The unreasonable effectiveness of Julia

Ayush Patnaik

November 11, 2024

{X}KDR

Julia

Julia was born out of addressing the limitations of MATLAB and other high-level languages.

1. To produce performant code in the existing frameworks, one had to resort to writing the heavy lifting code in other languages or opt for other “inelegant” solutions.
2. The existing scientific computing languages had proprietary code, limiting accessibility to their source code and imposing restrictions on customization and collaborative development.

To supplant the existing framework, Julia had be to significantly better in terms of elegance, performance, and fostering a collaborative ecosystem. This became the core culture of Julia.

Installation

Julia multiplexer gets installed with just a single command. No `sudo` required.

```
curl -fsSL https://install.julialang.org | sh
```

Everything is in the `.julia` folder.

```
ayush@woodpecker:~/.julia$ ls
artifacts  environments  logs      registries
compiled   juliaup       packages  scratchspaces
```

Syntax

Variable declaration

```
a = 10
b = 20
```

Arithmetic operations

```
sum = a + b
product = a * b
quotient = a / b
```

Conditional statement

```
age = 25
if age >= 18
  println("You’re an adult.")
else
  println("You’re a minor.")
```

Define a function

```
function greet(name)
  println("Hello ", name)
end
```

Call the function

```
greet("Julia")
```

Array creation

```
numbers = [1, 2, 3, 4, 5]
```

Loop through elements

```
for num in numbers
  println(num)
end
```

Syntax

Continued

List comprehension

```
squares = [i^2 for i in 1:5]
```

Dictionary creation

```
person = Dict(
"name" => "Alice",
"age" => 30,
"city" => "New York")
```

Access values by key

```
person["name"])
```

Create a matrix

```
A = [1 2 3; 4 5 6; 7 8 9]
```

Basic indexing

```
element = A[2, 3]
second_row = A[2, :]
third_column = A[:, 3]
submatrix = A[1:2, 2:3]
```

Broadcasting example

```
A = [1 2 3; 4 5 6; 7 8 9]
B = [2 2 2; 2 2 2; 2 2 2]
C = A .+ B
```

Multiple Dispatch

Methods are selected based on all argument types

Define methods for different types

```
function process(x::Number,
                 y::Number)
    return x + y
end

function process(x::String,
                 y::String)
    return x * y
end

function process(x::Array,
                 y::Array)
    return x .+ y
end
```

Julia automatically selects correct method

```
process(1, 2)         # 3
process("a", "b")     # "ab"
process([1,2], [3,4]) # [4,6]
```

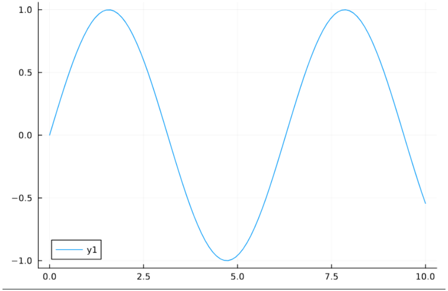
- No explicit method overloading needed
- Dispatch based on all arguments
- Compiler optimizes for each type combination

Multiple Dispatch in action

Same plot() function, different types

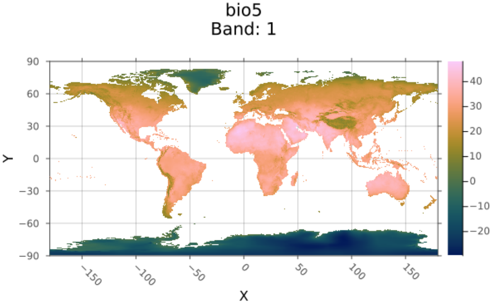
Vector plotting

```
using Plots
x = range(0, 10, length=100)
y = sin.(x)
plot(x, y)
```



Raster plotting

```
using Rasters
A = Raster(WorldClim{BioClim},
           5)
plot(A)
```



Package manager which doesn’t fail

Adding a package

```
using Pkg
Pkg.add("ExamplePackage")
```

Updating packages

```
Pkg.update()
```

Removing a package

```
Pkg.rm("ExamplePackage")
```

Listing installed packages

```
Pkg.status()
```

Thank you.

`https://xkdr.org`

Bibliography I