The Pragmatic Programmers

grox.io Learning

Programmer Passport

Julia

Bruce A. Tate

Edited by Jacquelyn Carter

Programmer Passport: Julia

Bruce Tate

The Pragmatic Bookshelf
Raleigh, North Carolina

Our Pragmatic books, screencasts, and audio books can help you and your team create better software and have more fun. Visit us at *https://pragprog.com*.

For sales, volume licensing, and support, please contact *support@pragprog.com*.

For international rights, please contact *rights@pragprog.com*.

ISBN-13: pending
Encoded using the finest acid-free high-entropy binary digits.
Book version: B1.0—Monthname, yyyy

Contents

Change History

The book you're reading is in beta. This means that we update it frequently. Here is the list of the major changes that have been made at each beta release of the book, with the most recent change first.

B1.0: Sept 15, 2020

- Welcome to the initial release for the Julia language! We're going to focus on the imperative features of Julia. Enjoy!

CHAPTER 1

Julia, the Imperative

Until the last few years, technical computing has always embraced performance as an overriding concern. Now, developers are increasingly embracing the Python language to implement machine learning, one of the most critical segments of technical computing. Dynamic languages like Python provide important *language productivity features* that many scientific languages don't.

Data scientists liked the way Python worked, but it wasn't fast enough. They had to spend large amounts of time optimizing for performance, and that took big chunks of time away from solving the critical concerns of data processing.

In 2009, Jeff Bezanson, Stefan Karpinski, Viral B. Shah, and Alan Edelman started work on Julia to bridge the gap between productivity and performance in technical computing languages. In 2012, the team began to publish their work, launching a web site promoting it.

Today, the language approaches 20 million downloads, supports a major conference, and is taught in universities around the world. Customers from Disney to Apple attest to its growing adoption. With just a little peek under the hood, you can see why.

Throughout the language, you can see powerful features supporting productivity, performance, and friendly features for scientists. The macro system makes higher level abstractions available, so most of Julia is built in Julia itself. Advanced concurrency features allow a higher level of parallelism than most other technical languages support. A rich type system provides the extra performance, documentation, and protection when you want it. Multiple dispatch allows an unusual amount of code reuse. Extensive support for type inference lets you specify types when it matters, and leave them off when you don't want them.

Control structures, method multiple dispatch, powerful libraries, commercial investment, and community support are fueling rapid adaption. The Julia team is assertively making a pitch to be a major player for scientific computing.

This book serves as a companion to the Groxio Julia language programming course. It's self-guided study links to other resources on the web, which are plentiful. For that reason, it will be short. This book will put Julia in the context of other programming languages. We'll look at the things that make it unique, demonstrating the major features. Then we'll get out of the way so you can explore.

The exercises at the end of this chapter will be a little more involved than the ones for the other chapters. We've curated a few interesting learning resources for you elsewhere. We also have a set of videos that demonstrate important Julia features, and how to use them.

We are *not* going to give you a full blow-by-blow of the Julia language. You can and should learn *syntax* and *basic types* elsewhere. Instead, both in this book and in our video series, we will focus on elements that shape the way you approach problems. Those coming from a functional language like Elixir will learn how Julia applies mutability in some places but not others. Those coming from Python or Java will learn how to build a project using multiple dispatch instead of inheritance. Those coming from R will learn how to shape code using a general purpose language, and so on.

With those concepts out of the way, let's get started.

Variables and Control Flow

If you haven't already installed Julia,[1] you should do so. We will be working with version 1.5 in this book. We're going to spend a good amount of time programming by example. Let's kick off our exploration in the Julia console, called the REPL. (I open it up by typing `julia`, but you may have to open an application.)

In essence, Julia is a language that makes high-performance libraries for scientists available in a language with general purpose features. Julia seeks to make it easy to use programming language techniques to explore complex datasets. To that end, we're going to show you a few features that will shape your Julia experience.

1. https://julialang.org/downloads/

Variables hold values that may change, and control flow refers to language features can change the order of the execution of programs. This section will focus on both from the REPL, and we'll expand from there. Let's open up a REPL to get started with a value and an interpolation, like this:

```
julia> x = 42
42

julia> println("The ultimate answer is $x")
The ultimate answer is 42
```

Julia's variable assignments and string interpolation look much like they do in other languages. Let's look at some more nuanced parts of the language.

Loops

If you are coming from a functional language, you should understand right away that Julia embraces some functional concepts, but is not a pure functional language and doesn't try to be. It's a multi-paradigm language.

Imperative languages use statements that change a program's *state*. Generally, in imperative languages, we focus on *how* to do a job by building a step-by-step list for how to complete a task. *Functional* programs transform inputs to outputs. Because Julia must focus on manipulating large blocks of data in place for performance reasons, you'll see plenty of support for imperative style of programming. Still, Julia supports many functional concepts and encourages that style of programming when possible. Founder Jeff Bezanson says "Good programs are often 80% functional," meaning there's plenty of room for both programming styles in Julia.

Right off the bat, you will see control structures for imperative programming. That means Julia has iteration control structures like `while` and `for`:

```
julia> for i in 1:3; println(i); end
1
2
3
```

That code looks something like Ruby, Java, or Python. The familiar `for` statement is decidedly imperative. It mutates the value of `i` twice. There's a similar structure for `while`:

```
julia> i = 1
1
julia> while i <= 3
         println(i)
         global i += 1
       end
```

```
1
2
3
```

That code gives you a hint that something is different, specifically the global expression. Julia makes us be careful when referencing variables declared outside the scope of an expression.

Julia has functional concepts, but the creators took a pragmatic approach to the language with an eye for adoption. This last statement illustrates why. Though functional concepts are popular today, many Julia users are coming from Python, so having imperative for and while statements makes sense, but features like global make it harder to make mistakes with scope.

Let's look at some of the other ways Julia focuses on the scientific and technical communities.

A Focus on Math and Sciences

Julia is a language built by scientists for scientists. Rapidly translating algorithms from scientific literature requires support for symbols and notations found there.

In high school algebra, you probably learned that 12x + 6 means 12 times x plus 6. Julia lets you use that notation:

```
julia> x = 6
6
julia> 12x + 10
82
```

This notation allows for a few ambiguities in things like hex numbers such as the Hex number 0xb, but the benefits in the friendly representation of formulas greatly outweighs the costs. There are many other examples of expressive functions.

You've seen a for statement, but those with a strong math background will recognize the ∈ symbol for membership. Julia allows both of these forms:

```
julia> for i ∈ 1:5
           println(i)
       end

julia> for i in 1:5
         println(i)
     end
```

The in keyword, ∈, and = are all valid. You can also get the square root of a number, like this:

```
julia> √4
2.0
```

You can work in the family of numbers called *imaginary numbers*. In this family of math, the constant `im` is defined as the square root of `-1`.

You can also work with *rational numbers*. A rational number includes numbers like the fraction `1/3`. There's no perfect representation of that number as a floating decimal, but you can use `//` in Julia to represent such a fraction, like this:

```
julia> 6 * (1 / 3)
2.0
```

```
julia> 6 * 1//3
2//1
```

Building symbols into the language makes it much easier to express mathematical programs reflecting the literature they're based on.

If you want to learn to type a character in the Julia REPL, you can hit the question mark, paste in a special character, and get the instructions for typing it:

```
help?> ∈
"∈" can be typed by \in<tab>
```

```
search: ∈
```

```
  in(x)
```

```
... more documentation about in ...
```

Perfect! In machine learning, this notation will come in handy. We'll be able to express common functions that reflect the traditional mathematical formulas they come from.

Julia has several ways to make assignments in scripts more understandable. That's next.

Multiline Assignments

Since Julia is in large measure about dealing with data, quickly understanding where the various components came from is important. As we work in the REPL, we'll often want to organize assignments across multiple lines. Doing so lets us mark concepts with the names of variables. Consider the differences between the following two code blocks:

```
julia> total = convert(round((3 * 530 * 1.08)))
1717
```

This simple statement doesn't do much, but the intention is difficult to understand. We're working with pennies, so we don't want any fractions laying around. That means we need to round and convert the answer to an integer.

Now, let's look at an alternative with multiline assignments:

```
julia> total = begin
         n = 3
         item_cost = 530
         tax = 0.08
         total = (n * item_cost * (1 + tax)) |> round
         convert(Int64, total)
       end
1717
```

The block of code lets us identify the basic concepts by temporary variable assignments. As you work in Julia to do things like data science, you'll find yourself building small scripts to build values, shaping data for use, and like tasks. The practice of labeling parts of formulas like this one is helpful.

Since you're not limited to a single line of code when you're working in the REPL, you have access to many of Julia's control structures as well.

Julia Types

As you might expect by now, we'll not repeat the full scope of the Julia type system documentation. Instead, we'll lay out a few basic concepts, and point you to the Julia documentation where you can find more. We'll focus on a couple of key pieces of the type system, and how you might interpret what's happening.

Julia is Dynamically Typed

Julia is a *dynamically typed* language. Stefan Karpinski says it this way. "In static languages, expressions have types; in dynamic languages, values have types." He then goes on to say that Julia values have types, but expressions don't. Let's see the type system in action:

```
julia> [1, 2, 3]
3-element Array{Int64,1}:
 1
 2
 3
```

Internally, it's the value that has the type. Based on contextual rules, Julia infers a type. There are ways we can force the type into something slightly

different, like a bigger integer. Since Julia is dynamic, we're going to check the type system at *run time*.

There's an overarching type, called Array, and a parametric[2] type, called Int64. We didn't have to assign this type. Julia inferred it based on program clues.

Let's add two arrays together:

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

Interesting. + between arrays will apply + to the first items, then the second, and so on. Keep in mind that Julia is dynamically typed, and determines which function to call using a mechanism called *multiple dispatch*. We will get to those details a little later.

Let's talk next about the relationships between types.

Julia Has a Type Tree

Before we get into types that depend on others, let's look at some of the base types in the language. You don't have to assume Julia is untyped because it is dynamic. It's non-numeric primitive types are mostly what you'd expect them to be:

```
julia> typeof(true)
Bool
julia> typeof('c')
Char
julia> typeof("char")
String
```

Julia has the types Bool, Char, and String to express boolean, character, and string values. The numeric types are a bit more complex, as you'd expect in a technical computing language.

From the very beginning, Julia has been about efficient representation of data, especially numeric data. In fact, the language supports *type trees*, meaning some types depend on others. Let's explore Julia's type hierarchy.

Julia has libraries for working with trees, so showing part of the type tree is trivial. Let's take a look at the numeric types in the language. Don't worry about the using and Pkg. statements for now.

2. https://docs.julialang.org/en/v1/manual/types/#Parametric-Types

```
julia> using Pkg
julia> Pkg.add("AbstractTrees")
julia> using AbstractTrees

julia> AbstractTrees.children(x::Type) = subtypes(x)
```

We've simply included the package manager, which in turn lets us add packages outside of the Julia core. We add a package called AbstractTrees, and then we tell the library how to find the children for a type. Now we can print out the tree:

```
julia> print_tree(Number)
Number
├─ Complex
└─ Real
   ├─ AbstractFloat
   │  ├─ BigFloat
   │  ├─ Float16
   │  ├─ Float32
   │  └─ Float64
   ├─ AbstractIrrational
   │  └─ Irrational
   ├─ Integer
   │  ├─ Bool
   │  ├─ Signed
   │  │  ├─ BigInt
   │  │  ├─ Int128
   │  │  ├─ Int16
   │  │  ├─ Int32
   │  │  ├─ Int64
   │  │  └─ Int8
   │  └─ Unsigned
   │     ├─ UInt128
   │     ├─ UInt16
   │     ├─ UInt32
   │     ├─ UInt64
   │     └─ UInt8
   └─ Rational
```

Interesting. That tree shows a good amount of Julia's primitive types, and a few composite ones as well. In particular, you can see that some of the classes are *abstract*. You can't actually build data with those types. They exist purely to help categorize child types. You can see that the type we saw inside our array, the Int64, depends on other types. In Julia, we'd say:

```
`Type <: Supertype`
```