12in23 - Julia - Trying Julia in Analytical April

Written by Seth Corker on Benevolent Bytes

Cover photo by Hal Gatewood on Unsplash

Why Julia?

The languages for April were much narrower than previous months. Julia, Python and R. I've used Python professionally as well as throughout university, so I thought I'd pick one of the other two instead. Julia sells itself as high-performance and while being dynamically typed. I've been working more with Rust for Mechanical March and the majority of the languages I've used, in my current and previous roles, have been dynamic languages. I thought it might be fun to see what this combination looks like and what the ecosystem has to offer.

Initial thoughts

Nothing in Julia particularly stands out, but maybe that's the point. It feels very familiar on the surface, though I'm sure there's more nuance I didn't uncover in my short time with it. It aims to slot into the scientific computing space and be compared with R, MATLAB and Python. Likewise, it's expressive at a high level, be dynamic to aid in speed of prototyping, but as performant as something like C. It seems to take some of the most useful features from other languages and package them up into something that just works.

Developer experience

Language features

Because Julia isn't particularly surprising or mind shifting like some other languages I've tried, let's have a look at some of the features that are a bit different.

Wider range of symbols APL is known for using esoteric symbols, Julia uses some extra mathematical operators. You can do division normally or use ÷ to truncate the result.

example	name	description
`x / y`	divide	performs division
`x ÷ y`	integer divide	x / y, truncated to an integer

Same with bitwise operations, we've got xor, nand and nor which use , , which I guess must be useful if you're used to seeing these in daily use (I'm not so typing some of these symbols takes some relearning).

example	name
`x | y`	bitwise or
`x  y`	bitwise xor (exclusive or)
`x  y`	bitwise nand (not and)
`x  y`	bitwise nor (not or)
`x >>> y`	logical shift right

Operators for vectors

Being in the scientific computing space means you might be working with vectors, so being able to perform operations on them easily is a must. Julia handles this by allowing every arithmetic operators to be prefixed with a `.`

Shorthand like this makes it easy to express operations on lists of numbers.

```
julia> [1,2,3] .+ 1
3-element Vector{Int64}:
 2
 3
 4
```

You'll also notice that when running these expressions in the REPL, I didn't provide any types, but the types are inferred and shown in the output.

```
julia> [1,2,3] ./ 3
3-element Vector{Float64}:
 0.3333333333333333
 0.6666666666666666
 1.0
```

It's a very natural syntax that's easy to see what's going on, at a glance.

```
julia> [1,2,3] .^ 3
3-element Vector{Int64}:
  1
  8
 27
```

Method specialisation

Julia takes an interesting approach to function definitions. You can define functions that act on specific types or arguments like so:

```
julia> f(x::Float64, y::Float64) = 2x + y
f (generic function with 1 method)
```

You'll notice we have a function with 1 method, this means we only support calls like `f(2.1, 3.5)` and not `f(2,3)`. Likewise, you can create more methods to handle different types, make more specific methods or more generic. If we wanted to handle more numbers, we could use the `Number` type, which `Float64` is a subclass of.

```
f(x::Number, y::Number) = 2x + y
f (generic function with 2 methods)
```

This is a useful pattern I like in Elixir with pattern matching and overloading in C#.

Takeaways

Nothing really stood out to me with Julia and maybe that was a good thing. I sense that it's a language that offers numerous features, and it seems pretty straightforward to get up and running. If an appealing project comes up that uses Julia, I'd be happy to dive back in and take another look, but for now, I'm happy with my time and don't intend to return to it in the short term.

Resources

- Julia on Exercism
- Official Julia website