Intro to Julia

Juan Pablo Vielma

May 28, 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 $\sqrt{}$ (\sqrt)

```
sin(2π/3) == √3/2
```

```
false
```

Wait. What???

```
sin(2π/3) - √3/2
```

```
1.1102230246251565e-16
```

Let's try again using $\approx$ (\approx).

```
sin(2π/3) ≈ √3/2
```

```
true
```

Note that this time we used $\approx$ 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)
```

```
Dict{String,Number} with 3 entries:
  "B" => 2.5
  "A" => 1
  "D" => 2-3im
```

Dictionaries can be nested

```
d2 = Dict("A" => 1, "B" => 2, "D" => Dict(:foo => 3, :bar => 4))
```

```
Dict{String,Any} with 3 entries:
  "B" => 2
  "A" => 1
  "D" => Dict(:bar=>4,:foo=>3)
```

```
d2["B"]
```

```
2
```

```
d2["D"][:foo]
```

```
3
```

1.7 For-Each Loops

Julia has native support for for-each style loops with the syntax `for <value> in <collection>` `end`.

```
for i in 1:5
    println(i)
end
```

```
1
2
3
4
5
```

```
for i in [1.2, 2.3, 3.4, 4.5, 5.6]
    println(i)
end
```

```
1.2
2.3
3.4
4.5
5.6
```

This for-each loop also works with dictionaries.

```
for (key, value) in Dict("A" => 1, "B" => 2.5, "D" => 2 - 3im)
    println("$key:  $value")
end
```

```
B: 2.5
A: 1
D: 2 - 3im
```

Note that in contrast to vector languages like Matlab and R, loops do not result in a significant performance degradation in Julia.

1.8 Control Flow

Julia control flow is similar to Matlab, using the keywords `if-elseif-else-end`, and the logical operators `||` and `&&` for *or* and *and* respectively.

```
i = 10
for i in 0:3:15
    if i < 5
        println("$(i)is less than 5")
    elseif i < 10
        println("$(i)is less than 10")
    else
        if i == 10
            println("the value is 10")
        else
            println("$(i)is bigger than 10")
        end
    end
end
```

```
0 is less than 5
3 is less than 5
6 is less than 10
9 is less than 10
12 is bigger than 10
15 is bigger than 10
```

1.9 Comprehensions

Similar to languages like Haskell and Python, Julia supports the use of simple loops in the construction of arrays and dictionaries, called comprehenions.

A list of increasing integers,

```
[i for i in 1:5]
```

```
5-element Array{Int64,1}:
 1
 2
 3
 4
 5
```

Matrices can be built by including multiple indices,

```
[i*j for i in 1:5, j in 5:10]
```

```
5×6 Array{Int64,2}:
  5   6   7   8   9  10
 10  12  14  16  18  20
 15  18  21  24  27  30
 20  24  28  32  36  40
 25  30  35  40  45  50
```

Conditional statements can be used to filter out some values,

```
[i for i in 1:10 if i%2 == 1]
```

```
5-element Array{Int64,1}:
 1
 3
 5
 7
 9
```

A similar syntax can be used for building dictionaries

```
Dict("$i" => i for i in 1:10 if i%2 == 1)
```

```
Dict{String,Int64} with 5 entries:
  "1" => 1
  "5" => 5
  "7" => 7
  "9" => 9
  "3" => 3
```

1.10 Functions

A simple function is defined as follows,

```
function print_hello()
    println("hello")
end
print_hello()
```

```
hello
```

Arguments can be added to a function,

```
function print_it(x)
    println(x)
end
print_it("hello")
```

```
hello
```

```
print_it(1.234)
```

```
1.234
```

```
print_it(:my_id)
```

```
my_id
```

Optional keyword arguments are also possible

```
function print_it(x; prefix="value:")
    println("$(prefix)$x")
end
print_it(1.234)
```

```
value: 1.234
```

```
print_it(1.234, prefix="val:")
```

```
val: 1.234
```

The keyword `return` is used to specify the return values of a function.

```
function mult(x; y=2.0)
    return x * y
end
mult(4.0)
```

```
8.0
```

```
mult(4.0, y=5.0)
```

```
20.0
```

1.11 Other notes on types

Usually, specifing types is not required to use Julia. However, it can be helpful to understand the basics of Julia types for debugging. For example this list has a type of `Array{Int64,1}` indicating that it is a one dimensional array of integer values.

```
[1, 5, -2, 7]
```

```
4-element Array{Int64,1}:
  1
  5
 -2
  7
```

In this example, the decimal values lead to a one dimensional array of floating point values, i.e. `Array{Float64,1}`. Notice that the integer `7` is promoted to a `Float64`, because all elements in the array need share a common type.

```
[1.0, 5.2, -2.1, 7]
```

```
4-element Array{Float64,1}:
  1.0
  5.2
 -2.1
  7.0
```

1.12 Mutable vs immutable objects

Some types in Julia are *mutable*, which means you can change the values inside them. A good example is an array. You can modify the contents of an array without having to make a new array. In contrast, types like `Float64` are *immutable*. You can't modify the contents of a `Float64`. This is something to be aware of when passing types into functions. For example:

```
function mutability_example(mutable_type::Vector{Int}, immutable_type::Int)
    mutable_type[1] += 1
    immutable_type += 1
    return
end

mutable_type = [1, 2, 3]
immutable_type = 1
```

```
mutability_example(mutable_type, immutable_type)
```

```
println("mutable_type:  $(mutable_type)")
```

```
mutable_type: [2, 2, 3]
```

```
println("immutable_type:  $(immutable_type)")
```

```
immutable_type: 1
```

Because `Vector{Int}` is a mutable type, modifying the variable inside the function changed the value outside of the function. In constrast, the change to `immutable_type` didn't modify the value outside the function. You can check mutability with the `isimmutable` function.

```
@show isimmutable([1, 2, 3])
```

```
isimmutable([1, 2, 3]) = false
```

```
@show isimmutable(1);
```

```
isimmutable(1) = true
```

1.13 Using Packages and the Package Manager

No matter how wonderful Julia's base language is, at some point you will want to use an extension package. Some of these are built-in, for example random number generation is available in the `Random` package in the standard library. These packages are loaded with the commands `using` and `import`.

```
using Random
[rand() for i in 1:10]
```

```
10-element Array{Float64,1}:
 0.12068932965998669
 0.7794263583810461
 0.8508881114818132
 0.24706719132999377
 0.9069377043911446
 0.23664001628942133
 0.2993202013041403
 0.6482771055726524
 0.8734699046200227
 0.8725673887333607
```

The Package Manager is used to install packages that are not part of Julia's standard library. For example the following can be used to install JuMP,

```
using Pkg
Pkg.add("JuMP")
```

For a complete list of registed Julia packages see the package listing at https://pkg.julialang.org/. From time to you may wish to use a Julia package that is not registered. In this case a git repository URL can be used to install the package.

```
using Pkg
Pkg.add("https://github.com/user-name/MyPackage.jl.git")
```

Note that for clarity this example uses the package manager `Pkg`. Julia 1.0 includes an interactive package manager that can be accessed using `]`. This video gives an overview of using the interactive package manager environment. The state of installed packages can also be saved in two files: `Project.toml` and `Manifest.toml`. If these files are stored in the same directory than a notebook, the state of the packages can be recovered by running

```
import Pkg
Pkg.activate(@__DIR__)
Pkg.instantiate()
```

1.14 HELP!

Julia 1.0 includes a help mode that can be accessed using `?`. Entering any object (e.g. function, type, struct, ...) into the help mode will show its documentation, if any is available.

1.15 Some Common Gotchas

1.15.1 MethodError

A common error in Julia is `MethodError`, which indicates that the function is not defined for the given value. For example, by default the `ceil` function is not defined for complex numbers. The "closest candidates" list suggest some Julia types that the function is defined for.

```
ceil(1.2 + 2.3im)
```

```
Error: MethodError: no method matching ceil(::Complex{Float64})
Closest candidates are:
  ceil(!Matched::Type{BigInt}, !Matched::BigFloat) at mpfr.jl:257
  ceil(!Matched::Missing) at missing.jl:108
  ceil(!Matched::Missing, !Matched::Integer) at missing.jl:108
  ...
```