

Julia

Feels like Python; Works like Lisp; Fast like C

Nick Eubank

CSDI, Vanderbilt University

Resources at github.com/nickeubank/JuliaOverview

April 24, 2018

Goals for Today

1. Why the need for *yet another* language?
2. Overview of Julia features
3. Brief hands-on tutorial
4. Leave you with resources for future exploration!

Who am I?

- Post-Doc at Center for Study of Democratic Institutions
- Study social networks using cell-phone meta-data
 - Lots of simulations on networks with $>10,000,000$ nodes
- Regularly work with Stata, R, Python, and Julia
 - Some contributions to Julia packages, but I am *not* a core Julia developer!

Easy To Use Languages

Python, R, Matlab

Fast Languages

C, Java

Easy To Use Languages

Python, R, Matlab

- Interactive

Fast Languages

C, Java

- Compiled

Easy To Use Languages

Python, R, Matlab

- Interactive
- Dynamic typed

Fast Languages

C, Java

- Compiled
- Static Typed

Easy To Use Languages

Python, R, Matlab

- Interactive
- Dynamic typed
- Fast to write

Fast Languages

C, Java

- Compiled
- Static Typed
- Slow to write

Easy To Use Languages

Python, R, Matlab

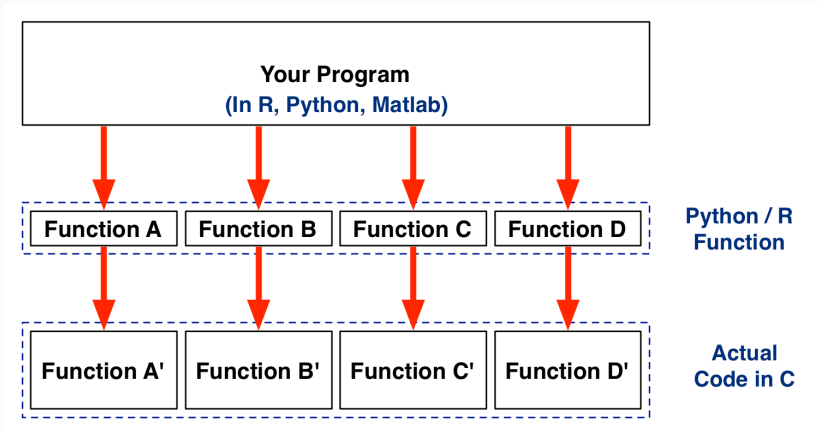
- Interactive
- Dynamic typed
- Fast to write
- Slow to run

Fast Languages

C, Java

- Compiled
- Static Typed
- Slow to write
- Fast to run

Hybrid Solution



Two Language Problem

Two Language Problem

- Hard to **understand** workings of packages

Two Language Problem

- Hard to **understand** workings of packages
- Hard to **modify** packages

Two Language Problem

- Hard to **understand** workings of packages
- Hard to **modify** packages
- Hard to **write** performant packages

Two Language Problem

- Hard to **understand** workings of packages
- Hard to **modify** packages
- Hard to **write** performant packages

⇒ True if you know C...

Two Language Problem

- Hard to **understand** workings of packages
- Hard to **modify** packages
- Hard to **write** performant packages

⇒ True if you know C...

⇒ *Extremely* true if you don't know C!

No Two Language Problem

Base Julia is written *in Julia*

- Even things like definitions of integers!

Most packages written in pure Julia

Python

```
def sum_sequence(start, stop):  
    total = 0  
    for i in range(start, stop):  
        total = total + i  
    return total
```

Julia

```
function sum_sequence(start, stop)  
    total = 0  
    for i in start:stop  
        total = total + i  
    end  
    return total  
end
```

Python

```
def sum_sequence(start, stop):  
    total = 0  
    for i in range(start, stop):  
        total = total + i  
    return total
```

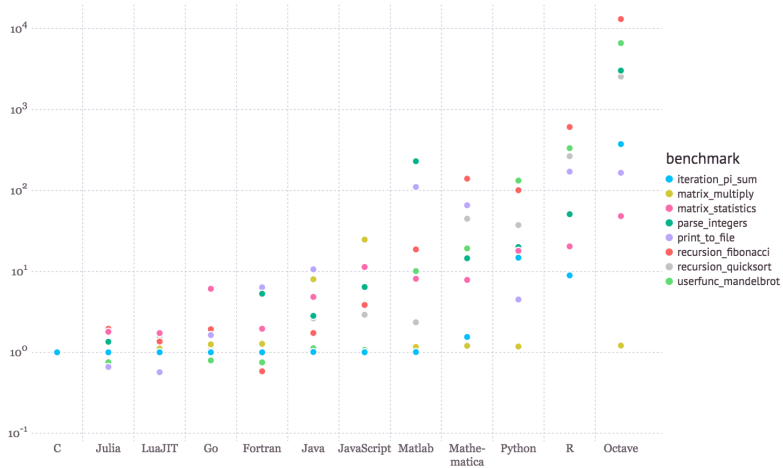
Julia

```
function sum_sequence(start, stop)  
    total = 0  
    for i in start:stop  
        total = total + i  
    end  
    return total  
end
```

Python: sum_sequence(0, 1000000): 78.8 milliseconds

R: sum_sequence(0, 1000000): 274 milliseconds

Julia: sum_sequence(0, 1_000_000): 0.0037 milliseconds



Why Python (and R) are slow

```
# Python
def sum_sequence(start, stop):
    total = 0
    for i in range(start, stop):
        total = total + i
    return total
sum_sequence(0, 10000000)
```

Why Python (and R) are slow

```
# Python
def sum_sequence(start, stop):
    total = 0
    for i in range(start, stop):
        total = total + i
    return total
sum_sequence(0, 10000000)
```

- Your processor doesn't know what "add total and i" means...

Why Python (and R) are slow

```
# Python
def sum_sequence(start, stop):
    total = 0
    for i in range(start, stop):
        total = total + i
    return total
sum_sequence(0, 10000000)
```

- Your processor doesn't know what "add total and i" means...
 - Not all numbers are created equal

Why Python (and R) are slow

```
# Python
def sum_sequence(start, stop):
    total = 0
    for i in range(start, stop):
        total = total + i
    return total
sum_sequence(0, 1000000)
```

- Your processor doesn't know what "add total and i" means...
 - Not all numbers are created equal
 - + actually has different meanings

Why Python (and R) are slow

```
# Python
def sum_sequence(start, stop):
    total = 0
    for i in range(start, stop):
        total = total + i
    return total
sum_sequence(0, 1000000)
```

- Your processor doesn't know what "add total and i" means...
 - Not all numbers are created equal
 - + actually has different meanings

⇒ Checks type of **total**, type of **i**, and looks up appropriate function + one million times!

Why Julia is Fast

```
# Julia
function sum_sequence(start, stop)
    total = 0
    for i in start:stop
        total = total + i
    end
    return total
end
sum_sequence(0, 1_000_000)
```

Why Julia is Fast

```
# Julia
function sum_sequence(start, stop)
    total = 0
    for i in start:stop
        total = total + i
    end
    return total
end
sum_sequence(0, 1_000_000)
```

- Treats function as a small program.

Why Julia is Fast

```
# Julia
function sum_sequence(start, stop)
    total = 0
    for i in start:stop
        total = total + i
    end
    return total
end
sum_sequence(0, 1_000_000)
```

- Treats function as a small program.
- Realizes that `total` and `i` are always going to be integers, so only checks once.

Why Julia is Fast

```
# Julia
function sum_sequence(start, stop)
    total = 0
    for i in start:stop
        total = total + i
    end
    return total
end
sum_sequence(0, 1_000_000)
```

- Treats function as a small program.
- Realizes that `total` and `i` are always going to be integers, so only checks once.
- Keeps copy of machine code once created so doesn't have to re-evaluate every time function is called.

Corollary 1: Julia is only fast inside functions

```
# Slow  
total = 0  
for i in 0:1_000_000  
    total = total + i  
end
```

Corollary 1: Julia is only fast inside functions

```
# Slow
```

```
total = 0
```

```
for i in 0:1_000_000  
    total = total + i  
end
```

```
# Fast
```

```
function sum_sequence(start, stop)  
    total = 0  
    for i in start:stop  
        total = total + i  
    end  
    return total  
end  
sum_sequence(0, 1_000_000)
```

Corollary 2: Type Stability for Max Speed

```
function return_if_even(a_number)
    if a_number % 2 == 0
        return a_number
    end
    if a_number % 2 != 0
        return "This is not even!"
    end
end
```

You can help the compiler by ensuring that, **conditional on the type of arguments**, all intermediate and output variables will always be of the same type.

This function is **not** type stable because:

- If `a_number` is an even integer, it returns an integer, but
- If `a_number` is an odd integer, it returns a string.

Features: Just Write the Loop

No more need to always vectorize!

```
x = rand(100)
```

```
# Loop
```

```
for i in 1:length(x)
    x[i] = sqrt(x[i])
end
```

But you can if you want with `.` notation.

```
# Vectorized
```

```
x = sqrt.(x)
```

Times: 6.651 ms (loop) and 7.682 ms (vectorized)

Features: Native Parallelism

Add workers:

```
addprocs(3)
```

Small jobs:

```
num_heads = @parallel (+) for i in 1:1_000_000  
    rand(Bool)  
end
```

Features: Native Parallelism

Add workers:

```
addprocs(3)
```

Small jobs:

```
num_heads = @parallel (+) for i in 1:1_000_000
    rand(Bool)
end
```

Or:

```
a = SharedArray{Float64}(1_000)
@parallel for i = 1:1_000
    a[i] = randn()
end
```

Features: Parallelism

Big jobs:

```
svds = pmap(svd, list_of_matrices)
```

Features: *Extensive* Linear Algebra Optimizations

```
julia> A = randn(n,n)
julia> Asym = A + A'
julia> issymmetric(Asym)
true
```

Features: *Extensive* Linear Algebra Optimizations

```
julia> A = randn(n,n)
julia> Asym = A + A'
julia> issymmetric(Asym)
true
```

Can also declare special structures (to deal with floating point errors):

- Triangular, Diagonal, Tridiagonal, Sparse Symmetric, etc..

Features: *Extensive* Linear Algebra Optimizations

```
julia> A = randn(n,n)
julia> Asym = A + A'
julia> issymmetric(Asym)
true
```

Can also declare special structures (to deal with floating point errors):

- Triangular, Diagonal, Tridiagonal, Sparse Symmetric, etc..

(Factorizations done using **LAPACK** and **UMFPACK** libraries)

Features: Scientific / Math Types Inbuilt

Base types:

- Rationals
- Imaginary Numbers
- BigInts

Features: Scientific / Math Types Inbuilt

Base types:

- Rationals
- Imaginary Numbers
- BigInts

Plus, user types as fast as Base types.

Features: Easy C Integration

If you need it, use `ccall`.

Features: Easy C Integration

If you need it, use `ccall`. Here's a call to `clock` function in C library `libc` that takes no arguments and returns an `Int32` value:

```
t = ccall((:clock, "libc"), Int32, ())
```

Features: Easy Python Integration

Import python `math` function and use its functions in Julia.

```
using PyCall
```

```
@pyimport math
```

```
math.sin(math.pi / 4) - sin(pi / 4)
```

Features: Support for Unicode

OLS with Unicode:

```
N = 4000
```

```
x = randn(N, 3)
```

```
ε = randn(N)
```

```
β = [2, 1, 90]
```

```
y = x * β + ε
```

```
 $\hat{\beta} = \text{inv}(x' * x) * x' * y$ 
```

```
 $\hat{\epsilon} = y - x * \hat{\beta}$ 
```

Features: Meta-Programming

You can write Julia code that writes Julia code!

Familiar:

- Duck-typing
- Pass by reference
- Iterators
- List (and array) comprehensions

Unfamiliar:

- No integer overflow checking
 - **SafeInts** package available
- Built in Package Manager
 - No **conda** needed!
 - No name spaces yet; coming in new package manager with 1.0 release.
- Not white-space sensitive
- Indexes start at 1, not 0

- Multiple dispatch for functions
- Loops as fast as vectorized functions

Familiar:

- Multiple dispatch
- Built in package manager

Unfamiliar:

- No integer overflow checking
 - **SafeInts** package available
- Pass-by-reference and mutable / immutable data types
- LOTS of syntactic sugar
- Loops as fast as vectorized functions

Not 1.0 Yet...

Currently Stable Release: 0.6.2 Pending Release: 0.7

- Expected this summer (~ June 2018?)
- 0.7 is 1.0 with depreciation warnings
 - If your code works with 0.7, syntax won't change!

Expected changes

- Handful of syntax changes
- Major compiler improvements for missing data
- New package manager

Hands-on Tutorials!

Go to `juliabox.com`, create an account, and navigate to `tutorials/intro-to-julia`.

Today we'll do:

- 1. Getting Started
- 4. Loops
- 6. Functions
- 10. Multiple Dispatch

Next Steps

See `Eubank_JuliaResources.pdf` at www.github.com/nickeubank/JuliaOverview for links for installing Julia, installing an IDE, where to find tutorials, and more!