Julia for R users

I know a lot of R and can do my daily job with it. Why should I learn Julia?

In my case, I was looking for some adventure. Haskell seemed too hard, Python too normal. So I went on a journey to learn Julia and was very happy with what I discovered.

I love R, it is my breadwinner (and has been for the past 6 years), and I know some of its limitations. So below is a (biased) list of features that may interest you in trying Julia:

You can run R inside of Julia

Not sure where to begin in Julia? Start with R!

```julia
using RCall;

R"""
median(1:5)
"""
```

```
RObject{IntSxp}
[1] 3
```

You can even pass objects from Julia to R:

```julia
x = [1:5;]

@rput x

R"median(x)"
```

```
RObject{IntSxp}
[1] 3
```

This quarto notebook is actually run using RCall! See RCall docs for more details.

You can see some differences between R and Julia here.

There is a tidyverse in Julia and it is awesome

Tidier.jl is a data analysis package inspired by R's tidyverse and crafted specifically for Julia. It is made with the macro magic described below. Behind the scenes, it is transformed in usual `Dataframes.jl` code.

Here's an example from TidierData docs:

```julia
using TidierData
using RDatasets

movies = dataset("ggplot2", "movies");

@chain movies begin
    @mutate(Budget = Budget / 1_000_000)
    @filter(Budget >= mean(skipmissing(Budget)))
    @select(Title, Budget)
    @slice(1:5)
end
```

It looks like `dplyr` code with `@`s.

Fast Julia code is written in Julia; fast R code is not written in R

In R, whenever you need some *really* fast code (as fast as you would get in C), you have to use C or Fortran code. R is simply slow. If you need speed in R, you will have to find a package that already implements what you need or learn C/Fortran, use RCpp and pray.

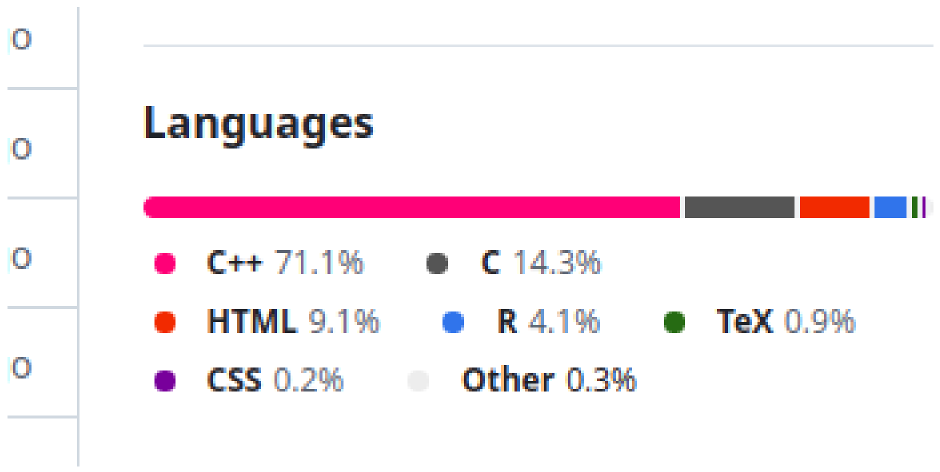


Figure 1: `stringi` package sourcecode.

In Julia, you won't need other language to get speed close to C. That's way they say that Julia solves the two language problem. Julia packages are almost always 100% Julia, which means that you can look to its sourcecode and learn a lot.

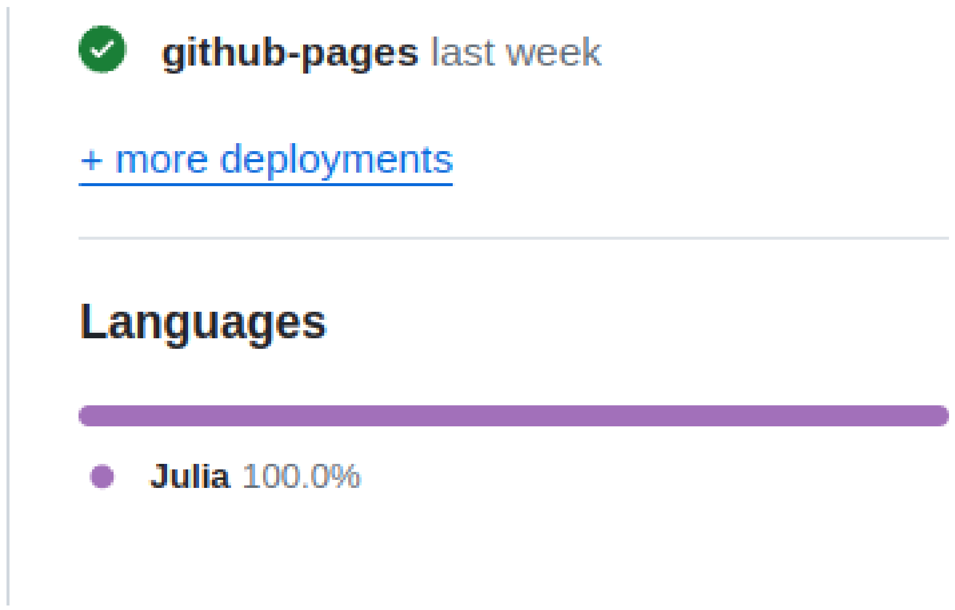
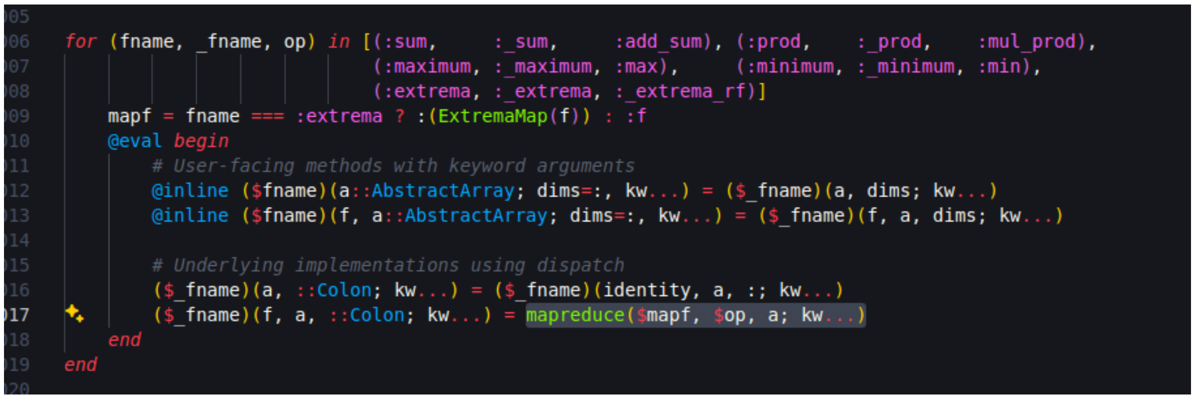


Figure 2: Images that make you cry: the deep learning package Flux.jl.

This is specially interesting if you read Julia Base sourcecode! How does Julia define the maximum of a vector? Type

```julia
@edit maximum([1:5;])
```

and you will see this:



```julia
for (fname, _fname, op) in [(:sum,     :_sum,     :add_sum), (:prod,    :_prod,    :mul_prod),
                            (:maximum, :_maximum, :max),     (:minimum, :_minimum, :min),
                            (:extrema, :_extrema, :_extrema_rf)]
    mapf = fname === :extrema ? :(ExtremaMap(f)) : :f
    @eval begin
        # User-facing methods with keyword arguments
        @inline ($fname)(a::AbstractArray; dims=:, kw...) = ($_fname)(a, dims; kw...)
        @inline ($fname)(f, a::AbstractArray; dims=:, kw...) = ($_fname)(f, a, dims; kw...)

        # Underlying implementations using dispatch
        ($_fname)(a, ::Colon; kw...) = ($_fname)(identity, a, :; kw...)
        ($_fname)(f, a, ::Colon; kw...) = mapreduce($mapf, $op, a; kw...)
    end
end
```

Figure 3: The sourcecode of the function maximum applied to a vector.

It takes some time to grasp the meaning, but in the end it says "apply a mapreduce into the vector, using the max function on each pair of numbers". In R, the sourcecode is a sad `.Primitive("max")`.

No need to vectorize code; loops, maps and broadcast are fast enough

Tired of writing loops? Julia has a special notation `.` (yes, a dot) to apply *any* function to a vector/array/iterable-object; this is called *broadcasting*. For example, you can apply the `power2` function in a vector as easy as

```julia
#julia
# define power2 for numbers
power2(x) = x^2;

# apply in vectors
power2.(1:10)
```

```
10-element Vector{Int64}:
   1
   4
   9
  16
  25
  36
  49
  64
  81
 100
```

or in a matrix

```
#julia
X = reshape([1:16;], (4, 4))
```

```
4×4 Matrix{Int64}:
 1  5   9  13
 2  6  10  14
 3  7  11  15
 4  8  12  16
```

```
#julia
power2.(X)
```

```
4×4 Matrix{Int64}:
  1  25   81  169
  4  36  100  196
  9  49  121  225
 16  64  144  256
```

When using infix functions like `+` or `=`, you put the dot before the operator, as in

```
#julia
[1:5;] .+ 10
```

```
5-element Vector{Int64}:
 11
 12
 13
 14
 15
```

In R, you always try to avoid loops because they are *slow*. Suppose you have a vector and want to sum 1 to every entry. As an experienced R programmer, you look for a vectorized approach:

```
# R
f1_vec = function(x) {
    y = x + 1
}
```

instead of a loop

```
# R
f1_loop = function(x) {
    y = x
    for (i in seq_along(x)) y[i] = x[i] + 1
    y
}
```

or a even a **purrr::map** approach (if you are in a functional programming mood)

```
#R
f1_map = function(x) {
    purrr::map_dbl(x, \(xi) xi + 1)
}
```

because the first options is faster. We can see the difference:

```
# R
x = 1:100000

bench::mark(
    f1_vec(x)
    ,f1_loop(x)
    ,f1_map(x)
    ,relative = TRUE
)
```

```
RObject{VecSxp}
# A tibble: 3 × 13
  expression   min median `itr/sec` mem_alloc `gc/sec` n_itr  n_gc total_time
  <bch:expr> <dbl>  <dbl>     <dbl>     <dbl>    <dbl> <int> <dbl>   <bch:tm>
```

```
1 f1_vec(x)    1      1       125.       1.50    11.0    763    11      500ms
2 f1_loop(x)  43.9   12.6      12.1      1.50     1       74     1      501ms
3 f1_map(x)  559.   151.        1        1        7.83     7     9      575ms
#   4 more variables: result <list>, memory <list>, time <list>, gc <list>
```

In my machine, the loop is ~40x slower and the map ~500x slower than the vectorized version.

The problem is when the function you want to apply have no vectorized form.

In Julia, the three approachs are similar:

```
#julia
f1_vec(x) = x .+ 1;

function f1_loop(x)
    y = similar(x)
    @inbounds for i   eachindex(x) y[i] = x[i] + 1 end
    y
end;

function f1_map(x)
    map(x) do xi
        xi + 1
    end
end;
```

```
#julia
using BenchmarkTools;
x = [1:100000;];

@benchmark f1_vec($x)
```

```
#julia
@benchmark f1_loop($x)
```

```
#julia
@benchmark f1_map($x)
```

This means that in Julia it is usual to *define a function using a scalar type* (a Number like Float64/Int or a String) *and then use broadcast* to apply the function to vectors/matrices/etc. No need to create vectorized forms of functions anymore!

The compiler is your friend

Sometimes you write code that won't make your parents proud. Suppose you create a function that sums all the numbers from 1 to `n`:

```julia
function f_sum(n)
    s = 0
    for i in 1:n
        s += i
    end

    s
end;
```

In R, the language will obediently execute each iteration of the loop, as you demanded. You are the boss.

But in Julia, we have this curious phenomena:

```julia
@time f_sum(100)
```

```
  0.000001 seconds
```

```
5050
```

```julia
@time f_sum(100_000)
```

```
  0.000001 seconds
```

```
5000050000
```

```julia
@time f_sum(100_000_000)
```

```
  0.000001 seconds
```

```
5000000050000000
```

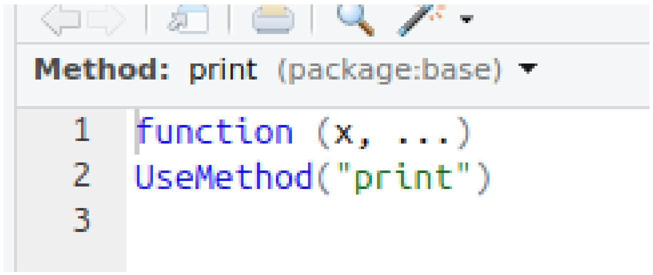
How is it possible that it took the same time for 100 iterations and 100 million? The magic is the compiler: it understood that what you are doing is the sum of the terms of an arithmetic progression. The legend says that Gauss deduced the formula for its sum, and the compiler did the same for you. No need to be as smart as Gauss while using Julia!

Julia is a just-in-time (JIT) compiled language, which means that each function is compiled when you first execute it. The "time to first compile" was a problem in the past, but from Julia 1.8 onwards it is not a big deal.

Multiple dispatch and type system

A type system is a way to organize data types within an hierarchy. Think of it as mathematical sets: you have the real numbers, and inside it are the rationals, the integers and so on. Each one of these types store data into memory in a different manner (integers can be stored more efficiently than arbitrary real numbers, for example). Julia has a really nice type system. Let's see some examples to better understand it.

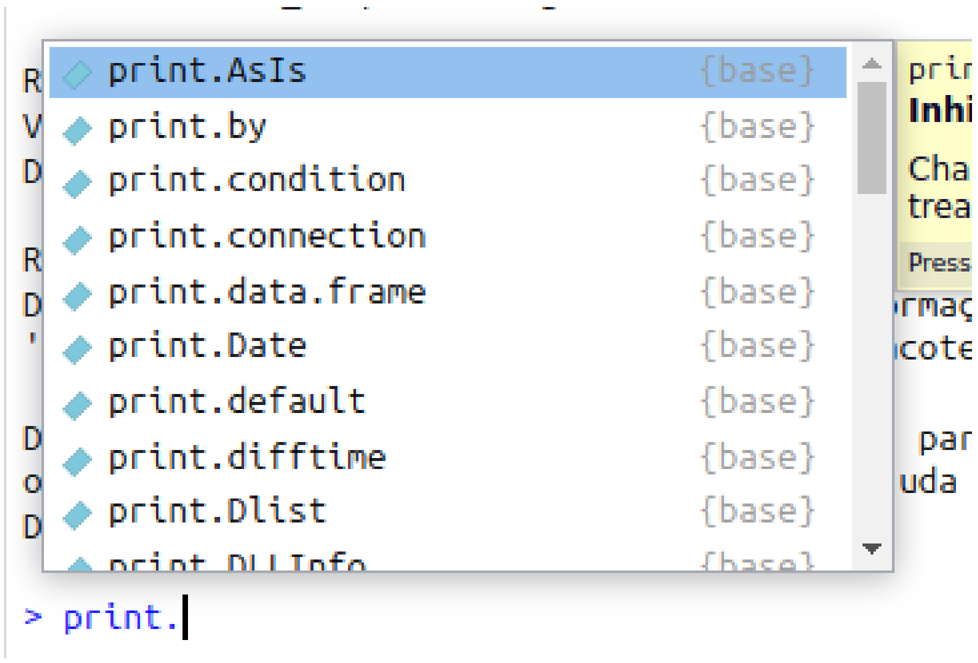
Consider the `print` function in R. It is a *generic* function, which means that its behaviour depends on the class/type of its first argument. This can be seen when we look to its misterious source code:



```
function (x, ...)
UseMethod("print")
```

Figure 4: The `print` function sourcecode.

which means that `print` will use several `methods` (implementations/pieces-of-code), one for each class/type. Actually, R just creates a different function for each class, with the pattern `{function}.{class}`:



```
> print.
```

Figure 5: Each method/implementation of the generic function `print`.

In Julia, *every function is generic*. This means that we can use the same function name and define different behaviours/implementations for each combination of classes/types of its arguments. We saw above the implementation of the `maximum` function in Julia for an arbitrary vector of numbers. But what if the vector is of a different kind, which is easier to determine the maximum?

Take, for example, the object

```
x = 1:10
```

```
1:10
```

It is *not* a vector (to create a usual vector from 1 to 10, you type `[1:10;]`). Actually, its type is

```
typeof(x)
```

```
UnitRange{Int64}
```

and its type hierarchy is

```
Base.show_supertypes(typeof(x))
```

```
UnitRange{Int64} <: AbstractUnitRange{Int64} <: OrdinalRange{Int64, Int64} <: AbstractRange{I
```

So `x` is a much simpler object than a vector: it is an increasing sequence of integer numbers, each 1 unity bigger than the previous one. It makes sense then that the `maximum` function can be defined much more simpler for an object of type `UnitRange{Int64}`: the biggest element is always the last.

If you look at the sourcecode

```
@edit maximum(1:10)
```

you will get

```
maximum(r::AbstractUnitRange) = isempty(r) ? throw(ArgumentError("range must be non-empty"))
```

which is exactly what we thought. The function `last` is defined for every element children of type `AbstractUnitRange`, and so `maximum` is well defined and performant.

In summary: Julia has an arbitrary `maximum` function for arbitrary vectors, but has specialized methods for some other specific types. This is a common pattern in Julia, and much of its performance depends on this.

Macros rewrite code without typing

Macros are one of the most powerful tools in Julia. They rewrite your code before executing it: it is a metaprogramming technique. When creating macros, you will have to understand how a bunch of characters are interpreted by the language and executed as code. This means that macros can rewrite pieces of code and add functionalities that are not possible simply with functions.

We already used some macros on this notebook; they all start with the `@` symbol.

How much time does it take to calculate the sin of a million numbers?

```
@time sin.(1:1_000_000);
```

```
  0.018330 seconds (2 allocations: 7.629 MiB)
```

Do you have a loop and want to see a progress bar? No need to change the code inside the loop:

```
using ProgressMeter;

@showprogress for i in 1:10
    sleep(0.1)
end
```

```
Progress:  20%|                                    |  ETA: 0:00:01Progress:  30%|
```

As we've seen, even the tidyverse could be recreated in Julia using macros!

Multithreading is trivial

Modules and packages are a joy to use

In R, you have 2 options to call a function from another package:

- use `library(PACKAGE)` and then import *every* function from PACKAGE to your namespace;
- use `PACKAGE::FUNCTION` every time you want to use a function.

Packages like `box` are a more "Pythonesque" approach to importing libraries.

In Julia, you have all these for free:

- a
- b
- c

There is also the possibility to create modules inside modules (which are like packages inside packages). For example, if you have a package to train machine learning models, you can have a module about Metrics, another one with Models and so on. Importing then can be done with

```
using MyPackage.Metrics

# or
import MyPackage.Models as MD

MD.model1(etc)
```

Math symbols for the math enthusiasts

Examples of correct Julia code:

```
[1, 2]   [2, 3]
```

```
1-element Vector{Int64}:
 2
```

```
2   [2, 3]
```

```
true
```

```
f(r) =  *r^2

f(3)
```

```
28.274333882308138
```

```
# Euler's identity
 ^(im *  ) + 1 |> round
```

```
0.0 + 0.0im
```