BasicIntroduction

November 12, 2018

0.1 A Basic Introduction to 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 Problems

Time to start using your noggin. Scattered in this document are problems for you to solve using Julia. Many of the details for solving these problems have been covered, some have not. You may need to use some external resources:

https://docs.julialang.org/en/stable/

https://gitter.im/JuliaLang/julia

Solve as many or as few problems as you can during these times. Please work at your own pace, or with others if that's how you're comfortable!

0.2 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 [1]: ?copy
```

```
search: copy copy! copyto! copysign deepcopy unsafe_copyto! circcopy! cospi
```

```
Out[1]:
```

```
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.

```
copy(A::Transpose)
copy(A::Adjoint)
```

Eagerly evaluate the lazy matrix transpose/adjoint. Note that the transposition is applied recursively to elements.

This operation is intended for linear algebra usage - for general data manipulation see `permutedims`, which is non-recursive.

1 Examples

```
julia> A = [1 2im; -3im 4]
2×2 Array{Complex{Int64},2}:
 1+0im  0+2im
 0-3im  4+0im

julia> T = transpose(A)
2×2 Transpose{Complex{Int64},Array{Complex{Int64},2}}:
 1+0im  0-3im
 0+2im  4+0im

julia> copy(T)
2×2 Array{Complex{Int64},2}:
 1+0im  0-3im
 0+2im  4+0im
```

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

```
In [2]: methods(+)

Out[2]: # 163 methods for generic function "+":
        [1] +(x::Bool, z::Complex{Bool}) in Base at complex.jl:277
        [2] +(x::Bool, y::Bool) in Base at bool.jl:104
        [3] +(x::Bool) in Base at bool.jl:101
        [4] +(x::Bool, y::T) where T<:AbstractFloat in Base at bool.jl:112
        [5] +(x::Bool, z::Complex) in Base at complex.jl:284
        [6] +(a::Float16, b::Float16) in Base at float.jl:392
        [7] +(x::Float32, y::Float32) in Base at float.jl:394
        [8] +(x::Float64, y::Float64) in Base at float.jl:395
        [9] +(z::Complex{Bool}, x::Bool) in Base at complex.jl:278
        [10] +(z::Complex{Bool}, x::Real) in Base at complex.jl:292
        [11] +(::Missing, ::Missing) in Base at missing.jl:92
        [12] +(::Missing) in Base at missing.jl:79
        [13] +(::Missing, ::Number) in Base at missing.jl:93
        [14] +(level::Base.CoreLogging.LogLevel, inc::Integer) in Base.CoreLogging at logging.
        [15] +(c::BigInt, x::BigFloat) in Base.MPFR at mpfr.jl:353
```

```
[16] +(a::BigInt, b::BigInt, c::BigInt, d::BigInt, e::BigInt) in Base.GMP at gmp.jl:443
[17] +(a::BigInt, b::BigInt, c::BigInt, d::BigInt) in Base.GMP at gmp.jl:442
[18] +(a::BigInt, b::BigInt, c::BigInt) in Base.GMP at gmp.jl:441
[19] +(x::BigInt, y::BigInt) in Base.GMP at gmp.jl:412
[20] +(x::BigInt, c::Union{UInt16, UInt32, UInt64, UInt8}) in Base.GMP at gmp.jl:449
[21] +(x::BigInt, c::Union{Int16, Int32, Int64, Int8}) in Base.GMP at gmp.jl:455
[22] +(a::BigFloat, b::BigFloat, c::BigFloat, d::BigFloat, e::BigFloat) in Base.MPFR at
[23] +(a::BigFloat, b::BigFloat, c::BigFloat, d::BigFloat) in Base.MPFR at mpfr.jl:496
[24] +(a::BigFloat, b::BigFloat, c::BigFloat) in Base.MPFR at mpfr.jl:490
[25] +(x::BigFloat, c::BigInt) in Base.MPFR at mpfr.jl:349
[26] +(x::BigFloat, y::BigFloat) in Base.MPFR at mpfr.jl:318
[27] +(x::BigFloat, c::Union{UInt16, UInt32, UInt64, UInt8}) in Base.MPFR at mpfr.jl:32
[28] +(x::BigFloat, c::Union{Int16, Int32, Int64, Int8}) in Base.MPFR at mpfr.jl:333
[29] +(x::BigFloat, c::Union{Float16, Float32, Float64}) in Base.MPFR at mpfr.jl:341
[30] +(x::Dates.CompoundPeriod, y::Dates.CompoundPeriod) in Dates at /buildworker/worke
[31] +(x::Dates.CompoundPeriod, y::Dates.Period) in Dates at /buildworker/worker/packag
[32] +(x::Dates.CompoundPeriod, y::Dates.TimeType) in Dates at /buildworker/worker/pack
[33] +(x::Dates.Date, y::Dates.Day) in Dates at /buildworker/worker/package_linux64/bu
[34] +(x::Dates.Date, y::Dates.Week) in Dates at /buildworker/worker/package_linux64/bu
[35] +(dt::Dates.Date, z::Dates.Month) in Dates at /buildworker/worker/package_linux64/
[36] +(dt::Dates.Date, y::Dates.Year) in Dates at /buildworker/worker/package_linux64/b
[37] +(dt::Dates.Date, t::Dates.Time) in Dates at /buildworker/worker/package_linux64/b
[38] +(t::Dates.Time, dt::Dates.Date) in Dates at /buildworker/worker/package_linux64/b
[39] +(x::Dates.Time, y::Dates.TimePeriod) in Dates at /buildworker/worker/package_linu
[40] +(dt::Dates.DateTime, z::Dates.Month) in Dates at /buildworker/worker/package_linu
[41] +(dt::Dates.DateTime, y::Dates.Year) in Dates at /buildworker/worker/package_linux
[42] +(x::Dates.DateTime, y::Dates.Period) in Dates at /buildworker/worker/package_linu
[43] +(B::BitArray{2}, J::LinearAlgebra.UniformScaling) in LinearAlgebra at /buildworke
[44] +(a::Pkg.Resolve.VersionWeights.VersionWeight, b::Pkg.Resolve.VersionWeights.Versi
[45] +(a::Pkg.Resolve.MaxSum.FieldValues.FieldValue, b::Pkg.Resolve.MaxSum.FieldValues
[46] +(y::AbstractFloat, x::Bool) in Base at bool.jl:114
[47] +(x::T, y::T) where T<:Union{Int128, Int16, Int32, Int64, Int8, UInt128, UInt16, U
[48] +(c::Union{UInt16, UInt32, UInt64, UInt8}, x::BigInt) in Base.GMP at gmp.jl:450
[49] +(c::Union{Int16, Int32, Int64, Int8}, x::BigInt) in Base.GMP at gmp.jl:456
[50] +(a::Integer, b::Integer) in Base at int.jl:791
[51] +(x::Integer, y::Ptr) in Base at pointer.jl:157
[52] +(z::Complex, w::Complex) in Base at complex.jl:266
[53] +(z::Complex, x::Bool) in Base at complex.jl:285
[54] +(x::Real, z::Complex{Bool}) in Base at complex.jl:291
[55] +(x::Real, z::Complex) in Base at complex.jl:303
[56] +(z::Complex, x::Real) in Base at complex.jl:304
[57] +(x::Rational, y::Rational) in Base at rational.jl:248
[58] +(x::Integer, y::AbstractChar) in Base at char.jl:208
[59] +(c::Union{UInt16, UInt32, UInt64, UInt8}, x::BigFloat) in Base.MPFR at mpfr.jl:32
[60] +(c::Union{Int16, Int32, Int64, Int8}, x::BigFloat) in Base.MPFR at mpfr.jl:337
[61] +(c::Union{Float16, Float32, Float64}, x::BigFloat) in Base.MPFR at mpfr.jl:345
[62] +(x::AbstractIrrational, y::AbstractIrrational) in Base at irrationals.jl:133
[63] +(x::Number) in Base at operators.jl:477
```

```
[64] +(x::T, y::T) where T<:Number in Base at promotion.jl:411
[65] +(x::Number, y::Number) in Base at promotion.jl:313
[66] +(r1::OrdinalRange, r2::OrdinalRange) in Base at range.jl:978
[67] +(r1::LinRange{T}, r2::LinRange{T}) where T in Base at range.jl:985
[68] +(r1::StepRangeLen{T,R,S} where S, r2::StepRangeLen{T,R,S} where S) where {R<:Twi
[69] +(r1::StepRangeLen{T,S,S1} where S1, r2::StepRangeLen{T,S,S1} where S1) where {T,
[70] +(r1::Union{LinRange, OrdinalRange, StepRangeLen}, r2::Union{LinRange, OrdinalRang
[71] +(x::Ptr, y::Integer) in Base at pointer.jl:155
[72] +(x::Base.TwicePrecision, y::Number) in Base at twiceprecision.jl:265
[73] +(x::Number, y::Base.TwicePrecision) in Base at twiceprecision.jl:268
[74] +(x::Base.TwicePrecision{T}, y::Base.TwicePrecision{T}) where T in Base at twicepr
[75] +(x::Base.TwicePrecision, y::Base.TwicePrecision) in Base at twiceprecision.jl:275
[76] +(A::Array, Bs::Array...) in Base at arraymath.jl:44
[77] +(A::BitArray, B::BitArray) in Base at bitarray.jl:1074
[78] +(A::Array{T,2} where T, B::LinearAlgebra.Diagonal) in LinearAlgebra at /buildwork
[79] +(A::Array{T,2} where T, B::LinearAlgebra.Bidiagonal) in LinearAlgebra at /buildwo
[80] +(A::Array{T,2} where T, B::LinearAlgebra.Tridiagonal) in LinearAlgebra at /buildw
[81] +(A::Array{T,2} where T, B::LinearAlgebra.SymTridiagonal) in LinearAlgebra at /bui
[82] +(A::Array{T,2} where T, B::LinearAlgebra.AbstractTriangular) in LinearAlgebra at
[83] +(A::Array, B::SparseArrays.SparseMatrixCSC) in SparseArrays at /buildworker/worke
[84] +(x::Union{DenseArray{#s549,N}, ReinterpretArray{#s549,N,S,A} where S where A<:Uni
[85] +(X::Union{DenseArray{#s549,N}, ReinterpretArray{#s549,N,S,A} where S where A<:Uni
[86] +(x::Union{DenseArray{#s549,N}, ReinterpretArray{#s549,N,S,A} where S where A<:Uni
[87] +(r::AbstractRange{#s549} where #s549<:Dates.TimeType, x::Dates.Period) in Dates a
[88] +(A::LinearAlgebra.SymTridiagonal, B::LinearAlgebra.SymTridiagonal) in LinearAlgeb
[89] +(A::LinearAlgebra.Tridiagonal, B::LinearAlgebra.Tridiagonal) in LinearAlgebra at
[90] +(A::LinearAlgebra.UpperTriangular, B::LinearAlgebra.UpperTriangular) in LinearAlg
[91] +(A::LinearAlgebra.LowerTriangular, B::LinearAlgebra.LowerTriangular) in LinearAlg
[92] +(A::LinearAlgebra.UpperTriangular, B::LinearAlgebra.UnitUpperTriangular) in Linea
[93] +(A::LinearAlgebra.LowerTriangular, B::LinearAlgebra.UnitLowerTriangular) in Linea
[94] +(A::LinearAlgebra.UnitUpperTriangular, B::LinearAlgebra.UpperTriangular) in Linea
[95] +(A::LinearAlgebra.UnitLowerTriangular, B::LinearAlgebra.LowerTriangular) in Linea
[96] +(A::LinearAlgebra.UnitUpperTriangular, B::LinearAlgebra.UnitUpperTriangular) in L
[97] +(A::LinearAlgebra.UnitLowerTriangular, B::LinearAlgebra.UnitLowerTriangular) in L
[98] +(A::LinearAlgebra.AbstractTriangular, B::LinearAlgebra.AbstractTriangular) in Lin
[99] +(Da::LinearAlgebra.Diagonal, Db::LinearAlgebra.Diagonal) in LinearAlgebra at /bui
[100] +(A::LinearAlgebra.Bidiagonal, B::LinearAlgebra.Bidiagonal) in LinearAlgebra at /
[101] +(UL::LinearAlgebra.UnitUpperTriangular, J::LinearAlgebra.UniformScaling) in Line
[102] +(UL::LinearAlgebra.UnitLowerTriangular, J::LinearAlgebra.UniformScaling) in Line
[103] +(A::SparseArrays.SparseMatrixCSC, J::LinearAlgebra.UniformScaling) in SparseArra
[104] +(A::AbstractArray{T,2} where T, J::LinearAlgebra.UniformScaling) in LinearAlgebr
[105] +(A::LinearAlgebra.Diagonal, B::LinearAlgebra.Bidiagonal) in LinearAlgebra at /bu
[106] +(A::LinearAlgebra.Bidiagonal, B::LinearAlgebra.Diagonal) in LinearAlgebra at /bu
[107] +(A::LinearAlgebra.Diagonal, B::LinearAlgebra.Tridiagonal) in LinearAlgebra at /b
[108] +(A::LinearAlgebra.Tridiagonal, B::LinearAlgebra.Diagonal) in LinearAlgebra at /b
[109] +(A::LinearAlgebra.Diagonal, B::Array{T,2} where T) in LinearAlgebra at /buildwor
[110] +(A::LinearAlgebra.Bidiagonal, B::LinearAlgebra.Tridiagonal) in LinearAlgebra at
[111] +(A::LinearAlgebra.Tridiagonal, B::LinearAlgebra.Bidiagonal) in LinearAlgebra at
```

```
[112] +(A::LinearAlgebra.Bidiagonal, B::Array{T,2} where T) in LinearAlgebra at /buildw
[113] +(A::LinearAlgebra.Tridiagonal, B::Array{T,2} where T) in LinearAlgebra at /build
[114] +(A::LinearAlgebra.SymTridiagonal, B::LinearAlgebra.Tridiagonal) in LinearAlgebra
[115] +(A::LinearAlgebra.Tridiagonal, B::LinearAlgebra.SymTridiagonal) in LinearAlgebra
[116] +(A::LinearAlgebra.SymTridiagonal, B::Array{T,2} where T) in LinearAlgebra at /bu
[117] +(A::LinearAlgebra.Diagonal, B::LinearAlgebra.SymTridiagonal) in LinearAlgebra at
[118] +(A::LinearAlgebra.SymTridiagonal, B::LinearAlgebra.Diagonal) in LinearAlgebra at
[119] +(A::LinearAlgebra.Bidiagonal, B::LinearAlgebra.SymTridiagonal) in LinearAlgebra
[120] +(A::LinearAlgebra.SymTridiagonal, B::LinearAlgebra.Bidiagonal) in LinearAlgebra
[121] +(A::LinearAlgebra.Diagonal, B::LinearAlgebra.UpperTriangular) in LinearAlgebra a
[122] +(A::LinearAlgebra.UpperTriangular, B::LinearAlgebra.Diagonal) in LinearAlgebra a
[123] +(A::LinearAlgebra.Diagonal, B::LinearAlgebra.UnitUpperTriangular) in LinearAlgeb
[124] +(A::LinearAlgebra.UnitUpperTriangular, B::LinearAlgebra.Diagonal) in LinearAlgeb
[125] +(A::LinearAlgebra.Diagonal, B::LinearAlgebra.LowerTriangular) in LinearAlgebra a
[126] +(A::LinearAlgebra.LowerTriangular, B::LinearAlgebra.Diagonal) in LinearAlgebra a
[127] +(A::LinearAlgebra.Diagonal, B::LinearAlgebra.UnitLowerTriangular) in LinearAlgeb
[128] +(A::LinearAlgebra.UnitLowerTriangular, B::LinearAlgebra.Diagonal) in LinearAlgeb
[129] +(A::LinearAlgebra.AbstractTriangular, B::LinearAlgebra.SymTridiagonal) in Linear
[130] +(A::LinearAlgebra.SymTridiagonal, B::LinearAlgebra.AbstractTriangular) in Linear
[131] +(A::LinearAlgebra.AbstractTriangular, B::LinearAlgebra.Tridiagonal) in LinearAlg
[132] +(A::LinearAlgebra.Tridiagonal, B::LinearAlgebra.AbstractTriangular) in LinearAlg
[133] +(A::LinearAlgebra.AbstractTriangular, B::LinearAlgebra.Bidiagonal) in LinearAlge
[134] +(A::LinearAlgebra.Bidiagonal, B::LinearAlgebra.AbstractTriangular) in LinearAlge
[135] +(A::LinearAlgebra.AbstractTriangular, B::Array{T,2} where T) in LinearAlgebra at
[136] +(A::SparseArrays.SparseMatrixCSC, B::SparseArrays.SparseMatrixCSC) in SparseArra
[137] +(A::SparseArrays.SparseMatrixCSC, B::Array) in SparseArrays at /buildworker/work
[138] +(x::SparseArrays.AbstractSparseArray{Tv,Ti,1} where Ti where Tv, y::SparseArrays
[139] +(x::AbstractArray{#s57,N} where N where #s57<:Number) in Base at abstractarrayma
[140] +(A::AbstractArray, B::AbstractArray) in Base at arraymath.jl:38
[141] +(x::T, y::Integer) where T<:AbstractChar in Base at char.jl:207
[142] +(index1::CartesianIndex{N}, index2::CartesianIndex{N}) where N in Base.Iterators
[143] +(::Number, ::Missing) in Base at missing.jl:94
[144] +(x::P, y::P) where P<:Dates.Period in Dates at /buildworker/worker/package_linux
[145] +(x::Dates.Period, y::Dates.Period) in Dates at /buildworker/worker/package_linux
[146] +(y::Dates.Period, x::Dates.CompoundPeriod) in Dates at /buildworker/worker/packa
[147] +(x::Union{CompoundPeriod, Period}) in Dates at /buildworker/worker/package_linux
[148] +(x::Dates.TimeType) in Dates at /buildworker/worker/package_linux64/build/usr/sh
[149] +(a::Dates.TimeType, b::Dates.Period, c::Dates.Period) in Dates at /buildworker/w
[150] +(a::Dates.TimeType, b::Dates.Period, c::Dates.Period, d::Dates.Period...) in Dat
[151] +(x::Dates.TimeType, y::Dates.CompoundPeriod) in Dates at /buildworker/worker/pac
[152] +(x::Dates.Instant) in Dates at /buildworker/worker/package_linux64/build/usr/sha
[153] +(y::Dates.Period, x::Dates.TimeType) in Dates at /buildworker/worker/package_lin
[154] +(x::AbstractArray{#s549,N} where N where #s549<:Dates.TimeType, y::Union{Compoun
[155] +(x::Dates.Period, r::AbstractRange{#s549} where #s549<:Dates.TimeType) in Dates
[156] +(y::Union{CompoundPeriod, Period}, x::AbstractArray{#s549,N} where N where #s549
[157] +(y::Dates.TimeType, x::Union{DenseArray{#s549,N}, ReinterpretArray{#s549,N,S,A}
[158] +(J::LinearAlgebra.UniformScaling, x::Number) in LinearAlgebra at /buildworker/wo
[159] +(x::Number, J::LinearAlgebra.UniformScaling) in LinearAlgebra at /buildworker/wo
```

```
[160] +(J1::LinearAlgebra.UniformScaling, J2::LinearAlgebra.UniformScaling) in LinearAl
[161] +(J::LinearAlgebra.UniformScaling, B::BitArray{2}) in LinearAlgebra at /buildwor
[162] +(J::LinearAlgebra.UniformScaling, A::AbstractArray{T,2} where T) in LinearAlgebr
[163] +(a, b, c, xs...) in Base at operators.jl:502
```

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

```
In [3]: fieldnames(UnitRange)
```

```
Out[3]: (:start, :stop)
```

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

```
In [4]: @which copy([1,2,3])
```

```
Out[4]: copy(a::T) where T<:Array in Base at array.jl:299
```

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

```
Out[5]: Array{Int64,1}
```

1.0.1 Array Syntax

The array syntax is similar to MATLAB's conventions.

```
In [6]: a = Vector{Float64}(undef,5) # Create a length 5 Vector (dimension 1 array) of Float64

        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}(undef,4,2) # Define a Matrix of Float64's of size (4,2) with undef

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

        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[6]: 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 [7]: a = [1;3;4]
        b = a
        b[1] = 10
        a
```

```
Out[7]: 3-element Array{Int64,1}:
         10
          3
          4
```

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

```
In [8]: a = [1;4;5]
        b = copy(a)
        b[1] = 10
        a
```

```
Out[8]: 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 [9]: c = similar(a)
        d = zero(a)
        e = ones(a)
        println(c); println(d); println(e)
```

```
        MethodError: no method matching ones(::Array{Int64,1})
    Closest candidates are:
      ones(!Matched::Union{Integer, AbstractUnitRange}...) at array.jl:463
      ones(!Matched::Type{T}, !Matched::Union{Integer, AbstractUnitRange}...) where T at array
      ones(!Matched::Tuple{Vararg{Union{Integer, AbstractUnitRange},N} where N}) at array.jl:4
      ...
```

```
Stacktrace:

 [1] top-level scope at In[9]:3
```

Note that arrays can be index'd by arrays:

```
In [10]: a[1:2]

Out[10]: 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 [11]: a = Vector{Vector{Float64}}(undef,3)
         a[1] = [1;2;3]
         a[2] = [1;2]
         a[3] = [3;4;5]
         a

Out[11]: 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 [12]: b = a
         b[1] = [1;4;5]
         a

Out[12]: 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 [13]: b = deepcopy(a)
         b[1] = [1;2;3]
         a
```

```
Out[13]: 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 `copyto!` which copies the values of over to the first array.

```
In [14]: a = [1;6;8]
         b = similar(a) # make an array just like a but with undefined values
         copyto!(b,a) # b changes
```

```
Out[14]: 3-element Array{Int64,1}:
          1
          6
          8
```

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

1.1 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 [15]: 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 [16]: for i=1:2,j=2:4
             println(i*j)
         end

2
3
4
4
6
8
```

1.2 Problems

Try the Starter Problems. If you need help, start looking through the next parts of this tutorial!

1.3 Function Syntax

```
In [17]: f(x,y) = 2x+y # Create an inline function

Out[17]: f (generic function with 1 method)

In [18]: f(1,2) # Call the function

Out[18]: 4

In [19]: function f(x)
           x+2
         end # Long form definition

Out[19]: f (generic function with 2 methods)
```

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

```
In [20]: f(2)

Out[20]: 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 [21]: f(x::Int,y::Int) = 3x+2y
```

```
Out[21]: f (generic function with 3 methods)
```

Julia will dispatch onto the strictest acceptible type signature.

```
In [22]: f(2,3) # 3x+2y
```

```
Out[22]: 12
```

```
In [23]: f(2.0,3) # 2x+y since 2.0 is not an Int
```

```
Out[23]: 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 [24]: f{T<:Number}(x::T,y::T) = 4x+10y
```

```
        UndefVarError: T not defined



        Stacktrace:

         [1] top-level scope at In[24]:1
```

```
In [25]: f(2,3) # 3x+2y since (::Int,::Int) is stricter
```

```
Out[25]: 12
```

```
In [26]: f(2.0,3.0) # 4x+10y
```

```
Out[26]: 7.0
```

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

```
Out[27]: f (generic function with 4 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 [28]: 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)
```

```
  0.002646 seconds (1.08 k allocations: 59.268 KiB)
  0.000005 seconds (4 allocations: 160 bytes)
  0.000002 seconds (4 allocations: 160 bytes)
  0.004057 seconds (5.03 k allocations: 275.235 KiB)
  0.000015 seconds (5 allocations: 176 bytes)
```

```
Out[28]: 4.0
```

Note that functions can also feature optional arguments:

```
In [29]: 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[29]: test_function (generic function with 1 method)
```

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

```
In [30]: x,y = test_function(1,2)
```

```
Out[30]: (3, 2)
```

```
In [31]: x,y = test_function(1,2;z=3)
```

```
Out[31]: (6, 6)
```

Notice that we also featured multiple return values.

```
In [32]: 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 [33]: 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)

Out[33]: (64, test)

In [34]: test()

Out[34]: 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 [35]: a = 2

Out[35]: 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 [36]: g = (x,y) -> 2x+y

Out[36]: #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 [37]: g = (x) -> 2x

Out[37]: #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.

1.4 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 [38]: mutable struct Car
             make
             model
         end
```

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

```
In [39]: mycar = Car("Toyota","Rav4")

Out[39]: 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 [40]: mycar.make

Out[40]: "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, Camel-Case) 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 [41]: mutable struct WorkshopParticipant
             name::String
             field::Symbol
         end
         tony = WorkshopParticipant("Tony",:physics)

Out[41]: 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 [42]: mutable struct StaffMember{T<:Number}
             name::String
             field::Symbol
             age::T
         end
         ter = StaffMember("Terry",:football,17)

Out[42]: 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 [43]: abstract type Person
         end
```

Then we can set types as a subtype of person

```
In [44]: mutable struct Student <: Person
             name
             grade
         end
```

You can define type heirarchies on abstract types. See the beautiful explanation at: http://docs.julialang.org/en/release-0.5/manual/types/#abstract-types

```
In [45]: abstract type AbstractStudent <: Person
         end
```

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 [46]: struct Field
             name
             school
         end
         ds = Field(:DataScience,[:PhysicalScience;:ComputerScience])

Out[46]: Field(:DataScience, Symbol[:PhysicalScience, :ComputerScience])
```

Question 3 Can you explain this interesting quirk? Thus Field is immutable, meaning that `ds.name` and `ds.school` cannot be changed:

```
In [47]: ds.name = :ComputationalStatistics
```

```
type Field is immutable


Stacktrace:

 [1] setproperty!(::Field, ::Symbol, ::Symbol) at ./sysimg.jl:19

 [2] top-level scope at In[47]:1
```

However, the following is allowed:

```
In [48]: push!(ds.school,:BiologicalScience)
         ds.school

Out[48]: 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 native, but all of the "native parts" are always accessible. For example, we can, if we so choose, get a function pointer:

```
In [49]: foo(x) = 2x
         cfunction(foo, Int, Tuple{Int})


UndefVarError: cfunction not defined


Stacktrace:

 [1] top-level scope at In[49]:2
```

1.5 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.

1.5.1 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 [50]: 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 [51]: println(a[2]); println(b[3])

2
5
```

But what is `b`?

```
In [52]: 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 [53]: fieldnames(StepRange)

Out[53]: (:start, :step, :stop)
```

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

```
In [54]: c = collect(a)

Out[54]: 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 [55]: @time a = 1:100000
         @time a = 1:100
         @time b = collect(1:100000);
```

```
  0.000004 seconds (5 allocations: 192 bytes)
  0.000003 seconds (5 allocations: 192 bytes)
  0.001093 seconds (7 allocations: 781.516 KiB)
```

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.

1.5.2 Dictionaries

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

```
In [56]: d = Dict(:test=>2,"silly"=>:suit)
         println(d[:test])
         println(d["silly"])
```

```
2
suit
```

1.5.3 Tuples

Tuples are immutable arrays. That means they can't be changed. However, they are super fast. They are made with the `(x,y,z,...)` syntax and are the standard return type of functions which return more than one object.

```
In [57]: tup = (2.,3) # Don't have to match types
         x,y = (3.0,"hi") # Can separate a tuple to multiple variables
```

```
Out[57]: (3.0, "hi")
```

1.6 Problems

Try problems 8-11 in the Basic Problems

1.7 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 [58]: macro my_time(ex)
           return quote
             local t0 = time()
             local val = $ex
             local t1 = time()
             println("elapsed time: ", t1-t0, " seconds")
             val
           end
         end
```

```
Out[58]: @my_time (macro with 1 method)
```

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).

1.8 Steps for Julia Parsing and Execution

1. The AST after parsing <- Macros
2. The AST after lowering (@code_typed)
3. The AST after type inference and optimization <- Generated Functions (@code_lowered)
4. The LLVM IR <- Functions (@code_llvm)
5. The assembly code (@code_native)