a small presentation of the julia language

Alejandro Pérez Velilla & Jingyu Xi

"Julia features optional typing, multiple dispatch, and good performance, achieved using type inference and just-in-time (JIT) compilation, implemented using LLVM. It is multi-paradigm, combining features of imperative, functional, and object-oriented programming. Julia provides ease and expressiveness for high-level numerical computing, in the same way as languages such as R, MATLAB, and Python, but also supports general programming. To achieve this, Julia builds upon the lineage of mathematical programming languages, but also borrows much from popular dynamic languages, including Lisp, Perl, Python, Lua, and Ruby." -https://docs.julialang.org/en/v1/

The language's release "manifesto" is a nice read on the subject: https://julialang.org/blog/2012/02/why-we-created-julia/

So, why julia?

"In short, because we are greedy."

let's jump in!

arrays

```
[1, 2, 3]
```

```
[1, 2, 3]
```

```
Vector{Int64} (alias for Array{Int64, 1})
```

```
typeof( [1, 2, 3] )
```

```
[1.0, 2.0, 3.0]
```

```
[1.0, 2.0, 3.0]
```

```
modsoc =  ["paul", "matt", "ket", "cody", "amin"]
```

```
modsoc = ["paul", "matt", "ket", "cody", "amin"]
```

```
"ket"
```

```
modsoc[3]
```

```
"paul"
```

```
modsoc[1]
```

```
[1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0]
```

```
ones(10)
```

```
my_zero_array =  [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
```

```
my_zero_array = zeros(10)
```

```
Vector{Float64} (alias for Array{Float64, 1})
```

```
typeof(my_zero_array)
```

```
Float64
```

```
eltype(my_zero_array)
```

```
[1, "paul"]
```

```
[1, "paul"]
```

```
[4.0e-323, 5.0e-324, 6.90819e-310]
```

```
Array{Float64, 1}(undef, 3) #create an array of a particular type and dim, and populate it
```

```
altered_array =  [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
```

```
altered_array = copy(my_zero_array)
```

```
[0.0, 0.0, 0.0, 0.0]
```

```
altered_array[4:7]
```

```
view(::Vector{Float64}, 5:6): [1.0, 1.0]
```

```
altered_array[5:6] .= 1 #broadcasting
```

```
[0.0, 1.0, 1.0, 0.0]
```

```
altered_array[4:7]
```

```
[0.0, 0.0, 0.0, 0.0, 1.0, 1.0, 0.0, 0.0, 0.0, 0.0]
```

```
altered_array
```

```
[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
```

```
my_zero_array
```

other convenient ways of generating arrays:

```
BitVector: [true, true, true, true, true, true, true, true, true, true]
```

```
trues(10)
```

```
BitVector: [false, false, false, false, false, false, false, false, false, false]
```

```
falses(10)
```

ranges are important:

```
1:10
```

```
1:10 #iterator object representing integers from 1 to 10
```

```
0:2:100
```

```
0:2:100 #start-stop-step
```

```
UnitRange{Int64}
```

```
typeof(1:10)
```

you can evaluate iterators (such as ranges) using `collect()`:

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

```
collect(1:10)
```

in general, you also want to use `collect()` to "unpack" lazy iterators and make immutable copies of lists (evaluating them in the process). In an ABM context, this will help you gather data from structures that are changing as the models run.

operations with arrays can be element-wise:

```
[2, 4, 6, 8, 10]
```

```
collect(1:5) + collect(1:5)
```

in this case elements are added by index. We can also operate on the array by broadcasting:

```
[2, 4, 6, 8, 10]
```

```
collect(1:5) .* 2
```

this is equivalent to

```
[2, 4, 6, 8, 10]
```

```
[n*2 for n in 1:5] #this is an array comprehension
```

and also to

```
[2, 4, 6, 8, 10]
```

```
begin
    local arr = Array{Int64, 1}()

    for n in 1:5
        push!(arr, n*2) #functions that end with ! modify one
    end                 #or several of their arguments
                        #they are called "in-place"
    arr
end
```

```
[2, 4, 6, 8, 10]
```

```
begin
    local arr = Array{Int64, 1}()

    local n = 1
    while n <= length(1:5)
        push!(arr, n*2)
        n += 1
    end

    arr
end
```

loops!

there are **many** useful functions for working with arrays. An example is the `setdiff(a, itrs)` function, which returns the elements in `a` that are not in `itrs`, where the latter can be a single iterator or a list of iterators:

```
[1.0]
```

```
begin
    my_arr = zeros(5) #define a vector of zeros
    my_arr[5] = 1 #change last element to 1
    setdiff( my_arr, zeros(5) ) #compare to a pristine zero vector
                                #and extract difference
end
```

again, there are many of these, and if we stop to look at all of them we will take a long time. So look in the documentation/tutorials section.

arrays also have concatenation:

```
[BitVector: [false, false, false, false, false], BitVector: [true, true, true, true, true
```

```
[falses(5), trues(5)]
```

this is just an array of arrays. What if we wanted to concatenate two arrays?

```
BitVector: [false, false, false, false, false, true, true, true, true, true]
```

```
[falses(5); trues(5)]
```

vectors in julia are assumed to be column vectors by default, so you can read `[a; b]` as "vertically concatenate `a` and `b`". And we already know how to concatenate horizontally:

```
5×2 BitMatrix:
 0  1
 0  1
 0  1
 0  1
 0  1
```

```
[falses(5) trues(5)]
```

```
5×2 BitMatrix:
 1  0
 1  0
 1  0
 1  0
 1  0
```

```
[falses(5) trues(5)] .== false #another example of broadcasting
```

horizontal concatenation gives you **matrices** (same as type Array, but with more than one dimension)

```
my_matrix = 2×2 Matrix{Int64}:
             1  2
             3  4
```

```
my_matrix = [1 2;
             3 4]
```

```
3
```

```
my_matrix[2, 1]
```

```
5×2 Matrix{Int64}:
 1  1
 2  2
 3  3
 4  4
 5  5
```

```
[1:5 1:5] #read as "horizontally concatenate these two ranges"
```

the library `LinearAlgebra` has most of the tools for generating matrices and performing algebraic operations with vectors and matrices, check the documentation!

dictionaries

```
my_dict =   Dict(:three => true, :two => 2.0, :four => "lol", :one => 1)
```

```
my_dict = Dict(
    :one => 1,
    :two => 2.0,
    :three => true,
    :four => "lol",
)
```

```
1
```

```
my_dict[:one]
```

```
"lol"
```

```
my_dict[:four]
```

tuples & named tuples

```
my_tup =   (2, 3, 4, 5)
```

```
my_tup = (2, 3, 4, 5)
```

```
3
```

```
my_tup[2]
```

```
MethodError: no method matching setindex!(::NTuple{4, Int64}, ::Int64, ::Int64)

  1. top-level scope  @ ⎡Local: 1   [inlined]
```

```
my_tup[3] = 2 #immutable!
```

```
my_named_tup =   (a = 1, b = 2, c = 3)
```

```
my_named_tup = (a = 1, b = 2, c = 3)
```

```
true
```

```
my_named_tup[1] == my_named_tup.a
```

Structs are extensions of tuples, and we can define mutable ones too:

```
mutable struct Peep
    age::Int
    height::Float64
    hobby::String
    dancer::Bool
end
```

```
alejandro =   Peep(30, 1.75, "julia", false)
```

```
alejandro = Peep(30, 1.75, "julia", false)
```

```
"julia"
```

```
alejandro.hobby
```

```
false
```

```
alejandro.dancer
```

functions: we have several options to define them

```
g1 (generic function with 1 method)
```

```
g1(x, y) = x + y
```

which is equivalent to

```
g2 (generic function with 1 method)
```

```
function g2(x, y)
    #block of code
    return x + y
end
```

```
true
```

```
g1(2, 3) == g2(2, 3)
```

keyword arguments (name matters)

```
t (generic function with 1 method)
```

```
function t(x, y; meow="meow", woof="woof")
    x = string(x) #convert x to a string
    return meow*x #string concatenation
end
```

```
"meow1"
```

```
t(1, 2)
```

```
"lol1"
```

```
t(1, 2, meow="lol")
```

functions can also be anonymous:

```
#3 (generic function with 1 method)
```

```
x -> 2*x
```

this is useful in many cases. One such case is when we want to use `filter()` to select only certain elements from an iterable based on some criterion. Our anonymous function will be the criterion, receiving the array element and returning a Boolean value:

```
[2]
```

```
filter( x -> x == 2, [1, 2, 3] )
```

we can return a tuple in a function when we want to return multiple things:

```
(2, 4)
```

```
begin
    function whatever(x, y)
        return (2*x, 2*y)
    end

    q, r = whatever(1, 2) #tuple unpacking
end
```

```
2
```

```
q
```

control flow: if, else, ternary operator

```
2
```

```
x = 3; y = 2
```

```
"yes!"
```

```
if x > y
    "yes!"
else        #can do more complex decision trees with elseif
    "no!"
end
```

if our conditional expression is simple enough, we can express it using the ternary operator:

```
"yes!"
```

```
x > y ? "yes!" : "no!"
```

we use `&&` and `||` as And and Or logical operators:

```
true
```

```
x > y && true
```

```
false
```

```
x < y && true
```

```
true
```

```
x > y || true
```

we can incorporate this for control flow in more complex functions, like so:

```
fact (generic function with 1 method)
```

```
function fact(n::Int)
    n >= 0 || error("n must be non-negative") #easily throw errors
    n == 0 && return 1
    n * fact(n - 1)
end
```

n must be non-negative

1. **error**(::String) @ *error.jl:35*
2. **fact**(::Int64) @ *Other: 2*
3. **top-level scope** @ *Local: 1* [inlined]

```
fact(-1)
```

```
1
```

```
fact(0)
```

```
1
```

```
fact(1)
```

```
6
```

```
fact(3)
```

extra: pipes, macros, random numbers, plotting, etc

Multiple dispatch:

```
f (generic function with 1 method)
```

```
f(x::Number, y::Number) = 2*x + y
```

```
f (generic function with 2 methods)
```

```
f(x::Float64, y::Float64) = 2*x - y
```

```
7
```

```
f(2, 3)
```

```
7.0
```

```
f(2.0, 3)
```

```
1.0
```

```
f(2.0, 3.0)
```

```
using Pipe    , Random    , Distributions    , PlutoUI
```

```
rng =   RandomDevice()
```

```
rng = RandomDevice() #will use computer's entropy pool
```

```
0.5261056186815309
```

```
rand(rng)
```

```
0.7601255691392914
```

```
rand()
```

```
3
```

```
rand(rng, [1,2,3])
```

```
[3, 2]
```

```
rand(rng, [1,2,3], 2)
```

let's sample from a normal distribution

```
-1.8027815310020958
```

```
@pipe Normal(0, 1) |> rand(rng, _)
```

the macro takes our piped code and re-writes it before the code compiles! It re-writes it to this:

```
1.4258583947907941
```

```
rand( rng, Normal(0,1) )
```

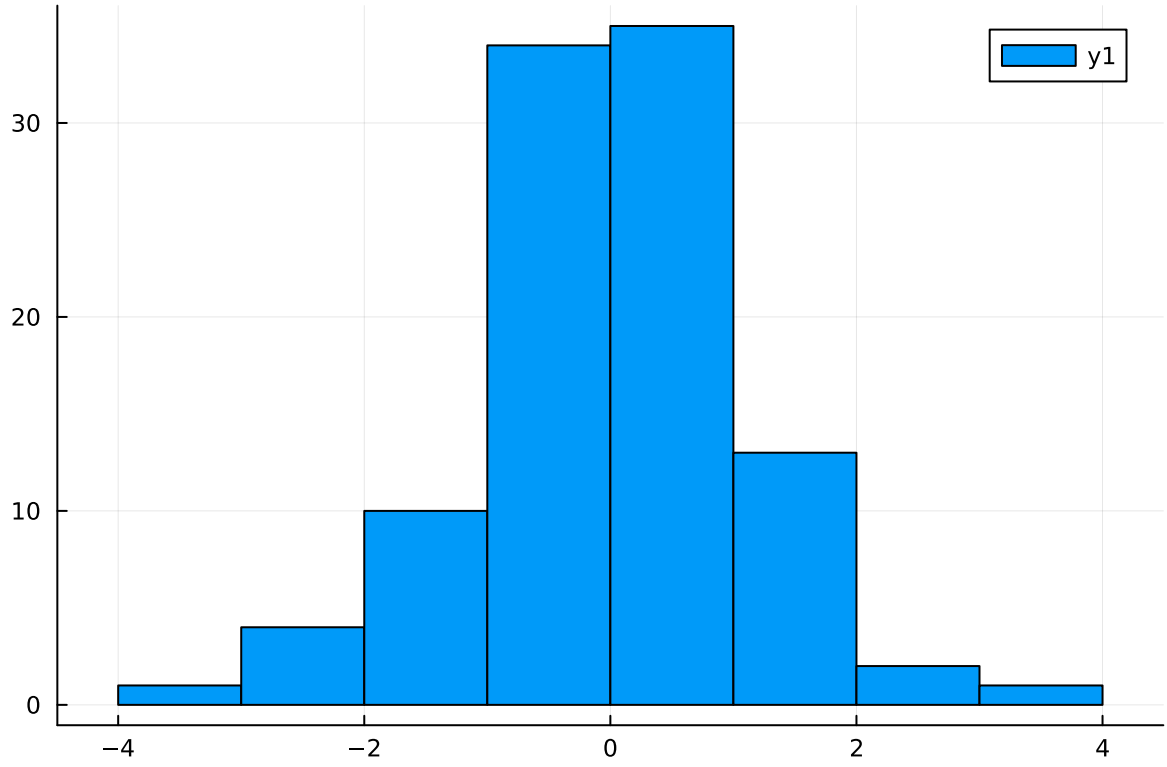
```
100
```

```
random_arr =
  [0.490195, -0.374821, -2.00048, 0.779187, 0.5422, -0.0755416, 0.77303, -0.266699, 0.1067
```

```
random_arr = @pipe Normal(0, 1) |> rand(rng, _, n)
```

time to plot it:

```
using Plots
```



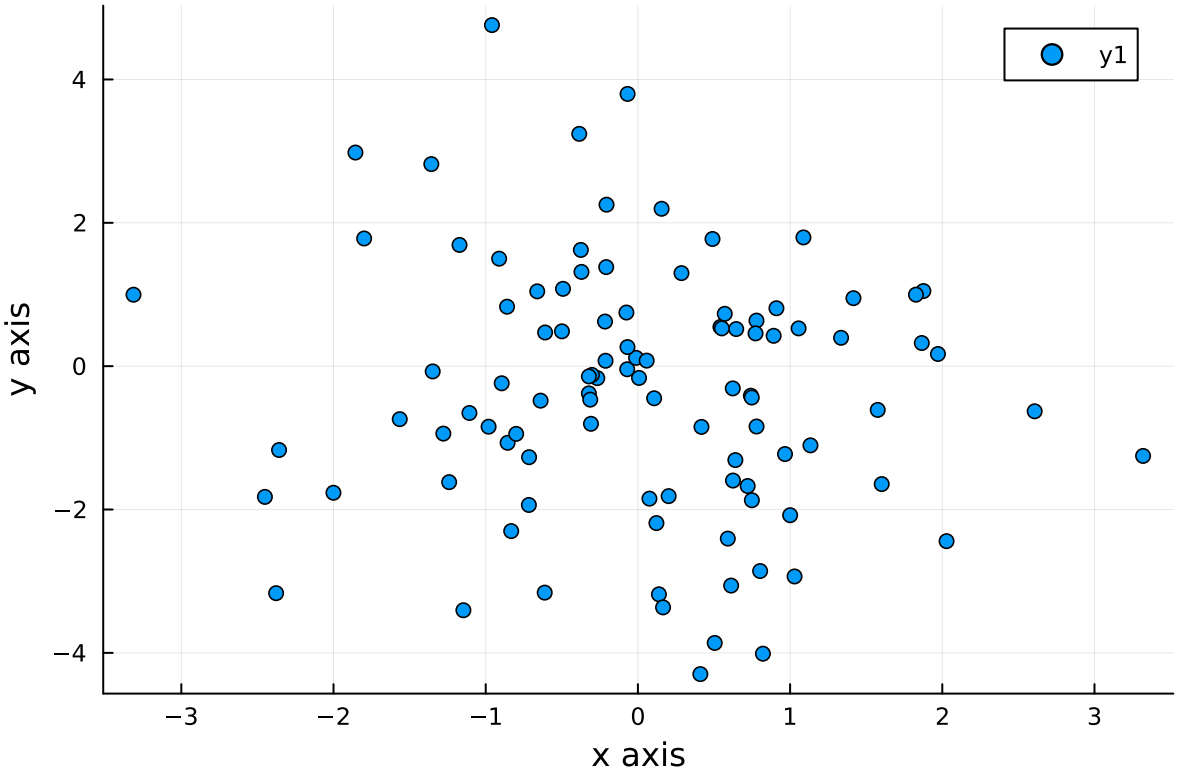

```
histogram(random_arr)
```

more! MORE!

```
@bind n Slider(100:100:1000) #Slider function from PlutoUI library
```

```
random_arr2 =
  [1.77439, 1.6215, -1.76647, 0.635595, 0.547134, 0.748341, 0.457167, -0.166937, -0.447938,
```

```
random_arr2 = @pipe 2 .* Normal(0, 1) |> rand(rng, _, n)
```




```
scatter(
    random_arr, random_arr2,
    xlabel = "x axis", ylabel = "y axis"
)
```

these are the basics, for now. We have made a very superficial treatment of things here, and many things we have not looked at, at all. Data types, for example, is an important subject that requires some time set aside for study. Examples of important subjects we didn't see but you should nevertheless explore: DataFrames, structs, modules, constructors, strings, scope of variables, type conversion and promotion, multithreading... and so much more.

The basics of julia are easy enough that one can get up and running with them. But its large variety of features and the flexibility they bring means this rabbit hole goes deep. For now, this taste of the language should help you, should you choose to embark on a learning journey.

Resources for learning:

1. Julia documentation manual: https://docs.julialang.org/en/v1/
2. Julia documentation tutorials section (https://julialang.org/learning/tutorials/) has a list of good tutorials, like
3. The Julia Express - https://github.com/bkamins/The-Julia-Express
4. A Concise Tutorial - https://syl1.gitbook.io/julia-language-a-concise-tutorial/
5. Julia Academy - https://juliaacademy.com/courses?preview=logged_out