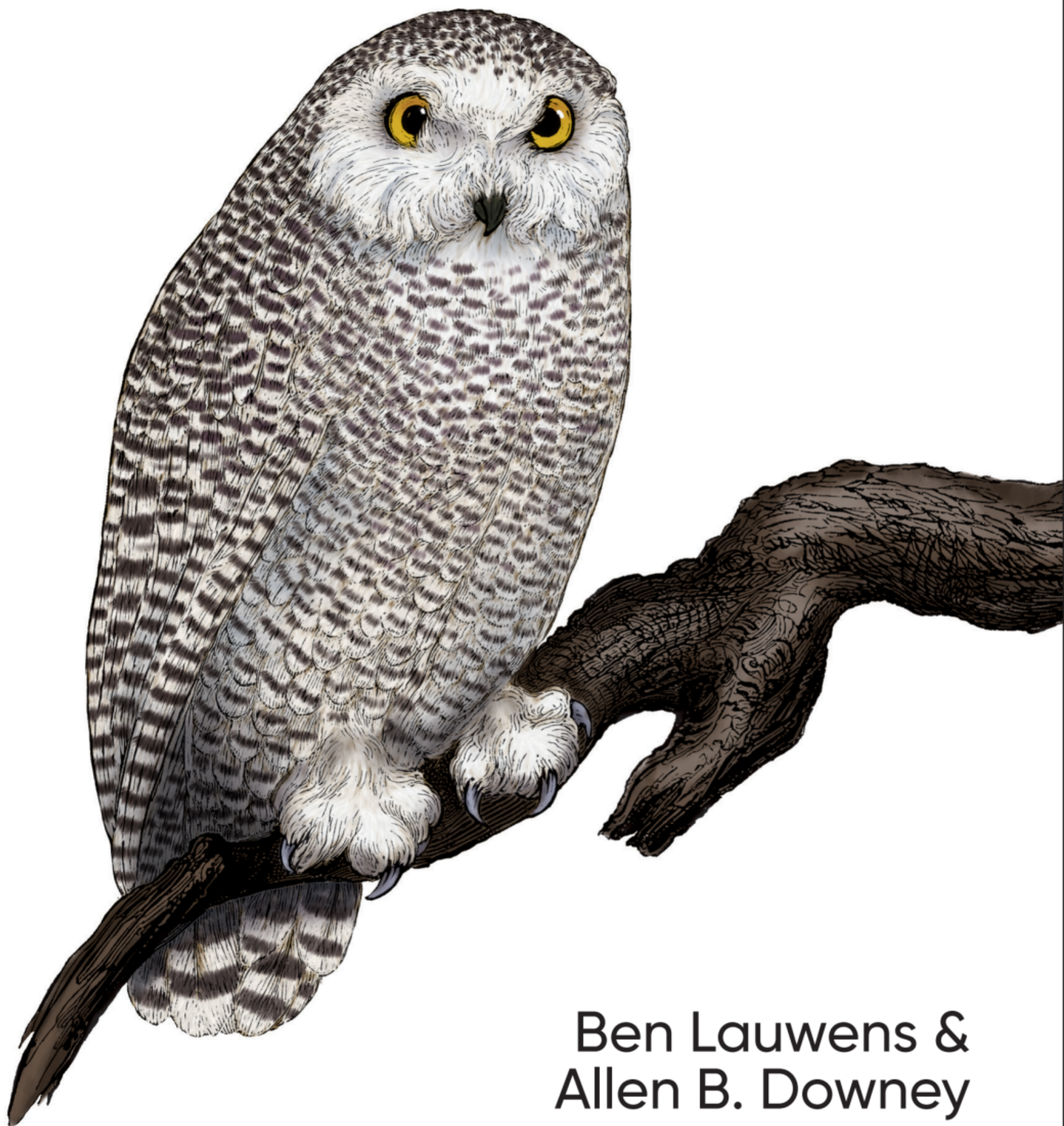
O'REILLY®

Think Julia

How to Think Like a Computer Scientist



Ben Lauwens &
Allen B. Downey

Copyrighted material

Think Julia

How to Think Like a Computer Scientist

Ben Lauwens and Allen B. Downey

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

Copyrighted material

Think Julia
by Ben Lauwens and Allen B. Downey

Copyright © 2019 Ben Lauwens and Allen Downey. All rights reserved.

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://oreilly.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
2019-05-03: Second Release
2019-12-06: Third Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781492045038* for release details.

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. *Think Julia*, the cover image, and related trade dress are trademarks of O'Reilly Media, Inc.

The views expressed in this work are those of the authors, and do not represent the publisher's views. While the publisher and the authors have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the authors disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

978-1-492-04503-8

[LSI]

Copyrighted material

Table of Contents

Preface . . . . **xiii**

1. The Way of the Program **1**
What Is a Program? 1
Running Julia 2
The First Program 3
Arithmetic Operators 3
Values and Types 4
Formal and Natural Languages 5
Debugging 6
Glossary 7
Exercises 9

2. Variables, Expressions, and Statements **11**
Assignment Statements 11
Variable Names 12
Expressions and Statements 12
Script Mode 13
Operator Precedence 14
String Operations 15
Comments 16
Debugging 16
Glossary 17
Exercises 19

3. Functions **21**
Function Calls 21
Math Functions 22

Copyrighted material

Composition 23
Adding New Functions 23
Definitions and Uses 24
Flow of Execution 25
Parameters and Arguments 26
Variables and Parameters Are Local 27
Stack Diagrams 27
Fruitful Functions and Void Functions 28
Why Functions? 29
Debugging 30
Glossary 30
Exercises 32

4. Case Study: Interface Design. 35
Turtles 35
Simple Repetition 36
Exercises 37
Encapsulation 39
Generalization 39
Interface Design 40
Refactoring 41
A Development Plan 42
Docstring 43
Debugging 44
Glossary 44
Exercises 45

5. Conditionals and Recursion. 49
Floor Division and Modulus 49
Boolean Expressions 50
Logical Operators 51
Conditional Execution 51
Alternative Execution 51
Chained Conditionals 52
Nested Conditionals 52
Recursion 53
Stack Diagrams for Recursive Functions 55
Infinite Recursion 55
Keyboard Input 56
Debugging 57
Glossary 58
Exercises 59

Copyrighted material

6. **Fruitful Functions**. 63
Return Values 63
Incremental Development 65
Composition 67
Boolean Functions 67
More Recursion 68
Leap of Faith 71
One More Example 71
Checking Types 72
Debugging 73
Glossary 74
Exercises 74

7. **Iteration**. 77
Reassignment 77
Updating Variables 78
The while Statement 79
break 80
continue 81
Square Roots 81
Algorithms 83
Debugging 84
Glossary 84
Exercises 85

8. **Strings**. 87
Characters 87
A String Is a Sequence 88
length 88
Traversal 89
String Slices 90
Strings Are Immutable 91
String Interpolation 91
Searching 92
Looping and Counting 93
String Library 93
The ∈ Operator 94
String Comparison 94
Debugging 95
Glossary 96
Exercises 97

Copyrighted material

9. Case Study: Word Play.. 101
Reading Word Lists 101
Exercises 102
Search 103
Looping with Indices 104
Debugging 106
Glossary 107
Exercises 107

10. Arrays.. 109
An Array Is a Sequence 109
Arrays Are Mutable 110
Traversing an Array 111
Array Slices 112
Array Library 112
Map, Filter, and Reduce 113
Dot Syntax 115
Deleting (Inserting) Elements 115
Arrays and Strings 116
Objects and Values 117
Aliasing 118
Array Arguments 119
Debugging 121
Glossary 122
Exercises 124

11. Dictionaries.. 127
A Dictionary Is a Mapping 127
Dictionaries as Collections of Counters 129
Looping and Dictionaries 130
Reverse Lookup 131
Dictionaries and Arrays 132
Memos 133
Global Variables 135
Debugging 137
Glossary 138
Exercises 139

12. Tuples.. 141
Tuples Are Immutable 141
Tuple Assignment 142
Tuples as Return Values 143

Copyrighted material

Variable-Length Argument Tuples 144
Arrays and Tuples 145
Dictionaries and Tuples 146
Sequences of Sequences 148
Debugging 149
Glossary 149
Exercises 150

13. Case Study: Data Structure Selection. 153
Word Frequency Analysis 153
Random Numbers 154
Word Histogram 155
Most Common Words 156
Optional Parameters 157
Dictionary Subtraction 158
Random Words 159
Markov Analysis 159
Data Structures 161
Debugging 163
Glossary 164
Exercises 165

14. Files. 167
Persistence 167
Reading and Writing 168
Formatting 168
Filenames and Paths 169
Catching Exceptions 170
Databases 171
Serialization 172
Command Objects 173
Modules 174
Debugging 175
Glossary 176
Exercises 177

15. Structs and Objects. 179
Composite Types 179
Structs Are Immutable 180
Mutable Structs 181
Rectangles 181
Instances as Arguments 182

Copyrighted material

Instances as Return Values 184
Copying 184
Debugging 185
Glossary 186
Exercises 186

16. Structs and Functions. 189
Time 189
Pure Functions 190
Modifiers 191
Prototyping Versus Planning 192
Debugging 194
Glossary 195
Exercises 195

17. Multiple Dispatch. 197
Type Declarations 197
Methods 198
Additional Examples 199
Constructors 200
show 201
Operator Overloading 202
Multiple Dispatch 203
Generic Programming 204
Interface and Implementation 205
Debugging 205
Glossary 206
Exercises 207

18. Subtyping. 209
Cards 209
Global Variables 210
Comparing Cards 211
Unit Testing 211
Decks 212
Add, Remove, Shuffle, and Sort 213
Abstract Types and Subtyping 213
Abstract Types and Functions 215
Type Diagrams 216
Debugging 217
Data Encapsulation 218
Glossary 219

Copyrighted material

Exercises 220

19. The Goodies: Syntax. 223
Named Tuples 223
Functions 224
Blocks 225
Control Flow 227
Types 228
Methods 230
Constructors 230
Conversion and Promotion 231
Metaprogramming 232
Missing Values 234
Calling C and Fortran Code 234
Glossary 235

20. The Goodies: Base and Standard Library. 237
Measuring Performance 237
Collections and Data Structures 238
Mathematics 240
Strings 240
Arrays 241
Interfaces 243
Interactive Utilities 244
Debugging 245
Glossary 246

21. Debugging. 247
Syntax Errors 247
Runtime Errors 249
Semantic Errors 253

A. Unicode Input. 257

B. JuliaBox. 259

Index. 263

Copyrighted material

Copyrighted material

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*

Copyrighted material

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.

Copyrighted material

`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 develop-

Copyrighted material

ment 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

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*.

Copyrighted material

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 *bookquestions@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.

Copyrighted material

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.

Copyrighted material

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:

Copyrighted material

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:

Copyrighted material

```
               _
   _       _ _(_)_     |  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:

Copyrighted material

```
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)
```


Copyrighted material

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

Copyrighted material

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*.

Copyrighted material

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.

Copyrighted material

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.

Copyrighted material

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?

Copyrighted material

Copyrighted material

CHAPTER 2

Variables, Expressions, and Statements

One of the most powerful features of a programming language is the ability to manipulate *variables*. A variable is a name that refers to a value.

Assignment Statements

An *assignment statement* creates a new variable and gives it a value:

```
julia> message = "And now for something completely different"
"And now for something completely different"
julia> n = 17
17
julia> π_val = 3.141592653589793
3.141592653589793
```

This example makes three assignments. The first assigns a string to a new variable named `message`, the second assigns the integer `17` to `n`, and the third assigns the (approximate) value of π to `π_val` (**`\pi TAB`**).

A common way to represent variables on paper is to write the name of each with an arrow pointing to its value. This kind of figure is called a *state diagram* because it shows what state each of the variables is in (think of it as the variable's state of mind). Figure 2-1 shows the result of the previous example.

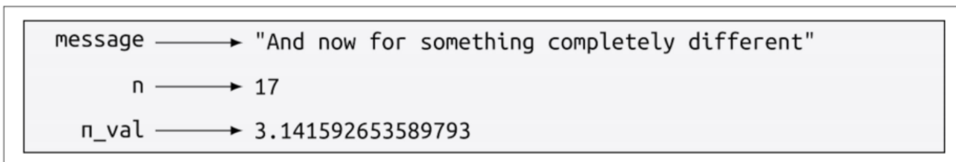


Figure 2-1. State diagram

Copyrighted material

Variable Names

Programmers generally choose names for their variables that are meaningful—they document what the variable is used for.

Variable names can be as long as you like. They can contain almost all Unicode characters (see "Characters" on page 87), but they can't begin with a number. It is legal to use uppercase letters, but it is conventional to use only lowercase for variable names.

Unicode characters can be entered via tab completion of LaTeX-like abbreviations in the Julia REPL.

The underscore character, _, can appear in a name. It is often used in names with multiple words, such as `your_name` or `airspeed_of_unladen_swallow`.

If you give a variable an illegal name, you get a syntax error:

```
julia> 76trombones = "big parade"
ERROR: syntax: "76" is not a valid function argument name
julia> more@ = 1000000
ERROR: syntax: extra token "@" after end of expression
julia> struct = "Advanced Theoretical Zymurgy"
ERROR: syntax: unexpected "="
```

`76trombones` is illegal because it begins with a number. `more@` is illegal because it contains an illegal character, `@`. But what's wrong with `struct`?

It turns out that `struct` is one of Julia's *keywords*. The REPL uses keywords to recognize the structure of the program, and they cannot be used as variable names.

Julia has these keywords:

```
abstract type   baremodule   begin     break     catch
const           continue     do        else      elseif
end             export       finally   for       false
function        global       if        import    in
let             local        macro     module    mutable struct
primitive type  quote        return    true      try
using           struct       where     while
```

You don't have to memorize this list. In most development environments, keywords are displayed in a different color; if you try to use one as a variable name, you'll know.

Expressions and Statements

An *expression* is a combination of values, variables, and operators. A value all by itself is considered an expression, and so is a variable, so the following are all legal expressions:

```
julia> 42
42
```


Copyrighted material

```
julia> n
17
julia> n + 25
42
```

When you type an expression at the prompt, the REPL *evaluates* it, which means that it finds the value of the expression. In this example, `n` has the value `17` and `n + 25` has the value `42`.

A *statement* is a unit of code that has an effect, like creating a variable or displaying a value:

```
julia> n = 17
17
julia> println(n)
17
```

The first line here is an assignment statement that gives a value to `n`. The second line is a print statement that displays the value of `n`.

When you type a statement, the REPL *executes* it, which means that it does whatever the statement says.

Script Mode

So far we have run Julia in *interactive mode*, which means that you interact directly with the REPL. Interactive mode is a good way to get started, but if you are working with more than a few lines of code, it can be clumsy.

The alternative is to save code in a file called a *script* and then run Julia in *script mode* to execute the script. By convention, Julia scripts have names that end with *.jl*.

If you know how to create and run a script on your computer, you are ready to go. Otherwise I recommend using JuliaBox again. Open a text file, write the script, and save the file with a *.jl* extension. The script can be executed in a terminal with the command **`julia`** ***`name_of_the_script`*****`.jl`**.

Because Julia provides both modes, you can test bits of code in interactive mode before you put them in a script. But there are differences between interactive mode and script mode that can be confusing.

For example, if you are using Julia as a calculator, you might type:

```
julia> miles = 26.2
26.2
julia> miles * 1.61
42.182
```


Copyrighted material

The first line assigns a value to `miles` and displays the value. The second line is an expression, so the REPL evaluates it and displays the result. It turns out that a marathon is about 42 kilometers.

But if you type the same code into a script and run it, you get no output at all. In script mode an expression, all by itself, has no visible effect. Julia actually evaluates the expression, but it doesn't display the value unless you tell it to:

```
miles = 26.2
println(miles * 1.61)
```

This behavior can be confusing at first.

A script usually contains a sequence of statements. If there is more than one statement, the results appear one at a time as the statements execute.

For example, the script:

```
println(1)
x = 2
println(x)
```

produces the output:

```
1
2
```

The assignment statement produces no output.

Exercise 2-1

To check your understanding, type the following statements in the Julia REPL and see what they do:

```
5
x = 5
x + 1
```

Now put the same statements in a script and run it. What is the output? Modify the script by transforming each expression into a print statement and then run it again.

Operator Precedence

When an expression contains more than one operator, the order of evaluation depends on the *operator precedence*. For mathematical operators, Julia follows mathematical convention. The acronym *PEMDAS* is a useful way to remember the rules:

- *P*arentheses have the highest precedence and can be used to force an expression to evaluate in the order you want. Since expressions in parentheses are evaluated first, `2*(3-1)` is `4`, and `(1+1)^(5-2)` is `8`. You can also use parentheses to make an

Copyrighted material

expression easier to read, as in `(minute * 100) / 60`, even if it doesn't change the result.

- *E*xponentiation has the next highest precedence, so `1+2^3` is `9`, not `27`, and `2*3^2` is `18`, not `36`.
- *M*ultiplication and *D*ivision have higher precedence than *A*ddition and *S*ubtraction. So, `2*3-1` is `5`, not `4`, and `6+4/2` is `8`, not `5`.
- Operators with the same precedence are evaluated from left to right (except exponentiation). So in the expression `degrees / 2 * π`, the division happens first and the result is multiplied by `π`. To divide by 2π, you can use parentheses, or write `degrees / 2 / π` or `degrees / 2π`.

I don't work very hard to remember the precedence of operators. If I can't tell by looking at the expression, I use parentheses to make it obvious.

String Operations

In general, you can't perform mathematical operations on strings, even if the strings look like numbers, so the following are illegal:

```
"2" - "1"    "eggs" / "easy"    "third" + "a charm"
```

But there are two exceptions, `*` and `^`.

The `*` operator performs *string concatenation*, which means it joins the strings by linking them end-to-end. For example:

```
julia> first_str = "throat"
"throat"
julia> second_str = "warbler"
"warbler"
julia> first_str * second_str
"throatwarbler"
```

The `^` operator also works on strings; it performs repetition. For example, `"Spam"^3` is `"SpamSpamSpam"`. If one of the values is a string, the other has to be an integer.

This use of `*` and `^` makes sense by analogy with multiplication and exponentiation. Just as `4^3` is equivalent to `4*4*4`, we expect `"Spam"^3` to be the same as `"Spam"*"Spam"*"Spam"`, and it is.

Copyrighted material

Comments

As programs get bigger and more complicated, they get more difficult to read. Formal languages are dense, and it is often difficult to look at a piece of code and figure out what it is doing, or why.

For this reason, it is a good idea to add notes to your programs to explain in natural language what the program is doing. These notes are called *comments*, and they start with the # symbol:

```
# compute the percentage of the hour that has elapsed
percentage = (minute * 100) / 60
```

In this case, the comment appears on a line by itself. You can also put comments at the end of a line:

```
percentage = (minute * 100) / 60   # percentage of an hour
```

Everything from the # to the end of the line is ignored—it has no effect on the execution of the program.

Comments are most useful when they document nonobvious features of the code. It is reasonable to assume that the reader can figure out *what* the code does; it is more useful to explain *why*.

This comment is redundant with the code and useless:

```
v = 5   # assign 5 to v
```

This comment contains useful information that is not in the code:

```
v = 5   # velocity in meters/second
```

Good variable names can reduce the need for comments, but long names can make complex expressions hard to read, so there is a trade-off.

Debugging

Three kinds of errors can occur in a program: syntax errors, runtime errors, and semantic errors. It is useful to distinguish between them in order to track them down more quickly:

Syntax error
: "Syntax" refers to the structure of a program and the rules about that structure. For example, parentheses have to come in matching pairs, so `(1 + 2)` is legal, but `8)` is a syntax error.

Copyrighted material

If there is a syntax error anywhere in your program, Julia displays an error message and quits, and you will not be able to run the program. During the first few weeks of your programming career, you might spend a lot of time tracking down syntax errors. As you gain experience, you will make fewer errors and find them faster.

Runtime error
: The second type of error is a runtime error, so called because the error does not appear until after the program has started running. These errors are also called *exceptions* because they usually indicate that something exceptional (and bad) has happened.

Runtime errors are rare in the simple programs you will see in the first few chapters, so it might be a while before you encounter one.

Semantic error
: The third type of error is "semantic," which means related to meaning. If there is a semantic error in your program, it will run without generating error messages, but it will not do the right thing. It will do something else. Specifically, it will do what you told it to do.

Identifying semantic errors can be tricky because it requires you to work backward by looking at the output of the program and trying to figure out what it is doing.

Glossary

variable
: A name that refers to a value.

assignment
: A statement that assigns a value to a variable.

state diagram
: A graphical representation of a set of variables and the values they refer to.

keyword
: A reserved word that is used to parse a program; you cannot use keywords like `if`, `function`, and `while` as variable names.

expression
: A combination of variables, operators, and values that represents a single result.

evaluate
: To simplify an expression by performing the operations in order to yield a single value.

Copyrighted material

statement
: A section of code that represents a command or action. So far, the statements we have seen are assignments and print statements.

execute
: To run a statement and do what it says.

interactive mode
: A way of using the Julia REPL by typing code at the prompt.

script mode
: A way of using Julia to read code from a script and run it.

script
: A program stored in a file.

operator precedence
: Rules governing the order in which expressions involving multiple mathematical operators and operands are evaluated.

concatenate
: To join two strings end-to-end.

comment
: Information in a program that is meant for other programmers (or anyone reading the source code) and has no effect on the execution of the program.

syntax error
: An error in a program that makes it impossible to parse (and therefore impossible to interpret).

runtime error or exception
: An error that is detected while the program is running.

semantics
: The meaning of a program.

semantic error
: An error in a program that makes it do something other than what the programmer intended.

Copyrighted material

Exercises

Exercise 2-2

Repeating my advice from the previous chapter, whenever you learn a new feature, you should try it out in interactive mode and make errors on purpose to see what goes wrong.

1. We've seen that `n = 42` is legal. What about `42 = n`?
2. How about `x = y = 1`?
3. In some languages every statement ends with a semicolon, `;`. What happens if you put a semicolon at the end of a Julia statement?
4. What if you put a period at the end of a statement?
5. In math notation you can multiply `x` and `y` like this: `x y`. What happens if you try that in Julia? What about `5x`?

Exercise 2-3

Practice using the Julia REPL as a calculator:

1. The volume of a sphere with radius r is $\frac{4}{3}\pi r^3$. What is the volume of a sphere with radius 5?
2. Suppose the cover price of a book is $24.95, but bookstores get a 40% discount. Shipping costs $3 for the first copy and 75 cents for each additional copy. What is the total wholesale cost for 60 copies?
3. If I leave my house at 6:52 a.m. and run 1 mile at an easy pace (8:15 per mile), then 3 miles at tempo (7:12 per mile) and 1 mile at easy pace again, what time do I get home for breakfast?

Copyrighted material

Copyrighted material

CHAPTER 3
Functions

In the context of programming, a *function* is a named sequence of statements that performs a computation. When you define a function, you specify the name and the sequence of statements. Later, you can "call" the function by name.

Function Calls

We have already seen one example of a function call:

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

The name of the function is `println`. The expression in parentheses is called the *argument* of the function.

It is common to say that a function "takes" an argument and "returns" a result. The result is also called the *return value*.

Julia provides functions that convert values from one type to another. The `parse` function takes a string and converts it to any number type, if it can, or complains otherwise:

```
julia> parse(Int64, "32")
32
julia> parse(Float64, "3.14159")
3.14159
julia> parse(Int64, "Hello")
ERROR: ArgumentError: invalid base 10 digit 'H' in "Hello"
```

`trunc` can convert floating-point values to integers, but it doesn't round off; it chops off the fraction part:

```
julia> trunc(Int64, 3.99999)
3
```


Copyrighted material

```
julia> trunc(Int64, -2.3)
-2
```

`float` converts integers to floating-point numbers:

```
julia> float(32)
32.0
```

Finally, `string` converts its argument to a string:

```
julia> string(32)
"32"
julia> string(3.14159)
"3.14159"
```

Math Functions

In Julia, most of the familiar mathematical functions are directly available. The following example uses `log10` to compute a signal-to-noise ratio in decibels (assuming that `signal_power` and `noise_power` are defined). `log`, which computes natural logarithms, is also provided:

```
ratio = signal_power / noise_power
decibels = 10 * log10(ratio)
```

This next example finds the sine of `radians`. The name of the variable is a hint that `sin` and the other trigonometric functions (`cos`, `tan`, etc.) take arguments in radians:

```
radians = 0.7
height = sin(radians)
```

To convert from degrees to radians, divide by 180 and multiply by π:

```
julia> degrees = 45
45
julia> radians = degrees / 180 * π
0.7853981633974483
julia> sin(radians)
0.7071067811865475
```

The value of the variable `π` is a floating-point approximation of π, accurate to about 16 digits.

If you know trigonometry, you can check the previous result by comparing it to the square root of 2 divided by 2:

```
julia> sqrt(2) / 2
0.7071067811865476
```


Copyrighted material

Composition

So far, we have looked at the elements of a program—variables, expressions, and statements—in isolation, without talking about how to combine them.

One of the most useful features of programming languages is their ability to take small building blocks and *compose* them. For example, the argument of a function can be any kind of expression, including arithmetic operators:

```
x = sin(degrees / 360 * 2 * π)
```

and even function calls:

```
x = exp(log(x+1))
```

Almost anywhere you can put a value, you can put an arbitrary expression, with one exception: the left side of an assignment statement has to be a variable name. We'll see exceptions to this later, but as a general rule any other expression on the left side is a syntax error:

```
julia> minutes = hours * 60 # right
120
julia> hours * 60 = minutes # wrong!
ERROR: syntax: "60" is not a valid function argument name
```

Adding New Functions

So far, we have only been using the functions that come with Julia, but it is also possible to add new functions. A *function definition* specifies the name of a new function and the sequence of statements that run when the function is called. Here is an example:

```
function printlyrics()
    println("I'm a lumberjack, and I'm okay.")
    println("I sleep all night and I work all day.")
end
```

`function` is a keyword that indicates that this is a function definition. The name of the function is `printlyrics`. The rules for function names are the same as for variable names: they can contain almost all Unicode characters (see "Characters" on page 87), but the first character can't be a number. You can't use a keyword as the name of a function, and you should avoid having a variable and a function with the same name.

The empty parentheses after the name indicate that this function doesn't take any arguments.

The first line of the function definition is called the *header*; the rest is called the *body*. The body is terminated with the keyword `end`, and it can contain any number of statements. For readability the body of the function should be indented.

Copyrighted material

The quotation marks must be "straight quotes," usually located next to Enter on the keyboard. “Curly quotes,” like the ones in this sentence, are not legal in Julia.

If you type a function definition in interactive mode, the REPL indents to let you know that the definition isn’t complete:

```
julia> function printlyrics()
       println("I'm a lumberjack, and I'm okay.")
```

To end the function, you have to enter `end`.

The syntax for calling the new function is the same as for built-in functions:

```
julia> printlyrics()
I'm a lumberjack, and I'm okay.
I sleep all night and I work all day.
```

Once you have defined a function, you can use it inside another function. For example, to repeat the previous refrain, we could write a function called `repeatlyrics`:

```
function repeatlyrics()
    printlyrics()
    printlyrics()
end
```

And then call `repeatlyrics`:

```
julia> repeatlyrics()
I'm a lumberjack, and I'm okay.
I sleep all night and I work all day.
I'm a lumberjack, and I'm okay.
I sleep all night and I work all day.
```

But that’s not really how the song goes.

Definitions and Uses

Pulling together the code fragments from the previous section, the whole program looks like this:

```
function printlyrics()
    println("I'm a lumberjack, and I'm okay.")
    println("I sleep all night and I work all day.")
end

function repeatlyrics()
    printlyrics()
    printlyrics()
end

repeatlyrics()
```


Copyrighted material

This program contains two function definitions: `printlyrics` and `repeatlyrics`. Function definitions get executed just like other statements, but the effect is to create *function objects*. The statements inside the function do not run until the function is called, and the function definition generates no output.

As you might expect, you have to create a function before you can run it. In other words, the function definition has to run before the function gets called.

Exercise 3-1

Restart the REPL and move the last line of this program to the top, so the function call appears before the definitions. Run the program and see what error message you get.

Now move the function call back to the bottom and move the definition of `printlyrics` after the definition of `repeatlyrics`. What happens when you run this program?

Flow of Execution

To ensure that a function is defined before its first use, you have to know the order statements run in, which is called the *flow of execution*.

Execution always begins at the first statement of the program. Statements are run one at a time, in order, from top to bottom.

Function definitions do not alter the flow of execution of the program, but remember that statements inside a function don't run until the function is called.

A function call is like a detour in the flow of execution. Instead of going to the next statement, the flow jumps to the body of the function, runs the statements there, and then comes back to pick up where it left off.

That sounds simple enough, until you remember that one function can call another. While in the middle of one function, the program might have to run the statements in another function. Then, while running that new function, the program might have to run yet another function!

Fortunately, Julia is good at keeping track of where it is, so each time a function completes, the program picks up where it left off in the function that called it. When it gets to the end of the program, it terminates.

In summary, when you read a program, you don't always want to read from top to bottom. Sometimes it makes more sense if you follow the flow of execution.

Copyrighted material

Parameters and Arguments

Some of the functions we have seen require arguments. For example, when you call `sin` you pass a number as an argument. Some functions take more than one argument: `parse` takes two, a number type and a string.

Inside the function, the arguments are assigned to variables called *parameters*. Here is a definition for a function that takes an argument:

```
function printtwice(bruce)
    println(bruce)
    println(bruce)
end
```

This function assigns the argument to a parameter named `bruce`. When the function is called, it prints the value of the parameter (whatever it is) twice.

This function works with any value that can be printed:

```
julia> printtwice("Spam")
Spam
Spam
julia> printtwice(42)
42
42
julia> printtwice(π)
π = 3.1415926535897...
π = 3.1415926535897...
```

The same rules of composition that apply to built-in functions also apply to programmer-defined functions, so we can use any kind of expression as an argument for `printtwice`:

```
julia> printtwice("Spam "^4)
Spam Spam Spam Spam
Spam Spam Spam Spam
julia> printtwice(cos(π))
-1.0
-1.0
```

The argument is evaluated before the function is called, so in these examples the expressions `"Spam "^4` and `cos(π)` are only evaluated once.

You can also use a variable as an argument:

```
julia> michael = "Eric, the half a bee."
"Eric, the half a bee."
julia> printtwice(michael)
Eric, the half a bee.
Eric, the half a bee.
```


Copyrighted material

The name of the variable we pass as an argument (`michael`) has nothing to do with the name of the parameter (`bruce`). It doesn't matter what the value was called back home (in the caller); here in `printtwice`, we call everybody `bruce`.

Variables and Parameters Are Local

When you create a variable inside a function, it is *local*, which means that it only exists inside the function. For example:

```
function cattwice(part1, part2)
    concat = part1 * part2
    printtwice(concat)
end
```

This function takes two arguments, concatenates them, and prints the result twice. Here is an example that uses it:

```
julia> line1 = "Bing tiddle "
"Bing tiddle "
julia> line2 = "tiddle bang."
"tiddle bang."
julia> cattwice(line1, line2)
Bing tiddle tiddle bang.
Bing tiddle tiddle bang.
```

When `cattwice` terminates, the variable `concat` is destroyed. If we try to print it, we get an exception:

```
julia> println(concat)
ERROR: UndefVarError: concat not defined
```

Parameters are also local. For example, outside `printtwice`, there is no such thing as `bruce`.

Stack Diagrams

To keep track of which variables can be used where, it is sometimes useful to draw a *stack diagram*. Like state diagrams, stack diagrams show the value of each variable, but they also show the function each variable belongs to.

Each function is represented by a *frame*. A frame is a box with the name of a function beside it and the parameters and variables of the function inside it. The stack diagram for the previous example is shown in Figure 3-1.

Copyrighted material

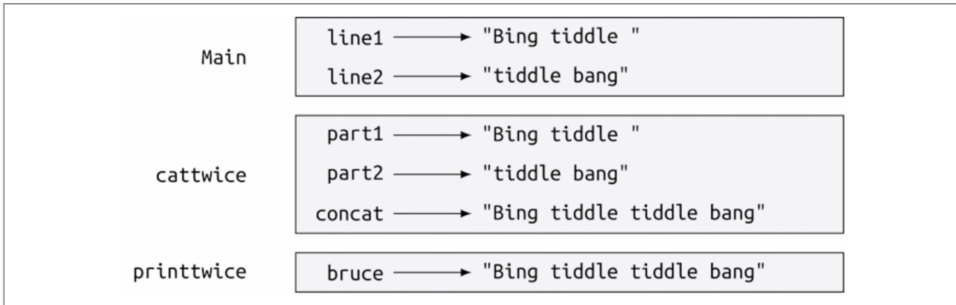


Figure 3-1. Stack diagram

The frames are arranged in a stack that indicates which function called which. In this example, `printtwice` was called by `cattwice`, and `cattwice` was called by `Main`, which is a special name for the topmost frame. When you create a variable outside of any function, it belongs to `Main`.

Each parameter refers to the same value as its corresponding argument. So, `part1` has the same value as `line1`, `part2` has the same value as `line2`, and `bruce` has the same value as `concat`.

If an error occurs during a function call, Julia prints the name of the function, the name of the function that called it, and the name of the function that called *that*, all the way back to `Main`.

For example, if you try to access `concat` from within `printtwice`, you get an `UndefVarError`:

```
ERROR: UndefVarError: concat not defined
Stacktrace:
 [1] printtwice at ./REPL[1]:2 [inlined]
 [2] cattwice(::String, ::String) at ./REPL[2]:3
```

This list of functions is called a *stacktrace*. It tells you what program file the error occurred in, and what line, and what functions were executing at the time. It also shows the line of code that caused the error.

The order of the functions in the stacktrace is the inverse of the order of the frames in the stack diagram. The function that is currently running is at the top.

Fruitful Functions and Void Functions

Some of the functions we have used, such as the math functions, return results; for lack of a better name, I call them *fruitful functions*. Other functions, like `printtwice`, perform an action but don't return a value. They are called *void functions*.

Copyrighted material

When you call a fruitful function, you almost always want to do something with the result. For example, you might assign it to a variable or use it as part of an expression:

```
x = cos(radians)
golden = (sqrt(5) + 1) / 2
```

When you call a function in interactive mode, Julia displays the result:

```
julia> sqrt(5)
2.23606797749979
```

But in a script, if you call a fruitful function all by itself, the return value is lost forever!

```
sqrt(5)
```

This script computes the square root of `5`, but since it doesn't store or display the result, it is not very useful.

Void functions might display something on the screen or have some other effect, but they don't have a return value. If you assign the result to a variable, you get a special value called `nothing`:

```
julia> result = printtwice("Bing")
Bing
Bing
julia> show(result)
nothing
```

To print the value `nothing`, you have to use the function `show`, which is like `print` but can handle this special value.

The value `nothing` is not the same as the string `"nothing"`. It is a special value that has its own type:

```
julia> typeof(nothing)
Nothing
```

The functions we have written so far are all void. We will start writing fruitful functions in a few chapters.

Why Functions?

It may not be clear why it is worth the trouble to divide a program into functions. There are several reasons:

- Creating a new function gives you an opportunity to name a group of statements, which makes your program easier to read and debug.
- Functions can make a program smaller by eliminating repetitive code. Later, if you make a change, you only have to make it in one place.

Copyrighted material

- Dividing a long program into functions allows you to debug the parts one at a time and then assemble them into a working whole.
- Well-designed functions are often useful for many programs. Once you write and debug one, you can reuse it.
- In Julia, functions can improve performance a lot.

Debugging

One of the most important skills you will acquire is debugging. Although it can be frustrating, debugging is one of the most intellectually rich, challenging, and interesting parts of programming.

In some ways debugging is like detective work. You are confronted with clues and you have to infer the processes and events that led to the results you see.

Debugging is also like an experimental science. Once you have an idea about what is going wrong, you modify your program and try again. If your hypothesis was correct, you can predict the result of the modification, and you take a step closer to a working program. If your hypothesis was wrong, you have to come up with a new one. As Sherlock Holmes pointed out,

> When you have eliminated the impossible, whatever remains, however improbable, must be the truth.
>
> —A. Conan Doyle, *The Sign of Four*

For some people, programming and debugging are the same thing. That is, programming is the process of gradually debugging a program until it does what you want. The idea is that you should start with a working program and make small modifications, debugging them as you go.

For example, Linux is an operating system that contains millions of lines of code, but it started out as a simple program Linus Torvalds used to explore the Intel 80386 chip. According to Larry Greenfield in *The Linux Users' Guide* (version beta-1), "One of Linus's earlier projects was a program that would switch between printing AAAA and BBBB. This later evolved to Linux."

Glossary

function
: A named sequence of statements that performs some useful operation. Functions may or may not take arguments and may or may not produce a result.

Copyrighted material

function call
: A statement that runs a function. It consists of the function name followed by an argument list in parentheses.

argument
: A value provided to a function when the function is called. This value is assigned to the corresponding parameter in the function.

return value
: The result of a function. If a function call is used as an expression, the return value is the value of the expression.

composition
: Using an expression as part of a larger expression, or a statement as part of a larger statement.

function definition
: A statement that creates a new function, specifying its name, parameters, and the statements it contains.

header
: The first line of a function definition.

body
: The sequence of statements inside a function definition.

function object
: A value created by a function definition. The name of the function is a variable that refers to a function object.

flow of execution
: The order statements run in.

parameter
: A name used inside a function to refer to the value passed as an argument.

local variable
: A variable defined inside a function. A local variable can only be used inside its function.

stack diagram
: A graphical representation of a stack of functions, their variables, and the values they refer to.

frame
: A box in a stack diagram that represents a function call. It contains the local variables and parameters of the function.

Copyrighted material

stacktrace
: A list of the functions that are executing, printed when an exception occurs.

fruitful function
: A function that returns a value.

void function
: A function that always returns `nothing`.

`nothing`
: A special value returned by void functions.

Exercises

These exercises should be done using only the statements and other features introduced so far.

Exercise 3-2

Write a function named `rightjustify` that takes a string named `s` as a parameter and prints the string with enough leading spaces so that the last letter of the string is in column 70 of the display:

```
julia> rightjustify("monty")
                                                                 monty
```

Use string concatenation and repetition. Also, Julia provides a built-in function called `length` that returns the length of a string, so the value of `length("monty")` is `5`.

Exercise 3-3

A function object is a value you can assign to a variable or pass as an argument. For example, `dotwice` is a function that takes a function object as an argument and calls it twice:

```
function dotwice(f)
    f()
    f()
end
```

Here's an example that uses `dotwice` to call a function named `printspam` twice:

Copyrighted material

```
function printspam()
    println("spam")
end

dotwice(printspam)
```

1. Type this example into a script and test it.
2. Modify `dotwice` so that it takes two arguments, a function object and a value, and calls the function twice, passing the value as an argument.
3. Copy the definition of `printtwice` from earlier in this chapter to your script.
4. Use the modified version of `dotwice` to call `printtwice` twice, passing `"spam"` as an argument.
5. Define a new function called `dofour` that takes a function object and a value and calls the function four times, passing the value as a parameter. There should be only two statements in the body of this function, not four.

Exercise 3-4

1. Write a function `printgrid` that draws a grid like the following:

```
julia> printgrid()
+ - - - - + - - - - +
|         |         |
|         |         |
|         |         |
|         |         |
+ - - - - + - - - - +
|         |         |
|         |         |
|         |         |
|         |         |
+ - - - - + - - - - +
```

2. Write a function that draws a similar grid with four rows and four columns.

Credit: This exercise is based on an exercise in *Practical C Programming*, by Steve Oualline (O'Reilly).

Copyrighted material

To print more than one value on a line, you can print a comma-separated sequence of values:

```
println("+", "-")
```

The function `print` does not advance to the next line:

```
print("+ ")
println("-")
```

The output of these statements is `"+ -"` on the same line. The output from the next print statement would begin on the next line.

Copyrighted material

CHAPTER 4

Case Study: Interface Design

This chapter presents a case study that demonstrates a process for designing functions that work together.

It introduces turtle graphics, a way to create programmatic drawings. Turtle graphics are not included in the standard library, so to use them you'll have to add the `ThinkJulia` module to your Julia setup.

The examples in this chapter can be executed in a graphical notebook on JuliaBox, which combines code, formatted text, math, and multimedia in a single document (see Appendix B).

Turtles

A *module* is a file that contains a collection of related functions. Julia provides some modules in its standard library. Additional functionality can be added from a growing collection of *packages*.

Packages can be installed in the REPL by entering the Pkg REPL mode using the key `]` and using the `add` command:

```
(v1.0) pkg> add https://github.com/BenLauwens/ThinkJulia.jl
```

This can take some time.

Before we can use the functions in a module, we have to import it with a *`using` statement*:

```
julia> using ThinkJulia

julia> 🐢 = Turtle()
Luxor.Turtle(0.0, 0.0, true, 0.0, (0.0, 0.0, 0.0))
```


Copyrighted material

The `ThinkJulia` module provides a function called `Turtle` that creates a `Luxor.Turtle` object, which we assign to a variable named 🐢 (**`\:turtle: TAB`**).

Once you create a turtle, you can call a function to move it around. For example, to move the turtle forward:

```
@svg begin
    forward(🐢, 100)
end
```

The `@svg` keyword runs a macro that draws an SVG picture (Figure 4-1). Macros are an important but advanced feature of Julia.

Figure 4-1. Moving the turtle forward

The arguments of `forward` are the turtle and a distance in pixels, so the actual size of the line that's drawn depends on your display.

Each turtle is holding a pen, which is either down or up; if the pen is down (the default), the turtle leaves a trail when it moves. Figure 4-1 shows the trail left behind by the turtle. To move the turtle without drawing a line, first call the function `penup`. To start drawing again, call `pendown`.

Another function you can call with a turtle as an argument is `turn` for turning. The second argument for `turn` is an angle in degrees.

To draw a right angle, modify the macro call:

```
🐢 = Turtle()
@svg begin
    forward(🐢, 100)
    turn(🐢, -90)
    forward(🐢, 100)
end
```

Exercise 4-1

Now modify the macro to draw a square. Don't go on until you've got it working!

Simple Repetition

Chances are you wrote something like this:

```
🐢 = Turtle()
@svg begin
    forward(🐢, 100)
```


Copyrighted material

```
    turn(🐢, -90)
    forward(🐢, 100)
    turn(🐢, -90)
    forward(🐢, 100)
    turn(🐢, -90)
    forward(🐢, 100)
end
```

We can do the same thing more concisely with a `for` statement:

```
julia> for i in 1:4
          println("Hello!")
       end
Hello!
Hello!
Hello!
Hello!
```

This is the simplest use of the `for` statement; we will see more later. But that should be enough to let you rewrite your square-drawing program. Don't go on until you do.

Here is a `for` statement that draws a square:

```
🐢 = Turtle()
@svg begin
    for i in 1:4
        forward(🐢, 100)
        turn(🐢, -90)
    end
end
```

The syntax of a `for` statement is similar to a function definition. It has a header and a body that ends with the keyword `end`. The body can contain any number of statements.

A `for` statement is also called a *loop* because the flow of execution runs through the body and then loops back to the top. In this case, it runs the body four times.

This version is actually a little different from the previous square-drawing code because it makes another turn after drawing the last side of the square. The extra turn takes more time, but it simplifies the code if we do the same thing every time through the loop. This version also has the effect of leaving the turtle back in the starting position, facing in the starting direction.

Exercises

The following is a series of exercises using turtles. They are meant to be fun, but they have a point, too. While you are working on them, think about what the point is.

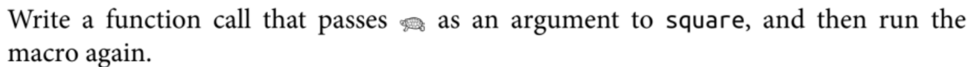
Copyrighted material

The following sections contain solutions to the exercises, so don't look until you have finished (or at least tried them).

Exercise 4-2

Write a function called `square` that takes a parameter named `t`, which is a turtle. It should use the turtle to draw a square.

Exercise 4-3

Write a function call that passes  as an argument to `square`, and then run the macro again.

Exercise 4-4

Add another parameter, named `len`, to `square`. Modify the body so the length of the sides is `len`, and then modify the function call to provide a second argument. Run the macro again. Test with a range of values for `len`.

Exercise 4-5

Make a copy of `square` and change the name to `polygon`. Add another parameter named `n` and modify the body so it draws an *n*-sided regular polygon.

The exterior angles of an *n*-sided regular polygon are $\frac{360}{n}$ degrees.

Exercise 4-6

Write a function called `circle` that takes a turtle, `t`, and radius, `r`, as parameters and that draws an approximate circle by calling `polygon` with an appropriate length and number of sides. Test your function with a range of values of `r`.

Figure out the circumference of the circle and make sure that `len * n == circumference`.

Copyrighted material

Exercise 4-7

Make a more general version of `circle` called `arc` that takes an additional parameter `angle`, which determines what fraction of a circle to draw. `angle` is in units of degrees, so when `angle = 360`, `arc` should draw a complete circle.

Encapsulation

The first exercise asks you to put your square-drawing code into a function definition and then call the function, passing the turtle as a parameter. Here is a solution:

```
function square(t)
    for i in 1:4
        forward(t, 100)
        turn(t, -90)
    end
end
🐢 = Turtle()
@svg begin
    square(🐢)
end
```

The innermost statements, `forward` and `turn`, are indented twice to show that they are inside the `for` loop, which is inside the function definition.

Inside the function, `t` refers to the same turtle 🐢, so `turn(t, -90)` has the same effect as `turn(🐢, -90)`. In that case, why not call the parameter 🐢? The idea is that `t` can be any turtle, not just 🐢 so you could create a second turtle and pass it as an argument to `square`:

```
🐫 = Turtle()
@svg begin
    square(🐫)
end
```

Wrapping a piece of code up in a function is called *encapsulation*. One of the benefits of encapsulation is that it attaches a name to the code, which serves as a kind of documentation. Another advantage is that if you reuse the code, it is more concise to call a function twice than to copy and paste the body!

Generalization

The next step is to add a `len` parameter to `square`. Here is a solution:

```
function square(t, len)
    for i in 1:4
        forward(t, len)
        turn(t, -90)
    end
```


Copyrighted material

```
end
🐢 = Turtle()
@svg begin
    square(🐢, 100)
end
```

Adding a parameter to a function is called *generalization* because it makes the function more general. In the previous version, the square is always the same size; in this version it can be any size.

The next step is also a generalization. Instead of drawing squares, `polygon` draws regular polygons with any number of sides. Here is a solution:

```
function polygon(t, n, len)
    angle = 360 / n
    for i in 1:n
        forward(t, len)
        turn(t, -angle)
    end
end
🐢 = Turtle()
@svg begin
    polygon(🐢, 7, 70)
end
```

This example draws a 7-sided polygon with side length 70.

Interface Design

The next step is to write `circle`, which takes a radius, `r`, as a parameter. Here is a simple solution that uses `polygon` to draw a 50-sided polygon:

```
function circle(t, r)
    circumference = 2 * π * r
    n = 50
    len = circumference / n
    polygon(t, n, len)
end
```

The first line computes the circumference of a circle with radius `r` using the formula `2πr`. `n` is the number of line segments in our approximation of a circle, so `len` is the length of each segment. Thus, `polygon` draws a 50-sided polygon that approximates a circle with radius `r`.

One limitation of this solution is that `n` is a constant, which means that for very big circles, the line segments are too long, and for small circles, we waste time drawing very small segments. One solution would be to generalize the function by taking `n` as a parameter. This would give the user (whoever calls `circle`) more control, but the interface would be less clean.

Copyrighted material

The *interface* of a function is a summary of how it is used: What are the parameters? What does the function do? And what is the return value? An interface is "clean" if it allows the caller to do what he wants without dealing with unnecessary details.

In this example, `r` belongs in the interface because it specifies the circle to be drawn. `n` is less appropriate because it pertains to the details of how the circle should be rendered.

Rather than cluttering up the interface, it is better to choose an appropriate value of `n` depending on `circumference`:

```
function circle(t, r)
    circumference = 2 * π * r
    n = trunc(circumference / 3) + 3
    len = circumference / n
    polygon(t, n, len)
end
```

Now the number of segments is an integer near `circumference/3`, so the length of each segment is approximately 3, which is small enough that the circles look good but big enough to be efficient, and acceptable for any size circle.

Adding `3` to `n` guarantees that the polygon has at least three sides.

Refactoring

When I wrote `circle`, I was able to reuse `polygon` because a many-sided polygon is a good approximation of a circle. But `arc` is not as cooperative; we can't use `polygon` or `circle` to draw an arc.

One alternative is to start with a copy of `polygon` and transform it into `arc`. The result might look like this:

```
function arc(t, r, angle)
    arc_len = 2 * π * r * angle / 360
    n = trunc(arc_len / 3) + 1
    step_len = arc_len / n
    step_angle = angle / n
    for i in 1:n
        forward(t, step_len)
        turn(t, -step_angle)
    end
end
```

The second half of this function looks like `polygon`, but we can't reuse `polygon` without changing the interface. We could generalize `polygon` to take an `angle` as a third argument, but then `polygon` would no longer be an appropriate name! Instead, let's call the more general function `polyline`:

Copyrighted material

```
function polyline(t, n, len, angle)
    for i in 1:n
        forward(t, len)
        turn(t, -angle)
    end
end
```

Now we can rewrite `polygon` and `arc` to use `polyline`:

```
function polygon(t, n, len)
    angle = 360 / n
    polyline(t, n, len, angle)
end

function arc(t, r, angle)
    arc_len = 2 * π * r * angle / 360
    n = trunc(arc_len / 3) + 1
    step_len = arc_len / n
    step_angle = angle / n
    polyline(t, n, step_len, step_angle)
end
```

Finally, we can rewrite `circle` to use `arc`:

```
function circle(t, r)
    arc(t, r, 360)
end
```

This process—rearranging a program to improve interfaces and facilitate code reuse—is called *refactoring*. In this case, we noticed that there was similar code in `arc` and `polygon`, so we "factored it out" into `polyline`.

If we had planned ahead, we might have written `polyline` first and avoided refactoring, but often you don't know enough at the beginning of a project to design all the interfaces. Once you start coding, you understand the problem better. Sometimes refactoring is a sign that you have learned something.

A Development Plan

A *development plan* is a process for writing programs. The process we used in this case study is "encapsulation and generalization." The steps of this process are:

1. Start by writing a small program with no function definitions.
2. Once you get the program working, identify a coherent piece of it, encapsulate the piece in a function, and give it a name.
3. Generalize the function by adding appropriate parameters.
4. Repeat steps 1–3 until you have a set of working functions. Copy and paste working code to avoid retyping (and redebugging).

Copyrighted material

5. Look for opportunities to improve the program by refactoring. For example, if you have similar code in several places, consider factoring it into an appropriately general function.

This process has some drawbacks—we will see alternatives later—but it can be useful if you don't know ahead of time how to divide the program into functions. This approach lets you design as you go along.

Docstring

A *docstring* is a string before a function that explains the interface ("doc" is short for "documentation"). Here is an example:

```
"""
polyline(t, n, len, angle)

Draws n line segments with the given length and
angle (in degrees) between them.  t is a turtle.
"""
function polyline(t, n, len, angle)
    for i in 1:n
        forward(t, len)
        turn(t, -angle)
    end
end
```

Documentation can be accessed in the REPL or in a notebook by typing ? followed by the name of a function or macro, and pressing Enter:

```
help?> polyline
search:

  polyline(t, n, len, angle)

  Draws n line segments with the given length and angle (in degrees) between
them. t is a turtle.
```

Docstrings are often triple-quoted strings, also known as "multiline" strings because the triple quotes allow the string to span more than one line.

A docstring contains the essential information someone would need to use the function. It explains concisely what the function does (without getting into the details of how it does it). It explains what effect each parameter has on the behavior of the function and what type each parameter should be (if it is not obvious).

Copyrighted material

Writing this kind of documentation is an important part of interface design. A well-designed interface should be simple to explain; if you have a hard time explaining one of your functions, maybe the interface could be improved.

Debugging

An interface is like a contract between a function and a caller. The caller agrees to provide certain parameters and the function agrees to do certain work.

For example, `polyline` requires four arguments: `t` has to be a turtle; `n` has to be an integer; `len` should be a positive number; and `angle` has to be a number, which is understood to be in degrees.

These requirements are called *preconditions* because they are supposed to be true before the function starts executing. Conversely, conditions at the end of the function are *postconditions*. Postconditions include the intended effect of the function (like drawing line segments) and any side effects (like moving the turtle or making other changes).

Preconditions are the responsibility of the caller. If the caller violates a (properly documented!) precondition and the function doesn't work correctly, the bug is in the caller, not the function.

If the preconditions are satisfied and the postconditions are not, the bug is in the function. If your pre- and postconditions are clear, they can help with debugging.

Glossary

module
: A file that contains a collection of related functions and other definitions.

package
: An external library with additional functionality.

`using` *statement*
: A statement that reads a module file and creates a module object.

loop
: A part of a program that can run repeatedly.

encapsulation
: The process of transforming a sequence of statements into a function definition.

Copyrighted material

generalization
: The process of replacing something unnecessarily specific (like a number) with something appropriately general (like a variable or parameter).

interface
: A description of how to use a function, including the name and descriptions of the arguments and return value.

refactoring
: The process of modifying a working program to improve function interfaces and other qualities of the code.

development plan
: A process for writing programs.

docstring
: A string that appears at the top of a function definition to document the function's interface.

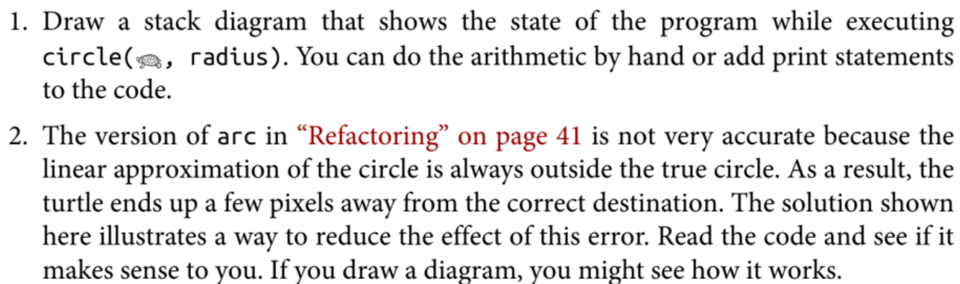
precondition
: A requirement that should be satisfied by the caller before a function starts.

postcondition
: A requirement that should be satisfied by the function before it ends.

Exercises

Exercise 4-8

Enter the code in this chapter in a notebook.

1. Draw a stack diagram that shows the state of the program while executing circle(, radius). You can do the arithmetic by hand or add print statements to the code.
2. The version of `arc` in "Refactoring" on page 41 is not very accurate because the linear approximation of the circle is always outside the true circle. As a result, the turtle ends up a few pixels away from the correct destination. The solution shown here illustrates a way to reduce the effect of this error. Read the code and see if it makes sense to you. If you draw a diagram, you might see how it works.

```
"""
arc(t, r, angle)

Draws an arc with the given radius and angle:

    t: turtle
    r: radius
```


Copyrighted material

```
    angle: angle subtended by the arc, in degrees
"""
function arc(t, r, angle)
    arc_len = 2 * π * r * abs(angle) / 360
    n = trunc(arc_len / 4) + 3
    step_len = arc_len / n
    step_angle = angle / n

    # making a slight left turn before starting reduces
    # the error caused by the linear approximation of the arc
    turn(t, -step_angle/2)
    polyline(t, n, step_len, step_angle)
    turn(t, step_angle/2)
end
```

Exercise 4-9

Write an appropriately general set of functions that can draw flowers as in Figure 4-2.

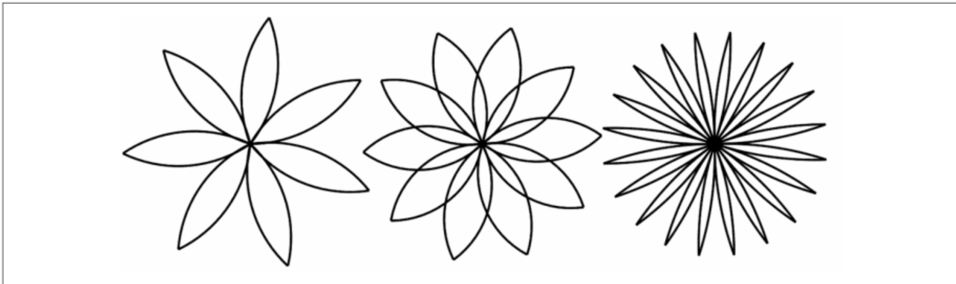


Figure 4-2. Turtle flowers

Exercise 4-10

Write an appropriately general set of functions that can draw shapes as in Figure 4-3.

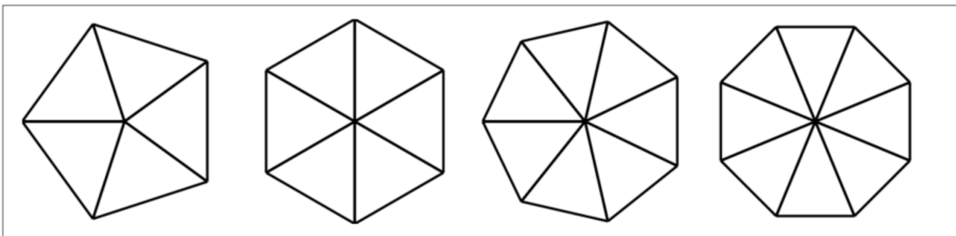


Figure 4-3. Turtle pies

Copyrighted material

Exercise 4-11

The letters of the alphabet can be constructed from a moderate number of basic elements, like vertical and horizontal lines and a few curves. Design an alphabet that can be drawn with a minimal number of basic elements and then write functions that draw the letters.

You should write one function for each letter, with names `draw_a`, `draw_b`, etc., and put your functions in a file named *letters.jl*.

Exercise 4-12

Read about spirals at *https://en.wikipedia.org/wiki/Spiral*; then write a program that draws an Archimedean spiral as in Figure 4-4.

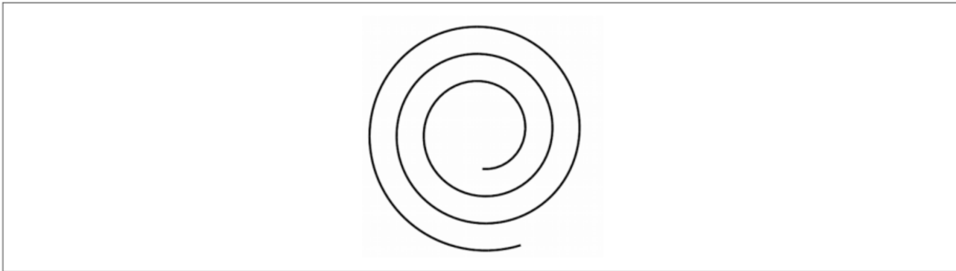


Figure 4-4. Archimedean spiral

Copyrighted material