<sup>1</sup>.

---

<sup>1</sup><https://julialang.org/>



# Julia Programming Language

Composable & Open Source

- Julia uses multiple dispatch as a paradigm, making it easy to express many object-oriented and functional programming patterns. The talk on the Unreasonable Effectiveness of Multiple Dispatch explains why it works so well.
- Julia is an open source project with over 1,000 contributors. It is made available under the MIT license. The source code is available on GitHub<sup>2</sup>.

---

<sup>2</sup><https://julialang.org/>



# Julia Programming Language

## Compilation to Binary

Julia code is compiled into binary executable via LLVM (Low-Level Virtual Machine).

```

julia> @code_native 2 - 5
.text
.file "_"
.globl "julia_-_199"           # -- Begin function julia_-_199
.p2align      4, 0x90
.type "julia_-_199",@function
"julia_-_199":                # @julia_-_199
; r @ int.jl:86 within `~`
# %bb.0:                       # %top
    push    rbp
    mov     rax, rdi
    mov     rbp, rsp
    sub    rax, rsi
    pop    rbp
    ret
.Lfunc_end0:
.size "julia_-_199", .Lfunc_end0-"julia_-_199"
; L
                                # -- End function
.section ".note.GNU-stack","",@progbits

```



# Julia Programming Language

First things first!

helloworld.jl file

```
println("Hello, world!")
```

```
julia> include("helloworld.jl")  
Hello, world!
```



# Julia Programming Language

Welcome!

```
Documentation: https://docs.julialang.org
Type "?" for help, "]"?" for Pkg help.
Version 1.10.0-rc1 (2023-11-03)
Official https://julialang.org/ release

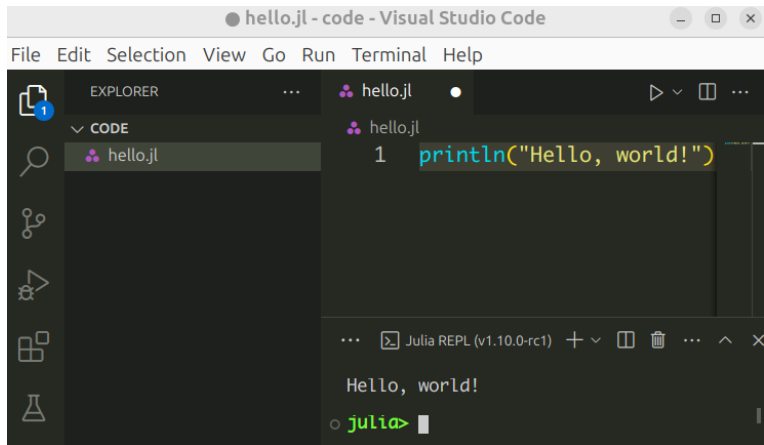
julia> println("Hello, world!")
Hello, world!

julia> 
```



# Julia Programming Language

The editor: Visual Studio Code



The screenshot shows the Visual Studio Code interface. The title bar reads "hello.jl - code - Visual Studio Code". The menu bar includes "File", "Edit", "Selection", "View", "Go", "Run", "Terminal", and "Help". The Explorer sidebar on the left shows a file named "hello.jl" under the "CODE" section. The main editor area displays the code: `1 println("Hello, world!")`. Below the editor is a terminal window titled "Julia REPL (v1.10.0-rc1)" which shows the output "Hello, world!" and the prompt `julia>`.





# Julia Programming Language

## Basics

Variables have types (Int, Float, Bool, String, etc.)

```
julia> a = 3
3
julia> b = 3.14159265
3.14159265
julia> typeof(a)
Int64
julia> typeof(b)
Float64
```



# Julia Programming Language

## Vectors

Vectors and Matrices are first-class citizens (no need for external libs)

```
julia> v = [1, 42, -8, 10]
4-element Vector{Int64}:
 1
42
-8
10
```



# Julia Programming Language

## Matrices

```
julia> hcat(zeros(5), ones(5), 1:5, 5:(-1):1)
5x4 Matrix{Float64}:
 0.0  1.0  1.0  5.0
 0.0  1.0  2.0  4.0
 0.0  1.0  3.0  3.0
 0.0  1.0  4.0  2.0
 0.0  1.0  5.0  1.0
```



# Julia Programming Language

## Matrices

```
julia> m = zeros(5, 3)
```

```
5x3 Matrix{Float64}:
```

```
0.0  0.0  0.0
```

```
0.0  0.0  0.0
```

```
0.0  0.0  0.0
```

```
0.0  0.0  0.0
```

```
0.0  0.0  0.0
```

```
julia> size(m)
```

```
(5, 3)
```



# Julia Programming Language

## Installing packages

```
julia> using Pkg  
julia> Pkg.add("JMCDM")
```

```
julia> ]  
@v1.10) pkg> add JMCDM
```



# Julia Programming Language

## Importing Data

```
using CSV, DataFrames

mydata = CSV.read("data.csv",
                  delim = ";",
                  DataFrame)

show(mydata)
```



# Julia Programming Language

## Importing Data

```
julia> using Latexify
julia> latexify(mydata, env = :table) |> println
```

X	Y
1	2
2	4
3	5
4	-1
5	2



# Julia Programming Language

## if/elseif/else

```
function numberofrealroots(delta)
    if delta > 0
        return 2
    elseif delta == 0
        return 1
    else
        return 0
    end
end
```





# Julia Programming Language

## Pattern Matching

```
using Rematch

function numberofrealroots(delta)
    @match delta begin
        x where x > 0    => 2
        x where x == 0  => 1
        -                => 0
    end
end
```



# Julia Programming Language

## Loops

For loops are single threaded by design

```
results = zeros(10)

for i in 1:10
    results[i] = dosomethingwith(i)
end
```



# Julia Programming Language

## Threads

Using multiple threads<sup>3</sup>

```
using Base.Threads

results = zeros(10)

@threads for i in 1:10
    results[i] = dosomethingwith(i)
end
```

---

<sup>3</sup># julia -t 10



# Julia Programming Language

## Distributed Programming

```
julia> using Distributed
julia> addprocs(5);
julia> pmap(abs, [1, 2, -5, 10, 100, -6])
6-element Vector{Int64}:
 1
 2
 5
10
100
 6
```



# Julia Programming Language

Functions are first-class citizens

```
function apply(f, x)
    return f(x)
end

julia> apply(abs, -10)
10
```

- Functions can take functions as arguments.
- Functions can return functions as values.



# Julia Programming Language

## Multiple Dispatch

```
struct Point2D
    x::Float64
    y::Float64
end
```

- Structs are user-defined concrete data types.
- An object instance can be created like `Point2D(1, 2)`.
- Object fields can be accessed like `p.x` and `p.y`.



# Julia Programming Language

## Multiple Dispatch

```
function Base.+(p::Point2D, other::Point2D)::Point2D
    Point2D(p.x + other.x, p.y + other.y)
end
```

```
julia> Point2D(1, 2) + Point2D(4, 5)
Point2D(5.0, 7.0)
```

- Operator `+` is overloaded for the type `Point2D`.
- Now, both `2 + 2` and `p1 + p2` are legal Julia codes where `p1` and `p2` are in type of `Point2D`.



# Julia Programming Language

## Multiple Dispatch

```
function Base.:*(p::Point2D, other::Point2D)::Float64
    return p.x * other.x + p.y * other.y
end
```

```
julia> Point2D(1, 2) * Point2D(4, 5)
12.0
```

- The operator `*` is overloaded for the type `Point2D`.
- `*` now operates like the *dot product* of vectors in linear algebra.





# Linear Regression

## The formulation

$$y = \beta_0 + \beta_1 x + \varepsilon \quad (1)$$

$$\hat{y} = \hat{\beta}_0 + \hat{\beta}_1 x \quad (2)$$

$$\hat{\beta} = (X'X)^{-1}X'y \quad (3)$$



# Linear Regression

## Sample Data

$$X = \begin{bmatrix} 1 & 1 \\ 1 & 2 \\ 1 & 3 \\ 1 & 4 \\ 1 & 5 \end{bmatrix}, y = \begin{bmatrix} 2 \\ 5 \\ 5 \\ 8 \\ 12 \end{bmatrix} \quad (4)$$



# Linear Regression

## The Matrix Solution

```
using LinearAlgebra

x = [1, 2, 3, 4, 5]
y = [2, 5, 5, 8, 12]
X = hcat(ones(5), x)
betahats = inv(X'X)X'y
println(betahats)
```

```
julia> include("reg-matrix.jl")
[-0.5, 2.3]
```



# Linear Regression

## Pseudo Inverse - Numerical Fit

```
x = [1, 2, 3, 4, 5]
y = [2, 5, 5, 8, 12]

betahats = hcat(ones(5), x) \ y
println(betahats)
```

```
julia> include("reg-simple.jl")
[-0.5, 2.3]
```



# Linear Regression

## The GLM package

```
using GLM

x = [1, 2, 3, 4, 5]
y = [2, 5, 5, 8, 12]

result = lm(hcat(ones(5), x), y)

println(result)
```



# Linear Regression

## The GLM package - Results

```
julia> include("reg-glm.jl")
```

Coefficients:

	Coef.	Std. Error	t	Pr(> t )	Lower 95%	Upper 95%
x1	-0.5	1.25565	-0.40	0.7171	-4.49605	3.49605
x2	2.3	0.378594	6.08	0.0090	1.09515	3.50485

```
julia> GLM.r2(result)
```

```
0.9248251748251748
```



# MLJ

## A Machine Learning Framework for Julia

```
julia> using MLJ
julia> models = MLJ.models()
julia> for m in models
    println(m[:name])
end
ARDRegressor
AdaBoostClassifier
AdaBoostRegressor
AdaBoostStumpClassifier
...
KMedoids
KNNClassifier
...
NeuralNetworkRegressor
...
RandomForestClassifier
RandomForestImputer
RandomForestRegressor
...
SRRegressor
```



# XOR

eXclusive OR

$x_1$	$x_2$	$y$
1	1	0
1	0	1
0	1	1
0	0	0

Table:  $y = \text{xor}(x_1, x_2)$





# Symbolic Regression

```
using SymbolicRegression, MLJ

x = (
  x1 = Float64[1, 1, 0, 0],
  x2 = Float64[1, 0, 1, 0]
)

y = Float64[0, 1, 1, 0]
```



# Symbolic Regression

```
model = SRRegressor(  
    niterations = 50,  
    binary_operators = [+ , - , *] ,  
    unary_operators = [abs] ,  
    should_simplify = true ,  
    save_to_file = false)
```



# Symbolic Regression

```
mach = machine(model, x, y)
fit!(mach)
report(mach)
@info predict(mach, x)
```



# Symbolic Regression

Hall of Fame:

```
-----  
Complexity    Loss          Score          Equation  
1             2.500e-01     3.604e+01     y = 0.5  
4             0.000e+00     1.201e+01     y = abs(x1 - x2)  
-----  
[ Info: [0.0, 1.0, 1.0, 0.0]
```



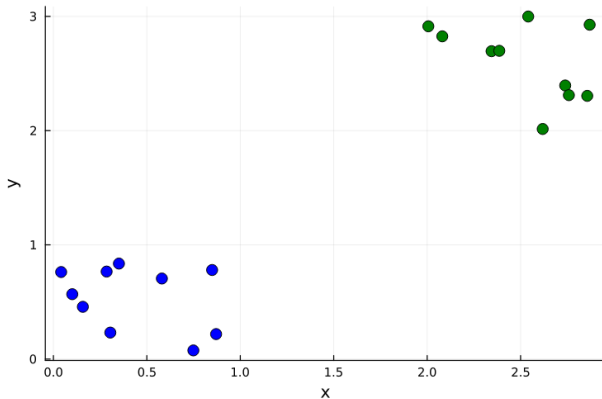


*one more cup of coffee?*



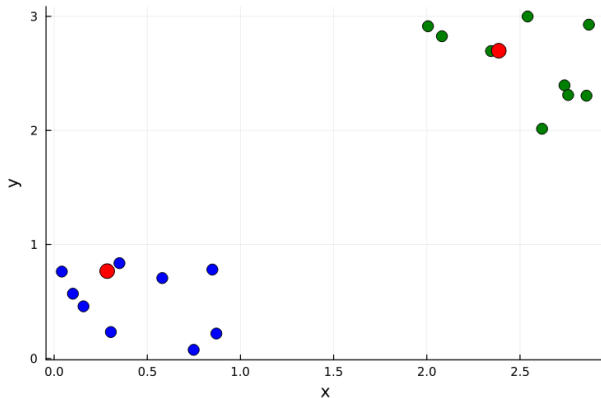
# Clustering Multivariate Data

## k-medoids



# Clustering Multivariate Data

## k-medoids



# Clustering Multivariate Data

## Problem of Distance Matrices

```
using Clustering, Plots, Distances

# data = Code for loading data...
plt = scatter(data[:, 1], data[:, 2])

dist = pairwise(euclidean, eachrow(data))

result = kmedoids(dist, 2)
centers = data[result.medoids, :];
scatter!(centers[:, 1], centers[:, 2])
```





# A Distance Matrix

$$D = \begin{bmatrix} D_{11} & D_{12} & D_{13} & \dots & D_{1n} \\ D_{21} & D_{22} & D_{23} & \dots & D_{2n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ D_{n1} & D_{n2} & D_{n3} & \dots & D_{nn} \end{bmatrix}_{n \times n}$$



# Clustering Multivariate Data

## Problem of Distance Matrices

```
dist = pairwise(euclidean, eachrow(data))
```

- A distance matrix holds the distance data of  $i$ th and  $j$ th points, e.g.,  $D_{ij} = D_{ji}$  due to the symmetry.
- If data has  $n$  rows then the distance matrix is in dimension of  $n \times n$ .
- Each distance is measured in 64-bits float numbers (Float64).
- If  $n$  is large, your machine will probably throw an *Out of Memory* error!



# Big Distance Matrices

## On-demand Distance Matrix

```
struct OnDemandDistanceMatrix <: AbstractMatrix{Float64}
    rawdata :: Matrix
end

function Base.getindex(odm::OnDemandDistanceMatrix, i::Int, j::Int)::Float64
    return euclidean(odm.rawdata[i, :], odm.rawdata[j, :])
end

function Base.size(odm::OnDemandDistanceMatrix)
    n, _ = size(odm.rawdata)
    return (n, n)
end
```



# Big Distance Matrices

## On-demand Distance Matrix

```
# Example
data = Float64[
    1 2;
    0 5;
    1 2]

d = OnDemandDistanceMatrix(data)

println(d[1, 3])
```



# Big Distance Matrices

## On-demand Distance Matrix

```
data = Float64[
    1 2;
    0 5;
    1 2]

d = OnDemandDistanceMatrix(data)

# d is a usual distance matrix now!
kmedoids(d, 2)
```



# Big Distance Matrices

## On-demand Distance Matrix

- On-demand distance matrix costs zero memory
- Caution: But it's really slow just because the requested distance is calculated on demand!
- But it makes it possible! 😊



# Big Matrices

## Memory Mapped IO

- We need an efficient way to cope with big distance matrices
- Memory-mapped IO is an OS level solution to this problem
- The content of a matrix is stored in files (on disk!)
- Access to data is really fast 😊 (contrast to the previous one!)



# Big Matrices

## Memory-mapped IO

```
import Mmap

xio = open("/tmp/X.dat", "w+")
yio = open("/tmp/y.dat", "w+")

X = Mmap.mmap(xio, Matrix{Float64}, (n, 2))
y = Mmap.mmap(yio, Vector{Float64}, n)
```

- $X$  and  $y$  are stored in files  $X.dat$  and  $y.dat$
- But they are stored in files and mapped to memory (RAM).





# Big Matrices

## Memory-mapped IO

$X$  and  $y$  are processed and accessed as normal matrices and vectors

```
X[1, :] = [1, 3]
```

```
y[5] = 9.7
```

```
betahats = inv(X'X)X'y
```



# Distributions

## The Normal Distribution

$$f(x; \mu, \sigma) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2}, \quad -\infty < x < \infty \quad (5)$$

$$f(x; 0, 1) = \frac{1}{\sqrt{2\pi}} e^{-\frac{1}{2}x^2}, \quad -\infty < x < \infty \quad (6)$$



# Distributions

## The Normal Distribution

```
julia> using Distributions

julia> quantile(Normal(), 0.05/2)
-1.9599639845400592

julia> quantile(Normal(), 0.10/2)
-1.6448536269514729

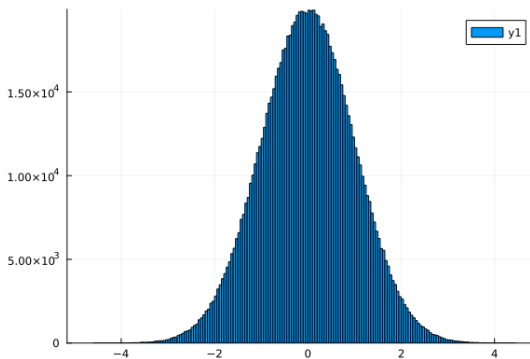
julia> quantile(Normal(), 0.01/2)
-2.5758293035489053
```



# Distributions

## Monte Carlo Simulations - Drawing Random Numbers

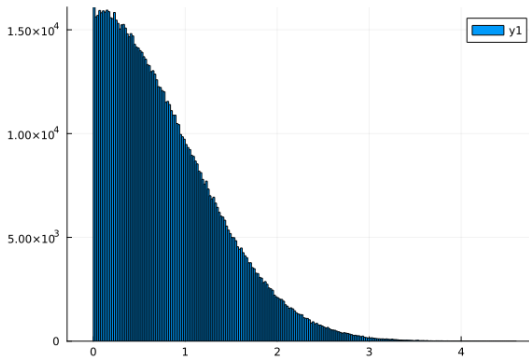
```
julia> using Plots, Distributions  
julia> x = rand(Normal(), 1000000);  
julia> histogram(x)
```



# Distributions

## Monte Carlo Simulations - Drawing Random Numbers

```
julia> histogram(abs.(x))
```



# Hypothesis Tests

## Jarque-Bera Test for Normality

```
julia> using HypothesisTests
julia> x = randn(30);
julia> JarqueBeraTest(x)
```

The null hypothesis is a joint hypothesis of the skewness being 0 and the kurtosis being 3.

$H_0$  : Data comes from a Normal distribution

$H_a$  : 



# Hypothesis Tests

## Jarque-Bera Test for Normality

### Jarque–Bera normality test

---

#### Population details:

parameter of interest:	skewness <b>and</b> kurtosis
value under $h_0$ :	"0 and 3"
point estimate:	"-0.065 and 1.873"

#### Test summary:

outcome with 95% confidence:	fail to reject $h_0$
one-sided p-value:	0.4474

#### Details:

number of observations:	30
JB statistic:	1.60881



# Numerical Integration

## QuadGK

$$\int_{-1}^1 \frac{1}{\sqrt{2\pi}} e^{-\frac{1}{2}x^2} dx =? \quad (7)$$

```
using QuadGK
```

```
function f(x)
```

```
    return 1/sqrt(2pi) * exp(-0.5x^2)
```

```
end
```

```
quadgk(f, -1.0, 1.0)
```

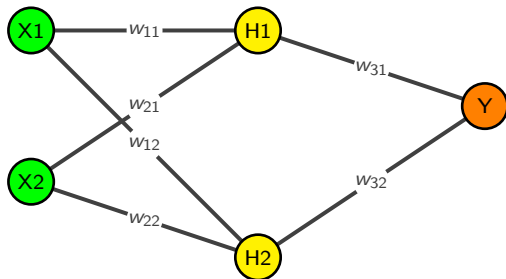




# Optimizations

## Simple Neural Network

$$H_1 = f(w_0 + x_1 w_{11} + x_2 w_{21})$$



# Optimizations

## Simple Neural Network

$$H_1 = f(w_{01} + x_1 w_{11} + x_2 w_{21})$$

$$H_2 = f(w_{02} + x_1 w_{12} + x_2 w_{22})$$

$$Y = f(w_{03} + w_{31} H_1 + w_{32} H_2)$$

What are the values of  $w_{ij}$ 's that minimize the total network error?



# Optimizations

## Simple Neural Network

```
function sigmoid(x)
    return 1.0/(1.0 + exp(-x))
end

function cost(w)
    error = 0.0
    for i in 1:4
        H1 = sigmoid(w[1] + w[2]*x1[i] + w[3]*x2[i])
        H2 = sigmoid(w[4] + w[5]*x1[i] + w[6]*x2[i])
        yhat = sigmoid(w[7] + w[8] * H1 + w[9] * H2)
        error += (yhat - y[i])^2
    end
    return error
end
```



# Optimizations

## Simple Neural Network

```
using Metaheuristics
```

```
x1 = [1, 1, 0, 0]
```

```
x2 = [1, 0, 1, 0]
```

```
y = [0, 1, 1, 0]
```

```
bounds = vcat([-10000.0 for i in 1:9]',  
              [10000.0 for i in 1:9]')
```

```
result = Metaheuristics.optimize(cost, bounds, MCCGA())
```

```
display(result)
```



## Feeding the trained network

```
function forward(w)
    yhat = zeros(length(y))
    for i in 1:4
        H1 = sigmoid(w[1] + w[2]*x1[i] + w[3]*x2[i])
        H2 = sigmoid(w[4] + w[5]*x1[i] + w[6]*x2[i])
        H3 = sigmoid(w[7] + w[8] * H1 + w[9] * H2)
        yhat[i] = H3
    end
    return yhat
end
```



# Optimizations

## Mathematical Programming

$$\max z = 2x_1 + 3x_2$$

Subject to:

$$x_1 + 2x_2 \leq 100$$

$$2x_1 + x_2 \leq 150$$

$$x_1, x_2 \geq 0$$



# Optimizations

## JuMP

```
using JuMP, HiGHS

m = Model(HiGHS.Optimizer)

@variable(m, x1 >= 0)
@variable(m, x2 >= 0)

@objective(m, Max, 2x1 + 3x2)

@constraint(m, x1 + 2x2 <= 100)
@constraint(m, 2x1 + x2 <= 150)
```



# Optimizations

JuMP

```
julia> optimize!(m)
Solving LP without presolve or with basis
Model      status      : Optimal
Objective value      : 1.83333333333e+02
HiGHS run time      : 0.00
```

```
julia> value.([x1, x2])
2-element Vector{Float64}:
66.66666666666667
16.666666666666657
```





# SQL Integration

## SQLite

```
using SQLite

db = SQLite.DB("database.db")

sqlst = """
    select item, price from Prices
    where date = '2023.12.01'
    order by price
"""

resultsql = DBInterface.execute(db, sqlst)

for row in resultsql
    println(row[:item], ": ", row[:price])
end

close(db)
```



- Julia can operate with R and Python.
- R and Python objects can be transferred in both ways.
- We don't need to give up on them, let's talk to the strangers!



# Talking to Strangers

## Calling into R

```
using RCall

x = [1, 2, 3, 4, 5]
y = [2, 5, 5, 8, 12]

@rput x y

R"result <- lm(y~x)"

jresult = @rget result
```



```
julia> jresult
:coefficients => [-0.5, 2.3]
:residuals    => [0.2, 0.9, -1.4, -0.7, 1.0]
:rank         => 2
:fitted_values => [1.8, 4.1, 6.4, 8.7, 11.0]
:assign       => [0, 1]
:df_residual  => 3
:xlevels      => OrderedDict{Symbol, Any}()
:terms        => y ~ x
```



# Talking to Strangers

## Calling into Python

```
using PyCall

np = pyimport("numpy")
linalg = pyimport("numpy.linalg")

x = np.matrix([1.0 1; 1 2; 1 3; 1 4; 1 5])
y = np.array([2.0, 5, 5, 8, 12])

result = linalg.lstsq(x, y)
```



# Talking to Strangers

## Calling into Python

```
julia> include("pycaller.jl")  
(  
    [-0.500000000000000023, 2.300000000000000003] ,  
    [4.2999999999999995] ,  
    2,  
    [7.69121313410482, 0.9193696350073228]  
)
```





Thank you!  
Any questions? ☕

