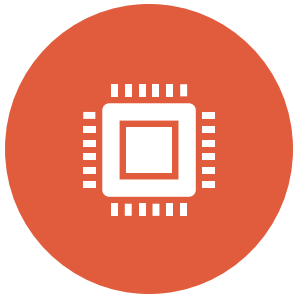
What is Julia and why you should learn it.

Come Julia presenta se stessa

Scientific computing has traditionally required the highest performance, yet domain experts have largely moved to slower dynamic languages for daily work. We believe there are many good reasons to prefer dynamic languages for these applications, and we do not expect their use to diminish. Fortunately, modern language design and compiler techniques make it possible to mostly eliminate the performance trade-off and provide a single environment productive enough for prototyping and efficient enough for deploying performance-intensive applications. The Julia programming language fills this role: it is a flexible dynamic language, appropriate for scientific and numerical computing, with performance comparable to traditional statically-typed languages.



Because Julia's compiler is different from the interpreters used for languages like Python or R, you may find that Julia's performance is unintuitive at first. If you find that something is slow, we highly recommend reading through the Performance Tips section before trying anything else. Once you understand how Julia works, it is easy to write code that is nearly as fast as C.

Fonte: Julia's Documentation

Julia has two priorities

- To be a dynamical language, meaning that variables types are only known at run time. This allows an easier and more intuitive syntax.
- Have performances comparable to statically typed languages like C and Fortran

Let's try to make some quantitative comparisons

PYTHON

```python
import numpy as np


def rand_mul(size):
    a = np.random.rand(size, size)
    b = np.random.rand(size, size)
    return np.dot(a, b)
```

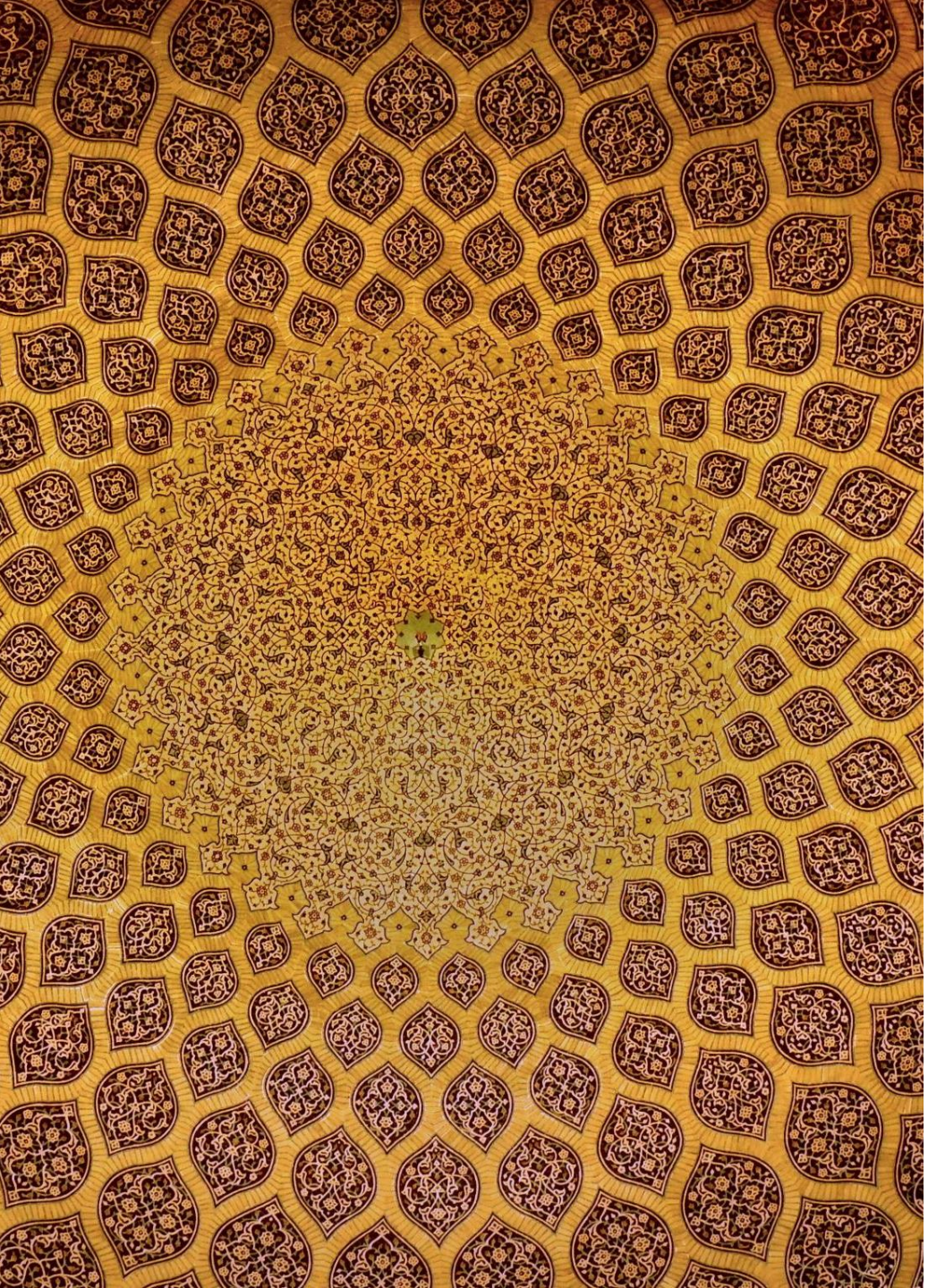
JULIA

```julia
function
    rand_mul(size::Int)
    a = rand(size, size)
    b = rand(size, size)
    return a * b
end
```

FORTRAN

```fortran
function rand_mul_fun(size) result(c)
    integer, intent(in) :: size
    real(8), dimension(size, size) :: a, b,
    c


    call random_number(a)
    call random_number(b)
    c = matmul(a, b)
end function rand_mul_fun
```



Syntax differences

- Unlike Julia and Fortran, Python is not designed for computational purposes, this means that it requires external libraries like NumPy. Julia and Fortran instead have N-dimensional arrays a built-in types.
- Even though Fortran is a static language, it doesn't require a big syntax overhead in this case, if not for the fact to declare variable types before using them.
- Assignments are different between Julia and Python respect to Fortran, the first 2 use "Reference Semantics", it means that the rand function allocates memory, and then does variables are bounded to that memory, while Fortran uses "Value Semantics", it means that the variable itself has the memory, and that memory is then used by the functions.

Performance Difference

- Even though the python function mostly utilizes NumPy, the function is still incredibly inefficient.
- It may look like Fortran has a worst performance compared to Julia, but actually the Fortran code can be further optimized, but to make the comparison fair and keep the same syntactic complexity, this has been avoided.