I Like Julia Because It Scales and Is Productive: Some Insights From A Julia Developer

Christopher Rackauckas[1]

[1]Affiliation not available

April 17, 2023

I Like Julia Because It Scales and Is Productive: Some Insights From A Julia Developer

CHRISTOPHER RACKAUCKAS

CORRESPONDENCE:
me@chrisrackauckas.com

In this post I would like to reflect a bit on the Julia programming language. These are my personal views and I have had more than a year developing a lot of packages for the Julia programming language. After roaming around many different languages including R, MATLAB, C, and Python; Julia is finally a language I am sticking to. In this post I would like to explain why. I want to go back through some thoughts about what the current state of the language is, who it's good for, and what changes I would like to see. My opinions changed a lot since first starting to work on Julia, so I'd just like to share the changed mindset one has after using the language deeply.

QUICK SUMMARY

Here's a quick summary of my views.

1. Julia is not only a fast language, but what makes it unique is how predictable the performance and the compilation process is.
2. The language gives you lots of introspection tools to be able to easily isolate issues.
3. The opt-in type checking and allowing many different architectures to be fast is a strong bonus for software development, especially when scaling to large software ecosystems, over other scripting languages since it allows you to write simple and self-documenting code.
4. Julia's unique features make it easy to make packages which are type-generic and parallel.
5. Most of these benefits will be seen by package developers. "Users" will probably not see as much of a difference in their own codes because the majority of their performance will be determined by the packages they use.
6. To attract a new wave of users, Julia needs to start taking a "package-first" mentality and push package-level unique features rather than language-level features. Language level is what developers care about, but the majority of programmers are not developers.
7. We have all of the basics in Julia, but we need to start showing off (and working towards) how we can be different. Every package should be picking some special features and types to support. Speed is just one feature.
8. Julia can have a bright future, but we may need to start advertising and teaching it differently.

I think that to start, I need to discuss a topic which might be new to newcomers.

JULIA'S JIT IS NOT LIKE OTHER JITS, AND IT HELPS PACKAGE DEVELOPMENT

LuaJIT is a runtime that just-in-time compiles Lua, so is it the same thing or similar to Julia? What about MATLAB's JIT? The answer is no, Julia's compilation strategy is very different. Other JIT setups for scripting languages use something that's known as a tracing JIT. What these do is they track the commands you are running and then via some algorithm (possibly probabilistic in some setups like LuaJIT) it determines what parts of the code are repetitively ran enough that they warrant compilation, and then code for those specific sets of commands are compiled and when the parser hits those areas again it runs the JIT code.

Julia on the other hand is almost static. When running any Julia function, it will first take your function calls and then auto-specialize it down to concrete types. So f(x::Number)=x^2 will specialize to Float64 when you call it with f(1.0). Using this type information, it has a compile-time stage where it propagates all type information as far as it can go. So if you internally do x^2, it will replace that with ^(::Float64,2) (the 2 is a compile-time constant), and then since it can determine the types of everything, it will push this all the way down to generate clean LLVM IR (and thus clean assembly code) for x^2 on Float64s. It will cache this compiled version of the function so that way f(2.0) uses the same compiled code (notice that the compilation process only needs the type and not the runtime values!). However, when you then call it with f(1), it compiles a separate version for Int.

In some sense, this is very different. The obvious difference is that Julia's compilation process is deterministic and is open to introspection. If you do

@code_warntype f(1.0)
@code_llvm f(1.0)
Julia will spit back at you its internal AST of the method specialized on Float64, and then the LLVM IR. The nice thing about this is that you can check every step of the process. I never participated in programming language development conversations before working with Julia, but this makes it easy. You see this kick something out, you find out yourself how to optimize it, and then you can easily ask someone "why doesn't the compiler optimize xyz away here?" and then that leads to code changes (i.e. "oh, array aliasing is preventing compiler optimizations!"), new work on compiler optimizations, and better performance. A recent place where just happened was in this Discourse discussion where a user points out that a separate faster LLVM instruction could be used than the one that is actually emitted.

But what I really like about this process is that it's clear and deterministic. I can know exactly what my packages are doing and why. This makes it very easy to optimize packages because you always know what's going on. When trying to piece together 50+ packages like DifferentialEquations.jl, deterministic rules for how to optimize code and easy means of introspection are essential to know what's going on. This kind of transparency is something that's really easy to appreciate after working on something like MATLAB where the internals are opaque.

There is a tradeoff though. In order for this process to work well it requires being able to propagate type information. This requires that the types of the variables inside of a function are what's known as "type-stable" or "type-inferrable". For example,

function g(a,n)
  x = 0
  for i in 1:n
    x += a
  end
end
will add a n times. If you call this with g(1.0,10), you'll notice this is not as fast as you'd hope. The reason is because the type of x starts as a Int, and then Int+Float64 produces a Float64 and so x changes types. Think about writing a C-code for doing this: you can't just have a variable change types like that? So what Julia does is compile x in a way such that it is dynamically typed, and the lost assumptions associated with doing this is why the performance drops down to that of Python/MATLAB/R since now it has the same amount of dynamicness as a real scripting language: it's all just a game of how much information is known by the compiler. So the way to think about it is, if your code can be type-stable, Julia will take full advantage of this and make a completely static function and compile that, otherwise it will make the necessary parts dynamic in order to handle the extra features. Together, this is a pretty nifty way to mix ease of use and performance. At any time, you can @code_warntype your function call and it will explicitly highlight which variables are "being dynamic", and that tells you exactly what to optimize. For example,

Variables:
  #self#::#g
  a::Float64
  n::Int64
  i::Int64

```
    #temp#@_5::Int64
    x::UNION{FLOAT64, INT64}
    #temp#@_7::Core.MethodInstance
    #temp#@_8::Float64

Body:
  begin
    x::UNION{FLOAT64, INT64} = 0 # line 3:
    SSAValue(2) = (Base.select_value)((Base.sle_int)(1, n::Int64)::Bool, n::Int64, (Base.sub_int)(1, 1)::Int64)::Int64
    #temp#@_5::Int64 = 1
    5:
    unless (Base.not_int)((#temp#@_5::Int64 === (Base.add_int)(SSAValue(2), 1)::Int64)::Bool)::Bool goto 29
    SSAValue(3) = #temp#@_5::Int64
    SSAValue(4) = (Base.add_int)(#temp#@_5::Int64, 1)::Int64
    #temp#@_5::Int64 = SSAValue(4) # line 4:
    unless (x::UNION{FLOAT64, INT64} isa Int64)::Bool goto 14
    #temp#@_7::Core.MethodInstance = MethodInstance for +(::Int64, ::Float64)
    goto 23
    14:
    unless (x::UNION{FLOAT64, INT64} isa Float64)::Bool goto 18
    #temp#@_7::Core.MethodInstance = MethodInstance for +(::Float64, ::Float64)
    goto 23
    18:
    goto 20
    20:
    #temp#@_8::Float64 = (x::UNION{FLOAT64, INT64} + a::Float64)::Float64
    goto 25
    23:
    #temp#@_8::Float64 = $(Expr(:invoke, :(#temp#@_7), :(Main.+), :(x), :(a)))
    25:
    x::UNION{FLOAT64, INT64} = #temp#@_8::Float64
    27:
    goto 5
    29:
    return
  end::Void
```

This is spitting back and telling me that x is typed as both Float64 and Int64 which is exactly the problem I identified. But have no fear, Julia is based around typing, so it has ways to handle this. For example, zero(a) gives you a 0 in the type of a, so

```
function g(a,n)
  x = zero(a)
  for i in 1:n
    x += a
  end
end
```

is a good generic type-stable function for any number which has zero and + defined. That's quite unique: I can put floating point numbers in here, symbolic expressions from SymEngine.jl, ArbFloats from the user-defined ArbFloats.jl fast arbitrary precision library, etc. Since Julia always auto-specializes on the types, no matter what I give it, it will know how to replace zero with whatever it needs to and do so at compile-time since at compile-time there's enough type information to know what that constant should be, and then it will replace the + with calls to the correct version of +. Thus there is no runtime overhead for handling genericness. All you have to do is remember to write your code in a manner such that the types match what comes in (other helpful functions for this include typeof, eltype, and similar). So Julia is not only built to make generic functions performant, but it also has the right tools for handling types to make this actually easy to do.

There is a little bit more of a cognitive burden here than in other languages. It's not much, but there are a few things to keep in mind. Tracing JITs will more automatically handle typing by using runtime information, but are harder to introspect and harder to determine when things are going wrong (also, it cannot fully statically compile ahead of time). With Julia, you do need to understand and think about your code in terms of types if you want full performance. But if you do so, then your code isn't "close to C", it literally has enough information that it can create and compile a function which is as fast as C (and any speed difference from C is usually due to the fact that you're using the LLVM (clang) compiler instead of the more standard GCC, or Julia in some cases is missing the ability to add an optimization. Extra compiler optimizations are part of the 1.x milestones. In practice you'll find that most codes are The inability to handle types well is one of the main reasons I see developers having hard times scaling up programs in scripting languages, but it's not an issue in Julia. So while LuaJIT and other well-designed tracing JITs can in many cases match Julia's speed with enough work optimizing runtime heuristics, they won't have the same ease of understanding and predictability as Julia, and won't have as much of a robust type-checking system to plug into as your program and contributor base grows.

Sure, Numba uses a similar strategy as Julia. If you stick to only using Float64s and a few other basic types (Float32, and some integer types?) you'll get something very similar to what's described here. But I've described before that making robust generic software with this type-inference approach requires a strong understanding and use of the type system, and I have never found Python+Numba close to matching Julia in its ability to let the user directly handle types, and it's this combination of the type system + the compilation strategy that makes Julia code scale well. This difference may not be appreciated in one-off scripts, but when building and maintaining robust software I have found it necessary.

But lastly, using this strategy Julia actually can produce static binaries like compiled C or Fortran code, so I think this fact will come in handy for making Julia into a package development language. Basically, you can write packages in Julia, and someday it will be easy to generate a runtime-free binary (like C/Fortran) others can link to. That leads me to my next point.

In some ways, I see Julia actually as a more productive C++ instead of a faster Python. At least as a package developer I tend to use it like that, and a lot of its features essentially give you full generic template programming that you'd attempt in C++, just with a lot less code. I think that for many developers, this is a more appropriate way to consider Julia. But that's not Julia's whole audience, which leads to my next point.

YOU WILL SEE THE MOST BENEFITS IN PACKAGE DEVELOPMENT

How much does this language choice actually effect real use cases? Well, I think it depends on who you are. One thing that you'll notice is that, for the vast majority of users, the performance of your scripts is almost entirely determined by the speed of your packages. This is true in pretty much any scientific computing application that I know of, in Julia, R, MATLAB, Python, etc. If 90% of your time is spent calling functions from DataFrames or JuMP, then the speed of your analysis code really doesn't impact the final runtime all that much.

So in some sense I don't believe that Julia's performance will directly effect most users of the language (and I think that this is true for R/Python/MATLAB as well). However, where it really manifests itself is indirectly through package development productivity. Simply put, a Python package written in pure Python is slow, like really really slow. Same with R and MATLAB. This impacts the development time and resources since all of the major projects like SciPy, NumPy, and Pandas are almost entirely C++ code underneath. And this also impacts features. It's really hard to make traditional languages handle things like arbitrary precision arithmetic well, and so you'll see that most functionality in things like SciPy do not support all of the glory that Python allows since it's not really Python code! Then Python packages are built on SciPy and its limitations, and that just further propagates the impossibility of extending the algorithms to something more generic. Python in fact has so many issues here that projects have had to internally develop their own JITs in order to get the performance, features, and syntax they want. JitCODE and PyDSTool are two ODE solvers which developed DSLs and ways to compile their code. One nice example is this video on PyTorch where the developer describes the tracing JIT they built into their package. I seriously commend these individuals for building such amazing systems, but dear god I do not want to spend my own development time building a

compilation scheme for each new scientific project I am involved in! To me, these are language level issues and should be addressed by the language. What the PyTorch developers had to build into their package is what Julia gives you package developers for free, and as a scientist who doesn't want to write compilers, I am super satisfied.

In Julia, packages can be written in Julia and be performant. As demonstrated above and in this post, they can also be type-agnostic or type-generic and still be performant. This means that it's much easier/quicker to develop "good" libraries in Julia, and that it's easy for any of these libraries to support just about any type. An example for this is GenericSVD.jl which just implements an SVD-factorization in Julia, and this compiles to fast code for things which are not supported by common Fortran bindings like BigFloats, ArbFloats, complex numbers, etc. and the code is simple. It's a great simple package which likely didn't take much development time but is already something very unique due to its type-genericness and speed.

This makes me think that most users should look at Julia a little differently. To me, Julia is a language which is designed to make it easy to roll your own package and have them be performant. If you want to develop libraries, then Julia is great for you and the benchmarks (among other things like multiple dispatch and type-genericness) show why you should choose Julia over Python/R/MATLAB/etc. To me it's almost a no-brainer to choose Julia for your next methods project or package. But if you're a user who uses libraries, you cannot expect a blanket performance difference from your previous language "just" because it's Julia. For example, Pandas is in Python, but it's fast because it's written in C++. Just because you're switching to Julia you shouldn't expect functions on data table objects to be faster (in fact, they might not be since Pandas is pretty well-optimized. One place where the package ecosystem is currently not as well-optimized is plotting, where the plotting libraries need some restructuring to improve performance.). What you can expect is that the development of the associated Julia libraries is more transparent (it's in Julia) and generally performant as well. But also, all of the tools you'd use to help out your own scripts make it easy to find out what's wrong in a library and contribute some code to speed it up. Going from user to contributor and actually helping package development is super trivial and probably the most underappreciated "feature" (or more, side effect of the design) of Julia.

But given the market for who actually chooses Julia, Julia packages do tend to be fast in the end Julia is really top-of-the-class in many disciplines, though that is not true all around. But the fact that development is fast and easy to write scalable packages in means that you can find a lot of odd unique packages: lots of small numerical linear algebra or new optimization algorithm packages which run performantly and just don't exist anywhere else make up a good chunk of what I find in Julia. To top it off, Julia makes it easy to setup continuous integration in Windows, Mac, and Linux to get your package off the ground and well-tested. Here's a video that goes through all of that. Again, package-development level features.

FAST MULTIPARADIGMS MAKES MAINTENANCE EASIER

The big thing which has helped me scale packages is that Julia doesn't care how you program. Looping, vectorization, functional styles with lots of recursion, etc.: it doesn't care. You can just choose the form that is natural for your problem, write a code which uses that style, and know it's going to be performant. This is very different than the R/MATLAB/Python mentality of "vectorize everything!", even if it's not natural to vectorize. I remember doing some very "fun" hacks in MATLAB and Python specifically in order to make good use of vectorized functions, and having to document inside of comments a step-by-step explanation of what the hell the code is actually doing. Many of those times, a 3 line loop which is almost copy-pasted from the paper's psudocode would've done the trick, but in the name of performance I mangled the code. Julia doesn't require this, and I have very much appreciated this when reviewing code from others.

This is probably the biggest difference. If you have appropriately vectorized code in MATLAB/Python/R, the performance difference from Julia isn't actually too big. There's still a difference because vectorization is particularly wasteful with memory and isn't actually optimal in most cases, but it's within an order of magnitude. However, you are required to program in a specific way in these other languages to get here, and if you're doing anything more than scripting you've probably found that this doesn't scale because, well, sometimes code gets mangled when you force vectorization!

THERE IS BEAUTY IN USING THE FULL LANGUAGE

Sure, Python isn't too bad if for performance you stick to arrays of 64-bit numbers as allowed in NumPy arrays, and you use pre-written functions (C-bindings) in SciPy. But what if you really want to create your own numbers as some object, and you want to build data structures using objects in order to more easily scale your software? This is where this setup begins to take its toll. When saying that a small part of the language is required to be used in order to get good performance, you get pidgeon-holed away from the most intuitive code and have to move towards big arrays of numbers. Instead of "naming" separate arrays, items are thrown into matrices with comments says "1:3 is chemical A, 4:7 is chemical B, ...", which I did not find scales very well.

But in Julia, using types is actually performant. Structs in Julia are value-types, which means that they create structures which inline the type-values instead of using indirection via pointers. This means that a Complex{Float64}, which is a struct of two values (the real and imaginary parts), is in memory as a 128-bit chunk with two numbers. This is as performant as if you implemented some kind of intrinsic type in C. You can use this without worry. And mutable structs, while including pointers, are much more performant than objects in most scripting languages because functions auto-specialize on them. The result is that you can make use of all of the language when building software that needs to be performant, once again making it easier to scale projects.

GPUS AND PARALLELISM IS STRAIGHTFORWARD AND GENERIC

Serial performance only gets you so far. Scaling up your project means two things: scaling up the size and features of the codebase, and scaling up the computing power. I discussed how Julia excels in the first part, but what about the second? What I have found really great about Julia is that its parallelization is dead simple. Add @threads in front of a loop and done, its multithreaded. @parallel and pmap do distributed parallelism via multiprocessing, and I have shown how in 5 minutes you can parallelize your code across a whole cluster. Explicitly writing GPU kernels and using multiple GPUs is easy.

I cannot forget one of my favorite developments as of late:GPUArrays.jl. Essentially, when you write vectorized (broadcasted) code with a GPUArray, it automatically parallelizes on the GPU. Neat! But this means that any generic function written in this style is already compatible to automatically compile a GPU version. So things like OrdinaryDiffEq.jl (the ODE solvers of DifferentialEquations.jl), even though they don't have a single piece of GPU-specific code in them, compile performant versions of their solvers for GPUs, automatically, because Julia feels like being nice. Like seriously, once you get a strong understanding about how types and dispatching works, Julia is absolutely mindblowing.

SUMMARY: JULIA IS GREAT FOR PACKAGE DEVELOPMENT, BUT USERS JUST SEE PACKAGES

And this leads me to my conclusion. In each of the sections above I note why Julia is great for building packages in. In comparison to the other scripting languages, I find nothing comes close in terms of productivity, scalability, resulting performance, and resulting features. No other languages makes it so easy to make a function which is performant yet doesn't care what number types you use! And being allowed to use the whole language "correctly" means that your code is much easier to understand and grow. If you're looking to publish a package along with your algorithm, Julia is definitely the right place to be. In that sense, this group will see Julia as an easier or more productive C++.

But for end users throwing together a 100 line script for a data analysis? I don't think that this crowd will actually see as much of a difference between other scripting languages if the packages in the other languages they are using are sufficiently performant (this isn't always true, but let's assume it is). To people who aren't "pros" in the language, it will probably look like it just has a different syntax. It will be a little faster than vectorized code in other languages if code is in type-stable functions, but most of the differences a user will notice will come from the mixture of features and performance of packages. Because of this, I am not sure if marketing the features of the language is actually the best way to approach the general audience. The general audience will be convinced Julia is worthwhile only by the package offering.

I want to end though with the fact that, since Julia packages are written in Julia, a Julia user is qualified to write, debug, and contribute to packages. I myself never saw myself becoming a package dev until about a year ago, and this transition only was because Julia makes the change so easy (it wasn't any different than the Julia development I was already doing!). While this is a good highlight to people who

would read a blog post about programming languages, I still think this is a small niche when considering the average programmer. I don't think that the average programmer sees this as an upside. Most don't have the time to invest in this kind of development, and see that push that "you can do it all yourself!" as a turn-off (even though it's "can" instead of "have to"!). It's a very different crowd to be catered to.

MY RECOMMENDATIONS

PACKAGE ACCESSIBILITY AND DISCOVERABILITY

So in the end, I see Julia as in a state of transition. The way it was marketed before in terms of performance benchmarks, the type-system, etc. are all things which appeal to package developers, and Julia already has had great adoption by developers. But now Julia needs to start targeting general audiences by sharing its packages. There are a few things in the Base Julia language like adding noalias scopes that I would like to see, but pretty much everything (other than a few compiler optimizations) I put as "not essential" now. What I see as essential is making packages more accessible.

As someone well-aware of the packages which are available, I can tell you that "lack of packages" isn't really a problem with the ecosystem: you can find a great package that does what you're looking for. In fact, this weird idea that Julia doesn't have packages yet leads to some silly chatroom discussions where a newcomer joins the Gitter channel and asks "I wanna contribute to the language... has x been done yet?", give back 10 example, "y?", give back 5 examples, "z?", 5 examples, "oh, I'm surprised how much is already done!".

This chat happens quite often, which is fine, but it points to a problem. We really need to work more on "discoverability" and "noob-friendly docs", both items which I'm not entirely sure what the easiest way to handle are. But, I think that this is what is required for Julia to grow (word choice is on purpose: grow instead of succeed, because Julia to me is clearly already successful enough to sustain a development community and thus invest your own time in it, though the community could grow much more). If we think of newcomers as coming in waves: wave 1 brought in language devs and was extremely successful (as evidenced by the over 500 committers to just the Base language, more than projects like cpython, Cython, or PyPy has ever had!), wave 2 brought in package devs and has been very successful as well, but now we need to gear up for wave 3: the package users. This means that package discoverability is what will bring in the third wave. Tools like JuliaObserver.com need to be better advertised as standard parts of the Julia-sphere.

PACKAGE UNIQUENESS

Julia packages should spend more time on their unique parts. Everyone has optimization packages, and people in Python are using things written in C/C++/Fortran like NLopt and IPOPT and so the performance difference is essentially nil in these well-developed cases. But a native Julia package can have other advantages as well. Working out of the box with complex and arbitrary precision numbers, using autodifferentiation, being compatible with DistributedArrays and GPUArrays to allow these forms of parallelism without having to transfer back to the CPU for the optimizer's parts. These features are huge for large classes of researchers (I know that quantum physicists want better complex number support everywhere!) and this is where Julia's packages shine: genericness. Supporting genericness and parallelism should be front and center with every big Julia package, and it should have unique examples to show it off.

We already have very strong libraries in quite a few domains of scientific computing. Here's what Julia sounds like when, instead of saying "Julia can do _____", you say some unique things about the libraries:

1. Constrainted optimization and mathematical programming. JuMP's DSL automatically defines Jacobians and Hessians via autodifferentiation. Its solver independence makes it easy to switch between libraries to choose the most efficient one for the problem.
2. Numerical linear algebra. Not only does th standard library expose more of BLAS and LAPACK via special matrix types than other languages, there are many special-purpose libraries for linear solving. IterativeSolvers.jl allows you to use any LinearMap, meaning that by passing a lazy LinearMap it automatically works as matrix-free preconditioned GMRES etc., and there are many

(and a growing number) of methods to choose from. This library also supported complex and arbitrary precision numbers. In case you were wondering, there does exist PETSc.jl is a binding to the infamous PETSc library, and it is compatible with MPI.

3. Differential equations. I've already discussed the feature set of DifferentialEquations.jl.
4. Autodifferentiation. With ForwardDiff.jl and its operator overloading approach, pretty much any Julia code can use forward-mode autodifferentiation without any modifications, including the Base library. This means there's almost no reason to use numerical differentiation of any pure-Julia code! Reverse-mode autodifferentiation exists as well, but I'll leave a note pointing to the future Cassette.jl.

I'm not an image processing guy but Images.jl is a large library developed by Tim Holy which must have some unique points. A major contributor to Julia is Steven Johnson, the creator of FFTW (and NLopt which has a Julia binding), the widely used FFT library, is a big Julia contributor and there has been discussions about building a generic FFT library in Julia. In addition, many applied mathematicians are pointing to mixed-precision algorithms as a big possibility for increasing the performance of scientific applications, and Julia's strong control over types and specialization on types is perfect for handling such algorithms.

The main thing is that, if we only talk about "Python does this, does Julia do _____?", then we can only talk about speed. We have lots of unique developments already, so we should change the conversation to highlight the things that packages in Julia can do that other packages cannot.

PACKAGE DISTRIBUTIONS AND BRANDING

Another thing I think we should be doing is bundling together packages as distributions. We somewhat do this with the organizations (JuliaStats, JuliaOpt, etc.), but it's not the same as having a single cohesive documentation. People know and respect SciPy. It's a hodgepodge of different things, but you know who's going to look at your bug reports and you've other parts of SciPy and that's why over time you respect it. It's very hard to get that kind of respect for a lone 5 star github package. I think cohesive documentation for orgs and metapackages, much like I have done with DifferentialEquations.jl is required in order to get big "mainstream" packages that users can know and trust.

"USER"-FOCUSED TUTORIALS

I think Julia can become big since it gives package developers so many tools to make big performant package ecosystems with ease. But this has made most people in the community focused on talking to other "developers". We should reach out to "users" with simple examples and tutorials, and understand that most people don't want to contribute to a package or anything like that, but really just want to add a package and use a function to spit out a plot as fast as possible. For the next wave of Julia users, we should show should how the package ecosystem enables this kind of usage, and that's how Julia will grow.

I believe that we should target teaching resources directly to them, similar to what's seen in other languages. I see workshops for "learn how to do regression in R!", "learn how to build websites with Shiny!", "learn how to use Pandas!", but for Julia I only seem to see "let's learn Julia in depth: how to write fast code and the type system". We should instead run workshops directly on Optim, JuMP, DifferentialEquations, etc. at various universities where we are already "teaching Julia", and have it setup as "direct to skill" for specific disciplines instead of teaching fancy language-level features. Although it's hard because I too am a purist in "learn the language and you can do it all!", I think we need to reach out more to those who only want to do a very specific thing, and train them in Julia for psychology research, Julia for climate models, etc. And honestly, I can't think off of the top of my head a good tutorial that says "here's how to do scientific computing in Julia", and works through some specific issues and skills that piece together a few important libraries. We need to write some resources along these lines.

At least, that's what I think. Feel free to agree/disagree in the comments below.