QuickIntroduction

September 18, 2016

0.1 A Quick Introduction to Basic Julia

This quick introduction assumes that you have basic knowledge of some scripting language and provides an example of the Julia syntax. So before we explain anything, let's just treat it like a scripting language, take a head-first dive into Julia, and see what happens.

You'll notice that, given the right syntax, almost everything will "just work". There will be some peculiarities, and these we will be the facts which we will study in much more depth. Usually, these oddies/differences from other scripting languages are "the source of Julia's power".

0.1.1 Array Syntax

The array syntax is similar to MATLAB's conventions.

```
In [11]: a = Vector{Float64}(5) # Create a length 5 Vector (dimension 1 array) of F

         a = [1;2;3;4;5] # Create the column vector [1 2 3 4 5]

         a = [1 2 3 4] # Create the row vector [1 2 3 4]

         a[3] = 2 # Change the third element of a (using linear indexing) to 2

         b = Matrix{Float64}(4,2) # Define a Matrix of Float64's of size (4,2)

         c = Array{Float64,4}((4,5,6,7)) # Define a (4,5,6,7) array of Float64's

         mat    = [1 2 3 4
                   3 4 5 6
                   4 4 4 6
                   3 3 3 3] #Define the matrix inline

         mat[1,2] = 4 # Set element (1,2) (row 1, column 2) to 4

         mat

Out[11]: 4×4 Array{Int64,2}:
          1  4  3  4
          3  4  5  6
          4  4  4  6
          3  3  3  3
```

Note that, in the console (called the REPL), you can use `;` to surpress the output. In a script this is done automatically. Note that the "value" of an array is its pointer to the memory location. This means that arrays which are set equal affect the same values:

```
In [12]: a = [1;3;4]
         b = a
         b[1] = 10
         a

Out[12]: 3-element Array{Int64,1}:
          10
           3
           4
```

To set an array equal to the values to another array, use copy

```
In [13]: a = [1;4;5]
         b = copy(a)
         b[1] = 10
         a

Out[13]: 3-element Array{Int64,1}:
          1
          4
          5
```

We can also make an array of a similar size and shape via the function `similar`, or make an array of zeros/ones with `zeros` or `ones` respectively:

```
In [14]: c = similar(a)
         d = zeros(a)
         e = ones(a)
         println(c); println(d); println(e)

[786432,0,0]
[0,0,0]
[1,1,1]
```

Note that arrays can be index'd by arrays:

```
In [15]: a[1:2]

Out[15]: 2-element Array{Int64,1}:
          1
          4
```

Arrays can be of any type, specified by the type parameter. One interesting thing is that this means that arrays can be of arrays:

```
In [12]: a = Vector{Vector{Float64}}(3)
         a[1] = [1;2;3]
         a[2] = [1;2]
         a[3] = [3;4;5]
         a

Out[12]: 3-element Array{Array{Float64,1},1}:
          [1.0,2.0,3.0]
          [1.0,2.0]
          [3.0,4.0,5.0]
```

Question 1 Can you explain the following behavior? Julia's community values consistancy of the rules, so all of the behavior is deducible from simple rules. (Hint: I have noted all of the rules involved here).

```
In [14]: b = a
         b[1] = [1;4;5]
         a

Out[14]: 3-element Array{Array{Float64,1},1}:
          [1.0,4.0,5.0]
          [1.0,2.0]
          [3.0,4.0,5.0]
```

To fix this, there is a recursive copy function: `deepcopy`

```
In [16]: b = deepcopy(a)
         b[1] = [1;2;3]
         a

Out[16]: 3-element Array{Array{Float64,1},1}:
          [1.0,4.0,5.0]
          [1.0,2.0]
          [3.0,4.0,5.0]
```

For high performance, Julia provides mutating functions. These functions change the input values that are passed in, instead of returning a new value. By convention, mutating functions tend to be defined with a `!` at the end and tend to mutate their first argument. An example of a mutating function in `scale!` which scales an array by a scalar (or array)

```
In [19]: a = [1;6;8]
         scale!(a,2) # a changes

Out[19]: 3-element Array{Int64,1}:
           2
          12
          16
```

The purpose of mutating functions is that they allow one to reduce the number of memory allocations which is crucial for achiving high performance.

0.2 Control Flow

Control flow in Julia is pretty standard. You have your basic for and while loops, and your if statements. There's more in the documentation.

```
In [16]: for i=1:5 #for i goes from 1 to 5
             println(i)
         end

         t = 0
         while t<5
             println(t)
             t+=1 # t = t + 1
         end

         school = :UCI

         if school==:UCI
             println("ZotZotZot")
         else
             println("Not even worth discussing.")
         end
```

```
1
2
3
4
5
0
1
2
3
4
ZotZotZot
```

One interesting feature about Julia control flow is that we can write multiple loops in one line:

```
In [17]: for i=1:2,j=2:4
             println(i*j)
         end
```

```
2
3
4
4
6
8
```

0.3 Function Syntax

```
In [20]: f(x,y) = 2x+y # Create an inline function
```

```
Out[20]: f (generic function with 1 method)
```

```
In [22]: f(1,2) # Call the function
```

```
Out[22]: 4
```

```
In [24]: function f(x)
           x+2
         end # Long form definition
```

```
WARNING: Method definition f(Any) in module Main at In[23]:2 overwritten at In[24]:


Out[24]: f (generic function with 2 methods)
```

By default, Julia functions return the last value computed within them.

```
In [26]: f(2)
```

```
Out[26]: 4
```

A key feature of Julia is multiple dispatch. Notice here that there is "one function", `f`, with two methods. Methods are the actionable parts of a function. Here, there is one method defined as `(::Any,::Any)` and `(::Any)`, meaning that if you give `f` two values then it will call the first method, and if you give it one value then it will call the second method.

Multiple dispatch works on types. To define a dispatch on a type, use the `::Type` signifier:

```
In [35]: f(x::Int,y::Int) = 3x+2y
```

```
WARNING: Method definition f(Int64, Int64) in module Main at In[28]:1 overwritten a


Out[35]: f (generic function with 4 methods)
```

Julia will dispatch onto the strictest acceptible type signature.

```
In [30]: f(2,3) # 3x+2y
```

```
Out[30]: 12
```

```
In [32]: f(2.0,3) # 2x+y since 2.0 is not an Int
```

```
Out[32]: 7.0
```

Types in signatures can be parametric. For example, we can define a method for "two values are passed in, both Numbers and having the same type". Note that `<:` means "a subtype of".

```
In [3]: f{T<:Number}(x::T,y::T) = 4x+10y
```

```
Out[3]: f (generic function with 1 method)

In [37]: f(2,3) # 3x+2y since (::Int,::Int) is stricter

Out[37]: 12

In [4]: f(2.0,3.0) # 4x+10y

Out[4]: 38.0
```

Note that type parameterizations can have as many types as possible, and do not need to declare a supertype. For example, we can say that there is an `x` which must be a Number, while `y` and `z` must match types:

```
In [5]: f{T<:Number,T2}(x::T,y::T2,z::T2) = 5x + 5y + 5z

Out[5]: f (generic function with 2 methods)
```

We will go into more depth on multiple dispatch later since this is the core design feature of Julia. The key feature is that Julia functions specialize on the types of their arguments. This means that `f` is a separately compiled function for each method (and for parametric types, each possible method). The first time it is called it will compile.

Question 2 Can you explain these timings?

```
In [13]: f(x,y,z,w) = x+y+z+w
         @time f(1,1,1,1)
         @time f(1,1,1,1)
         @time f(1,1,1,1)
         @time f(1,1,1,1.0)
         @time f(1,1,1,1.0)

  0.003228 seconds (257 allocations: 13.354 KB)
  0.000001 seconds (3 allocations: 144 bytes)
  0.000001 seconds (3 allocations: 144 bytes)
  0.002551 seconds (274 allocations: 14.007 KB)
  0.000001 seconds (4 allocations: 160 bytes)


WARNING: Method definition f(Any, Any, Any, Any) in module Main at In[12]:1 overwri


Out[13]: 4.0
```

Note that functions can also feature optional arguments:

```
In [42]: function test_function(x,y;z=0) #z is an optional argument
           if z==0
             return x+y,x*y #Return a tuple
           else
           return x*y*z,x+y+z #Return a different tuple
           #whitespace is optional
           end #End if statement
         end #End function definition

Out[42]: test_function (generic function with 1 method)
```

Here, if z is not specified, then it's 0.

```
In [45]: x,y = test_function(1,2)

Out[45]: (3,2)

In [46]: x,y = test_function(1,2;z=3)

Out[46]: (6,6)
```

Notice that we also featured multiple return values.

```
In [47]: println(x); println(y)

6
6
```

The return type for multiple return values is a Tuple. The syntax for a tuple is `(x,y,z,...)` or inside of functions you can use the shorthand `x,y,z,...` as shown.

Note that functions in Julia are "first-class". This means that functions are just a type themselves. Therefore functions can make functions, you can store functions as variables, pass them as variables, etc. For example:

```
In [7]: function function_playtime(x) #z is an optional argument
            y = 2+x
            function test()
                2y # y is defined in the previous scope, so it's available here
            end
            z = test() * test()
            return z,test
        end #End function definition
        z,test = function_playtime(2)

WARNING: Method definition function_playtime(Any) in module Main at In[6]:2 overwri


Out[7]: (64,test)
```

```
In [14]: test()

Out[14]: 8
```

Notice that `test()` does not get passed in `y` but knows what `y` is. This is due to the function scoping rules: an inner function can know the variables defined in the same scope as the function. This rule is recursive, leading us to the conclusion that the top level scope is global. Yes, that means

```
In [18]: a = 2

Out[18]: 2
```

defines a global variable. We will go into more detail on this.

Lastly we show the anonymous function syntax. This allows you to define a function inline.

```
In [20]: g = (x,y) -> 2x+y

Out[20]: (::#5) (generic function with 1 method)
```

Unlike named functions, `g` is simply a function in a variable and can be overwritten at any time:

```
In [21]: g = (x) -> 2x

Out[21]: (::#7) (generic function with 1 method)
```

An anonymous function cannot have more than 1 dispatch. However, as of v0.5, they are compiled and thus do not have any performance disadvantages from named functions.

0.4 Type Declaration Syntax

A type is what in many other languages is an "object". If that is a foreign concept, thing of a type as a thing which has named components. A type is the idea for what the thing is, while an instantiation of the type is a specific one. For example, you can think of a car as having an make and a model. So that means a Toyota RAV4 is an instantiation of the car type.

In Julia, we would define the car type as follows:

```
In [48]: type Car
             make
             model
         end
```

We could then make the instance of a car as follows:

```
In [49]: mycar = Car("Toyota","Rav4")

Out[49]: Car("Toyota","Rav4")
```

Here I introduced the string syntax for Julia which uses "..." (like most other languages, I'm glaring at you MATLAB). I can grab the "fields" of my type using the `.` syntax:

```
In [51]: mycar.make

Out[51]: "Toyota"
```

To "enhance Julia's performance", one usually likes to make the typing stricter. For example, we can define a WorkshopParticipant (notice the convention for types is capital letters, CamelCase) as having a name and a field. The name will be a string and the field will be a Symbol type, (defined by :Symbol, which we will go into plenty more detail later).

```
In [52]: type WorkshopParticipant
             name::String
             field::Symbol
         end
         tony = WorkshopParticipant("Tony",:physics)

Out[52]: WorkshopParticipant("Tony",:physics)
```

As with functions, types can be set "parametrically". For example, we can have an StaffMember have a name and a field, but also an age. We can allow this age to be any Number type as follows:

```
In [1]: type StaffMember{T<:Number}
            name::String
            field::Symbol
            age::T
        end
        ter = StaffMember("Terry",:football,17)

Out[1]: StaffMember{Int64}("Terry",:football,17)
```

The rules for parametric typing is the same as for functions. Note that most of Julia's types, like Float64 and Int, are natively defined in Julia in this manner. This means that there's no limit for user defined types, only your imagination. Indeed, many of Julia's features first start out as a prototyping package before it's ever moved into Base (the Julia library that ships as the Base module in every installation).

Lastly, there exist abstract types. These types cannot be instantiated but are used to build the type hierarchy. You've already seen one abstract type, Number. We can define one for Person using the Abstract keyword

```
In [15]: abstract Person
```

Then we can set types as a subtype of person

```
In [16]: type Student <: Person
             grade
         end
```

Question 3 Can you make a function `k(x)` where `x` has to be a Person?

Another "version" of type is `immutable`. When one uses `immutable`, the fields of the type cannot be changed. However, Julia will automatically stack allocate immutable types, whereas standard types are heap allocated. If this is unfamiliar terminology, then think of this as meaning that immutable types are able to be stored closer to the CPU and have less cost for memory access (this is a detail not present in many scripting languages). Many things like Julia's built-in Number types are defined as `immutable` in order to give good performance.

```
In [63]: immutable Field
             name
             school
         end
         ds = Field(:DataScience,[:PhysicalScience;:ComputerScience])

WARNING: Method definition (::Type{Main.Field})(Any, Any) in module Main at In[58]:


Out[63]: Field(:DataScience,Symbol[:PhysicalScience,:ComputerScience])
```

Question 4 Can you explain this interesting quirk? Thus Field is immutable, meaning that `ds.name` and `ds.school` cannot be changed:

```
In [64]: ds.name = :ComputationalStatistics


        LoadError: type Field is immutable
    while loading In[64], in expression starting on line 1
```

However, the following is allowed:

```
In [65]: push!(ds.school,:BiologicalScience)
         ds.school

Out[65]: 3-element Array{Symbol,1}:
          :PhysicalScience
          :ComputerScience
          :BiologicalScience
```

(Hint: recall that an array is not the values itself, but a pointer to the memory of the values)

One important detail in Julia is that everything is a type (and every piece of code is an Expression type, more on this later). Thus functions are also types, which we can access the fields of. Not only is everything compiled down to C, but all of the "C parts" are always accessible. For example, we can, if we so choose, get a function pointer:

```
In [46]: foo(x) = 2x
         first(methods(foo)).lambda_template.fptr

Out[46]: Ptr{Void} @0x0000000000000000
```

0.5 Documentation and "Hunting"

The main source of information is the Julia Documentation. Julia also provides lots of built-in documentation and ways to find out what's going on. The number of tools for "hunting down what's going on / available" is too numerous to explain in full detail here, so instead this will just touch on what's important. For example, the ? gets you to the documentation for a type, function, etc.

```
In [23]: ?copy

search: copy copy! copysign deepcopy unsafe_copy! cospi complex Complex



Out[23]:

copy(x)
```

Create a shallow copy of `x`: the outer structure is copied, but not all internal values. For example, copying an array produces a new array with identically-same elements as the original.

To find out what methods are available, we can use the `methods` function. For example, let's see how + is defined:

```
In [25]: methods(+)

Out[25]: # 163 methods for generic function "+":
         +(x::Bool, z::Complex{Bool}) at complex.jl:136
         +(x::Bool, y::Bool) at bool.jl:48
         +(x::Bool) at bool.jl:45
         +{T<:AbstractFloat}(x::Bool, y::T) at bool.jl:55
         +(x::Bool, z::Complex) at complex.jl:143
         +(x::Bool, A::AbstractArray{Bool,N<:Any}) at arraymath.jl:91
         +(x::Float32, y::Float32) at float.jl:239
         +(x::Float64, y::Float64) at float.jl:240
         +(z::Complex{Bool}, x::Bool) at complex.jl:137
         +(z::Complex{Bool}, x::Real) at complex.jl:151
         +(a::Float16, b::Float16) at float16.jl:136
         +(x::Char, y::Integer) at char.jl:40
         +(c::BigInt, x::BigFloat) at mpfr.jl:240
```

```
+(a::BigInt, b::BigInt, c::BigInt, d::BigInt, e::BigInt) at gmp.jl:298
+(a::BigInt, b::BigInt, c::BigInt, d::BigInt) at gmp.jl:291
+(a::BigInt, b::BigInt, c::BigInt) at gmp.jl:285
+(x::BigInt, y::BigInt) at gmp.jl:255
+(x::BigInt, c::Union{UInt16,UInt32,UInt64,UInt8}) at gmp.jl:310
+(x::BigInt, c::Union{Int16,Int32,Int64,Int8}) at gmp.jl:326
+(a::BigFloat, b::BigFloat, c::BigFloat, d::BigFloat, e::BigFloat) at mpfr
+(a::BigFloat, b::BigFloat, c::BigFloat, d::BigFloat) at mpfr.jl:381
+(a::BigFloat, b::BigFloat, c::BigFloat) at mpfr.jl:375
+(x::BigFloat, c::BigInt) at mpfr.jl:236
+(x::BigFloat, y::BigFloat) at mpfr.jl:205
+(x::BigFloat, c::Union{UInt16,UInt32,UInt64,UInt8}) at mpfr.jl:212
+(x::BigFloat, c::Union{Int16,Int32,Int64,Int8}) at mpfr.jl:220
+(x::BigFloat, c::Union{Float16,Float32,Float64}) at mpfr.jl:228
+{T}(B::BitArray{2}, J::UniformScaling{T}) at linalg/uniformscaling.jl:38
+(a::Base.Pkg.Resolve.VersionWeights.VWPreBuildItem, b::Base.Pkg.Resolve.V
+(a::Base.Pkg.Resolve.VersionWeights.VWPreBuild, b::Base.Pkg.Resolve.Versi
+(a::Base.Pkg.Resolve.VersionWeights.VersionWeight, b::Base.Pkg.Resolve.Ve
+(a::Base.Pkg.Resolve.MaxSum.FieldValues.FieldValue, b::Base.Pkg.Resolve.M
+(x::Base.Dates.CompoundPeriod, y::Base.Dates.CompoundPeriod) at dates/per
+(x::Base.Dates.CompoundPeriod, y::Base.Dates.Period) at dates/periods.jl:
+(x::Base.Dates.CompoundPeriod, y::Base.Dates.TimeType) at dates/periods.j
+(dt::DateTime, z::Base.Dates.Month) at dates/arithmetic.jl:37
+(dt::DateTime, y::Base.Dates.Year) at dates/arithmetic.jl:13
+(x::DateTime, y::Base.Dates.Period) at dates/arithmetic.jl:64
+(x::Date, y::Base.Dates.Day) at dates/arithmetic.jl:62
+(x::Date, y::Base.Dates.Week) at dates/arithmetic.jl:60
+(dt::Date, z::Base.Dates.Month) at dates/arithmetic.jl:43
+(dt::Date, y::Base.Dates.Year) at dates/arithmetic.jl:17
+(y::AbstractFloat, x::Bool) at bool.jl:57
+{T<:Union{Int128,Int16,Int32,Int64,Int8,UInt128,UInt16,UInt32,UInt64,UInt
+(x::Integer, y::Ptr) at pointer.jl:108
+(z::Complex, w::Complex) at complex.jl:125
+(z::Complex, x::Bool) at complex.jl:144
+(x::Real, z::Complex{Bool}) at complex.jl:150
+(x::Real, z::Complex) at complex.jl:162
+(z::Complex, x::Real) at complex.jl:163
+(x::Rational, y::Rational) at rational.jl:199
+(x::Integer, y::Char) at char.jl:41
+{N}(i::Integer, index::CartesianIndex{N}) at multidimensional.jl:58
+(c::Union{UInt16,UInt32,UInt64,UInt8}, x::BigInt) at gmp.jl:314
+(c::Union{Int16,Int32,Int64,Int8}, x::BigInt) at gmp.jl:327
+(c::Union{UInt16,UInt32,UInt64,UInt8}, x::BigFloat) at mpfr.jl:216
+(c::Union{Int16,Int32,Int64,Int8}, x::BigFloat) at mpfr.jl:224
+(c::Union{Float16,Float32,Float64}, x::BigFloat) at mpfr.jl:232
+(x::Irrational, y::Irrational) at irrationals.jl:88
+(x::Number) at operators.jl:115
+{T<:Number}(x::T, y::T) at promotion.jl:255
```

```
+(x::Number, y::Number) at promotion.jl:190
+(r1::OrdinalRange, r2::OrdinalRange) at operators.jl:505
+{T<:AbstractFloat}(r1::FloatRange{T}, r2::FloatRange{T}) at operators.jl:
+{T<:AbstractFloat}(r1::LinSpace{T}, r2::LinSpace{T}) at operators.jl:531
+(r1::Union{FloatRange,LinSpace,OrdinalRange}, r2::Union{FloatRange,LinSpa
+(x::Ptr, y::Integer) at pointer.jl:106
+(A::BitArray, B::BitArray) at bitarray.jl:1042
+(A::Array{T<:Any,2}, B::Diagonal) at linalg/special.jl:121
+(A::Array{T<:Any,2}, B::Bidiagonal) at linalg/special.jl:121
+(A::Array{T<:Any,2}, B::Tridiagonal) at linalg/special.jl:121
+(A::Array{T<:Any,2}, B::SymTridiagonal) at linalg/special.jl:130
+(A::Array{T<:Any,2}, B::Base.LinAlg.AbstractTriangular) at linalg/special
+(A::Array, B::SparseMatrixCSC) at sparse/sparsematrix.jl:1711
+{P<:Union{Base.Dates.CompoundPeriod,Base.Dates.Period}}(x::Union{Base.Res
+(A::AbstractArray{Bool,N<:Any}, x::Bool) at arraymath.jl:90
+(A::SymTridiagonal, B::SymTridiagonal) at linalg/tridiag.jl:96
+(A::Tridiagonal, B::Tridiagonal) at linalg/tridiag.jl:494
+(A::UpperTriangular, B::UpperTriangular) at linalg/triangular.jl:357
+(A::LowerTriangular, B::LowerTriangular) at linalg/triangular.jl:358
+(A::UpperTriangular, B::Base.LinAlg.UnitUpperTriangular) at linalg/triang
+(A::LowerTriangular, B::Base.LinAlg.UnitLowerTriangular) at linalg/triang
+(A::Base.LinAlg.UnitUpperTriangular, B::UpperTriangular) at linalg/triang
+(A::Base.LinAlg.UnitLowerTriangular, B::LowerTriangular) at linalg/triang
+(A::Base.LinAlg.UnitUpperTriangular, B::Base.LinAlg.UnitUpperTriangular)
+(A::Base.LinAlg.UnitLowerTriangular, B::Base.LinAlg.UnitLowerTriangular)
+(A::Base.LinAlg.AbstractTriangular, B::Base.LinAlg.AbstractTriangular) at
+(Da::Diagonal, Db::Diagonal) at linalg/diagonal.jl:110
+(A::Bidiagonal, B::Bidiagonal) at linalg/bidiag.jl:256
+(UL::UpperTriangular, J::UniformScaling) at linalg/uniformscaling.jl:55
+(UL::Base.LinAlg.UnitUpperTriangular, J::UniformScaling) at linalg/unifor
+(UL::LowerTriangular, J::UniformScaling) at linalg/uniformscaling.jl:55
+(UL::Base.LinAlg.UnitLowerTriangular, J::UniformScaling) at linalg/unifor
+(A::Diagonal, B::Bidiagonal) at linalg/special.jl:120
+(A::Bidiagonal, B::Diagonal) at linalg/special.jl:121
+(A::Diagonal, B::Tridiagonal) at linalg/special.jl:120
+(A::Tridiagonal, B::Diagonal) at linalg/special.jl:121
+(A::Diagonal, B::Array{T<:Any,2}) at linalg/special.jl:120
+(A::Bidiagonal, B::Tridiagonal) at linalg/special.jl:120
+(A::Tridiagonal, B::Bidiagonal) at linalg/special.jl:121
+(A::Bidiagonal, B::Array{T<:Any,2}) at linalg/special.jl:120
+(A::Tridiagonal, B::Array{T<:Any,2}) at linalg/special.jl:120
+(A::SymTridiagonal, B::Tridiagonal) at linalg/special.jl:129
+(A::Tridiagonal, B::SymTridiagonal) at linalg/special.jl:130
+(A::SymTridiagonal, B::Array{T<:Any,2}) at linalg/special.jl:129
+(A::Diagonal, B::SymTridiagonal) at linalg/special.jl:138
+(A::SymTridiagonal, B::Diagonal) at linalg/special.jl:139
+(A::Bidiagonal, B::SymTridiagonal) at linalg/special.jl:138
+(A::SymTridiagonal, B::Bidiagonal) at linalg/special.jl:139
```

```
+(A::Diagonal, B::UpperTriangular) at linalg/special.jl:150
+(A::UpperTriangular, B::Diagonal) at linalg/special.jl:151
+(A::Diagonal, B::Base.LinAlg.UnitUpperTriangular) at linalg/special.jl:15
+(A::Base.LinAlg.UnitUpperTriangular, B::Diagonal) at linalg/special.jl:15
+(A::Diagonal, B::LowerTriangular) at linalg/special.jl:150
+(A::LowerTriangular, B::Diagonal) at linalg/special.jl:151
+(A::Diagonal, B::Base.LinAlg.UnitLowerTriangular) at linalg/special.jl:15
+(A::Base.LinAlg.UnitLowerTriangular, B::Diagonal) at linalg/special.jl:15
+(A::Base.LinAlg.AbstractTriangular, B::SymTridiagonal) at linalg/special.
+(A::SymTridiagonal, B::Base.LinAlg.AbstractTriangular) at linalg/special.
+(A::Base.LinAlg.AbstractTriangular, B::Tridiagonal) at linalg/special.jl:
+(A::Tridiagonal, B::Base.LinAlg.AbstractTriangular) at linalg/special.jl:
+(A::Base.LinAlg.AbstractTriangular, B::Bidiagonal) at linalg/special.jl:1
+(A::Bidiagonal, B::Base.LinAlg.AbstractTriangular) at linalg/special.jl:1
+(A::Base.LinAlg.AbstractTriangular, B::Array{T<:Any,2}) at linalg/special
+{P<:Union{Base.Dates.CompoundPeriod,Base.Dates.Period}}(Y::Union{Base.Res
+{P<:Union{Base.Dates.CompoundPeriod,Base.Dates.Period},Q<:Union{Base.Date
+{T<:Base.Dates.TimeType,P<:Union{Base.Dates.CompoundPeriod,Base.Dates.Per
+{T<:Base.Dates.TimeType}(r::Range{T}, x::Base.Dates.Period) at dates/rang
+{Tv1,Ti1,Tv2,Ti2}(A_1::SparseMatrixCSC{Tv1,Ti1}, A_2::SparseMatrixCSC{Tv2
+(A::SparseMatrixCSC, B::Array) at sparse/sparsematrix.jl:1709
+(A::SparseMatrixCSC, J::UniformScaling) at sparse/sparsematrix.jl:3811
+(x::AbstractSparseArray{Tv<:Any,Ti<:Any,1}, y::AbstractSparseArray{Tv<:An
+(x::Union{Base.ReshapedArray{T<:Any,1,A<:DenseArray,MI<:Tuple{Vararg{Base
+(x::AbstractSparseArray{Tv<:Any,Ti<:Any,1}, y::Union{Base.ReshapedArray{T
+{T<:Number}(x::AbstractArray{T,N<:Any}) at abstractarraymath.jl:91
+{R,S}(A::AbstractArray{R,N<:Any}, B::AbstractArray{S,N<:Any}) at arraymat
+(A::AbstractArray, x::Number) at arraymath.jl:94
+(x::Number, A::AbstractArray) at arraymath.jl:95
+{N}(index1::CartesianIndex{N}, index2::CartesianIndex{N}) at multidimensi
+{N}(index::CartesianIndex{N}, i::Integer) at multidimensional.jl:57
+(J1::UniformScaling, J2::UniformScaling) at linalg/uniformscaling.jl:37
+(J::UniformScaling, B::BitArray{2}) at linalg/uniformscaling.jl:39
+(J::UniformScaling, A::AbstractArray{T<:Any,2}) at linalg/uniformscaling.
+(J::UniformScaling, x::Number) at linalg/uniformscaling.jl:41
+(x::Number, J::UniformScaling) at linalg/uniformscaling.jl:42
+{TA,TJ}(A::AbstractArray{TA,2}, J::UniformScaling{TJ}) at linalg/uniforms
+{T}(a::Base.Pkg.Resolve.VersionWeights.HierarchicalValue{T}, b::Base.Pkg.
+{P<:Base.Dates.Period}(x::P, y::P) at dates/periods.jl:70
+(x::Base.Dates.Period, y::Base.Dates.Period) at dates/periods.jl:311
+(y::Base.Dates.Period, x::Base.Dates.CompoundPeriod) at dates/periods.jl:
+(y::Base.Dates.Period, x::Base.Dates.TimeType) at dates/arithmetic.jl:66
+{T<:Base.Dates.TimeType}(x::Base.Dates.Period, r::Range{T}) at dates/rang
+(x::Union{Base.Dates.CompoundPeriod,Base.Dates.Period}) at dates/periods.
+{P<:Union{Base.Dates.CompoundPeriod,Base.Dates.Period}}(x::Union{Base.Dat
+(x::Base.Dates.TimeType) at dates/arithmetic.jl:8
+(a::Base.Dates.TimeType, b::Base.Dates.Period, c::Base.Dates.Period) at c
+(a::Base.Dates.TimeType, b::Base.Dates.Period, c::Base.Dates.Period, d::B
```

```
+(x::Base.Dates.TimeType, y::Base.Dates.CompoundPeriod) at dates/periods.j
+(x::Base.Dates.Instant) at dates/arithmetic.jl:4
+{T<:Base.Dates.TimeType}(x::AbstractArray{T,N<:Any}, y::Union{Base.Dates.
+{T<:Base.Dates.TimeType}(y::Union{Base.Dates.CompoundPeriod,Base.Dates.Pe
+{P<:Union{Base.Dates.CompoundPeriod,Base.Dates.Period}}(y::Base.Dates.Tim
+(a, b, c, xs...) at operators.jl:138
```

We can inspect a type by finding its fields with `fieldnames`

```
In [40]: fieldnames(LinSpace)

Out[40]: 4-element Array{Symbol,1}:
          :start
          :stop
          :len
          :divisor
```

and find out which method was used with the `@which` macro:

```
In [43]: @which copy([1,2,3])

Out[43]: copy{T<:Array{T,N}}(a::T) at array.jl:70
```

Notice that this gives you a link to the source code where the function is defined.
Lastly, we can find out what type a variable is with the `typeof` function:

```
In [44]: a = [1;2;3]
         typeof(a)

Out[44]: Array{Int64,1}
```

0.6 Some Basic Types

Julia provides many basic types. Indeed, you will come to know Julia as a system of multiple dispatch on types, meaning that the interaction of types with functions is core to the design.

0.6.1 Lazy Iterator Types

While MATLAB or Python has easy functions for building arrays, Julia tends to side-step the actual "array" part with specially made types. One such example are ranges. To define a range, use the `start:stepsize:end` syntax. For example:

```
In [45]: a = 1:5
         println(a)
         b = 1:2:10
         println(b)

1:5
1:2:9
```

We can use them like any array. For example:

```
In [47]: println(a[2]); println(b[3])

2
5
```

But what is `b`?

```
In [50]: println(typeof(b))

StepRange{Int64,Int64}
```

`b` isn't an array, it's a StepRange. A StepRange has the ability to act like an array using its fields:

```
In [52]: fieldnames(StepRange)

Out[52]: 3-element Array{Symbol,1}:
          :start
          :step
          :stop
```

Question 4 If you know `start`, `step`, and `stop`, how do you calculate the `i`th value? Can you create a function MyRange which where for `a` being a `MyRange`, `value(a,i) == a[i]` for `a` a `StepRange`? (I will show you how to make the `a[i]` syntax available for your own types later. As noted before, all of these Julia types are implemented in Julia, and therefore you have the power to implement it yourself).

Note that at any time we can get the array from these kinds of type via the `collect` function:

```
In [55]: c = collect(a)

Out[55]: 5-element Array{Int64,1}:
          1
          2
          3
          4
          5
```

The reason why lazy iterator types are preferred is that they do not do the computations until it's absolutely necessary, and they take up much less space. We can check this with `@time`:

```
In [73]: @time a = 1:100000
         @time a = 1:100
         @time b = collect(1:100000);
```

```
  0.000003 seconds (5 allocations: 192 bytes)
  0.000001 seconds (5 allocations: 192 bytes)
  0.000076 seconds (8 allocations: 781.547 KB)
```

Notice that the amount of time the range takes is much shorter. This is mostly because there is a lot less memory allocation needed: only a `StepRange` is built, and all that holds is the three numbers. However, `b` has to hold `100000` numbers, leading to the huge difference.

0.6.2 Dictionaries

Another common type is the Dictionary. It allows you to access (key,value) pairs in a named manner. For example:

```
In [76]: d = Dict(:test=>2,"silly"=>:suit)
         println(d[:test])
         println(d["silly"])

2
suit
```

0.7 Package Management

Julia's package system is similar to R/Python in that a large number of packages are freely available. You search for them in places like Julia's Package Genie, or from the Julia Package Listing. Let's take a look at the Plots.jl package by Tom Breloff. This is a highly reguarded plotting package which we will make extensive use of later in this workshop. To add a package, use `Pkg.add`

```
In [1]: Pkg.update() # You may need to update your local packages first
        Pkg.add("Plots")

INFO: Initializing package repository /home/juser/.julia/v0.5
INFO: Cloning METADATA from https://github.com/JuliaLang/METADATA.jl
INFO: Cloning cache of ColorTypes from https://github.com/JuliaGraphics/ColorTypes.
INFO: Cloning cache of Colors from https://github.com/JuliaGraphics/Colors.jl.git
INFO: Cloning cache of Compat from https://github.com/JuliaLang/Compat.jl.git
INFO: Cloning cache of FixedPointNumbers from https://github.com/JeffBezanson/Fixed
INFO: Cloning cache of FixedSizeArrays from https://github.com/SimonDanisch/FixedSi
INFO: Cloning cache of Iterators from https://github.com/JuliaLang/Iterators.jl.git
INFO: Cloning cache of Measures from https://github.com/dcjones/Measures.jl.git
INFO: Cloning cache of PlotUtils from https://github.com/JuliaPlots/PlotUtils.jl.gi
INFO: Cloning cache of Plots from https://github.com/tbreloff/Plots.jl.git
INFO: Cloning cache of RecipesBase from https://github.com/JuliaPlots/RecipesBase.j
INFO: Cloning cache of Reexport from https://github.com/simonster/Reexport.jl.git
INFO: Cloning cache of Showoff from https://github.com/JuliaGraphics/Showoff.jl.git
INFO: Installing ColorTypes v0.2.6
INFO: Installing Colors v0.6.7
INFO: Installing Compat v0.9.2
```

```
INFO: Installing FixedPointNumbers v0.1.6
INFO: Installing FixedSizeArrays v0.2.3
INFO: Installing Iterators v0.1.10
INFO: Installing Measures v0.0.3
INFO: Installing PlotUtils v0.0.4
INFO: Installing Plots v0.9.1
INFO: Installing RecipesBase v0.0.6
INFO: Installing Reexport v0.0.3
INFO: Installing Showoff v0.0.7
INFO: Building Plots
INFO: Cannot find deps/plotly-latest.min.js... downloading latest version.
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100 1746k  100 1746k    0     0  9420k      0 --:--:-- --:--:-- --:--:-- 9439k
INFO: Package database updated
```

This will install the package to your local system. However, this will only work for registered packages. To add a non-registered package, go to the Github repository to find the clone URL and use `Pkg.clone`. For example, to install the `ParameterizedFunctions` package, we can use:

```
In [1]: Pkg.clone("https://github.com/JuliaDiffEq/ParameterizedFunctions.jl")

INFO: Cloning ParameterizedFunctions from https://github.com/JuliaDiffEq/Parameteri
INFO: Computing changes...
WARNING: julia is fixed at 0.5.0-rc4+0 conflicting with requirement for Parameteriz
INFO: Cloning cache of BinDeps from https://github.com/JuliaLang/BinDeps.jl.git
INFO: Cloning cache of Conda from https://github.com/JuliaPy/Conda.jl.git
INFO: Cloning cache of JSON from https://github.com/JuliaLang/JSON.jl.git
INFO: Cloning cache of MacroTools from https://github.com/MikeInnes/MacroTools.jl.g
INFO: Cloning cache of PyCall from https://github.com/JuliaPy/PyCall.jl.git
INFO: Cloning cache of SHA from https://github.com/staticfloat/SHA.jl.git
INFO: Cloning cache of SymPy from https://github.com/JuliaPy/SymPy.jl.git
INFO: Cloning cache of URIParser from https://github.com/JuliaWeb/URIParser.jl.git
INFO: Installing BinDeps v0.4.5
INFO: Installing Conda v0.3.2
INFO: Installing JSON v0.7.0
INFO: Installing MacroTools v0.3.2
INFO: Installing PyCall v1.7.2
INFO: Installing SHA v0.2.1
INFO: Installing SymPy v0.3.1
INFO: Installing URIParser v0.1.6
INFO: Building PyCall
INFO: PyCall is using python (Python 2.7.6) at /usr/bin/python, libpython = libpyth
```

To use a package, you have to import the package. The `import` statement will import the package without exporting the functions to the namespace. For example:

```
In [2]: using Plots
        Plots.plot(rand(4,4))
```

```
INFO: Precompiling module Plots.
WARNING: Method definition cgrad(Any, Any) in module PlotUtils at /home/juser/.juli
WARNING: Method definition #cgrad(Array{Any, 1}, PlotUtils.#cgrad, Any, Any) in mod
```

```
[Plots.jl] Initializing backend: plotly
```

```
WARNING: Method definition show(IO, Base.Multimedia.MIME{:text/plain}, Plots.Plot)
```

(Note that the first time a package is run, it will precompile a lot of the functionality.) To instead export the functions (of the developers choosing) to the namespace, we can use the `using` statement. Since Plots.jl exports the `plot` command, we can then use it without reference to the package that it came from:

```
In [4]: using Plots
        plot(rand(4,4))
```

What really makes this possible in Julia but not something like Python is that namespace clashes are usually avoided by multiple dispatch. Most packages will define their own types in order to use dispatches, and so when they export the functionality, the methods are only for their own types and thus do not clash with other packages. Therefore it's common in Julia for concise syntax like `plot` to be part of packages, all without fear of clashing.

Since Julia is currently under lots of development, you may wish to checkout newer versions. By default, `Pkg.add` is the "latest release", meaning the latest tagged version. However, the main version shown in the Github repository is usually the "master" branch. It's good development practice that the latest release is kept "stable", while the "master" branch is kept "working", and development takes place in another branch (many times labelled "dev"). You can choose which branch your local repository takes from. For example, to checkout the master branch, we can use:

```
In [6]: Pkg.checkout("Plots")
```

```
INFO: Checking out Plots master...
INFO: Pulling Plots latest master...
INFO: No packages to install, update or remove
```

This will usually gives us pretty up to date features (if you are using a "unreleased version of Julia" like building from the source of the Julia nightly, you may need to checkout master in order to get some packages working). However, to go to a specific branch we can give the branch as another argument:

```
In [8]: Pkg.checkout("Plots","dev")
```

```
INFO: Checking out Plots dev...
INFO: Pulling Plots latest dev...
INFO: No packages to install, update or remove
```

This is not advised if you don't know what you're doing (i.e. talk to the developer or read the pull requests (PR)), but this is common if you talk to a developer and they say "yes, I already implemented that. Checkout the dev branch and use `plot(...)`.

0.8 Metaprogramming

Metaprogramming is a huge feature of Julia. The key idea is that every statement in Julia is of the type `Expression`. Julia operators by building an Abstract Syntax Tree (AST) from the Expressions. You've already been exposed to this a little bit: a `Symbol` (like `:PhysicalSciences` is not a string because it is part of the AST, and thus is part of the parsing/expression structure. One interesting thing is that symbol comparisons are O(1) while string comparisons, like always, are O(n)) is part of this, and macros (the weird functions with an `@`) are functions on expressions.

Thus you can think of metaprogramming as "code which takes in code and outputs code". One basic example is the `@time` macro:

```
In [85]: macro time(ex)
           return quote
             local t0 = time()
             local val = $ex
             local t1 = time()
             println("elapsed time: ", t1-t0, " seconds")
             val
           end
         end
```

```
        LoadError: error in method definition: function Base.@time must be explicit
    while loading In[85], in expression starting on line 1
```

This takes in an expression `ex`, gets the time before and after evaluation, and prints the elapsed time between (the real time macro also calculates the allocations as seen earlier). Note that `$ex` "interpolates" the expression into the macro. Going into detail on metaprogramming is a large step from standard scripting and will be a later session.

Why macros? One reason is because it lets you define any syntax you want. Since it operates on the expressions themselves, as long as you know how to parse the expression into working code, you can "choose any syntax" to be your syntax. A case study will be shown later. Another reason is because these are done at "parse time" and those are only called once (before the function compilation).