O'REILLY®

Think Julia

How to Think Like a Computer Scientist

Grayscale Edition

For Sale in the Indian Subcontinent & Select Countries Only*

*Refer Back Cover

Ben Lauwens &
Allen B. Downey

SPD®

Think Julia

How to Think Like a Computer Scientist

Ben Lauwens and Allen B. Downey

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

SHROFF PUBLISHERS & DISTRIBUTORS PVT. LTD.

Mumbai Bangalore Kolkata New Delhi

Think Julia

by Ben Lauwens and Allen B. Downey

Copyright © 2019 Ben Lauwens and Allen Downey. All rights reserved. ISBN: 978-1-492-04503-8
Originally printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://my.safaribooksonline.com*). For more information, contact our corporate/institutional sales department: (800) 998-9938 or *corporate@oreilly.com*.

Editors: Jonathan Hassell and Melissa Potter
Production Editor: Christopher Faucher
Copyeditor: Rachel Head
Proofreader: Christina Edwards
Indexer: Angela Howard
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Rebecca Demarest

April 2019: First Edition

Revision History for the First Edition: 2019-04-05: First Release
See http://oreilly.com/catalog/errata.csp?isbn=9781492045038 for release details.

First Indian Reprint: April 2019

ISBN: 978-93-5213-829-6

Published by **Shroff Publishers and Distributors Pvt. Ltd.** B-103, Railway Commercial Complex, Sector 3, Sanpada (E), Navi Mumbai 400705 • TEL: (91 22) 4158 4158 • FAX: (91 22) 4158 4141• E-mail : spdorders@shroffpublishers.com
Web : www.shroffpublishers.com Printed at Jasmine Art Printers Pvt. Ltd., Mumbai

For Emeline, Arnaud, and Tibo.

Table of Contents

Preface

In January 2018 I started the preparation of a programming course targeting students without programming experience. I wanted to use Julia, but I found that there existed no book with the purpose of learning to program with Julia as the first programming language. There are wonderful tutorials that explain Julia's key concepts, but none of them pay sufficient attention to learning how to think like a programmer.

I knew the book *Think Python* by Allen Downey, which contains all the key ingredients to learn to program properly. However, this book was based on the Python programming language. My first draft of the course notes was a melting pot of all kinds of reference works, but the longer I worked on it, the more the content started to resemble the chapters of *Think Python*. Soon, the idea of developing my course notes as a port of that book to Julia came to fruition.

All the material was available as Jupyter notebooks in a GitHub repository. After I posted a message on the Julia Discourse site about the progress of my course, the feedback was overwhelming. A book about basic programming concepts with Julia as the first programming language was apparently a missing link in the Julia universe. I contacted Allen to ask if I could start an official port of *Think Python* to Julia, and his answer was immediate: "Go for it!" He put me in touch with his editor at O'Reilly Media, and a year later I was putting the finishing touches on this book.

It was a bumpy ride. In August 2018 Julia v1.0 was released, and like all my fellow Julia programmers I had to do a migration of the code. All the examples in the book were tested during the conversion of the source files to O'Reilly-compatible AsciiDoc files. Both the toolchain and the example code had to be made Julia v1.0–compliant. Luckily, there are no lectures to give in August....

I hope you enjoy working with this book, and that it helps you learn to program and think like a computer scientist, at least a little bit.

— *Ben Lauwens*

Why Julia?

Julia was originally released in 2012 by Alan Edelman, Stefan Karpinski, Jeff Bezanson, and Viral Shah. It is a free and open source programming language.

Choosing a programming language is always subjective. For me, the following characteristics of Julia are decisive:

- Julia is developed as a high-performance programming language.
- Julia uses multiple dispatch, which allows the programmer to choose from different programming patterns adapted to the application.
- Julia is a dynamically typed language that can easily be used interactively.
- Julia has a nice high-level syntax that is easy to learn.
- Julia is an optionally typed programming language whose (user-defined) data types make the code clearer and more robust.
- Julia has an extended standard library and numerous third-party packages are available.

Julia is a unique programming language because it solves the so-called "two languages problem." No other programming language is needed to write high-performance code. This does not mean it happens automatically. It is the responsibility of the programmer to optimize the code that forms a bottleneck, but this can done in Julia itself.

Who Is This Book For?

This book is for anyone who wants to learn to program. No formal prior knowledge is required.

New concepts are introduced gradually and more advanced topics are described in later chapters.

Think Julia can be used for a one-semester course at the high school or college level.

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic
: Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`
: Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

`Constant width bold`
: Shows commands or other text that should be typed literally by the user.

`Constant width italic`
: Shows text that should be replaced with user-supplied values or by values determined by context.

This element signifies a tip or suggestion.

This element signifies a general note.

This element indicates a warning or caution.

Using Code Examples

All code used in this book is available from a Git repository on GitHub. If you are not familiar with Git, it is a version control system that allows you to keep track of the files that make up a project. A collection of files under Git's control is called a "repository." GitHub is a hosting service that provides storage for Git repositories and a convenient web interface.

A convenience package is provided that can be directly added to Julia. Just type **`add https://github.com/BenLauwens/ThinkJulia.jl`** in the REPL in Pkg mode, see "Turtles" on page 35.

The easiest way to run Julia code is by going to *https://juliabox.com* and starting a free session. Both the REPL and a notebook interface are available. If you want to have Julia locally installed on your computer, you can download JuliaPro for free from Julia Computing. It consists of a recent Julia version, the Juno interactive development environment based on Atom, and a number of preinstalled Julia packages. If you are more adventurous, you can download Julia from *https://julialang.org*, install the editor you like (e.g., Atom or Visual Studio Code), and activate the plug-ins for Julia integration. To a local install, you can also add the `IJulia` package and run a Jupyter notebook on your computer.

This book is here to help you get your job done. In general, you may use example code in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Think Julia* by Ben Lauwens and Allen B. Downey (O'Reilly). Copyright 2019 Allen B. Downey, Ben Lauwens, 978-1-492-04503-8."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at *permissions@oreilly.com*.

O'Reilly Online Learning

O'REILLY®

For almost 40 years, *O'Reilly Media* has provided technology and business training, knowledge, and insight to help companies succeed.

Our unique network of experts and innovators share their knowledge and expertise through books, articles, conferences, and our online learning platform. O'Reilly's online learning platform gives you on-demand access to live training courses, in-depth learning paths, interactive coding environments, and a vast collection of text and video from O'Reilly and 200+ other publishers. For more information, please visit *http://oreilly.com*.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at *https://oreil.ly/think-julia*.

To comment or ask technical questions about this book, please send an email to *book-questions@oreilly.com*.

For more information about our books, courses, conferences, and news, see our website at *http://www.oreilly.com*.

Find us on Facebook: *http://facebook.com/oreilly*

Follow us on Twitter: *http://twitter.com/oreillymedia*

Watch us on YouTube: *http://www.youtube.com/oreillymedia*

Acknowledgments

I really want to thank Allen for writing *Think Python* and allowing me to port his book to Julia. Your enthusiasm is contagious!

I would also like to thank the technical reviewers for this book, who made many helpful suggestions: Tim Besard, Bart Janssens, and David P. Sanders.

Thanks to Melissa Potter from O'Reilly Media, who made this a better book. You forced me to do things right and make this book as original as possible.

Thanks to Matt Hacker from O'Reilly Media, who helped me out with the Atlas toolchain and some syntax highlighting issues.

Thanks to all the students who worked with an early version of this book and all the contributors (listed below) who sent in corrections and suggestions.

Contributor List

If you have a suggestion or correction, please send email to *ben.lauwens@gmail.com* or open an issue on GitHub. If I make a change based on your feedback, I will add you to the contributor list (unless you ask to be omitted).

Let me know what version of the book you are working with, and what format. If you include at least part of the sentence the error appears in, that will make it easy for me to search. Page and section numbers are fine, too, but not quite as easy to work with. Thanks!

- Scott Jones pointed out the name change of `Void` to `Nothing`, and this started the migration to Julia v1.0.
- Robin Deits found some typos in Chapter 2.
- Mark Schmitz suggested turning on syntax highlighting.
- Zigu Zhao caught some bugs in Chapter 8.
- Oleg Soloviev caught an error in the URL to add the `ThinkJulia` package.
- Aaron Ang found some rendering and naming issues.
- Sergey Volkov caught a broken link in Chapter 7.
- Sean McAllister suggested mentioning the excellent package `BenchmarkTools`.
- Carlos Bolech sent a long list of corrections and suggestions.
- Krishna Kumar corrected the Markov example in Chapter 18.

CHAPTER 1

The Way of the Program

The goal of this book is to teach you to think like a computer scientist. This way of thinking combines some of the best features of mathematics, engineering, and natural science. Like mathematicians, computer scientists use formal languages to denote ideas (specifically computations). Like engineers, they design things, assembling components into systems and evaluating trade-offs among alternatives. Like scientists, they observe the behavior of complex systems, form hypotheses, and test predictions.

The single most important skill for a computer scientist is *problem solving*. Problem solving means the ability to formulate problems, think creatively about solutions, and express a solution clearly and accurately. As it turns out, the process of learning to program is an excellent opportunity to practice problem-solving skills. That's why this chapter is called "The Way of the Program."

On one level, you will be learning to program, a useful skill by itself. On another level, you will use programming as a means to an end. As we go along, that end will become clearer.

What Is a Program?

A *program* is a sequence of instructions that specifies how to perform a computation. The computation might be something mathematical, such as solving a system of equations or finding the roots of a polynomial, but it can also be a symbolic computation, such as searching for and replacing text in a document, or something graphical, like processing an image or playing a video.

The details look different in different languages, but a few basic instructions appear in just about every language:

Input
: Get data from the keyboard, a file, the network, or some other device.

Output
: Display data on the screen, save it in a file, send it over the network, etc.

Math
: Perform basic mathematical operations like addition and multiplication.

Conditional execution
: Check for certain conditions and run the appropriate code.

Repetition
: Perform some action repeatedly, usually with some variation.

Believe it or not, that's pretty much all there is to it. Every program you've ever used, no matter how complicated, is made up of instructions that look pretty much like these. So you can think of programming as the process of breaking a large, complex task into smaller and smaller subtasks until the subtasks are simple enough to be performed with one of these basic instructions.

Running Julia

One of the challenges of getting started with Julia is that you might have to install it and related software on your computer. If you are familiar with your operating system, and especially if you are comfortable with the command-line interface, you will have no trouble installing Julia. But for beginners, it can be painful to learn about system administration and programming at the same time.

To avoid that problem, I recommend that you start out running Julia in a browser. Later, when you are comfortable with Julia, I'll make suggestions for installing Julia on your computer.

In the browser, you can run Julia on JuliaBox. No installation is required—just point your browser there, log in, and start computing (see Appendix B).

The Julia *REPL* (Read–Eval–Print Loop) is a program that reads and executes Julia code. You can start the REPL by opening a terminal on JuliaBox and typing **`julia`** on the command line. When it starts, you should see output like this:

```
               _
   _       _ _(_)_     |  Documentation: https://docs.julialang.org
  (_)     | (_) (_)    |
   _ _   _| |_  __ _   |  Type "?" for help, "]?" for Pkg help.
  | | | | | | |/ _` |  |
  | | |_| | | | (_| |  |  Version 1.1.0 (2019-01-21)
 _/ |\__'_|_|_|\__'_|  |  Official https://julialang.org/ release
|__/                   |

julia>
```

The first lines contain information about the REPL, so it might be different for you. But you should check that the version number is at least `1.0.0`.

The last line is a *prompt* that indicates that the REPL is ready for you to enter code. If you type a line of code and hit Enter, the REPL displays the result:

```
julia> 1 + 1
2
```

Code snippets can be copied and pasted verbatim, including the `julia>` prompt and any output.

Now you're ready to get started. From here on, I assume that you know how to start the Julia REPL and run code.

The First Program

Traditionally, the first program you write in a new language is called "Hello, World!" because all it does is display the words "Hello, World!" In Julia, it looks like this:

```
julia> println("Hello, World!")
Hello, World!
```

This is an example of a *print statement*, although it doesn't actually print anything on paper. It displays a result on the screen.

The quotation marks in the program mark the beginning and end of the text to be displayed; they don't appear in the result.

The parentheses indicate that `println` is a function. We'll get to functions in Chapter 3.

Arithmetic Operators

After "Hello, World!" the next step is arithmetic. Julia provides *operators*, which are symbols that represent computations like addition and multiplication.

The operators `+`, `-`, and `*` perform addition, subtraction, and multiplication, as in the following examples:

```
julia> 40 + 2
42
julia> 43 - 1
42
julia> 6 * 7
42
```

The operator / performs division:

```
julia> 84 / 2
42.0
```

You might wonder why the result is `42.0` instead of `42`. I'll explain in the next section.

Finally, the operator ^ performs exponentiation; that is, it raises a number to a power:

```
julia> 6^2 + 6
42
```

Values and Types

A *value* is one of the basic things a program works with, like a letter or a number. Some values we have seen so far are `2`, `42.0`, and `"Hello, World!"`.

These values belong to different *types*: `2` is an *integer*, `42.0` is a *floating-point number*, and `"Hello, World!"` is a *string*, so called because the letters it contains are strung together.

If you are not sure what type a value has, the REPL can tell you:

```
julia> typeof(2)
Int64
julia> typeof(42.0)
Float64
julia> typeof("Hello, World!")
String
```

Integers belong to the type `Int64`, strings belong to `String`, and floating-point numbers belong to `Float64`.

What about values like `"2"` and `"42.0"`? They look like numbers, but they are in quotation marks like strings. These are strings too:

```
julia> typeof("2")
String
julia> typeof("42.0")
String
```

When you type a large integer, you might be tempted to use commas between groups of digits, as in `1,000,000`. This is not a legal *integer* in Julia, but it is legal:

```
julia> 1,000,000
(1, 0, 0)
```

That's not what we expected at all! Julia parses `1,000,000` as a comma-separated sequence of integers. We'll learn more about this kind of sequence later.

You can get the expected result using `1_000_000`, however.

Formal and Natural Languages

Natural languages are the languages people speak, such as English, Spanish, and French. They were not designed by people (although people try to impose some order on them); they evolved naturally.

Formal languages are languages that are designed by people for specific applications. For example, the notation that mathematicians use is a formal language that is particularly good at denoting relationships among numbers and symbols. Chemists use a formal language to represent the chemical structure of molecules. And most importantly, programming languages are formal languages that have been designed to express computations.

Formal languages tend to have strict *syntax* rules that govern the structure of statements. For example, in mathematics the statement $3 + 3 = 6$ has correct syntax, but $3 + \ = 3\$6$ does not. In chemistry, H_2O is a syntactically correct formula, but $_2Zz$ is not.

Syntax rules come in two flavors: *tokens* and *structure*. Tokens are the basic elements of the language, such as words, numbers, and chemical elements. One of the problems with $3 + \ = 3\$6$ is that \$ is not a legal token in mathematics (at least as far as I know). Similarly, $_2Zz$ is not legal because there is no element with the abbreviation Zz.

The second type of syntax rule pertains to the way tokens are combined. The equation $3 + \ = 3$ is illegal because even though + and = are legal tokens, you can't have one right after the other. Similarly, in a chemical formula the subscript comes after the element name, not before.

This is @ well-structured Engli$h sentence with invalid t*kens in it. This sentence all valid tokens has, but invalid structure with.

When you read a sentence in English or a statement in a formal language, you have to figure out the structure (although in a natural language you do this subconsciously). This process is called *parsing*.

Although formal and natural languages have many features in common—tokens, structure, and syntax—there are some differences:

Ambiguity
: Natural languages are full of ambiguity, which people deal with by using contextual clues and other information. Formal languages are designed to be nearly or

completely unambiguous, which means that any statement has exactly one meaning, regardless of context.

Redundancy
: In order to make up for ambiguity and reduce misunderstandings, natural languages employ lots of redundancy. As a result, they are often verbose. Formal languages are less redundant and more concise.

Literalness
: Natural languages are full of idiom and metaphor. If I say, "The penny dropped," there is probably no penny and nothing dropping (this idiom means that someone understood something after a period of confusion). Formal languages mean exactly what they say.

Because we all grow up speaking natural languages, it is sometimes hard to adjust to formal languages. The difference between formal and natural language is like the difference between poetry and prose, but more so:

Poetry
: Words are used for their sounds as well as for their meaning, and the whole poem together creates an effect or emotional response. Ambiguity is not only common but often deliberate.

Prose
: The literal meaning of words is more important, and the structure contributes more meaning. Prose is more amenable to analysis than poetry but still often ambiguous.

Programs
: The meaning of a computer program is unambiguous and literal, and can be understood entirely by analysis of the tokens and structure.

Formal languages are more dense than natural languages, so it takes longer to read them. Also, the structure is important, so it is not always best to read from top to bottom, left to right. Instead, you'll learn to parse the program in your head, identifying the tokens and interpreting the structure. Finally, the details matter. Small errors in spelling and punctuation, which you can get away with in natural languages, can make a big difference in a formal language.

Debugging

Programmers make mistakes. For whimsical reasons, programming errors are called *bugs* and the process of tracking them down is called *debugging*.

Programming, and especially debugging, sometimes brings out strong emotions. If you are struggling with a difficult bug, you might feel angry, despondent, or embarrassed.

There is evidence that people naturally respond to computers as if they were people. When they work well, we think of them as teammates, and when they are obstinate or rude, we respond to them the same way we respond to rude, obstinate people.[1]

Preparing for these reactions might help you deal with them. One approach is to think of the computer as an employee with certain strengths, like speed and precision, and particular weaknesses, like lack of empathy and inability to grasp the big picture.

Your job is to be a good manager: find ways to take advantage of the strengths and mitigate the weaknesses. And find ways to use your emotions to engage with the problem, without letting your reactions interfere with your ability to work effectively.

Learning to debug can be frustrating, but it is a valuable skill that is useful for many activities beyond programming. At the end of each chapter there is a section, like this one, with my suggestions for debugging. I hope they help!

Glossary

problem solving
: The process of formulating a problem, finding a solution, and expressing it.

program
: A sequence of instructions that specifies a computation.

REPL
: A program that repeatedly reads input, executes it, and outputs results.

prompt
: Characters displayed by the REPL to indicate that it is ready to take input from the user.

print statement
: An instruction that causes the Julia REPL to display a value on the screen.

operator
: A symbol that represents a simple computation like addition, multiplication, or string concatenation.

1 Reeves, Byron, and Clifford Ivar Nass. 1996. “The Media Equation: How People Treat Computers, Television, and New Media Like Real People and Places.” Chicago, IL: Center for the Study of Language and Information; New York: Cambridge University Press.

value
: A basic unit of data, like a number or string, that a program manipulates.

type
: A category of values. The types we have seen so far are integers (`Int64`), floating-point numbers (`Float64`), and strings (`String`).

integer
: A type that represents whole numbers.

floating-point
: A type that represents numbers with a decimal point.

string
: A type that represents sequences of characters.

natural language
: Any one of the languages that people speak that evolved naturally.

formal language
: Any one of the languages that people have designed for specific purposes, such as representing mathematical ideas or computer programs. All programming languages are formal languages.

syntax
: The rules that govern the structure of a program.

token
: One of the basic elements of the syntactic structure of a program, analogous to a word in a natural language.

structure
: The way tokens are combined.

parse
: To examine a program and analyze the syntactic structure.

bug
: An error in a program.

debugging
: The process of finding and correcting bugs.

Exercises

It is a good idea to read this book in front of a computer so you can try out the examples as you go.

Exercise 1-1

Whenever you are experimenting with a new feature, you should try to make mistakes. For example, in the “Hello, World!” program, what happens if you leave out one of the quotation marks? What if you leave out both? What if you spell `println` wrong?

This kind of experiment helps you remember what you read; it also helps when you are programming, because you get to know what the error messages mean. It is better to make mistakes now and on purpose rather than later and accidentally.

1. In a print statement, what happens if you leave out one of the parentheses, or both?
2. If you are trying to print a string, what happens if you leave out one of the quotation marks, or both?
3. You can use a minus sign to make a negative number like `-2`. What happens if you put a plus sign before a number? What about `2++2`?
4. In math notation, leading zeros are okay, as in `02`. What happens if you try this in Julia?
5. What happens if you have two values with no operator between them?

Exercise 1-2

Start the Julia REPL and use it as a calculator.

1. How many seconds are there in 42 minutes 42 seconds?
2. How many miles are there in 10 kilometers? Note that there are 1.61 kilometers in a mile.
3. If you run a 10-kilometer race in 37 minutes 48 seconds, what is your average pace (time per mile in minutes and seconds)? What is your average speed in miles per hour?

O'REILLY®

Think Julia

If you're just learning how to program, Julia is an excellent JIT-compiled, dynamically typed language with a clean syntax. This hands-on guide uses Julia 1.0 to walk you through programming one step at a time, beginning with basic programming concepts before moving on to more advanced capabilities, such as creating new types and multiple dispatch.

Designed from the beginning for high performance, Julia is a general-purpose language ideal for not only numerical analysis and computational science but also web programming and scripting. Through exercises in each chapter, you'll try out programming concepts as you learn them. *Think Julia* is perfect for students at the high school or college level as well as self-learners and professionals who need to learn programming basics.

- Start with the basics, including language syntax and semantics
- Get a clear definition of each programming concept
- Learn about values, variables, statements, functions, and data structures in a logical progression
- Discover how to work with files and databases
- Understand types, methods, and multiple dispatch
- Use debugging techniques to fix syntax, runtime, and semantic errors
- Explore interface design and data structures through case studies

"This book is a great introduction to fundamental programming concepts and the Julia programming language. Highly recommended!"

—**David P. Sanders**
Faculty of Sciences, Universidad Nacional Autónoma de México

Ben Lauwens is a professor of mathematics at the Royal Military Academy (RMA Belgium). He has a PhD in engineering from KU Leuven and master's degrees from both KU Leuven and RMA.

Allen B. Downey is a professor of computer science at Olin College of Engineering. He's also taught at Wellesley College, Colby College, and UC Berkeley. Allen has a PhD in Computer Science from UC Berkeley and a master's degree from MIT.

PROGRAMMING

ISBN : 978-93-5213-829-6

9 789352 138296

First Edition/2019/Paperback/English

MRP: ₹ 1,000 .00

Twitter: @oreillymedia
facebook.com/oreilly

SPD®

SHROFF PUBLISHERS & DISTRIBUTORS PVT. LTD.