Intro to Julia

Juan Pablo Vielma

June 5, 2019

1 Introduction

Since JuMP is embedded in Julia, knowing some basic Julia is important for learning JuMP. This notebook is designed to provide a minimalist crash course in the basics of Julia. You can find resources that provide a more comprehensive introduction to Julia here.

1.1 How to Print

In Julia, we usually use println() to print

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

```
Hello, World!
```

1.2 Basic Data Types

Integers

```
typeof(1 + -2)
```

```
Int64
```

Floating point numbers

```
typeof(1.2 - 2.3)
```

```
Float64
```

There are also some cool things like an irrational representation of π. To make π (and most other greek letters), type \pi and then press [TAB].

```
π
```

```
π = 3.1415926535897...
```

```
typeof(π)
```

```
Irrational{:π}
```

Julia has native support for complex numbers

```
typeof(2 + 3im)
```

```
Complex{Int64}
```

Double quotes are used for strings

```
typeof("This is Julia")
```

```
String
```

Unicode is fine in strings

```
typeof("π is about 3.1415")
```

```
String
```

Julia symbols provide a way to make human readable unique identifiers.

```
:my_id
typeof(:my_id)
```

```
Symbol
```

1.3 Arithmetic and Equality Testing

Julia is great for math

```
1 + 1
```

```
2
```

Even math involving complex numbers

```
(2 + 1im) * (1 - 2im)
```

```
4 - 3im
```

We can also write things like the following using √ (\sqrt)

```
sin(2π/3) == √3/2
```

```
false
```

Wait. What???

```
sin(2π/3) - √3/2
```

```
1.1102230246251565e-16
```

Let's try again using ≈ (\approx).

```
sin(2π/3) ≈ √3/2
```

```
true
```

Note that this time we used ≈ instead of ==. That is because computers don't use real numbers. They use a discrete representation called floating point. If you aren't careful, this can throw up all manner of issues. For example:

```
1 + 1e-16 == 1
```

```
true
```

It even turns out that floating point numbers aren't associative!

```
(1 + 1e-16) - 1e-16 == 1 + (1e-16 - 1e-16)
```

```
false
```

1.4 Vectors, Matrices and Arrays

Similar to Matlab, Julia has native support for vectors, matrices and tensors; all of which are represented by arrays of different dimensions. Vectors are constructed by comma-separated elements surrounded by square brackets:

```
b = [5, 6]
```

```
2-element Array{Int64,1}:
 5
 6
```

Matrices can by constructed with spaces separating the columns, and semicolons separating the rows:

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

```
2×2 Array{Int64,2}:
 1  2
 3  4
```

We can do linear algebra:

```
x = A \ b
```

```
2-element Array{Float64,1}:
 -4.0
  4.5
```

```
A * x
```

```
2-element Array{Float64,1}:
 5.0
 6.0
```

```
A * x == b
```

```
true
```

Note that when multiplying vectors and matrices, dimensions matter. For example, you can't multiply a vector by a vector:

```
b * b
```

```
Error: MethodError: no method matching *(::Array{Int64,1}, ::Array{Int64,1}
)
Closest candidates are:
  *(::Any, ::Any, !Matched::Any, !Matched::Any...) at operators.jl:502
  *(!Matched::LinearAlgebra.Adjoint{#s576,#s575} where #s575<:Union{DenseAr
ray{T<:Union{Complex{Float32}, Complex{Float64}, Float32, Float64},2}, Rein
terpretArray{T<:Union{Complex{Float32}, Complex{Float64}, Float32, Float64}
```

```
,2,S,A} where S where A<:Union{SubArray{T,N,A,I,true} where I<:Tuple{Abstra
ctUnitRange,Vararg{Any,N} where N} where A<:DenseArray where N where T, Den
seArray}, ReshapedArray{T<:Union{Complex{Float32}, Complex{Float64}, Float3
2, Float64},2,A,MI} where MI<:Tuple{Vararg{SignedMultiplicativeInverse{Int6
4},N} where N} where A<:Union{ReinterpretArray{T,N,S,A} where S where A<:Un
ion{SubArray{T,N,A,I,true} where I<:Tuple{AbstractUnitRange,Vararg{Any,N} w
here N} where A<:DenseArray where N where T, DenseArray} where N where T, S
ubArray{T,N,A,I,true} where I<:Tuple{AbstractUnitRange,Vararg{Any,N} where
N} where A<:DenseArray where N where T, DenseArray}, SubArray{T<:Union{Comp
lex{Float32}, Complex{Float64}, Float32, Float64},2,A,I,L} where L where I<
:Tuple{Vararg{Union{Int64, AbstractRange{Int64}, AbstractCartesianIndex},N}
 where N} where A<:Union{ReinterpretArray{T,N,S,A} where S where A<:Union{S
ubArray{T,N,A,I,true} where I<:Tuple{AbstractUnitRange,Vararg{Any,N} where
N} where A<:DenseArray where N where T, DenseArray} where N where T, Reshap
edArray{T,N,A,MI} where MI<:Tuple{Vararg{SignedMultiplicativeInverse{Int64}
,N} where N} where A<:Union{ReinterpretArray{T,N,S,A} where S where A<:Unio
n{SubArray{T,N,A,I,true} where I<:Tuple{AbstractUnitRange,Vararg{Any,N} whe
re N} where A<:DenseArray where N where T, DenseArray} where N where T, Sub
Array{T,N,A,I,true} where I<:Tuple{AbstractUnitRange,Vararg{Any,N} where N}
 where A<:DenseArray where N where T, DenseArray} where N where T, DenseArr
ay}} where #s576, ::Union{DenseArray{S,1}, ReinterpretArray{S,1,S,A} where
S where A<:Union{SubArray{T,N,A,I,true} where I<:Tuple{AbstractUnitRange,Va
rarg{Any,N} where N} where A<:DenseArray where N where T, DenseArray}, Resh
apedArray{S,1,A,MI} where MI<:Tuple{Vararg{SignedMultiplicativeInverse{Int6
4},N} where N} where A<:Union{ReinterpretArray{T,N,S,A} where S where A<:Un
ion{SubArray{T,N,A,I,true} where I<:Tuple{AbstractUnitRange,Vararg{Any,N} w
here N} where A<:DenseArray where N where T, DenseArray} where N where T, S
ubArray{T,N,A,I,true} where I<:Tuple{AbstractUnitRange,Vararg{Any,N} where
N} where A<:DenseArray where N where T, DenseArray}, SubArray{S,1,A,I,L} wh
ere L where I<:Tuple{Vararg{Union{Int64, AbstractRange{Int64}, AbstractCart
esianIndex},N} where N} where A<:Union{ReinterpretArray{T,N,S,A} where S wh
ere A<:Union{SubArray{T,N,A,I,true} where I<:Tuple{AbstractUnitRange,Vararg
{Any,N} where N} where A<:DenseArray where N where T, DenseArray} where N w
here T, ReshapedArray{T,N,A,MI} where MI<:Tuple{Vararg{SignedMultiplicative
Inverse{Int64},N} where N} where A<:Union{ReinterpretArray{T,N,S,A} where S
 where A<:Union{SubArray{T,N,A,I,true} where I<:Tuple{AbstractUnitRange,Var
arg{Any,N} where N} where A<:DenseArray where N where T, DenseArray} where
N where T, SubArray{T,N,A,I,true} where I<:Tuple{AbstractUnitRange,Vararg{A
ny,N} where N} where A<:DenseArray where N where T, DenseArray} where N whe
re T, DenseArray}}) where {T<:Union{Complex{Float32}, Complex{Float64}, Flo
at32, Float64}, S} at /buildworker/worker/package_linux64/build/usr/share/j
ulia/stdlib/v1.0/LinearAlgebra/src/matmul.jl:98
  *(!Matched::LinearAlgebra.Adjoint{#s576,#s575} where #s575<:LinearAlgebra
.AbstractTriangular where #s576, ::AbstractArray{T,1} where T) at /buildwor
ker/worker/package_linux64/build/usr/share/julia/stdlib/v1.0/LinearAlgebra/
src/triangular.jl:1805
  ...
```

But multiplying transposes works:

```
@show b' * b
```

```
b' * b = 61
```

```
@show b * b';
```

```
b * b' = [25 30; 30 36]
```

1.5 Tuples

Julia makes extensive use of a simple data structure called Tuples. Tuples are immutable collections of values. For example,

```
t = ("hello", 1.2, :foo)
```

```
("hello", 1.2, :foo)
```

```
typeof(t)
```

```
Tuple{String,Float64,Symbol}
```

Tuples can be accessed by index, similar to arrays,

```
t[2]
```

```
1.2
```

And can be "unpacked" like so,

```
a, b, c = t
b
```

```
1.2
```

The values can also be given names, which is a convenient way of making light-weight data structures.

```
t = (word="hello", num=1.2, sym=:foo)
```

```
(word = "hello", num = 1.2, sym = :foo)
```

Then values can be accessed using a dot syntax,

```
t.word
```

```
"hello"
```

1.6 Dictionaries

Similar to Python, Julia has native support for dictionaries. Dictionaries provide a very generic way of mapping keys to values. For example, a map of integers to strings,

```
d1 = Dict(1 => "A", 2 => "B", 4 => "D")
```

```
Dict{Int64,String} with 3 entries:
  4 => "D"
  2 => "B"
  1 => "A"
```

Looking up a values uses the bracket syntax,

```
d1[2]
```

```
"B"
```

Dictionaries support non-integer keys and can mix data types,

```
Dict("A" => 1, "B" => 2.5, "D" => 2 - 3im)
```