

# Considerando Julia

---

## Una breve exposición de pícaros por qué para considerar si Julia puede ser bueno para tí

---

por Miguel Raz Guzmán (@miguelraz)

Gracias a @categorical y a @xalapacode por la invitación

## Julia ... sí de Ju + Py + R => JuPyteR o.0

---

Julia tiene muchas cosas que ya conocen:

- dinámico, pero ~tan rápido como C
- basado en el paradigma de despacho múltiple (cercano a OOP)
- altamente interactivo
- manejo de memoria automatizado (tiene un GC)
- homoicónico => metaprogramación
- Empezó por hartazgo a MATLAB y otros, pero ha robado lo bueno de los demás
- código libre, todo el desarrollo pasa ahorita en GitHub
- disponible aquí **para descargar (usen 1.6!)**
- joven (circa 2012ish, v1.0 en 2018)
- tiene un manejador de paquetes muuuuy pulido (estilo cargo!)
- te encantan los REPLs e iterar tus diseños rápidamente
- test driven development (TDD) incluido con la paquetería base
- Código de Conducta **internacionalmente reconocido** para evitar 🐻



```
• 🌐 = "Hola XalapaCode! 🏠"
```

```
• function misuma(xs)
•     res = 0
•     for i in xs
•         res += i
•     end
•     res
• end
```

Esta plática con suerte y te va a gustar si

- te gusta la programación funcional estilo Lisp pero no la lentitud 🤡
- te encanta prototipar en Python pero deploy en C++ 🧑
- te gusta el sentimiento de "empoderamiento" que te da el mundo de Rust
- estás buscando dejar todos los dolores de cabeza de `pip install ya-mátame`
- quieres la estadística fácil de R pero da mucho meyo meterse a los lares del compilador
- si quieres que tus loops no sean lentísimos 🐢

`xs =`

```
• xs = 1:100
```

```
• misuma(xs)
```

`α1 =`

```
• α1 = [i for i in 1:10 if i % 2 == 0] # Haskell y Python, alguien?
```

`β =`

```
• β = 3
```

```
• # pueden sacar el ensamblador en Python? 🐞
• with_terminal() do
•     @code_native debuginfo=:none misuma(xs)
• end
```

Esta plática es para contar

- las ventajas de Julia
- las desventajas de Julia
- en qué es muy, muy útil
- para qué no lo es tanto

No vengo a evangelizar o a decir que deberían usar Julia, o que Julia es la respuesta a todo. El chiste es exponer las razones por las cuáles vale la pena considerar a Julia como una alternativa a su workflow si viven con el problema de los 2 lenguajes.

# ¿Cuál es el problema de los 2 lenguajes?

---

Es un problema muy conocido por cualquier que hace bastante cómputo interactivo y de producción.

**Tienes 1 lenguaje** para prototipar (digamos Python, un lenguaje dinámico) usado por un grupo de gente (tus data scientists) y 1 lenguaje para producción (digamos C++, compilado) que usan tus devs de backend.

Todos son felices hasta que un día los requisitos cambian y tu separación de problemas ya no es tan fácil y un grupo tiene que hablar con el otro y acabas reimplementando todo porque tu código no es componible. Aunado a eso, hay una separación de tus equipos, pues ni siquiera hablan el mismo lenguaje por las distintas restricciones.

## Ok, y si me quiero unir al culto de Julia? ¿Reescribo todo o qué?

¡No es necesario! **La interoperabilidad es lo de hoy, no** hay porque dejar de usar herramientas o lenguajes que ya sirven.

## Demo interop

## ¡Oh wow! ¡Esa terminal de Julia se ve súper chula!

Clarín clarinetes, por eso mismo hice un tutorial para aprender todos los truquitos. La **liga aquí merito**

## ¿Cómo es distinto Julia?

"Métele Numpy y ya - no veo el gran problema..."


1. Pierdes algoritmos genéricos
2. Si no juegas dentro del arenero de Numpy, dios te ampare
3. Otra vez tienes la división entre los magos que saben las entrañas de C de Numpy y los usuarios...

Julia es distinto porque

1. Usa un JIT para optimizar el código en run-time

2. usa un algoritmo de subtipado para saber los tipos de los argumentos
3. cachear los resultados para generar código óptimo la primera vez

## Demo

1. Primera vez es lento,
2. Segunda vez 

```
• begin
•     # Esto ya no es Python..
•
•     abstract type AmfibioAbstracto end
•
•     struct Sapo <: AmfibioAbstracto
•     end
•
•     struct Rana <: AmfibioAbstracto
•     end
•
•     pedrada(::AmfibioAbstracto) = "Ouch"
•
•     canta(x::Sapo) = "Ribbit"
•     canta(x::Rana) = "Burp"
•
• end
```

rana =

```
• rana = Rana()
```

sapo =

```
• sapo = Sapo()
```

```
• pedrada(rana)
```

```
• pedrada(sapo)
```

```
• canta(sapo)
```

```
• canta(rana)
```

```
• Enter cell code...
```

Aquí es donde yo digo que Julia es capaz de resolver el **problema de expresión**:

The expression problem is a new name for an old problem.[2][3] The goal is to define a datatype by cases, where one can add new cases to the datatype and new functions over the datatype, without recompiling existing code, and while retaining static type safety

(e.g., no casts).

Como se cubre súper bien en [esta plática de Stefan Karpinski, co-creador de Julia](#)

1. En los lenguajes OOP, es fácil agregar nuevos tipos/clases a los que les aplican operadores ya existentes, pero difícil definir nuevas operaciones a tipos ya existentes
2. En los lenguajes funcionales, es fácil definir nuevas operaciones a tipos ya existentes, pero difícil definir nuevos tipos a los que les apliquen métodos ya existentes.

```
# Sin embargo en Julia puedo hacer esto
struct Renacuajo <: AmfibioAbstracto # Esto cuesta en C++
end
```

```
canta(x::Renacuajo) = "Mimimimimi" # Esto cuesta en Lisp/Haskell
```

a =

```
a = canta(Sapo())
```

b =

```
b = canta(Rana())
```

c =

```
c = canta(Renacuajo())
```

```
# Pero ya tengo esto de a gratis, pues Renacuajo <: Amfibio
pedrada(Renacuajo())
```

## En Julia no existen los problemas de identidad, o de Rectángulo/Cuadrado, porque los operadores no le pertenecen a nadie

3.foo(2) vs foo(x::Rana, y::Sapo) = ...

N.B: Julia no fue el primer lenguaje en tener despacho múltiple. Viene desde Lisp en los 80s, y Dylan lo implementó, pero no de una manera fácil, reusable, y rápida para que se adoptara en todo su ecosistema. Es difícil pensarlo sin JITs, que son algo recientes.

## ¡Esto no es nuevo! Pero sí choca con OOP...

1.  $1/2 + 1/2$

2.  $.5 + .5$

3.  $.5 + 1/2$

```
• 1//2 + 1//2
```

```
• .5 + .5
```

```
• .5 + 1//2
```

```
• Enter cell code...
```

```
• +(x::Rana, y::Sapo) = "lol"
```

```
x =
```

```
• x = Rana()
```

```
y =
```

```
• y = Sapo()
```

```
• x + y
```

```
• f(x) = x^2
```

```
• map(f, ["a", 1, 3.0])
```

```
• findfirst(<(0), [1, 2, 3, 4, -5, -8])
```

## Entonces, ¿dónde NO vale la pena Julia?

HOY:

1. Hard real time - puedes apagar el GC para ciertas actividades, pero si eres hiper-sensible a microlatencias probablemente no es lo mejor.
2. Sistemas embebidos - es posible apagar el JIT, pero no ha sido el enfoque de la comunidad
3. Quieres poder copy/pastear décadas de resultados en Stack Overflow y que todo funcione "out of the box". Julia tiene muchas veces mejores herramientas que otros ecosistemas, pero hay que familiarizarse eso no es opción para todo.
4. Microservicios - empezar y apagar Julia puede doler si no conoces el sistema. Pensar que es "otro Python/R" te va a doler, y hay que invertir algo de tiempo en saber la interacción del JIT y el subtipado.

## ¿Dónde vale la pena clavarse más?

1. Leer el manual, y de perdis los "[Performance Tips](#)"
2. [Checa la liga en julialang.org/learning](https://julialang.org/learning)
3. 🗯 con gente en el [Slack/Zulip](#)

4. ¿📺 Videos? las charlas de JuliaCon en Youtube son lo mejor
5. ¿📖 Cursos universitarios? Computational Thinking de MIT
6. ¿Y en español para la 🦄👉🔪? El libro de "Intro A Julia" empieza desde o
7. Dudas en Discourse para discusiones más a gusto (porque luego Stack Overflow es muy escueto)

Sobra, pero sobra...

1. Despacho múltiple
2. Paralelismo estilo fork-join
3. GPUs, TPUs, Deep Learning
4. Manejo de paquetes
5. Deploys con BinaryBuilder
6. Optimización
7. Técnicas del Garbage Collector

Y falta el ecosistema de DifferentialEquations.jl, LightGraphs.jl, JuMP.jl...