

Computational Thinking for Actuaries and Financial Professionals

With Applications in Julia

Alec Loudenback and Yun-Tien Lee

2024-12-16

Table of contents

Preface	1
1. Draft of Cover	3
I. Introduction	5
The approach	8
What you will learn	8
Prerequisites	9
The Contents of This Book	10
Notes on formatting	11
2. Why Program?	13
2.1. In this Chapter	13
2.2. The Long View	13
2.3. What's coding got to do with this?	15
2.4. The 10x Actuary	16
2.5. Risk Governance	17
2.6. Managing and Leading the Transformation	18
2.7. Outlook	18
3. Why use Julia?	19
3.1. Expressiveness and Syntax	20
3.1.1. Example: Retention Analysis	20
3.1.2. Example: Random Sampling	22
3.2. The Speed	23
3.3. More of Julia's benefits	24
3.4. The Tradeoff	25
3.5. Package Ecosystem	26
3.6. Conclusion	27

II. Conceptual Foundations	29
4. Elements of Financial Modeling	31
4.1. In this Chapter	31
4.2. What is a model?	31
4.3. What is a <i>Financial</i> Model?	32
4.4. The Process of Building a Financial Model	33
4.5. Predictive versus Explanatory Models	34
4.5.1. A Historical Example	34
4.5.2. Examples in the Financial Context	37
4.6. What makes a good model?	38
4.6.1. Achieving original purpose	38
4.6.2. Usability	38
4.6.3. Performance	39
4.7. What makes a good modeler?	40
4.7.1. Domain Expertise	40
4.7.2. Model Theory	40
4.7.3. Curiosity	40
4.7.4. Rigor	40
4.7.5. Toolset	40
5. Elements of Programming	41
5.1. In this section	41
5.2. Computer Science, Programming, and Coding	41
5.3. Assignment and Variables	43
5.4. Data Types	44
5.4.1. Numbers	44
5.4.2. Type Hierarchy	48
5.4.3. Arrays	48
5.4.4. Characters, Strings, and Symbols	53
5.4.5. Tuples	55
5.4.6. Parametric Types	56
5.4.7. Types for things not there	57
5.4.8. Union Types	58
5.4.9. Creating User Defined Types	58
5.4.10. Mutable structs	62
5.5. Expressions and Control Flow	62
5.5.1. Compound Expression	63
5.5.2. Conditional Expressions	64
5.5.3. Assignment and Variables	66
5.5.4. Loops	67

Table of contents

5.5.5. Performance of loops	69
5.6. Functions	69
5.6.1. Special Operators	69
5.6.2. General Functions	71
5.6.3. Anonymous Functions	73
5.6.4. Passing by Sharing	73
5.6.5. Broadcasting	74
5.6.6. First Class Nature	78
5.7. Scope	79
5.7.1. Modules and Namespaces	81
6. Patterns of Abstraction	83
6.1. In this section	83
6.2. Introduction	83
6.3. Interfaces	84
6.3.1. Conceptual Strategies	85
6.4. Programming Interfaces and Patterns	86
6.4.1. (Multiple) Dispatch	89
6.4.2. Programming Paradigms	90
6.5. Misc Techniques	91
6.5.1. Recursion	91
6.5.2. Iterators	91
7. Elements of Computer Science	93
7.1. In this section	93
7.2. Computer Science for Financial Professionals	93
7.3. Algorithms	95
7.3.1. Computational Complexity	95
7.4. Data Structures	95
7.5. Information Theory	95
7.5.1. Signal vs Noise	95
7.6. Formal Verification	95
7.7. The Discipline of Software Engineering	95
7.7.1. Patterns	95
8. Hardware and It's Implications	97
8.1. In this section	97
9. Applying Software Engineering Principles	99
9.1. In this section	99

Table of contents

III. Computational Thinking in an Actuarial and Financial Context	101
10. Modeling	103
10.1. In This Chapter	103
10.2. Parsimony	103
11. Optimization	105
11.1. In This Chapter	105
12. Sensitivity Analysis	107
12.1. In This Chapter	107
13. Stochastic Modeling	109
14. Visualizations	111
14.1. In This Chapter	111
15. Matrices and Their Uses	113
15.1. In This Chapter	113
16. Learning from Data	115
16.1. In this chapter	115
IV. Applications in Practice	117
17. Stochastic Mortality Projections	119
17.1. In This Chapter	119
17.2. Setup	119
17.3. The Data	120
17.4. Running the projection	123
17.4.1. Stochastic Projection	124
17.5. Benchmarking	125
17.6. Further Optimization	126
18. Scenario Generation	127
18.1. In This Chapter	127
19. Similarity Analysis	129
19.1. In This Chapter	129

20. Portfolio Optimization	131
20.1. In This Chapter	131
21. Other Useful Techniques	133
21.1. In this chapter	133
21.2. Taking things to the Extreme	133
21.3. Range Bounding	133
 V. Appendices	 135
22. Set up Julia and the Computing Environment	137
22.1. Installation	137
22.2. Package Management	137
22.3. Editors	138
22.3.1. Visual Studio Code	138
22.3.2. Notebooks	139
 23. The Julia Ecosystem Today	 141
23.0.1. Data	142
23.0.2. Plotting	142
23.0.3. Statistics	142
23.0.4. Machine Learning	143
23.0.5. Differentiable Programming	143
23.0.6. Utilities	144
23.0.7. Other packages	145
23.0.8. Actuarial packages	145
 References	 147

Preface

This book is intended to enable practitioners and advanced students of financial disciplines to utilize the tools, language, and ideas of computational sciences in their own discipline.

1. Draft of Cover

1. Draft of Cover

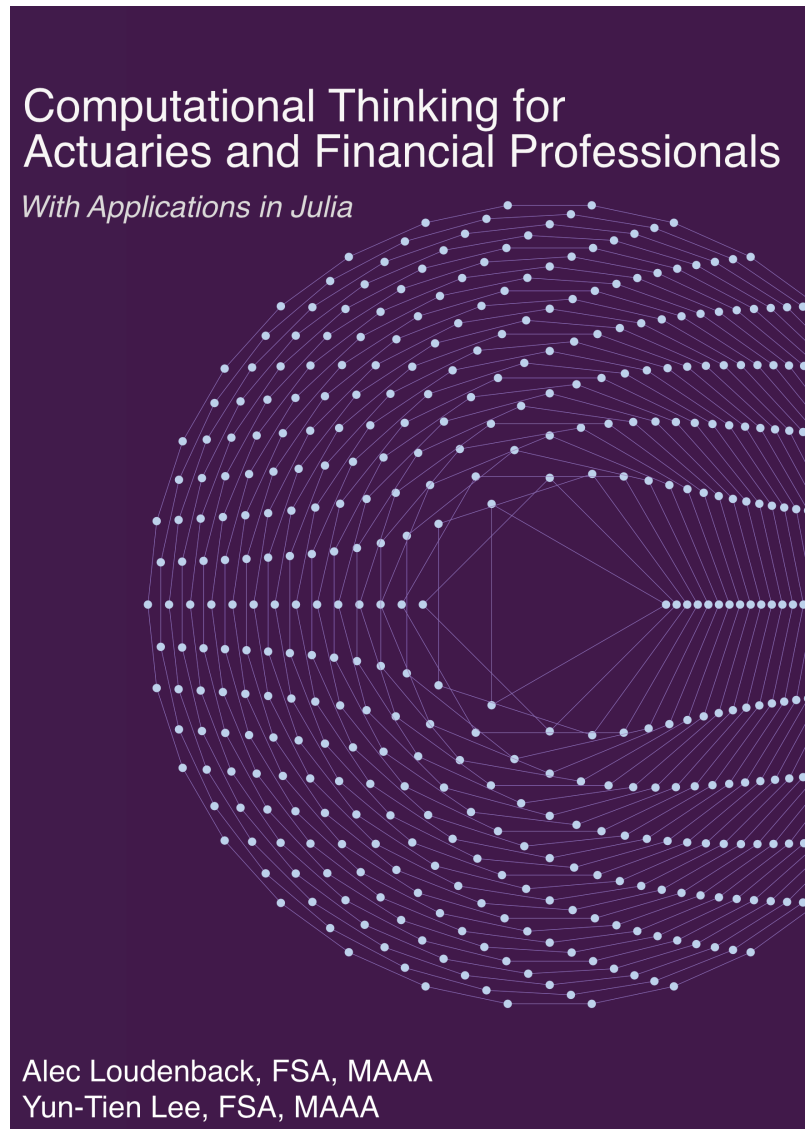


Figure 1.1.: Draft of Cover

Part I.

Introduction

“I think one of the things that really separates us from the high primates is that we’re tool builders. I read a study that measured the efficiency of locomotion for various species on the planet. The condor used the least energy to move a kilometer. And, humans came in with a rather unimpressive showing, about a third of the way down the list. It was not too proud a showing for the crown of creation. So, that didn’t look so good. But, then somebody at Scientific American had the insight to test the efficiency of locomotion for a man on a bicycle. And, a man on a bicycle, a human on a bicycle, blew the condor away, completely off the top of the charts.

And that’s what a computer is to me. What a computer is to me is it’s the most remarkable tool that we’ve ever come up with, and it’s the equivalent of a bicycle for our minds.” - Steve Jobs (1990)

The world of financial modeling is incredibly complex and variegated. It, along with many of the sciences, is a place where practical goals harness computational tools to arrive at answers that (we hope) are meaningful in a way that tells us more about the world we live in. What this usually means specifically is that practitioners utilize computers to do the heavy work of processing data or running simulations which reveal the something about the complex systems we seek to represent. In this way, then, financial modelers must also be a craftsman who seeks not only to design new products, but must also think carefully about the tools and the process used therein.

This book seeks to aid the practitioner in developing that workmanship: we will develop new ways to look at the *process*, think about how to most clearly represent ideas, dive into details about computer hardware and bring it back up to the most abstract levels, and develop a vocabulary to more clearly express and communicate these concepts. The book contains a large number of practical examples to demonstrate that the end result is better for the journey we will take.

This book looks at programming for the applied financial professional and we will start by answering a very basic question:

“why is this relevant for financial modeling?”. The answer is simple: financial modeling is complex, data intensive, and often very abstract. Programming is the best tool humans have so far developed for rigorously transforming ideas and data into results. A builder may be the most skilled person in the world with a hammer but another with some basic training in a richer set of tools will build a better house. This book will enhance your toolkit with experience with multiple tools: a specific programming language, yes, but much more than that: a language to talk about solving problems, a deeper understanding of specific problem solving techniques, how to make decisions about what the architecture of a solution looks like, and practical advice from experienced practitioners.

The approach

The authors of the book are practicing actuaries, but we intend for the content to be applicable to nearly all practitioners in the financial industry. The discussion and examples may have an orientation towards insurance topics, but the concepts and patterns are applicable to a wide variety of related disciplines.

We will pull from examples on both sides of the balance sheet: the left (assets) and right (liabilities). We may also take the liberty to, at times, abuse traditional accounting notions: a liability is just an asset with the obligor and obligee switched. When the accounting conventions are important (such as modeling a total balance sheet) we will be mindful in explaining the accounting perspective. In practice, this means that we’ll take examples that use examples of assets (fixed income, equity, derivatives) or liabilities (life insurance, annuities, long term care) and show that similar modeling techniques can be used for both.

What you will learn

It is our hope that with the help of this book, you will find it more efficient to discuss aspects of modeling with colleagues,

borrow problem solving language from computer science, spot recurring structural patterns in problems that arise, and understand how best to make use of the “bicycle for your mind” in the context of financial modeling.

It is the experience of the authors that many professionals that do complex modeling as a part of their work have gotten to be very proficient *in spite of* not having substantive formal training on problem solving, algorithms, or model architecture. This book serves to fill that gap and provide the “missing semester” (or “years of practical learning”!). After reading this book, we hope that you will *appreciate* the attributes of Microsoft Excel that made it so ubiquitous, but that you *prefer* to use a programming language for the ability to more naturally express the relevant abstractions which make your models simpler, faster, or more usable by others.

Prerequisites

Basic experience with financial modeling is not strictly required, but it will benefit the reader to be familiar so that the examples will not be attempting to teach both financial maths and computer science simultaneously.

Advanced financial maths (e.g. stochastic calculus) is *not* required. Indeed, this book is not oriented to the advanced technicalities of Wall Street “quants” and is instead directed at the multitudes of financial practitioners focused on producing results that are not measured in the microseconds of high-frequency trading.

Prior programming experience is *not* required either: Chapter 5 introduces the basic syntax and concepts while Chapter 22 covers setting up your environment to follow along. For readers with background in programming, we recommend skimming Chapter 5 and reading in full the sections which have a symbol in the margin, which is our way of highlighting Julia-specific content to be aware of.

i TODO

Create a venn diagram showing financial modeling at the intersection of statistics, financial math, computer science.

The Contents of This Book

Part 1 of the book addresses the theoretical and technical foundations of programming, as well as the conceptual basis for financial modelling. It familiarizes the readers with key functional programming principles, alongside introducing important aspects of software engineering relevant to financial modelling.

Parts 2 and 3 bridge the gap between theory and practical applications, underlining the features of Julia that make it a robust tool for real-world financial and actuarial contexts. Through a careful exploration of topics like sensitivity analysis, optimization, stochastic modeling, visualization, and practical financial applications, the book demonstrates how Julia's high-level, high-performance programming capabilities can enhance accuracy and efficiency in financial modelling. As an up-and-coming language loved for its speed and simplicity, Julia is ripe for wide adoption in the financial sector. The time for this book is ripe, as it will satiate the growing demand for professionals who want to blend programming skills with financial modelling acumen.

While we have chosen to use Julia for the examples in this book, the vast majority of the concepts presented are not Julia-specific. We will attempt to motivate why Julia works so well as a language for financial modeling but like mathematics and applied mathematics, the concepts are portable even if the numbers (language) changes. Readers are encouraged to follow along the examples on their own computer (see instructions for Julia in Chapter 22) and the entire book is available on GitHub at [#TODO: determine book URL].

Notes on formatting

When a concept is defined for the first time, the term will be **bold**. Code, or references to pieces of code will be formatted in inline code style like `1+1` or in separate code blocks:

```
"This is a code block that doesn't show any results"
```

```
"This is a code block that does show output"
```

```
"This is a code block that does show output"
```

There will be various callout blocks which indicate tips or warnings. These should be self-evident but we wanted to point to a particular callout which is intended to convey advice that stems from practical modeling experience of the authors:

Financial Modeling Pro-tip

This box indicates a side note that's particularly applicable to improving your financial modeling.

2. Why Program?

“Humans are allergic to change. They love to say, ‘We’ve always done it this way.’ I try to fight that. That’s why I have a clock on my wall that runs counterclockwise.” - Grace Hopper (1987)

[Drafting Note: This chapter is pulled from the article published in 2020 and needs to be adapted for the book’s audience. Also to include: why not low-code solutions?]

2.1. In this Chapter

We motivate why a financial professional should adopt programming skills which will improve their own capabilities and enjoyment of the discipline, whilst allowing themselves to better themselves and the industry we work in.

2.2. The Long View

It might be odd to say that technology and its use in insurance is on a one-hundred-year cycle, but that seems to be the case.

130 years ago, actuaries crowded into a room at a meeting of the Actuarial Society of America to watch a demonstration that would revolutionize the industry: Herman Hollerith’s tabulating punch card machine¹.

For the next half-century, the increasing automation — from tabulating machines to early-adopting mainframes and computers — was a critical competitive differentiator. Companies like Prudential, MetLife, and others partnered with technology companies in the development of hardware and software².

¹ Co-evolution of Information Processing Technology and Use: Interaction Between the Life Insurance and Tabulating Industries

² From Tabulators to Early Computers in the U.S. Life Insurance Industry

2. Why Program?

The dramatic embodiment of this information-driven cycle was portrayed in the infamous Billion Dollar Bubble movie, which showcased the power and abstraction of the computer to commit millions of dollars of fraud by creating and maintaining fake insurance policies.

The movie also starts to hint at the oscillation away from the technological-competitive focus of insurance companies. I argue that the focus on technology was lost over the last 50 years with the rise of Wall Street finance, investment-oriented life insurance, industry consolidation, and the explosion of financial structuring like derivatives, reserve financing, or other advanced forms of reinsurance.

Value-add came from the C-Suite, not from the underlying business processes, operations, and analysis. The result is, e.g., ever-more complicated reinsurance treaties layered into mainframes and admin systems older than most of the actuaries interfacing with them.

The pace of *strategic value-add* isn't slowing, though it must stretch further (in complexity and risk) to find comparable opportunities as the past. Having more agile, data-oriented operations enables companies to be able to react to and implement those opportunities. *Technological value-add* can improve a company's bottom line through lower expenses and higher top-line growth, but often with a more favorable risk profile than some of the "strategic" opportunities.

Today, there is a trend reverting back to technological value-creation and is evident across many traditional sectors. Tesla claims that it's a technology company; Amazon is the #1 product retailer because of its vehement focus on internal information sharing³; Airlines are so dependent on their systems that the skies become quieter on the rare occasion that their computers give way.

Why is it, that companies that are so involved in *things* (cars, shopping) and *physical services* (flights) are so much more focused on improving their technological operations than insurance companies *whose very focus is 'information-based'*? **The market has rewarded those who have prioritized their internal technological solutions.**

³ Have you had your Bezos moment?
What you can learn from Amazon

2.3. What's coding got to do with this?

Commoditized investing services and low yield environments have reduced insurance companies' comparative advantage to "manage money". Yield compression and the explosion of consumer-oriented investment services means a more competitive focus on the ability to manage the entire policy lifecycle efficiently (digitally), perform more real-time analysis of experience and risk management, and handle the growing product and regulatory complexity.

These are problems that have technological solutions and are waiting for insurance company adoption.

Companies that treat data like coordinates on a grid (spreadsheets) *will get left behind*. Two main hurdles have prevented technology companies from breaking into insurance:

1. High regulatory barriers to entry, and
2. Difficulty in selling complex insurance products without traditional distribution.

Once those two walls are breached, traditional insurance companies without a strong technology core will struggle to keep up. The key to thriving is not just adding "developers" to an organization; it's going to be **getting domain experts like actuaries to be an integral part of the technology transformation.**

2.3. What's coding got to do with this?

Everything. Programming is the optimal way to interact between the computer and actuary — and importantly between computer and computer. Programming is the actionable expression of ideas, math, analysis, and information. Think of programming as the 21st-century leap in the actuary's toolkit, just as spreadsheets were in the preceding 40 years. Versus a spreadsheet-oriented workflow:

- More natural automation of, and between processes
- Better reproducibility
- Scaling to fit any size dataset and workload
- Statistics and machine learning capabilities

2. Why Program?

- Advanced visualizations to garner new views into your data

This list isn't comprehensive and some benefits are subtle — when you are code-oriented instead of spreadsheet-oriented, you tend to want to structure your data in a portable and shareable way. For example, relying more on data warehouses instead of email attachments. This, in turn, enables data discovery and insights that otherwise wouldn't be there. Investing in a code-oriented workflow is playing the long-game.

The actuary of the future needs to have coding as one of their core skills. Already today, the advances of business processes, insurance products, and financial ingenuity are written with lines of code — *not* spreadsheets. Not being able to code *necessarily* means that you are *following* what others are doing today.

It's commonly accepted now that to gather insights from your data, you need to know how to code. Similar to your data, your business architecture, modeling needs, and product peculiarities are often better suited to customized solutions. Why stop at data science when learning how to solve problems with a computer?

2.4. The 10x Actuary

As we swing back to a technological focus, we do not leave the finance-driven complexity behind. The increasingly complex business needs will highlight a large productivity difference between an actuary who can code and one who can't — simply because the former can react, create, synthesize, and model faster than the latter. From the efficiency of transforming administration extracts, summarizing and aggregating valuation output, to analyzing claims data in ways that spreadsheets simply can't handle, you can become a “**10x Actuary**”⁴.

⁴ The 10x [Rockstar] developer is NOT a myth

Flipping switches in a graphical user interface versus being able to *build models* is the difference between having a surface-level familiarity and having full command over the analysis and the

2.5. Risk Governance

concepts involved — with the flexibility to do what your software can't.

Your current software might be able to perform the first layer of analysis but be at a loss when you want to visualize, perform sensitivity analysis, statistics, stochastic analysis, or process automation. Things that, when done programmatically, are often just a few lines of additional code.

Do I advocate dropping the license for your software vendor? No, not yet anyway. But the ability to supplement and break out of the modeling box has been an increasingly important part of most actuaries' work.

Additionally, code-based solutions can leverage the entire-technology sector's progress to solve problems that are *hard* otherwise: scalability, data workflows, integration across functional areas, version control and versioning, model change governance, reproducibility, and more.

30-40 years ago, there were no vendor-supplied modeling solutions and so you had no choice but to build models internally. This shifted with the advent of vendor-supplied modeling solutions. Today, it's never been better for companies to leverage open source to support their custom modeling, risk analysis/monitoring, and reporting workflows.

2.5. Risk Governance

Code-based workflows are highly conducive to risk governance frameworks as well. If a modern software project has all of the following benefits, then why not a modern insurance product and associated processes?

- Access control and approval processes
- Version control, version management, and reproducibility
- Continuous testing and validation of results
- Open and transparent design
- Minimization of manual overrides, intervention, and opportunity for user error
- Automated trending analysis, system metrics, and summary statistics

2. Why Program?

- Continuously updated, integrated, and self-generating documentation
- Integration with other business processes through a formal boundary (e.g. via an API)
- Tools to manage collaboration in parallel and in sequence

2.6. Managing and Leading the Transformation

The ability to understand the concepts, capabilities, challenges, and lingo is not a dichotomy, it's a spectrum. Most actuaries, even at fairly high levels, are still often involved in analytical work. Still above that, it's difficult to lead something that you don't understand.

Conversely, the skill and practice of coding enhances managerial capabilities. When you are really skilled at pulling apart a problem or process into its constituent parts and designing optimal solutions; that's a core attribute of leadership: having the vision of where the organization *should be* instead of thinking about where it is now.

Nor is the skillset described here limiting in any other aspect of career development any more than mathematical ability, project collaboration, or financial acumen — just to name a few.

2.7. Outlook

It will increasingly be essential for companies to modernize to remain competitive. That modernization isn't built with big black-box software packages; it will be with domain experts who can translate the expertise into new forms of analysis - doing it faster and more robustly than the competition.

SpaceX doesn't just hire rocket scientists - they hire rocket scientists who code.

Be an actuary who codes.

3. Why use Julia?

[Drafting Note: This chapter is pulled from the article published in 2021 and needs to be adapted for the book’s audience.
]

Julia is relatively new⁵, and *it shows*. It is evident in its pragmatic, productivity-focused design choices, pleasant syntax, rich ecosystem, thriving communities, and its ability to be both very general purpose and power cutting edge computing.

With Julia: math-heavy code looks like math; it’s easy to pick up, and quick-to-prototype. Packages are well-integrated, with excellent visualization libraries and pragmatic design choices.

Julia’s popularity continues to grow across many fields and there’s a growing body of online references and tutorials, videos, and print media to learn from.

Large financial services organizations have already started realizing gains: BlackRock’s Aladdin portfolio modeling, the Federal Reserve’s economic simulations, and Aviva’s Solvency II-compliant modeling⁶. The last of these has a great talk on YouTube by Aviva’s Tim Thornham, which showcases an on-the-ground view of what difference the right choice of technology and programming language can make. Moving from their vendor-supplied modeling solution was **1000x faster, took 1/10 the amount of code, and was implemented 10x faster**.

The language is not just great for data science — but also modeling, ETL, visualizations, package control/version management, machine learning, string manipulation, web-backends, and many other use cases. Julia is well suited for financial modeling work: easy to read and write and very performant.

⁵ Python first appeared in 1990. R is an implementation of S, which was created in 1976, though depending on when you want to place the start of an independent R project varies (1993, 1995, and 2000 are alternate dates). The history of these languages is long and substantial changes have occurred since these dates.

⁶ Aviva Case Study

3. Why use Julia?

3.1. Expressiveness and Syntax

Expressiveness is the *manner in which* and *scope of* ideas and concepts that can be represented in a programming language. **Syntax** refers to how the code *looks* on the screen and its readability.

In a language with high expressiveness and pleasant syntax, you:

- Go from idea in your head to final product faster.
- Encapsulate concepts naturally and write concise functions.
- Compose functions and data naturally.
- Focus on the end-goal instead of fighting the tools.

Expressiveness can be hard to explain, but perhaps two short examples will illustrate.

3.1.1. Example: Retention Analysis

This is a really simple example relating Cessions, Policys, and Lives to do simple retention analysis.

First, let's define our data:

```
# Define our data structures
struct Life
    policies
end

struct Policy
    face
    cessions
end

struct Cession
    ceded
end
```

3.1. Expressiveness and Syntax

Now to calculate amounts retained. First, let's say what retention means for a `Policy`:

```
# define retention
function retained(pol::Policy)
    pol.face - sum(cession.ceded for cession in pol.cessions)
end
```

And then what retention means for a `Life`:

```
function retained(l::Life)
    sum(retained(policy) for policy in life.policies)
end
```

It's almost exactly how you'd specify it English. No joins, no boilerplate, no fiddling with complicated syntax. You can express ideas and concepts the way that you think of them, not, for example, as a series of dataframe joins or as row/column coordinates on a spreadsheet.

We defined `retained` and adapted it to mean related, but different things depending on the specific context. That is, we didn't have to define `retained_life(...)` and `retained_pol(...)` because Julia can be *dispatch* based on what you give it. This is, as some would call it, unreasonably effective.

Let's use the above code in practice then.

The `julia>` syntax indicates that we've moved into Julia's interactive mode (REPL mode):

```
# create two policies with two and one cessions respectively
julia> pol_1 = Policy( 1000, [ Cession(100), Cession(500)] )
julia> pol_2 = Policy( 2500, [ Cession(1000) ] )

# create a life, which has the two policies
julia> life = Life([pol_1, pol_2])

julia> retained(pol_1)
400
```

3. Why use Julia?

```
julia> retained(life)
1900
```

And for the last trick, something called “broadcasting”, which automatically vectorizes any function you write, no need to write loops or create if statements to handle a single vs repeated case:

```
julia> retained.(life.policies) # retained amount for each policy
[400 , 1500]
```

3.1.2. Example: Random Sampling

As another motivating example showcasing multiple dispatch, here’s random sampling in Julia, R, and Python.

We generate 100:

- Uniform random numbers
- Standard normal random numbers
- Bernoulli random number
- Random samples with a given set

Table 3.1.: A comparison of random outcome generation in Julia, R, and Python.

Julia	R	Python
<pre>using Distributions rand(100) rand(Normal(), 100) rand(Bernoulli(0.5), 100) rand(["Preferred", "Standard"], 100)</pre>	<pre>runif(100) rnorm(100) rbern(100, 0.5) sample(c("Preferred", "Standard"), 100, replace=TRUE)</pre>	<pre>import scipy.stats as sps import numpy as np sps.uniform.rvs(size=100) sps.norm.rvs(size=100) sps.bernoulli.rvs(p=0.5, size=100) np.random.choice(["Preferred", "Standard"], size=100)</pre>

3.2. *The Speed*

By understanding the different types of things passed to `rand()`, it maintains the same syntax across a variety of different scenarios. We could define `rand(Cession)` and have it generate a random `Cession` like we used above.

3.2. The Speed

As the journal *Nature* said, “Come for the Syntax, Stay for the Speed”.

Recall the Solvency II compliance which ran 1000x faster than the prior vendor solution mentioned earlier: what does it mean to be 1000x faster at something? It’s the difference between something taking 10 seconds instead of 3 hours — or 1 hour instead of 42 days.

**What analysis would you like to do if it took less time?
A stochastic analysis of life-level claims? Machine
learning with your experience data? Daily valuation
instead of quarterly?**

Speaking from experience, speed is not just great for production time improvements. During development, it’s really helpful too. When building something, I can see that I messed something up in a couple of seconds instead of 20 minutes. The build, test, fix, iteration cycle goes faster this way.

Admittedly, most workflows don’t see a 1000x speedup, but 10x to 1000x is a very common range of speed differences vs R or Python or MATLAB.

Sometimes you will see less of a speed difference; R and Python have already circumvented this and written much core code in low-level languages. This is an example of what’s called the “two-language” problem where the language productive to write in isn’t very fast. For example, more than half of R packages use C/C++/Fortran and core packages in Python like Pandas, PyTorch, NumPy, SciPy, etc. do this too.

Within the bounds of the optimized R/Python libraries, you can leverage this work. Extending it can be difficult: what if

3. Why use Julia?

you have a custom retention management system running on millions of policies every night?

Julia packages you are using are almost always written in pure Julia: you can see what's going on, learn from them, or even contribute a package of your own!

3.3. More of Julia's benefits

Julia is easy to write, learn, and be productive in:

- It's free and open-source
 - Very permissive licenses, facilitating the use in commercial environments (same with most packages)
- Large and growing set of available packages
- Write how you like because it's multi-paradigm: vectorizable (R), object-oriented (Python), functional (Lisp), or detail-oriented (C)
- Built-in package manager, documentation, and testing-library
- Jupyter Notebook support (it's in the name! **Julia-Python-R**)
- Many small, nice things that add up:
 - Unicode characters like α or β
 - Nice display of arrays
 - Simple anonymous function syntax
 - Wide range of text editor support
 - First-class support for missing values across the entire language
 - Literate programming support (like R-Markdown)
- Built-in `Dates` package that makes working with dates pleasant
- Ability to directly call and use R and Python code/packages with the `PyCall` and `RCall` packages
- Error messages are helpful and tell you *what line* the error came from, not just the type of error
- Debugger functionality so you can step through your code line by line

3.4. The Tradeoff

For power-users, advanced features are easily accessible: parallel programming, broadcasting, types, interfaces, metaprogramming, and more.

These are some of the things that make Julia one of the world’s most loved languages on the StackOverflow Developer Survey.

For those who are enterprise-minded: in addition to the liberal licensing mentioned above, there are professional products from organizations like Julia Computing that provide hands-on support, training, IT governance solutions, behind-the-firewall package management, and deployment/scaling assistance.

3.4. The Tradeoff

Julia is fast because it’s compiled, unlike R and Python where (loosely speaking) the computer just reads one line at a time. Julia compiles code “just-in-time”: right before you use a function for the first time, it will take a moment to pre-process the code section for the machine. Subsequent calls don’t need to be re-compiled and are very fast.

A hypothetical example: running 10,000 stochastic projections where Julia needs to precompile but then runs each 10x faster:

- Julia runs in 2 minutes: the first projection takes 1 second to compile and run, but each 9,999 remaining projections only take 10ms.
- Python runs in 17 minutes: 100ms of a second for each computation.

Typically, the compilation is very fast (milliseconds), but in the most complicated cases it can be several seconds. One of these is the “time-to-first-plot” issue because it’s the most common one users encounter: super-flexible plotting libraries have a lot of things to pre-compile. So in the case of plotting, it can take several seconds to display the first plot after starting Julia, but then it’s remarkably quick and easy to create an animation of your model results. The time-to-first plot is a solvable problem

3. *Why use Julia?*

that's receiving a lot of attention from the core developers and will get better with future Julia releases.

For users working with a lot of data or complex calculations (like actuaries!), the runtime speedup is worth a few seconds at the start.

3.5. Package Ecosystem

Using packages as dependencies in your project is assisted by Julia's bundled package manager.

For each project, you can track the exact set of dependencies and replicate the code/process on another machine or another time. In R or Python, dependency management is notoriously difficult and it's one of the things that the Julia creators wanted to fix from the start.

Packages can be one of the thousands of publicly available, or private packages hosted internally behind a firewall.

Another powerful aspect of the package ecosystem is that due to the language design, packages can be combined/extended in ways that are difficult for other common languages. This means that Julia packages often interop without any additional coordination.

For example, packages that operate on data tables work without issue together in Julia. In R/Python, many features tend to come bundled in a giant singular package like Python's Pandas which has Input/Output, Date manipulation, plotting, resampling, and more. There's a new Consortium for Python Data API Standards which seeks to harmonize the different packages in Python to make them more consistent (R's Tidyverse plays a similar role in coordinating their subset of the package ecosystem).

In Julia, packages tend to be more plug-and-play. For example, every time you want to load a CSV you might not want to transform the data into a dataframe (maybe you want a matrix or a plot instead). To load data into a dataframe, in Julia the practice is to use both the CSV and DataFrames packages, which

help separate concerns. Some users may prefer the Python/R approach of less modular but more all-inclusive packages.

3.6. Conclusion

Looking at other great tools like R and Python, it can be difficult to summarize a single reason to motivate a switch to Julia, but hopefully this article piqued an interest to try it for your next project.

That said, Julia shouldn't be the only tool in your tool-kit. SQL will remain an important way to interact with databases. R and Python aren't going anywhere in the short term and will always offer a different perspective on things!

In an earlier article, I talked about becoming a **10x Actuary** which meant being proficient in the language of computers so that you could build and implement great things. In a large way, the choice of tools and paradigms shape your focus. Productivity is one aspect, expressiveness is another, speed one more. There are many reasons to think about what tools you use and trying out different ones is probably the best way to find what works best for you.

It is said that you cannot fully conceptualize something unless your language has a word for it. Similar to spoken language, you may find that breaking out of spreadsheet coordinates (and even a dataframe-centric view of the world) reveals different questions to ask and enables innovated ways to solve problems. In this way, you reward your intellect while building more meaningful and relevant models and analysis.

Part II.

Conceptual Foundations

4. Elements of Financial Modeling

“Truth ... is much too complicated to allow anything but approximations” - John von Neumann

4.1. In this Chapter

We explain what constitutes a financial model and what are common uses of a model. We explain what makes an adept practitioner.

4.2. What is a model?

A **model** represents aspects of the world around us distilled down into simpler, more tractable components. It is impossible to fully capture the everything that may affect the objects of our interest.

For example, say we want to simulate the returns for the stocks in our retirement portfolio. It would be impossible to try to build a model which would capture all of the individual people working jobs and making decisions, weather events that damage property, political machinations, etc. Instead, we try to capture certain fundamental characteristics. For example, it is common to model equity returns as cumulative pluses and minuses from random movements where those movements have certain theoretical or historical characteristics.

Whether we are using this model of equity returns to estimate available retirement income or replicate an exotic option price, a key aspect of the model is the **assumptions** used therein.

4. *Elements of Financial Modeling*

For the retirement income scenario we might *assume* a healthy eight percent return on stocks and conclude that such a return will be sufficient to retire at age 53. Alternatively, we may assume that future returns will follow a stochastic path with a certain distribution of volatility and drift. These two assumption sets will produce **output** - results from our model that must be inspected, questioned, and understood in the context of the “small world” of the model’s mechanistic workings. Lastly, to be effective practitioners we must be able to contextualize the “small world” results withing the “large world” that exists around us.

More on the “small world” vs “large world”: say that our model is one that discounts a fixed set of future cashflows using the US Treasury rate curve. If I run my model using current rates today, and then re-run my model tomorrow with the same future cashflows and the present value of those cashflows has increased by 5% I may ask why the result has changed so much in such a short period of time! In the “small”, mechanistic world of the model I may be able to see that the rates I used to discount the cashflows with have fallen substantially. The “small world” answer is that the inputs have changed which produced a mechanical change in the output. The “big world” answer may be that the Federal Reserve lowered the Federal Funds Rate to prevent the economy from entering a deflationary recession. Of course, we can’t completely explain the relation between our model and the real world (otherwise we could capture that relationship in our model!). An effective practitioner will always try to look up from the immediate work and take stock of how the world at large *is* or *is not* reflected in the model.

4.3. What is a *Financial Model*?

Financial models are those used extensively to ascertain better understanding of complex contracts, perform scenario analysis, and inform market participants’ decisions related to perceived value (and therefore price). It can’t be quantified directly, but it is likely not an exaggeration that many billions of dollars is transacted each day as a result of decisions made from the output of financial models.

4.4. The Process of Building a Financial Model

Most financial models can be characterized with a focus on the first or both of:

1. Attempting to project pattern of cashflows or obligations at future timepoints
2. Reducing the projected obligations into a current value

Examples of this:

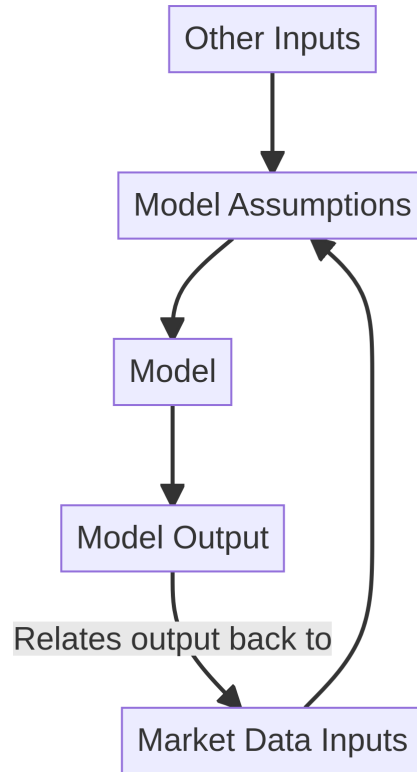
- Projecting a retiree's savings through time (1), and determining how much they should be saving today for their retirement goal (2)
- Projecting the obligation of an exotic option across different potential paths (1), and determining the premium for that option (2)

Models are sometimes taken a step further, such as transforming the underlying **economic view** into an accounting or regulatory view (such as representing associated debits and credits, capital requirements, or associated intangible, capitalized balances).

We should also distinguish a financial model from a purely statistical model, where often the inputs and output data are known and the intention is to estimate relationships between variables (example: linear regressions). That said, a financial model may have statistical components and many aspects of modeling is shared between the two kinds.

4.4. The Process of Building a Financial Model

i TODO: Describe model building process and make associated diagram



4.5. Predictive versus Explanatory Models

Given a set of inputs, our model will generate an output and we are generally interested in its accuracy. *The model need not have a realistic mechanism for how the world works.* That is, we may primarily be interested in accurately calculating an output value without the model having any scientific, explanatory power of how different parts of the real-world system interact.

4.5.1. A Historical Example

Consider the classic underdog story where Copernicus overthrew the status quo when he proposed (correctly) that the earth orbited the sun instead of the other way around⁷.

⁷ Prof. Richard Fitzpatrick has excellent coverage of the associated mathematics and implications in “A Modern Almagest”: <https://farside.ph.utexas.edu/books/Syntaxis/Almagest/Almagest.html>

4.5. Predictive versus Explanatory Models

The existing Ptolemaic model used a geocentric view of the solar system in which the planets and sun orbited the Earth in perfect circles with an epicycle used to explain retrograde motion (as see in Figure 4.1). Retrograde motion is the term used to describe the apparent, temporarily reversed motion of a planet as viewed from Earth when the Earth is overtaking the other planet in orbit around the sun. This was accurate enough to match the observational data that described the position of the planets in the sky.

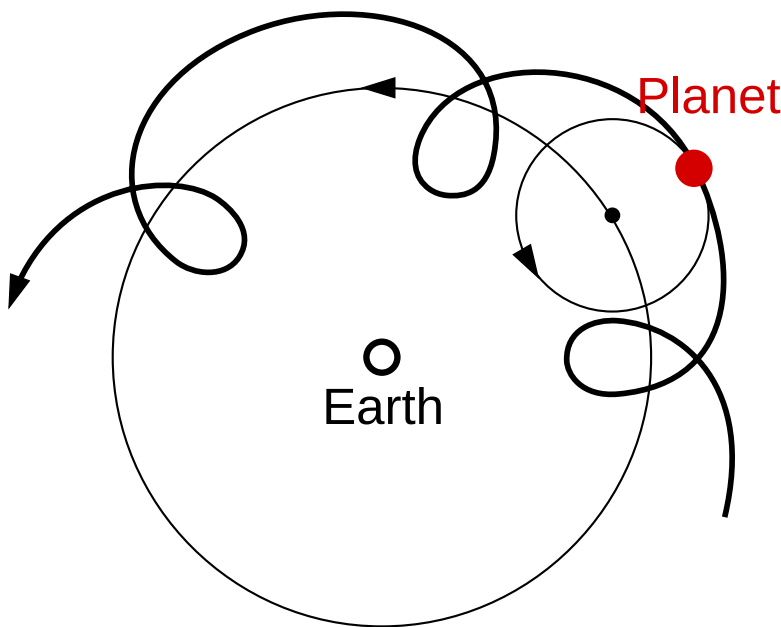


Figure 4.1.: In the Ptolemaic solar model, the retrograde motion of the planets was explained by adding an epicycle to the circular orbit around the earth.

Famously, Copernicus came along and said that the sun, not the Earth, should be at the center (a heliocentric model). Earth revolves around the sun! Today, we know this to be a much better description of reality than one in which the Earth arrogantly sits at the center of the universe. However the model was actually slightly *less* accurate in predicting the apparent position of the planets (to the limits of observational precision

4. Elements of Financial Modeling

at the time)! Why would this be?

First, the Copernican proposal still used perfectly circular orbits with an epicycle adjustment, which we know today to be inaccurate (in favor of an elliptical orbit consistent with the theory of gravity). *Despite being more scientifically correct, it was still not the complete picture.*

Second, the geocentric model was already very accurate because it was essentially a Taylor-series approximation which described to sufficient observational accuracy the apparent position of the planet relative to the Earth. *The heliocentric model was effectively a re-parameterization of the orbital approximation.*

Third, we have considered a limited criteria for which we are evaluating the model for accuracy, namely apparent position of the planets. *It's not until we contemplate other observational data that the Copernican model would demonstrate greater modeling accuracy:* apparent brightness of the planets as they undergo retrograde motion and angular relationship of the planets to the sun.

For modelers today, this demonstrates a few things to keep in mind:

1. Predictive models need not have a scientific, causal structure to make accurate predictions.
2. It is difficult to capture the complete scientific inter-relationships of a system and much care and thought needs to be given in what aspects are included in our model.
3. We should look at, or seek out, additional data that is related to our model because we may accurately fit (or overfit) to one outcome while achieving an increasingly poor fit to other related variables.

Striving to better understand the world is a *good thing to do* but trying to include more components into the model is not always going to help achieve our goals.

4.5. Predictive versus Explanatory Models

4.5.2. Examples in the Financial Context

4.5.2.1. Home Prices

American home prices which have a strong degree of seasonality and have the strongest prices around April of each year. We may find that including a simple oscillating term in our model captures the variability in prices *better* than if we tried to imperfectly capture the true market dynamics of home sales: supply and demand curves varying by personal (job bonus payment timing, school calendars), local (new homes built, company relocation), and national (monetary policy, tax incentives for home-ownership). In other words, one could likely predict a stable pattern like this with a model that contains a simple sinusoidal periodic component. One could likely spend months trying to build a more scientific model and not achieve as good of fit, *even though the latter tries to be more conceptually accurate*.

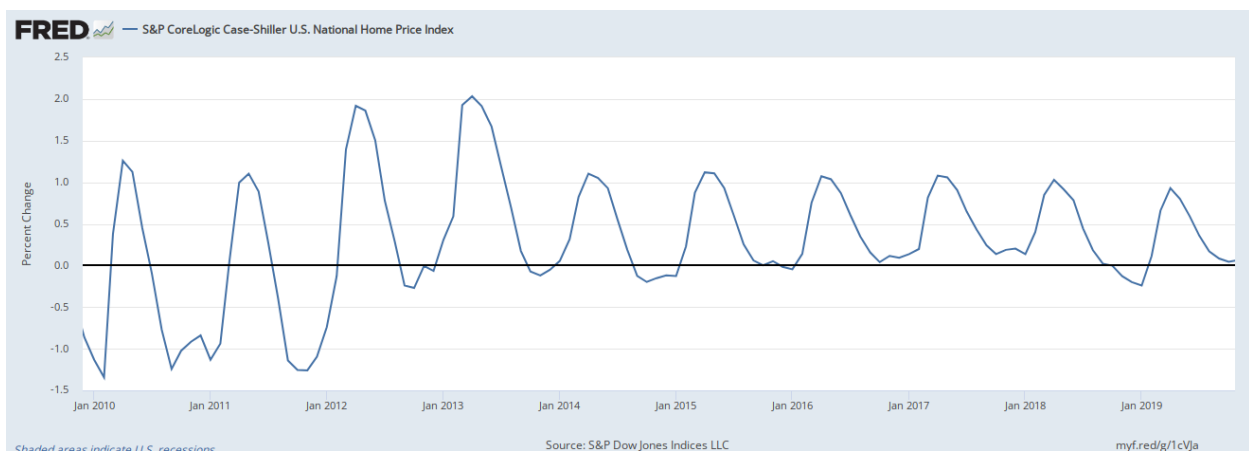


Figure 4.2.: House prices peak around April of each year.

4.5.2.2. Replicating Portfolio

Another example in the financial modeling realm: in attempting to value a portfolio of insurance contracts a **replicating portfolio** of hypothetical assets will sometimes be constructed⁸. The point of this is to create a basket of assets that can be more

⁸ See, e.g., SOA Investment Symposium March 2010. *Replicating Portfolios in the Insurance Industry* (Curt Burmeister Mike Dorsel Patricia Matson)

4. Elements of Financial Modeling

quickly (minutes to hours) valued in response to changing market conditions than it would take to run the actuarial model (hours to days). This is an example where the basket of assets has no ability to explain why the projected cashflows are what they are - but retains strong predictive accuracy.

4.6. What makes a good model?

The answer is: *it depends*.

4.6.1. Achieving original purpose

A model is built for a specific set of reasons and therefore we must evaluate a model in terms of achieving that goal. We should not critique a model if we want to use it outside of what it was intended to do. This includes: contents of output and required level of accuracy.

A model may have been created to for scenario analysis to value all assets in a portfolio to within half a percent of a more accurate, but much more computationally expensive model. If we try to add a never-before-seen asset class or use the model to order trades we may be extending the design scope of the original model.

4.6.2. Usability

How easy is it for someone to use? Does it require pages and pages of documentation, weeks of specialized training and an on-call help desk? *All else equal*, it is an indicator of how usable the model is by the amount of support and training. However, one may sometimes wish to create a highly capable, complex model which is known to require a high amount of experience and expertise. An analogy here might be the cockpit of a small Cessna aircraft versus a fighter jet: the former is a lot simpler and takes less training to master but is also more limited.

Figure 4.3 illustrates this concept and shows that if your goal is very high capability that you may need to expect to develop

4.6. What makes a good model?

training materials and support the more complex model. On this view, a better model is one that is able to have a shorter amount of time and experience to achieve the same level of capability.

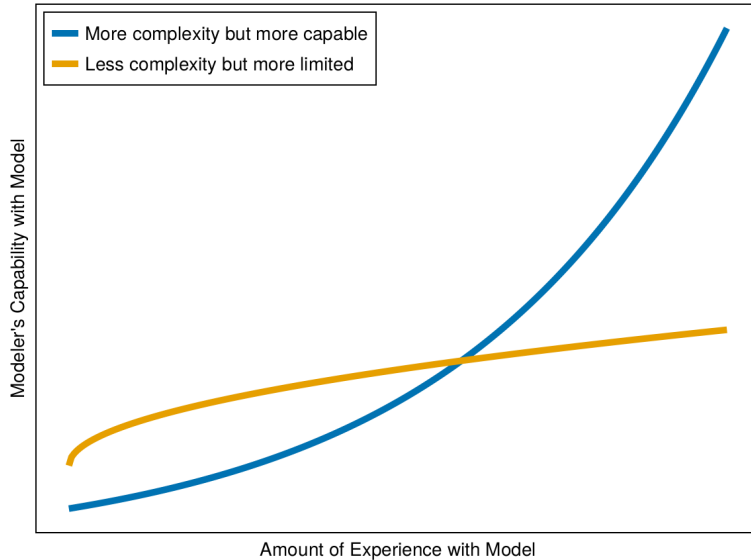


Figure 4.3.: Tradeoff between complexity and capability

4.6.3. Performance

Financial models are generally not used for their awe-inspiring beauty - users are results oriented and the faster a model returns the requested results, the better. Aside from direct computational costs such as server runtime, a shorter model runtime means that one can iterate faster, test new ideas on the fly, and stay focused on the problem at hand.

Many readers may be familiar with the cadence of (1) try running model overnight, (2) see results failed in the morning, (3) spend day developing, (4) repeat step 1. It is preferred if this cycle can be measured in minutes instead of hours or days.

Of course, requirements must be considered here too: needs for high frequency trading, daily portfolio rebalancing, and quarterly valuations are different when it comes to performance.

4.7. What makes a good modeler?

A model is nothing without it's operator, and a skilled practitioner is worth their weight in gold. What elements separate a good modeler from a mediocre modeler?

4.7.1. Domain Expertise

4.7.2. Model Theory

4.7.3. Curiosity

4.7.4. Rigor

4.7.5. Toolset

... The skills in this book!

5. Elements of Programming

“Programming is not about typing, it’s about thinking.” — Rich Hickey (2011)

5.1. In this section

Start building up computer science concepts by introducing tangible programming essentials. Data types, variables, control flow, functions, and scope are introduced.

5.2. Computer Science, Programming, and Coding

Computer Science is the study of computing and information. As a science, it is distinct from programming languages which are merely coarse implementations of specific computer science concepts⁹. Programming (or “coding”) is the art and science of writing code in programming languages to have the computer perform desired tasks. While this may sound mechanistic, programming truly is one of the highest forms of abstract thinking and the design space of potential solutions is so large and potentially complex that much art and experience is needed to create a well-made program.

The language of computer science also provides a lexicon so that financial practitioners can discuss model architecture and problem characteristics. Having the language to describe a concept will also help see aspects of the problem in new ways, opening one up to more innovative solutions.

In the context of this financial modeling that we do, we can consider a financial model to be a type of computer program.

⁹ Said differently, computer science may contemplate ideas and abstractions more generally than a specific implementation, as in mathematics where a theorem may be proved ($a^2 + b^2 = c^2$) without resorting to specific numeric examples ($3^2 + 4^2 = 5^2$).

5. *Elements of Programming*

It takes as input abstract information (data), performs calculations (an algorithm), and returns new data as an output. In this context, we generally do not need to consider many things that a software engineer may contemplate such as a graphical user interface, networking, or access restrictions. But there are many similarities: a good financial modeler must understand data types, algorithms, and some hardware details.

We will build up the concepts over this and the following chapter:

- This chapter will provide a survey of important concepts in computer science that will prove useful for our financial modeling. First, we will talk about data types, boolean logic, and basic expressions. We'll build on those to discuss algorithms (functions) which perform useful work and use control flow and recursion.
- The following chapter will step back and discuss higher level concepts: the “schools of thought” around organizing the relationship between data and functions (functional versus object-oriented programming), design patterns, computational complexity, and compilation.

Tip

There will be brief references to hardware considerations for completeness, but hardware knowledge is not necessary to understand most programming languages (including Julia). It's impossible to completely avoid talking about hardware when you care about the performance of your code, so feel free to gloss over the reference to hardware details on the first read and come back later after Chapter 8.

It's highly recommended that you follow along and have a Julia session open (e.g. a REPL or a notebook) when first going through this chapter. See Chapter 22 if you haven't gotten that set up yet. Follow along with the examples as we go.

Tip

You can get some help in the REPL by typing a `?` followed by the symbol you want help with, for example:

```
help?> sum
search: sum sum! summary cumsum cumsum! ...

sum(f, itr; [init])

Sum the results of calling function f on each element of itr.

... More text truncated...
```

Caution

This introductory chapter is intended to provide a survey of the important concepts and building blocks, not to be a complete reference. For full details on available functions, more complete definitions, and a more complete tour of all language features, see the Manual at docs.julialang.org.

5.3. Assignment and Variables

One of the first things it will be convenient to understand is the concept of variables. In virtually every programming language, we can assign values to make our program more organized and meaningful to the human reader. In the following example, we assign values to intermediate symbols to benefit us humans as we convert (silly!) American distance units:

```
feet_per_yard = 3
yards_per_mile = 1760

feet = 3000
miles = feet / feet_per_yard / yards_per_mile
```

5. Elements of Programming

0.5681818181818182

Beyond readability, variables are a form of **abstraction** which allows us to think beyond specific instances of data and numbers to a more general representation. For example, the last line in the prior code example is a very generic computation of a unit conversion relationship and `feet` could be any number and the expression remains a valid calculation.

We will return to this subject in more detail in ([ref-assignment?](#)).

?quarto-cite:ref-assignment

5.4. Data Types

Data types are a way of categorizing information by intrinsic characteristics. We instinctively know that `13.24` is different than "this set of words" and types are how we will formalize this distinction. This is a key conceptual point, and mathematically it's like we have different sets of objects to perform specialized operations on. Beyond this set-like abstraction is implementation details related to computer hardware. You probably know that computers only natively "speak" in binary zeros and ones. Data types are a primary way that a computer can understand if it should interpret `01000010` as `B` or as `66`¹⁰.

Each `0` or `1` within a computer is called a **bit** and eight bits in a row form a **byte** (such as `01000010`). This is where we get terms like "gigabytes" or "kilobits per second" as a measure of the quantity or rate of bits something can handle¹¹.

5.4.1. Numbers

Numbers are usually grouped into two categories: **integers** and **floating-point**¹² numbers. Integers are like the mathematical set of integers while floating-point is a way of representing decimal numbers. Both have some limitations since computers can only natively represent a finite set of numbers due to the hardware (more on this in Chapter 8). Here are three integers that are input into the **REPL** (Read-Eval-Print-Loop)¹³ and the

¹⁰ This binary representations correspond to `B` and `66` with the *ASCII character set* and 8-bit integer encodings respectively, discussed later in this chapter.

¹¹ Some distinctions you may encounter: in short-form, "kb" means *kilobits* while the upper-case "B" in "kB" means *kilobytes*. Also confusingly, sometimes the "k" can be binary or decimal - because computers speak in binary, a binary "k" means 1024 (equal to 2^{10}) instead of the usual decimal 1000. In most computer contexts, the binary (multiples of 1024) is more common.

¹² The term floating point refers to the fact that the number's radix (decimal) point can "float" between the significant digits of the number.

¹³ That is, it *reads* the code input from the user, *evaluates* what code was given to it, *prints* the result of the input to the screen, and *loops* through the process again.

5.4. Data Types

result is **printed** below the input:

```
2
```

```
2
```

```
423
```

```
423
```

```
1929234
```

```
1929234
```

And three floating-point numbers:

```
0.2
```

```
0.2
```

```
-23.3421
```

```
-23.3421
```

```
14e3      # the same as 14,000.0
```

```
14000.0
```

On most systems, `0.2` will be interpreted as a 64-bit floating point type called `Float64` in Julia since most architectures these days are 64-bit¹⁴, while on a 32-bit system `0.2` would be interpreted as a `Float32`. Given that there are a finite amount of bits attempting to represent a continuous, infinite set of numbers means that some numbers are not able to be represented with perfect precision. For example, if we ask for `0.2`, the closest representations in 64 and 32 bit are:

¹⁴ This means that their central processing units (CPUs) use instructions that are 64 bits long.

5. Elements of Programming

- 0.200000000298023223876953125 in 32-bit
- 0.2000000000000000011102230246251565404236316680908203125 in 64-bit

This leads to special considerations that computers take when performing calculations on floating point maths, some of which will be covered in more detail in Chapter 8. For now, just note that floating point numbers have limited precision and even if we input 0.2, your computations will use the above decimal representations even if it will print out a number with fewer digits shown:

```
x = 0.2 ①  
big(x) ②
```

- ① Here, we **assign** the value 0.2 to a **variable** x. More on variables/assignments in Section 5.5.3.
- ② `big(x)` is a arbitrary precision floating point number and by default prints the full precision that was embedded in our variable x, which was originally `Float64`.

0.2000000000000000011102230246251565404236316680908203125

Note

Note the difference in what printed between the last example and when we input 0.2 earlier in the chapter. The former had the same (not-exactly equal to 0.2) *value*, but it printed an abbreviated set of digits as a nicety for the user, who usually doesn't want to look at floating point numbers with their full machine precision. The system has the full precision (0.20...3125) but is truncating the output.

In the last example, we've converted the normal `Float64` to a `BigFloat` which will not truncate the output when printing.

Integers are similarly represented as 32 or 64 bits (with `Int32` and `Int64`) and are limited to exact precision:

5.4. Data Types

- -32,767 to 32,767 for Int32
- -2,147,483,647 to 2,147,483,647 for Int64

Additional range in the positive direction if one chooses to use “unsigned”, non-negative numbers (UInt32 and UInt64). Unlike floating point numbers, the integers have a type Int which will use the system bit architecture by default (that is, Int(30) will create a 64 bit integer on 64-bit systems and 32-bit on 32-bit systems).

Financial Modeling Pro-tip

Excel’s numeric storage and routine is complex and not quite the same as most programming languages, which follow the Institute of Electrical and Electronics Engineer’s standards (such as the IEEE 754 standard for double precision floating point numbers). Excel uses IEEE for the *computations* but results (and therefore the cells that comprise many calculations interim values) are stored with 15 significant digits of information. In some ways this is the worst of both worlds: having the sometimes unusual (but well-defined) behavior of floating point arithmetic *and* having additional modifications to various steps of a calculation. In general, you can assume that the programming language result (following the IEEE 754 standard) is a better result because there are aspects to the IEEE 754 defines techniques to minimize issues that arise in floating point math. Some of the issues (round-off or truncation) can be amplified instead of minimized with Excel.

In practice, this means that it can be difficult to exactly replicate a calculation in Excel in a programming language and vice-versa. It’s best to try to validate a programming model versus Excel model using very small unit calculations (e.g. a single step or iteration of a routine) instead of an all-in result. You may need to define some tolerance threshold for comparison of a value that is the result of a long chain of calculation.

5.4.2. Type Hierarchy

We can describe a *hierarchy* of types. Both `Float64` and `Int64` are examples of `Real` numbers (here, `Real` is an **abstract** Julia type which corresponds to the mathematical set of real numbers commonly denoted with \mathbb{R}). Both `Float64` and `Int32` are `Real` numbers, so why not just define all numbers as a `Real` type? Because for performant calculations, the computer must know in advance how many bits each number is represented with.

Figure 5.1 shows the type hierarchy for most built-in Julia number types.

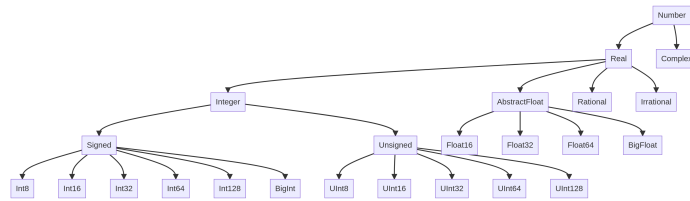


Figure 5.1.: Numeric Type Hierarchy in Julia. Leafs of the tree are concrete types.

The integer and floating point types described in the prior section are known as **concrete** types because there are no possible sub types (child types). Further, a concrete type can be a **bit type** if the data type will always have the same number of bits in memory: a `Float32` will always be 32 bits in memory, for example. Contrast this with strings (described below) which can contain an arbitrary number of characters.

5.4.3. Arrays

Arrays are the most common way to represent a collection of similar data. For example, we can represent a set of integers as follows:

Julia has very powerful and friendly array types.

5.4. Data Types

```
[1, 10, 300]
```

```
3-element Vector{Int64}:
```

```
 1  
10  
300
```

And a floating point array:

```
[0.2, 1.3, 300.0]
```

```
3-element Vector{Float64}:
```

```
0.2  
1.3  
300.0
```

Note the above two arrays are different types of arrays. The first is `Vector{Int64}` and the second is `Vector{Float64}`. These are arrays of concrete types and so Julia will know that each element of an array is the same amount of bits which will enable more efficient computations. With the following set of mixed numbers, Julia will **promote** the integers to floating point since the integers can be accurately represented¹⁵ in floating point.

```
[1, 1.3, 300.0, 21]
```

```
4-element Vector{Float64}:
```

```
1.0  
1.3  
300.0  
21.0
```

However, if we explicitly ask Julia to use a `Real`-typed array, the type is now `Vector{Real}`. Recall that `Real` is an abstract type. Having heterogeneous types within the array is conceptually fine, but in practice limits performance. Again, this will be covered in more detail in Chapter 8.

¹⁵ Accurate only to a limited precision, as described in Section 5.4.1.

5. Elements of Programming

In Julia, arrays can be multi-dimensional. Here are two three-dimensional arrays with length three in each dimension:

```
rand(3, 3, 3)
```

```
3×3×3 Array{Float64, 3}:  
[:, :, 1] =  
 0.189937  0.313227  0.756255  
 0.155365  0.560377  0.0781305  
 0.511681  0.530971  0.100398  
  
[:, :, 2] =  
 0.250718  0.275126  0.0377606  
 0.703835  0.587014  0.542175  
 0.350468  0.358308  0.817599  
  
[:, :, 3] =  
 0.56447  0.588792  0.730087  
 0.278002  0.333277  0.387758  
 0.899206  0.41445  0.860141
```

```
[x + y + z for x in 1:3, y in 11:13, z in 21:23]
```

```
3×3×3 Array{Int64, 3}:  
[:, :, 1] =  
 33 34 35  
 34 35 36  
 35 36 37  
  
[:, :, 2] =  
 34 35 36  
 35 36 37  
 36 37 38  
  
[:, :, 3] =  
 35 36 37  
 36 37 38  
 37 38 39
```

5.4. Data Types

The above example demonstrates **array comprehension** syntax which is a convenient way to create arrays in Julia.

A two-dimensional array has the rows by semi-colons (;):

```
x = [1 2 3; 4 5 6]
```

```
2×3 Matrix{Int64}:
```

```
1  2  3
4  5  6
```

Note

In Julia, a `Vector{Float64}` is simply a one-dimensional array of floating points and a `Matrix{Float64}` is a two-dimensional array. More precisely, they are **type aliases** of the more generic `Array{Float64,1}` and `Array{Float64,2}` names.

5.4.3.1. Array indexing

Array elements are accessed with the integer position, starting at 1 for the first element¹⁶ ¹⁷:

```
v = [10, 20, 30, 40, 50]
v[2]
```

20

We can also access a subset of the vector's contents by passing a range:

```
v[2:4]
```

```
3-element Vector{Int64}:
```

```
20
30
40
```

¹⁶ Whether an index starts at 1 or 0 is sometimes debated. Zero-based indexing is natural in the context of low-level programming which deal with bits and positional *offsets* in computer memory. For higher level programming one-based indexing is more natural: in a set of data stored in an array, it is much more natural to reference the *first* (through n^{th}) datum instead of the *zeroth* (through $(n-1)^{th}$ datum).

¹⁷ Arrays in Julia can actually be indexed with an arbitrary starting point: see the package `OffsetArrays.jl`

5. Elements of Programming

And we can generically reference the array's contents, such as:

```
v[begin+1:end-1]
```

3-element Vector{Int64}:

```
20  
30  
40
```

We can assign values into the array as well, as well as combine arrays and push new elements to the end:

```
v[2] = -1  
push!(v, 5)  
vcat(v, [1, 2, 3])
```

9-element Vector{Int64}:

```
10  
-1  
30  
40  
50  
5  
1  
2  
3
```

5.4.3.2. Array Alignment

When you have an $M \times N$ matrix (M rows, N columns), a choice must be made as to which elements are next to each other in memory. Typical math convention and fundamental computer linear algebra libraries (dating back decades!) are column major and Julia follows that legacy. **Column major** means that elements going down the rows of a column are stored next to each other in memory. This is important to know so that (1) you remember that vectors are treated like a column vector

5.4. Data Types

when working with arrays (a N element 1D vector is like a Nx1 matrix), and (2) when iterating through an array, it will be faster for the computer to access elements next to each other column-wise. A 10x10 matrix is actually stored in memory as 100 elements coming in order, one after another in single file.

This 3x4 matrix is stored with the elements of columns next to each other, which we can see with `vec`:

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

3x3 Matrix{Int64}:

```
1  2  3
4  5  6
7  8  9
```

```
vec(mat)
```

9-element Vector{Int64}:

```
1
4
7
2
5
8
3
6
9
```

5.4.4. Characters, Strings, and Symbols

Characters are represented in most programming languages as letters within quotation marks. In Julia, individual characters are represented using single quotes:

```
'a'
```

'a': ASCII/Unicode U+0061 (category Ll: Letter, lowercase)

5. Elements of Programming

Letters and other characters present more difficulties than numbers to represent within a computer (think of how many languages and alphabets exist!), and it essentially only works because the world at large has agreed to a given representation. Originally **ASCII** (American Standard Code for Information Interchange) was used to represent just 95 of the most common English characters (“a” through “z”, zero through nine, etc.). Now, UTF (Unicode Transformation Format) can encode more than a million characters and symbols from many human languages.

¹⁸ Under the hood, strings are essentially a vector of characters but there are complexities with character encoding that don’t allow a lossless conversion to individual characters of uniform bit length. This is for historical compatibility reasons and to avoid making most documents’ file sizes larger than it needs to be.

Strings are a collection¹⁸ of characters, and can be created in Julia with double quotes:

```
"hello world"
```

```
"hello world"
```

It’s easy to ascertain how ‘normal’ characters can be inserted into a string, but what about things like new lines or tabs? They are represented by their own characters but are normally not printed in computer output. However, those otherwise invisible characters do exist. For example, here we will use a **string literal** (indicated by the `"""`) to tell Julia to interpret the string as given, including the invisible new line created by hitting return on the keyboard between the two words:

```
"""  
hello  
world  
"""
```

```
"hello\nworld\n"
```

The output above shows the `\n` character contained within the string.

Symbols are a way of representing an identifier which cannot be seen as a collection of individual characters. `:helloworld` is distinct from `"helloworld"` - you can kind of think of the former

as an un-executed bit of code - if we were to execute it (with `eval(:helloworld)`), we would get an error `UndefVarError: `a` not defined`. Symbols can *look* like strings but do not behave like them. For now, it is best to not worry about symbols but it is an important aspect of Julia which allows the language to represent aspects of itself as data. This allows for powerful self-reference and self-modification of code but this is a more advanced topic generally out of scope of this book.

5.4.5. Tuples

Tuples are a set of values that belong together and are denoted by a values inside parenthesis and separated by a comma. An example might be x-y coordinates in 2 dimensional space:

```
x = 3
y = 4
p1 = (x, y)
```

`(3, 4)`

Tuple's values can be accessed like arrays:

```
p1[1]
```

3

Tuples fill a middle ground between scalar types and arrays in more ways than one:

- Tuples have no problem having heterogeneous types in the different slots.
- Tuples are **immutable**, meaning that you cannot overwrite the value in memory (an error will be thrown if we try to do `p[1] = 5`).

5. Elements of Programming

- It's generally expected that within an array, you would be able to apply the same operation to all the elements (e.g. square each element) or do something like sum all of the elements together which isn't generally case for a tuple.
- Tuples are generally stack allocated instead of being heap allocated like arrays¹⁹, meaning that a lot of times they can be faster than arrays.

¹⁹ What this means will be explained in Chapter 8 .

5.4.5.1. Named Tuples

Named tuples provide a way to give each field within the tuple a specific name. For example, our x-y coordinate example above could become:

```
p2 = (x=3, y=4)
```

```
(x = 3, y = 4)
```

The benefit is that we can give more meaning to each field and access the values in a nicer way. Previously, we used `location[1]` to access the x-value, but with the new definition we can access it by name:

```
p2.x
```

```
3
```

5.4.6. Parametric Types

We just saw how tuples can contain heterogeneous types of data inside a common container. Let's look at this a little bit closer by looking at the full type:

```
typeof(p1)
```

```
Tuple{Int64, Int64}
```


5.4. Data Types

`location` is a `Tuple{Int64,Int64}` type, which means that its first and second elements are both `Int64`. Contrast this with:

```
typeof(("hello", 1.0))
```

`Tuple{String, Float64}`

These tuples are both of the form `Tuple{T,U}` where `T` and `U` are both types. Why does this matter? We and the compiler can distinguish between a `Tuple{Int64,Int64}` and a `Tuple{String,Float64}` which allows us to reason about things (“I can add the first element of tuple together only if both are numbers”) and the compiler to optimize (sometimes it can know exactly how many bits in memory a tuple of a certain kind will need and be more efficient about memory use). Further, we will see how this can become a powerful force in writing appropriately abstracted code and more logically organize our entire program when we encounter “multiple dispatch” later on.

5.4.7. Types for things not there

`nothing` represents that there’s nothing to be returned - for example if there’s no solution to an optimization problem or if a function just doesn’t have any value to return (such as in the case with input/output like `println`).

`missing` is to represent something *should* be there but it’s not, as is all too common in real-world data. Julia natively supports `missing` and three-value logic, which is an extension of the two-value boolean (true/false) logic, to handle missing logical values:

Tip

`Missing` and `Nothing` are the *types* while `missing` and `nothing` are the values here. This is analogous to `Float64` being a type and `2.0` being a value.

5. Elements of Programming

Table 5.1.: Three value logic with true, missing, and false.

(a) Not logic		(b) And logic			(c) Or Logic			
NOT (!)	Value	AND (&)	true	missing	OR ()	true	missing	false
true	false	true	true	missing	false	true	true	true
missing	missing	missing	missing	missing	missing	true	missing	missing
false	true	false	false	false	true	true	missing	false

5.4.8. Union Types

When two types may arise in a context, **union types** are a way to represent that. For example, if we have a data feed and we know that it will produce *either* a `Float64` or a `Missing` type then we can say that the value for this is `Union{Float64,Missing}`. This is much better for the compiler (and our performance!) than saying that the type of this is `Any`.

5.4.9. Creating User Defined Types

We've talked about some built-in types but so much additional capabilities come from being able to define our own types. For example, taking the x-y-coordinate example from above, we could do the following instead of defining a tuple:

```
struct BasicPoint
    x::Int64
    y::Int64
end

p3 = BasicPoint(3, 4)
```

`BasicPoint(3, 4)`

Fields are accessed the same way as named tuples:

5.4. Data Types

```
p3.x, p3.y
```

①

- ① Note that here, Julia will return a tuple instead of a single value due to the comma separated expressions.

```
(3, 4)
```

structs in Julia are immutable like tuples above.

But wait, didn't tuples let us mix types too via parametric types? Yes, and we can do the same with our type!

```
struct Point{T}
    x::T
    y::T
end
```

Line 1 The {T} after the type's name allows for different Points to be created depending on what the type of the underlying x and y is.

Here's two new points which now have different types:

```
p4 = Point{1, 4}
p5 = Point{2.0, 3.0}

p4, p5
```

```
(Point{Int64}(1, 4), Point{Float64}(2.0, 3.0))
```

Note that the types are not equal because they have different type parameters!

```
typeof(p4), typeof(p5), typeof(p4) == typeof(p5)
```

```
(Point{Int64}, Point{Float64}, false)
```

But both are now subtypes of PPoint2D. The expression `X isa Y` is true when X is a (sub)type of Y:

5. Elements of Programming

```
p4 isa Point, p5 isa Point
```

```
(true, true)
```

Note though, that the `x` and `y` are both of the same type in each `PPoint2D` that we created. If instead we wanted to allow the coordinates to be of different types, then we could have defined `PPoint2D` as follows:

```
struct Point{T,U}
    x::T
    y::U
end
```

Note

Can we define the structs above without indicating a (parametric) type? Yes!

```
struct Point
    x # no type here!
    y # no type declared here either!
end
```

But! `x` and `y` will both be allowed to be `Any`, which is the fallback type where Julia says that it doesn't know any more about the type until runtime (the time at which our program encounters the data when running). This means that the compiler (and us!) can't reason about or optimize the code as effectively as when the types are explicit or parametric. This is an example of how Julia can provide a nice learning curve - don't worry about the types until you start to get more sophisticated about the program design or need to extract more performance from the code.

The above structs that we have defined are examples of **concrete types** which hold data. **Abstract types** don't directly hold data themselves but are used to define a hierarchy of types which we will later exploit (Chapter 6) to implement custom behavior depending on what type our data is.

```

abstract type Coordinate end
abstract type CartesianCoordinate <: Coordinate end
abstract type PolarCoordinate <: Coordinate end

```

Here's an example of (1) defining a set of related types that sits above our `Point2D{T}` <: `CartesianCoordinate`

```

struct Point2D{T} <: CartesianCoordinate
    x::T
    y::T
end

struct Point3D{T} <: CartesianCoordinate
    x::T
    y::T
    z::T
end

struct Polar2D{T} <: PolarCoordinate
    r::T
    θ::T
end

```

💡 Unicode Characters

Julia has wonderful Unicode support, meaning that it's not a problem to include characters like θ . The character can be typed in Julia editors by entering `\theta` and then pressing the TAB key on the keyboard.

Unicode is helpful for following conventions that you may be used to in math. For example, the math formula $\text{circumference}(r) = 2 \times r \times \pi$ can be written in Julia with `circumference(r) = 2 * r * π`.

The name for the characters follows the same for LaTeX, so you can search the internet for, e.g. “theta LaTeX” to find the appropriate name. Further, you can use the REPL help mode to find out how to enter a character if you can copy and paste it from somewhere:

5. Elements of Programming

```
help?> θ  
"θ" can be typed by \theta<tab>
```

5.4.10. Mutable structs

It is possible to define structs where the data can be modified - such a data field is said to be **mutable** because it can be changed or mutated. Here's an example of what it would look like if we made `Point2D` mutable:

```
mutable struct Point2D{T}  
    x::T  
    y::T  
end
```

You may find that this more naturally represents what you are trying to do. However, recall that an advantage of an immutable datatype is that costly memory doesn't necessarily have to be allocated for it. So you may think that you're being more efficient by re-using the same object... but it may not actually be faster. Again, more will be revealed in Chapter 8.



Financial Modeling Pro-tip

Generally you should default to using immutable types and consider only moving to mutable types in specific circumstances. You'll see some examples in the applications later in the book.

5.5. Expressions and Control Flow

Having already seen some more illustrative examples above, we can zoom in onto smaller pieces called **expressions** which are effectively the basic block of code that gets evaluated. Here is an expression that adds two integers together that evaluate to a new integer (3 in this case):

```
1 + 2
```

```
3
```

5.5.1. Compound Expression

There's two kinds of blocks where we can ensure that subexpressions get evaluated in order and return the last expression as the overall return value: `begin` and `let` blocks.

```
c = begin
  a = 3
  b = 4
  a + b
end

a, b, c
```

```
(3, 4, 7)
```

The variables inside the `begin` block are evaluated in the same scope as `c` and therefore have the assigned values when we call `a` and `b` in the last line. Contrast that with the `let` block below, where `d` and `e` are not available when we try to get the value of `f`. This is because `let` creates a new inner scope that's not available in `f`'s scope. More on scope later in the chapter.

```
f = let
  d = 1
  e = 2
  d + e
end

f
```

```
3
```

```
d
```

```
LoadError: UndefVarError: `d` not defined
```

5.5.2. Conditional Expressions

Conditionals are expressions that evaluate to a **boolean** `true` or `false`. This is the beginning of really being able to assemble complex logic to perform useful work. Here are a handful of expressions that would evaluate to `true`:

```
1 > 0
1 == 1 # check for equality
Float64 isa Rational
(5 > 0) & (-1 < 2) # "and" expression
(5 > 0) | (-1 > 2) # "or" expression
1 != 2
```

Note

In Julia, the booleans have an integer equality: `true` is equal to 1 (`true == 1`) and `false` is equal to 0 (`false == 0`). However:

- `true != 5`. Only 1 is equal to `true` (in some languages, any non-zero number is “truthy”).
- `true` is not equal to 1 (equal is defined later in this chapter).

Conditionals can be used to assemble different logical paths for the program to follow and the general pattern is an `if` block:

```
if condition
    # do one thing
elseif condition
    # do something else
else
    # do something if none of the
    # other conditions are met
end
```

A complete example:

5.5. Expressions and Control Flow

```
function buy_or_sell(my_value, market_price)
  if my_value > market_price
    "buy more"
  elseif my_value < market_price
    "sell"
  else
    "hold"
  end
end

buy_or_sell(10, 15), buy_or_sell(15, 10), buy_or_sell(10, 10)
```

("sell", "buy more", "hold")

5.5.2.1. Equality

The “Ship of Theseus²⁰” problem is an example of how equality can be philosophically complex concept. In computer science we have the advantage that while we may not be able to resolve what’s the “right” type of equality, we can be more precise about it.

Here is an example for which we can see the difference between two types of equality:

- **Egal** equality is when a program could not distinguish between two objects at all
- **Equal** equality is when the values of two objects are the same

If two things are egal, then they are also equal.

In the following example, s and t are equal but not egal:

```
s = [1, 2, 3]
t = [1, 2, 3]
s == t, s === t
```

(true, false)

²⁰ The Ship of Theseus problem specifically refers to a legendary ancient Greek ship, owned by the hero Theseus. The paradox arises from the scenario where, over time, each wooden part of the ship is replaced with identical materials, leading to the question of whether the fully restored ship is still the same ship as the original. The Ship of Theseus problem is a thought experiment in philosophy that explores the nature of identity and change. It questions whether an object that has had all of its components replaced remains fundamentally the same object.

5. Elements of Programming

One way to think about this is that while the values are equal, there is a way that one of the arrays could be made not equal to the other:

```
t[2] = 5  
t
```

3-element Vector{Int64}:

```
1  
5  
3
```

Now `t` is no longer equal to `s`:

```
s == t
```

false

Recall that arrays are able to be modified, but other types like tuples are immutable. Immutable types with the same value are equal because there is no way for us to make them different:

```
(2, 4) === (2, 4)
```

true

Using this terminology, we could now interpret the “Ship of Theseus” as that his ship is “equal” but not “egal”.

5.5.3. Assignment and Variables

When we say `x = 2` we are **assigning** the integer value of 2 to the variable `x`. This is an expression that lets us bind a something to a variable so that it can be referenced more concisely or in different parts of our code. When we re-assign the variable we are not mutating the value: `x = 3` does not change the 2.

5.5. Expressions and Control Flow

When we have a mutable object (e.g. an Array or a mutable struct), we can mutate the value inside the referenced container. For example:

```
x = [1, 2, 3]           ①  
x[1] = 5                ②  
x
```

- ① x refers to the array which currently contains the elements 1, 2, and 3.
- ② We re-assign the first element of the array to be the value 5 instead of 1

```
3-element Vector{Int64}:  
 5  
 2  
 3
```

In the above example, x has not been reassigned. It is possible for two variables to refer to the same object:

```
x = [1,2,3]  
y = x  
x[1] = 6  
y
```

- ① y refers to the *same* underlying array as x

```
3-element Vector{Int64}:  
 6  
 2  
 3
```

5.5.4. Loops

Loops are ways for the program to move through a program and repeat expressions while we want it to. There are two primary loops: for and while.

5. Elements of Programming

for loops are loops that iterate over a defined range or set of values. Let's assume that we have the array $v = [6,7,8]$. Here are multiple examples of using a for loop in order to print each value to output (println):

```
# use fixed indices
for i in 1:3
    println(v[i])
end
# Use indices the of the array
for i in eachindex(v)
    println(v[i])
end
# Use the elements of the array
for x in v
    println(x)
end
# Use the elements of the array
for x in v          # ∈ is typed \in<tab>
    println(x)
end
```

while loops will run repeatedly until an expression is false. Here's some examples of printing each value of v again:

```
# index the array
i = 1
while i <= length(v)
    println(v[i])
    global i += 1 #<1>
end
```

- ① `global` is used to increment `i` by 1. `i` is defined outside the scope of the while loop (see Section 5.7).

```

# index the array
i = 1
while true
    println(v[i])
    if i >= length(v)
        break
    end
    global i += 1
end

```

①

- ① `break` is used to terminate the loop manually, since the condition that follows the `while` will never be false.

5.5.5. Performance of loops

Loops are highly performant in Julia and often the fastest way to accomplish things. Those coming from Python or R may have developed a habit to avoid writing loops. *Fear the for loop not!*

5.6. Functions

Functions are a set of expressions that take inputs and return specified outputs.

5.6.1. Special Operators

Operators are the glue of expressions which combine values. We've already seen quite a few, but let's develop a little bit of terminology for these functions.

Unary operators are operators which only take a single argument. Examples include the `!` which negates a boolean value or `-` which negates a number:

```
!true, -5
```

```
(false, -5)
```

5. Elements of Programming

Binary operators take two arguments and are some of the most common functions we encounter, such as `+` or `-` or `>`:

```
1 + 2, 1 - 2, 1 > 2
```

```
(3, -1, false)
```

The above unary and binary operators are special kinds of functions which don't require the use of parenthesis. However, they can be written with parenthesis for greater clarity:

```
!(true), -(5), +(1,2), -(1,2)
```

```
(false, -5, 3, -1)
```

In Julia, we distinguish between **functions** which define behavior that maps a set of inputs to outputs. But a single function can adapt its behavior to the arguments themselves. We have just seen the function `-` be used in two different ways: negation and subtraction depending on whether it had one or two arguments given to it. In this way there is a conceptual hierarchy of functions that complements the hierarchy we have discussed in relation to types:

- `-` is the overall function
- `-(x)` is a unary function which negates its values, `-(x,y)` subtracts `y` from `x`
- Specific methods are then created for each combination of concrete types: `-(x::Float64)` is a different method than `-(x::Int)`

Methods are specific compiled versions of the function for specific types. This is important because at a hardware level, operations for different types (e.g. integers versus floating point) differ considerably. By optimizing for the specific types Julia is able to achieve nearly ideal performance without the same sacrifices of other dynamic languages. We will develop more with respect to methods when we talk about dispatch in Chapter 6.

5.6.2. General Functions

Functions more generally are defined like so:

```
function distance(point)           ①
    return sqrt(point.x^2 + point.y^2)  ②
end
```

- ① A function block is declared with the name `distance` which takes a single argument called `point`
- ② We compute the distance formula for a point with `x` and `y` coordinates. The `return` value makes explicit what value the function will output.

`distance` (generic function with 1 method)

Note

An alternate, simpler function syntax for `distance` would be:

```
distance(point) = sqrt(point.x^2 + point.y^2)
```

However, we might at this point note a flaw in our function's definition if we think about the various `Coordinates` we defined earlier: our definition would currently only work for `Point2D`. For example, if we try a `Point3D` we will get the wrong answer:

```
distance(Point3D(1, 1, 1))
```

```
1.4142135623730951
```

The above value should be $\sqrt{3}$, or approximately 1.73205. What we need to do is define a refined distance for each type, which we'll call `dist` to distinguish from the earlier definition. We'll also use the opportunity to introduce the syntax for documenting functions in Julia, which is simply to put a string (`"..."`) or string literal (`"""..."""`) right above the definition.

5. Elements of Programming

```
"""
    dist(point)

The euclidean distance of a point from the origin.
"""
dist(p::Point2D) = sqrt(p.x^2 + p.y^2)
dist(p::Point3D) = sqrt(p.x^2 + p.y^2 + p.z^2)
dist(p::Polar2D) = p.r
```

dist (generic function with 3 methods)

Now our result will be correct:

```
dist(Point3D(1, 1, 1,))
```

1.7320508075688772

In the next chapter we'll develop some more tools, which would, for example let us define the function `dist(p::CartesianCoordinate)` and generically define the distance for all of `CartesianCoordinate`'s subtypes.



Defining Methods for Parametric Types

We learned that `Float64 <: Real` in the type hierarchy. However, note that `Tuple{Float64}` is not a subtype of `Tuple{Real}`. This is called being **invariant** in type theory... but for our purposes this just practically means that when we define a method we need to specify that we want it to apply to all subtypes.

For example, `myfunction(x::Tuple{Real})` would *not* be called if `x` was a `Tuple{Float64}` because it's not a subtype of `Tuple{Real}`. To act the way we want, would define the method with the signature of `myfunction(Tuple{<:Real})` or `myfunction{Tuple{T}} where {T<:Real}`.

5.6.3. Anonymous Functions

Anonymous functions are functions that have no name and are used in contexts where the name does not matter. The syntax is $x \rightarrow \dots \text{expression}$ with $x \dots$. As an example, say that we want to create a vector from another where each element is squared. `map` applies a function to each member of a given collection:

```
v = [4, 1, 5]
map(x → x^2, v) ①
```

① The $x \rightarrow x^2$ is the anonymous function in this example.

```
3-element Vector{Int64}:
 16
  1
 25
```

They are often used when constructing something from another value, or defining a function within optimization or solving routines.

5.6.4. Passing by Sharing

Arguments to a function in Julia are **passed-by-sharing** which means that an outside variable can be mutated from within a function. We can modify the array in the outer scope (scope discussed later in this chapter) from within the function. In this example, we modify the array that is assigned to `v` by doubling each element:

```
v = [1, 2, 3]

function double!(v)
    for i in eachindex(v)
        v[i] = 2 * v[i]
    end
end
```

5. Elements of Programming

```
double!(v)
```

```
v
```

```
3-element Vector{Int64}:
```

```
6
```

```
2
```

```
3
```

Tip

Convention in Julia is that a function that modifies its arguments has a `!` in its name and we follow this convention in `double!` above. Another example would be the built-in function `sort!` which will sort an array in-place without allocating a new array to store the sorted values.

We won't discuss all potential ways that programming languages can behave in this regard, but an alternative that one may have seen before (e.g. in Matlab) is pass-by-value where a modification to an argument only modifies the value within the scope. Here's how to replicate that in Julia by copying the value before handing it to a function. This time, `v` is not modified because we only passed a copy of the array and not the array itself:

```
v = [1, 2, 3]
```

```
double!(copy(v))
```

```
v
```

```
3-element Vector{Int64}:
```

```
1
```

```
2
```

```
3
```

5.6.5. Broadcasting

Looking at the prior definition of `dist`, what if we wanted to compute the squared distance from the origin for a set of points?

If those points are stored in an array, we can **broadcast** functions to all members of a collection at the same time. This is accomplished using the **dot-syntax** as follows:

```
points = [Point2D(1, 2), Point2D(3, 4), Point2D(6, 7)]
dist.(points) .^ 2
```

```
3-element Vector{Float64}:
 5.000000000000001
25.0
85.0
```

Let's unpack that a bit more:

1. The `.` in `dist.(points)` tells Julia to apply the function `dist` to each element in `points`.
2. The `.` in `.^` tells Julia to square each values as well

Why broadcasting is useful:

1. Without needing any redefinition of functions we were able to transform the function `dist` and exponentiation (`^`) to work on a collection of data. This means that we can keep our code simpler and easier to reason about (operating on individual things is easier than adding logic to handle collections of things).
2. When multiple broadcasted operations are joined together, Julia can **fuse** the operations so that each operation is performed at the same time instead of each step sequentially. That is, if the operation were not fused, the computer would first calculate `dist` for each point, and then apply the square on the collection of distances. When it's fused, the operations can happen at the same time without creating an interim set of values.

i Note

For readers coming from numpy-flavored Python or R, broadcasting is a way that can feel familiar to the array-oriented behavior of those two languages. Once you feel comfortable with Julia in general, you may find yourself

relaxing and relying less on array-oriented design and instead picking whichever iteration paradigm feels most natural for the problem at hand: loops or broadcasting over arrays.

5.6.5.1. Broadcasting Rules

What happens if one of the collections is not the same size as the others? When broadcasting, singleton dimensions (i.e. the 1 in 1xN, “1-by-N”, dimensions) will be expanded automatically when it makes sense. For example, if you have a single element and a one dimensional array, the single element will be expanded in the function call without using any additional memory (if that dimension matches one of the dimensions of the other array).

The rules with an MxN and a PxQ array:

- either (M and P) or (N and Q) need to be the same, *and*
- one of the non-matching dimensions needs to be 1

Some examples might clarify. This 1x1 element is being combined with a 4x1, so there is a compatible dimension (N and Q match, M is 1):

```
2 .^ [0, 1, 2, 3]
```

4-element Vector{Int64}:

```
1  
2  
4  
8
```

Here, this 1x3 works with the 2x3 (N and Q match, M is 1)

```
[1 2 3] .+ [1 2 3; 4 5 6]
```

5.6. Functions

```
2x3 Matrix{Int64}:
```

```
 2  4  6
 5  7  9
```

This 3x1 isn't compatible with this 2x3 array (neither M and P nor N and Q match)

```
[1, 2, 3] .+ [1 2 3; 4 5 6]
```

LoadError: DimensionMismatch: arrays could not be broadcast to a common size; got a dimension with lengths 3 and 2

This 2x4 isn't compatible with the 2x3 (M and P match, but N nor Q is 1):

```
[1 2; 3 4] .+ [1 2 3; 4 5 6]
```

LoadError: DimensionMismatch: arrays could not be broadcast to a common size; got a dimension with lengths 2 and 3

5.6.5.2. Not Broadcasting

What if you do not want the array to be used element-wise when broadcasting? Then you can wrap the array in a `Ref`, which is used in broadcasting to make the array be treated like a scalar. In the example below, `in(needle, haystack)` searches a collection (`haystack`) for an item (`needle`) and returns `true` or `false` if the item is in the collection:

```
in(4, [1 2 3; 4 5 6])
```

true

What if we had an array of things (“needles”) that we wanted to search for? By default, broadcasting would effectively split the array up into collections of individual elements to search:

```
in.([1, 9], [1 2 3; 4 5 6])
```

5. Elements of Programming

2×3 BitMatrix:

```
1 0 0
0 0 0
```

Effectively, the result above is the result of this broadcasted result:

```
in(1, [1,2,3]) # the first row of the above result
in(9, [4,5,6])
```

If we were expecting Julia to return [1,0] (that the first needle is in the haystack but the second needle is not), then we need to tell Julia not to broadcast along the second array with Ref:

```
in.([1, 9], Ref([1 2 3; 4 5 6]))
```

2-element BitVector:

```
1
0
```

5.6.6. First Class Nature

Functions in many languages including Julia are **first class** which means that functions can be assigned and moved around like data variables.

In this example, we have a general approach to calculate the error of a modeled result compared to a known truth. In this context, there are different ways to measure error of the modeled result and we can simplify the implementation of `loss` by keeping the different kinds of error defined separately. Then, we can assign a function to a variable and use it as an argument to another function:

```
function square_error(guess, correct)
    (correct - guess)^2
end

function abs_error(guess, correct)
    abs(correct - guess)
end
```

```

end

# obs meaning "observations"
function loss(modeled_obs,
  actual_obs,
  loss_function
)
  sum(
    loss_function.(modeled_obs, actual_obs)
  )
end

let
  a = loss([1, 5, 11], [1, 4, 9], square_error)
  b = loss([1, 5, 11], [1, 4, 9], abs_error)
  a, b
end

```

- ① `loss_function` is a variable that will refer to a function instead of data.
- ② Using a `let` block here is good practice to not have temporary variables `a` and `b` scattered around our workspace.
- ③ Using a function as an argument to another function is an example of functions being treated as “first class”.

(5, 3)

5.7. Scope

In projects of even modest complexity, it can be challenging to come up with unique identifiers for different functions or variables. **Scope** refers to the bounds for which an identifier is available. We will often talk about the **local scope** that’s inside some expression that creates a narrowly defined scope (such as a function or `let` or `module` block) or the **global scope** which is the top level scope that contains everything else inside of it. Here are a few examples to demonstrate scope.

5. Elements of Programming

```
i = 1 #<1>
let #<2>
  j = 3 #<3>
  i + j
end
```

- ① `i` is defined in the global scope and would be available to other inner scopes.
- ② The `let ... end` block creates a local scope which inherits the defined global scope definitions.
- ③ `j` is only defined in the local scope created by the `let` block.

4

In fact, if we try to use `j` outside of the scope defined above we will get an error:

```
j
```

LoadError: UndefVarError: `j` not defined

Here is an example with functions:

```
x = 2
base = 10
foo() = base^x #<1>
foo(x) = base^x #<2>
foo(x, base) = base^x #<3>

foo(), foo(4), foo(4, 4)
```

- ① Both `base` and `x` are inherited from the global scope.
- ② `x` is based on the local scope from the function's arguments and `base` is inherited from the global scope.
- ③ Both `base` and `x` are defined in the local scope via the function's arguments.

(100, 10000, 256)

In Julia, it's always best to explicitly pass arguments to functions rather than relying on them coming from an inherited scope. This is more straight-forward and easier to reason about and it also allows Julia to optimize the function to run faster because all relevant variables coming from outside the function are defined at the function's entry point (the arguments).

5.7.1. Modules and Namespaces

Modules are ways to encapsulate related functionality together. Another benefit is that the variables inside the module don't "pollute" the **namespace** of your current scope. Here's an example:

```
module Shape #<1>

    struct Triangle{T}
        base::T
        height::T
    end

    function area(t::Triangle)           ②
        return 1/2 * t.base * t.height
    end
end

t = Shape.Triangle(4,2)                 ③
area = Shape.area(t)                    ④
```

- ① `module` defines an encapsulated block of code which is anchored to the namespace `Shape`
- ② Here, `area` a *function* defined within the `Shape` module.
- ③ Outside of `Shape` module, we can access the definitions inside via the `Module.identifier` syntax.
- ④ Here, `area` is a *variable* in our global scope that *does not* conflict with the `area` defined within the `Shape` module. If `Shape.area` were not within a module then when we said `area = ...` we would have reassigned `area` to no longer refer to the function and instead would refer to the area of our triangle.

5. *Elements of Programming*

4.0

i Note

Summarizing related terminology:

- A **module** is a block of code such as `module MySimulation ... end`
- A **package** is a module that has a specific set of files and associated metadata. Essentially, it's a module with a `Project.toml` file that has a name and unique identifier listed, and a file in a `src/` directory called `MySimulation.jl`
 - **Library** is just another name for a package, and the most common context this comes up is when talking about the packages that are bundled with Julia itself called the **standard library** (`stdlib`).

6. Patterns of Abstraction

“Simple things should be simple, complex things should be possible.” — Alan Kay (1970s)

6.1. In this section

We extend the building blocks from the prior section and talk about how to combine them into more abstract patterns to simplify the design of our programs. Data driven design, object oriented design versus composition, multiple dispatch, and interfaces.

6.2. Introduction

Abstraction is a selective ignorance—focusing on the aspects of the problem that are relevant, and ignoring the others. At different times we are interested in different **ladder of abstraction**: sometimes we are interested in the small details, but other times we are interested in understanding the behavior of systems at a higher level.

Say we are an insurance company with a portfolio of fixed income assets supporting long term insurance liabilities. We might delineate different levels of abstraction like so:

Table 6.1.: An example of the different levels of abstraction when thinking about modeling an insurance company’s assets and liabilities.

	Item
More Abstract	Sensitivity of an entire company’s solvency position



Figure 6.1.: Think about moving up and down a ladder of abstraction when analyzing a problem.

6. Patterns of Abstraction

	Item
	Sensitivity of a portfolio of assets
	Behavior over time of an individual contract
More granular	Mechanics of an individual bond or insurance policy

At different times, we are often interested in different aspects of a problem. In general, you start to be able to obtain more insights and a greater understanding of the system when you move up the ladder of abstraction.

In fact, a lot of designing a model is essentially trying to figure out where to put the right abstractions. What is the right level of detail to model this in and what is the right level of detail to expose to other systems?

Let us also distinguish between **vertical abstraction**, as described above, and **horizontal abstraction** which will refer to encapsulating different properties, or mechanics of components of model that effectively exist on the same level of vertical abstraction. For example, both asset and liability mechanics sit at the most granular level in Table 6.1, But it may make sense in our model to separate the data and behavior from each other. If we were to do that, that would be an example of creating horizontal abstraction in service of our overall modeling goals.

6.3. Interfaces

Interfaces are the boundary between different encapsulated abstractions. Financial model this might mean that there is an interface for bonds, or there is an interface for interest-rate swaps. There may be a different interface for calculating risk metrics or visualizing the results.

Financial model this might mean that there is an interface for bonds, or there is an interface for interest-rate swaps. There may be a different interface for calculating risk metrics or visualizing the results. A better system design will separate the concern of visualizing output from the mechanics of a fixed income contract. This is what it means to put boundaries on different parts of a models logic.

One of the easiest places to see this is with the available open source packages. There are packages available for visualizations, data frames, file, storage, statistical analysis, etc. for many of these it's easy to see where the natural boundary lies, However, it's often difficult to find where to draw lines within financial models. For example, should bonds and interest-rate swaps be in separate packages? Or both part of a broader fixed income package? This is where much of the art and domain expertise of the financial professional comes to bear in modeling. There would be no way for a pure software engineer to think about the right design for the system without understanding how underlying components share, similarities or differences and how those components interact.

6.3.1. Conceptual Strategies

Let's consider some strategies that you could think about when deciding where to draw different boundaries inside the model.

6.3.1.1. Behavior-Oriented

This strategies is to effectively group together components with a model that behaves similarly. So, in our example of bonds and interest-rate swaps fundamentally, they share many characteristics and are used in very similar ways within a model. Therefore, it might make sense to group them together when developing a model.

6.3.1.2. Domain Expertise

It may be that components of the model require sufficient expertise that different persons or groups are involved in the development. This may warrant separating a models design, So that different groups contributing to the model can focus on any more narrow aspect, Regardless of inherent similarity of components. For example, at a higher vertical level of obstruction, financial derivatives may fall under similar grouping, but sufficient differences exist for equity credit or foreign exchange

6. Patterns of Abstraction

derivatives that the model should separate those three asset classes for development purposes.

6.3.1.3. Composability versus All-in-One

For some model design goals, it may be warranted to attempt to bundle together more functionality instead of allowing users to compose a functionality that comes from different packages. For example, perhaps a certain visualization of a model result is particularly useful. It is not easy to create from scratch, and virtually everyone using the model, will desire to see the model output visualized that way. Instead of relying on the user to install a separate visualization package and develop the visualization themselves, it could make sense to bundle visualization functionality with a model that is otherwise unconcerned with graphical capabilities.

In general, though it is preferred to try to loosely couple systems, you can pick and choose which components you use and that those components work well together.

6.4. Programming Interfaces and Patterns

Chapter 5 Described a number of tools that we can utilize as interfaces within our model. We use these tools that are provided by our programming language *in service of* the conceptual abstraction described above.

- Functions let us implement behavior, where we need trouble ourselves with the low level details.
- Data types provide a hierarchical structure to provide meaning to things, and to group those things together into more meaningful structures.
- Modules allow us to combine data, and or function, into a related group of concepts which can be shared in different parts of our model

We will also discuss here some **patterns** which are ways of doing things that seem to appear repeatedly and specific design

6.4. Programming Interfaces and Patterns

choices have proven to work well in the past and should be considered when similar conditions arise in the future.

Let's develop a simplified system to value simple fixed income assets in order to illustrate some patterns. Inside a module called `Asset`, we'll define a short hierarchy of types and then a function `value` with multiple methods for the relevant types.

```
module Asset

  ## Data type definitions
  abstract type AbstractAsset end #<1>

  struct Cash <: AbstractAsset
    balance::Float64
  end

  abstract type AbstractBond <: AbstractAsset end #<2>

  struct CouponBond <: AbstractBond #<2>
    par::Float64
    coupon::Float64
    tenor::Int
  end

  struct ZeroCouponBond <: AbstractBond #<2>
    par::Float64
    tenor::Int
  end

  ## Functions

  """
      value(asset,discount_rate)

  The value of an asset with the given discount rate for it's cashflows.
  """
  value(asset::Cash, r) = asset.balance
```

6. Patterns of Abstraction

```
function value(asset::AbstractBond, r) #<4>
    discount_factor = 1.0
    value = 0.0
    for t in 1:asset.tenor
        discount_factor /= (1 + r)
        value += discount_factor * cashflow(asset, t)
    end
    return value
end

function cashflow(bond::CouponBond, time)
    if time == bond.tenor
        (1 + bond.coupon) * bond.par
    else
        bond.coupon * bond.par
    end
end

function value(bond::ZeroCouponBond, r)
    return bond.par / (1 + r)^bond.tenor
end
```

- ① General convention is to name abstract types beginning with `Abstract...`
- ② We define two simple bonds: a coupon paying and a zero-coupon instrument.
- ③ `x /= y`, `x += y`, etc. are shorthand ways to write `x = x / y` or `x = x + y`
- ④ `value` is defined for `AbstractBonds` in general...
- ⑤ ... and then more specifically for `ZeroCouponBonds`. This will be explained when discussing “dispatch” below.

Here’s an example of how this would be used:

```
portfolio = [
    Asset.Cash(50.0),
    Asset.CouponBond(100.0, 0.05, 5),
    Asset.ZeroCouponBond(100.0, 5),
```



```
]
Asset.value.(portfolio, 0.05)
```

```
3-element Vector{Float64}:
 50.0
 99.99999999999999
 78.35261664684589
```

i Note

In the example above, a docstring was included over `value(asset::Cash)` - but not over the others. That's okay. Julia will show docstrings for the *function* value not just individual *methods*.

There are quite a few things demonstrated here: dispatch, programming paradigms, and [!!what else?] . As each are addressed in turn, we will review how we could have designed the interface differently.

6.4.1. (Multiple) Dispatch

When a function is called, the computer has to decide which method to use. In the example above, when we want to `value` a `ZeroCouponBond`, does the `value(asset::AbstractBond, r)` or `value(bond::ZeroCouponBond, r)` version get used? **Dispatch** is the process of determining the right method to use and the rule is that *the most specific defined method gets used*. In this case, that means that even though our `ZeroCouponBond` is an `AbstractBond`, the routine that will be used is the more specific `value(bond::ZeroCouponBond, r)`.

Already, this is a powerful tool to simplify our code. Imagine the alternative of a long chain of conditional statements trying to find the right logic to use:

```
# don't do this!
function value(asset,r)
```

6. Patterns of Abstraction

```
if asset.type == "ZeroCouponBond"
  # special code for Zero coupon bonds
  # ...
elseif asset.type == "ParBond"
  # special code for Par bonds
  # ...
elseif asset.type == "AmortizingBond"
  # special code for Amortizing Bonds
  # ...
else
  # here define the generic AbstractBond logic
end
end
```

A more general concept is that of **multiple dispatch**, where the types of *all arguments* are used to determine which method to use. This is a very general paradigm, and in many ways is more extensible than traditional object oriented approaches, (more on that in the next section).

In our definition of `value` above, we used a simple scalar Interest rate to determine the rate to discount the cash flows. What if instead of a scalar interest rate value we wanted to instead pass an object that represented a term structure of interest rates? All we have to do is find a new method where the first argument is our `AbstractBonds` as already written above, but the second argument is our new type.

what if we wanted to extend from a rate to a yield curve and fixed to floating. Where does logic lie? In yield curve or in bond?)

6.4.2. Programming Paradigms

- OO vs Data Oriented Design
- Alternative designs:
 - `ZeroCouponBond(par,tenor) = CouponBond(par,0.0,tenor)`

6.5. Misc Techniques

6.5.1. Recursion

6.5.2. Iterators

7. Elements of Computer Science

“Fundamentally, computer science is a science of abstraction—creating the right model for a problem and devising the appropriate mechanizable techniques to solve it. Confronted with a problem, we must create an abstraction of that problem that can be represented and manipulated inside a computer. Through these manipulations, we try to find a solution to the original problem.” - Al Aho and Jeff Ullman (1992)

7.1. In this section

Adapting computer science concepts to work for financial professionals. Concepts like computability, computational complexity, the language of algorithms and problem solving, looking for and using patterns, and adopting digital-first practices to automate the boring parts of the job.

7.2. Computer Science for Financial Professionals

Computer science as a term can be a bit misleading because of the overwhelming association with the physical desktop or laptop machines that we call “computers”. The discipline of computer science is much richer than consumer electronics: at its core, computer science concerns itself with areas of research and answering tough questions:

7. Elements of Computer Science

- **Algorithms and Optimization.** How can a problem be solved efficiently? How can that problem be solved *at all*? Given constraints, how can one find an optimal solution?
- **Information Theory.** Given limited data, what *can* be known or inferred from it?
- **Theory of Computation.** What sorts of questions are even answerable? Is an answer easy to computer or will resolving it require more resources than the entire known universe? Will a computation ever stop calculating?
- **Data Structures.** How to encode, store, and use data? How does that data relate to each other and what are the trade-offs between different representations of that data?

For a reader in the twenty-first century we hope that it's patently obvious how impactful the *applied* computer science has been as an end-user of the internet, artificial intelligence, computational photography, safety control systems, etc., etc. have been to our lives. It is a testament to the utility of being able to harness some of the ideas of this science is. Many of the most impactful advances occur at the boundary between two disciplines. It's here in this chapter that we desire to bring together the financial discipline together with computer science and to provide the financial practitioner with the language and concepts to leverage some of computer science's most relevant ideas.

7.3. Algorithms

7.3.1. Computational Complexity

7.4. Data Structures

7.5. Information Theory

7.5.1. Signal vs Noise

7.6. Formal Verification

7.7. The Discipline of Software Engineering

7.7.1. Patterns

8. Hardware and It's Implications

8.1. In this section

A discussion of why a cursory understanding of modern computing hardware and architecture is important for making the right design decisions within a modeling context. Stack vs heap allocations, pointers, and bit types. A discussion of parallelism and the different kinds of parallelism.

9. Applying Software Engineering Principles

“Programs must be written for people to read,
and only incidentally for machines to execute.” —
Harold Abelson and Gerald Jay Sussman (1984)

9.1. In this section

We describe modern software engineering practices such as testing, documentation, and pipelines which can be utilized by the financial professional to make their own work more robust and automated.

Part III.

Computational Thinking in an Actuarial and Financial Context

10. Modeling

10.1. In This Chapter

We discuss how to approach a problem and identify the key attributes to include in the model, what are the inherent trade-offs with different approaches, and how to work with data that feeds your model.

10.2. Parsimony

11. Optimization

11.1. In This Chapter

Optimization as root finding or minimization/maximization of defined objectives. Differentiable programming and the benefits to optimization problems. Model fitting as an optimization problem.

12. Sensitivity Analysis

12.1. In This Chapter

Different approaches to understanding the sensitivity of a model to changes in its inputs: derivatives, finite differences, global sensitivity analysis approaches, and statistical approaches.

13. Stochastic Modeling

The Monte Carlo Method: (i) A last resort when doing numerical integration, and (ii) a way of wastefully using computer time. - Malvin H. Kalos²¹ (c. 1960)

²¹ Kalos was a pioneer in Monte Carlo techniques, quoted via https://doi.org/10.1007/978-3-540-74686-7_3

14. Visualizations

14.1. In This Chapter

The evolved brain and pattern recognition, recommended principles for looking at data, and avoiding common mistakes. Exploratory visualization versus visualizations intended for an audience.

15. Matrices and Their Uses

15.1. In This Chapter

Matrices and their myriad uses: reframing problems through the eyes of linear algebra, an intuitive refreshing on applicable maths, and recurring patterns of matrix operations in financial modeling.

16. Learning from Data

16.1. In this chapter

Using data to inform a model: fitting parameters, forecasting, and fundamental limitations on prediction. Also covered are elements of practical review such as static and dynamic validations, and implied rate analysis.

Part IV.

Applications in Practice

17. Stochastic Mortality Projections

[Drafting note: taken from a tutorial on JuliaActuary.org. Needs to be revised with more exposition.]

17.1. In This Chapter

A term life insurance policy is used to illustrate: selecting key model features, design tradeoffs between a few different approaches, and a discussion of the performance impacts of the different approaches to parallelism.

17.2. Setup

```
using CSV, DataFrames
using MortalityTables, ActuaryUtilities
using Dates
using ThreadsX
using BenchmarkTools
using Random
using CairoMakie
```

Define a datatype. Not strictly necessary, but will make extending the program with more functions easier.

Type annotations are optional, but providing them is able to coerce the values to be all plain bits (i.e. simple, non-referenced values like arrays are) when the type is constructed. This makes

17. Stochastic Mortality Projections

the whole data be stored in the stack and is an example of data-oriented design. It's much slower without the type annotations (~0.5 million policies per second, ~50x slower).

```
@enum Sex Female = 1 Male = 2
@enum Risk Standard = 1 Preferred = 2

struct Policy
  id::Int
  sex::Sex
  benefit_base::Float64
  COLA::Float64
  mode::Int
  issue_date::Date
  issue_age::Int
  risk::Risk
end
```

17.3. The Data

```
sample_csv_data =
  IOBuffer(
    raw"id,sex,benefit_base,COLA,mode,issue_date,issue_age,risk
    1,M,100000.0,0.03,12,1999-12-05,30,Std
    2,F,200000.0,0.03,12,1999-12-05,30,Pref"
  )
```

```
IOBuffer(data=UInt8[...], readable=true, writable=false, seekable=true, ap
```

```
policies = let

  # read CSV directly into a dataframe
  # df = CSV.read("sample_inforce.csv",DataFrame) # use local string
  df = CSV.read(sample_csv_data, DataFrame)

  # map over each row and construct an array of Policy objects
  map(eachrow(df)) do row
```



```

    Policy(
      row.id,
      row.sex == "M" ? Male : Female,
      row.benefit_base,
      row.COLA,
      row.mode,
      row.issue_date,
      row.issue_age,
      row.risk == "Std" ? Standard : Preferred,
    )
  end
end

```

2-element Vector{Policy}:

```

Policy(1, Male, 100000.0, 0.03, 12, Date("1999-12-05"), 30, Standard)
Policy(2, Female, 200000.0, 0.03, 12, Date("1999-12-05"), 30, Preferred)

```

Define what mortality gets used:

```

mort = Dict(
  Male => MortalityTables.table(988).ultimate,
  Female => MortalityTables.table(992).ultimate,
)

function mortality(pol::Policy, params)
  return params.mortality[pol.sex]
end

```

mortality (generic function with 1 method)

This defines the core logic of the policy projection and will write the results to the given out container (here, a named tuple of arrays).

This is using a threaded approach where it could be operating on any of the computer's available threads, thus achieving

17. Stochastic Mortality Projections

thread-based parallelism - as opposed to multi-processor (multi-machine) or GPU-based computation, which requires formulating the problem a bit differently (array/matrix based). For the scale of computation here, I think I'd apply this model of parallelism.

```
function pol_project!(out, policy, params)
    # some starting values for the given policy
    dur = duration(policy.issue_date, params.val_date)
    start_age = policy.issue_age + dur - 1
    COLA_factor = (1 + policy.COLA)
    cur_benefit = policy.benefit_base * COLA_factor^(dur - 1)

    # get the right mortality vector
    qs = mortality(policy, params)

    # grab the current thread's id to write to results container with
    tid = Threads.threadid()

    ω = lastindex(qs)

    # inbounds turns off bounds-checking, which makes hot loops faster
    @inbounds for t in 1:min(params.proj_length, ω - start_age)

        q = qs[start_age+t] # get current mortality

        if (rand() < q)
            return # if dead then just return and don't increment the
        else
            # pay benefit, add a life to the output count, and increment
            out.benefits[t, tid] += cur_benefit
            out.lives[t, tid] += 1
            cur_benefit *= COLA_factor
        end
    end
end
```

pol_project! (generic function with 1 method)

Parameters for our projection:

17.4. Running the projection

```
params = (  
    val_date=Date(2021, 12, 31),  
    proj_length=100,  
    mortality=mort,  
)
```

```
(val_date = Date("2021-12-31"), proj_length = 100, mortality = Dict{Sex, OffsetArrays.OffsetVector{Float64, V
```

Check the number of threads we're using:

```
Threads.nthreads()
```

```
1
```

```
function project(policies, params)  
    threads = Threads.nthreads()  
    benefits = zeros(params.proj_length, threads)  
    lives = zeros{Int, params.proj_length, threads}  
    out = (; benefits, lives)  
    ThreadsX.foreach(policies) do pol  
        pol_project!(out, pol, params)  
    end  
    map(x → vec(reduce(+, x, dims=2)), out)  
end
```

project (generic function with 1 method)

17.4. Running the projection

Example of a single projection:

```
project(repeat(policies, 100_000), params)
```

```
(benefits = [5.627193330102836e10, 5.670982424536237e10, 5.706276664169161e10, 5.739882816358932e10, 5.7618
```

17. Stochastic Mortality Projections

17.4.1. Stochastic Projection

Loop through and calculate the results n times (this is only running the two policies in the sample data” n times).

```
function stochastic_proj(policies, params, n)

    ThreadsX.map(1:n) do i
        project(policies, params)
    end
end
```

stochastic_proj (generic function with 1 method)

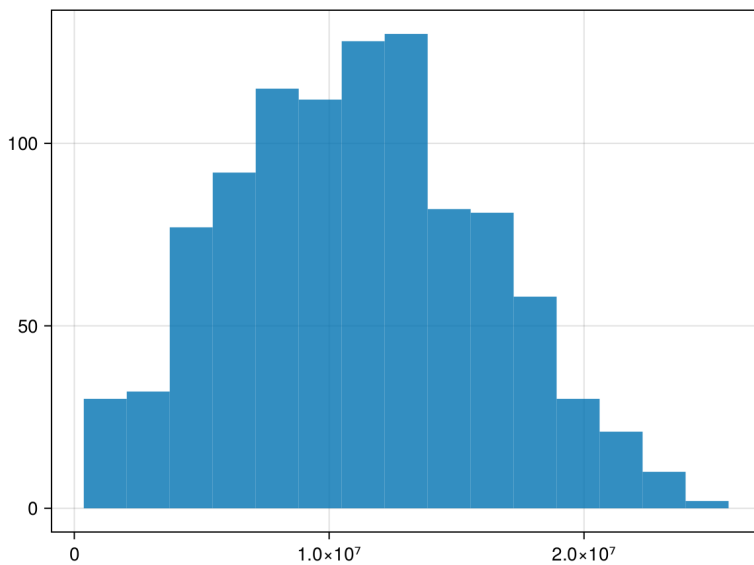
```
stoch = stochastic_proj(policies, params, 1000)
```

```
1000-element Vector{@NamedTuple{benefits::Vector{Float64}, lives::Vector{
(benefits = [574831.0226582347, 592075.9533379817, 609838.2319381211, 628
(benefits = [574831.0226582347, 592075.9533379817, 609838.2319381211, 628
(benefits = [574831.0226582347, 592075.9533379817, 609838.2319381211, 628
(benefits = [383220.68177215644, 394717.3022253211, 406558.82129208074, 4
(benefits = [574831.0226582347, 592075.9533379817, 609838.2319381211, 628
(benefits = [574831.0226582347, 394717.3022253211, 406558.82129208074, 41
(benefits = [574831.0226582347, 592075.9533379817, 609838.2319381211, 628
(benefits = [574831.0226582347, 592075.9533379817, 609838.2319381211, 628
(benefits = [574831.0226582347, 592075.9533379817, 609838.2319381211, 628
(benefits = [574831.0226582347, 592075.9533379817, 609838.2319381211, 628
(benefits = [574831.0226582347, 592075.9533379817, 609838.2319381211, 628
(benefits = [574831.0226582347, 592075.9533379817, 609838.2319381211, 628
(benefits = [383220.68177215644, 394717.3022253211, 406558.82129208074, 4
(benefits = [574831.0226582347, 592075.9533379817, 609838.2319381211, 628
:
(benefits = [574831.0226582347, 592075.9533379817, 609838.2319381211, 628
(benefits = [574831.0226582347, 592075.9533379817, 609838.2319381211, 628
(benefits = [574831.0226582347, 592075.9533379817, 609838.2319381211, 628
(benefits = [574831.0226582347, 592075.9533379817, 609838.2319381211, 628
(benefits = [191610.34088607822, 197358.65111266056, 203279.41064604037,
(benefits = [574831.0226582347, 592075.9533379817, 609838.2319381211, 628
(benefits = [574831.0226582347, 592075.9533379817, 609838.2319381211, 628
(benefits = [574831.0226582347, 394717.3022253211, 406558.82129208074, 41
```

17.5. Benchmarking

```
(benefits = [574831.0226582347, 592075.9533379817, 609838.2319381211, 628133.3788962648, 646977.3802631528,  
(benefits = [574831.0226582347, 592075.9533379817, 406558.82129208074, 418755.5859308432, 431318.2535087685,  
(benefits = [574831.0226582347, 394717.3022253211, 406558.82129208074, 418755.5859308432, 431318.2535087685,  
(benefits = [574831.0226582347, 592075.9533379817, 609838.2319381211, 628133.3788962648, 646977.3802631528,
```

```
let  
  v = [pv(0.03, s.benefits) for s in stoch]  
  hist(v,  
    bins=15,  
    xlabel="Present Value of Benefits",  
    ylabel="Number of scenarios")  
end
```



17.5. Benchmarking

Using a 2022 Macbook Air M2 laptop, about 30 million policies able to be stochastically projected per second:

```
policies_to_benchmark = 3_000_000  
# adjust the 'repeat' depending on how many policies are already in the array  
# to match the target number for the benchmark
```

17. Stochastic Mortality Projections

```
n = policies_to_benchmark ÷ length(policies)
```

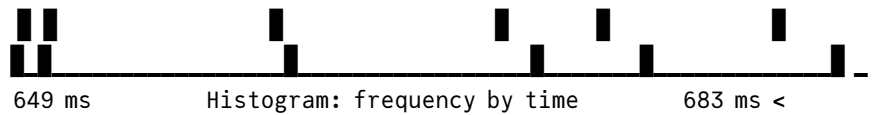
```
@benchmark project(p, r) setup = (p = repeat($policies, $n); r = $par
```

BenchmarkTools.Trial: 6 samples with 1 evaluation.

Range (min ... max): 649.393 ms ... 682.532 ms | GC (min ... max): 0.00% ... 0.00%

Time (median): 665.489 ms | GC (median): 0.00%

Time (mean ± σ): 664.738 ms ± 13.340 ms | GC (mean ± σ): 0.00% ± 0.00%



Memory estimate: 9.28 KiB, allocs estimate: 79.

17.6. Further Optimization

In no particular order:

- the RNG could be made faster: <https://bkamins.github.io/julialang/2020/01/04/rng.html>
- Could make the stochastic set distributed, but at the current speed the overhead of distributed computing is probably more time than it would save. Same thing with GPU projections
- ...

18. Scenario Generation

18.1. In This Chapter

How to generate synthetic data for your model using sub-models, with applications to economic scenario generation and portfolio composition.

19. Similarity Analysis

19.1. In This Chapter

Given a set of interest, understanding the relative similarity (or not) of features of interest is useful in classification and data compression techniques.

20. Portfolio Optimization

20.1. In This Chapter

Optimization in a portfolio context with examples of asset selection under different constraints and objectives.

21. Other Useful Techniques

21.1. In this chapter

Other useful techniques are surveyed, such as: memoization to avoid repeated computations, psuedo-monte carlo, creating a model office, and tips on modeling a complete balance sheet.

21.2. Taking things to the Extreme

Consider what happens if something is taken to an extreme. For example, what happens in the model if we input negative rates? Where should negative rates be allowed and can the model handle them?

21.3. Range Bounding

Sometimes you just need to know that an outcome is within a certain range - if you can develop a “high” and “low” estimate by making assumptions that you know are outside of feasible ranges, then you can determine whether something is reasonable or within tolerances.

To take an example from the pages of interview questions: say you need to determine if a mortgaged property’s value is greater than the amount of the outstanding loan (say \$100,000). You don’t have an appraisal, but know that it’s in reasonable condition and that (1) a comparable house with many more issues sold for \$100 per square foot. You also don’t know the square footage of the house, but know from the number of rooms and layout that it must be at least 1000 square feet. Therefore you know that the value should at least be greater than:

21. *Other Useful Techniques*

$$\frac{\$100}{\text{sq. ft}} \times 1000\text{sq. ft} = \$100,000$$

We'd then conclude that the value of the house very likely exceeds the outstanding balance of the loan and resolves our query without complex modeling or expensive appraisals.

Part V.

Appendices

22. Set up Julia and the Computing Environment

22.1. Installation

Julia is open source and can be downloaded from [JuliaLang.org](https://julialang.org) and is available for all major operating systems. After you download and install, then you have Julia installed and can access the **REPL**, or Read-Eval-Print-Loop, which can run complete programs or function as powerful day-to-day calculator. However, many people find it more comfortable to work in a text editor or **IDE** (Integrated Development Environment).

If you are looking for managed installations with a curated set of packages for use within an organization, there are ways to self-host package repositories and otherwise administratively manage packages. Julia Computing offers managed support with enterprise solutions, including push-button cloud compute capabilities.

22.2. Package Management

Julia comes with `Pkg`, a built-in package manager. With it, you can install packages, pin certain versions, recreate environments with the same set of dependencies, and upgrade/remove/develop packages easily. It's one of the things that *just works* and makes Julia stand out versus alternative languages that don't have a de-facto way of managing or installing packages.

Package installation is accomplished interactively in the REPL or executing commands.

22. Set up Julia and the Computing Environment

- In the REPL, you can change to the Package Management Mode by hitting `]` and, e.g., add `DataFrames` `CSV` to install the two packages. Hit `[backspace]` to exit that mode in the REPL.
- The same operation without changing REPL modes would be: `using Pkg; Pkg.add(["DataFrames", "CSV"])`

Related to packages, are **environments** which are a self-contained workspaces for your code. This lets you install only packages that are relevant to the current work. It also lets you ‘remember’ the exact set of packages and versions that you used. In fact, you can share the environment with others, and it will be able to recreate the same environment as when you ran the code. This is accomplished via a `Project.toml` file, which tracks the direct dependencies you’ve added, along with details about your project like its version number. The `Manifest.toml` tracks the entire dependency tree.

Reproducibility via the environment tools above is a really key aspect that will ensure Julia code is consistent across time and users, which is important for financial controls.

22.3. Editors

Because Julia is very extensible and amenable to analysis of its own code, you can typically find plugins for whatever tool you prefer to write code in. A few examples:

22.3.1. Visual Studio Code

Visual Studio Code is a free editor from Microsoft. There’s a full-featured Julia plugin available, which will help with auto-completion, warnings, and other code hints that you might find in a dedicated editor (e.g. PyCharm or RStudio). Like those tools, you can view plots, search documentation, show datasets, debug, and manage version control.

22.3.2. Notebooks

Notebooks are typically more interactive environments than text editors - you can write code in cells and see the results side-by-side.

The most popular notebook tool is Jupyter (“Julia, Python, R”). It is widely used and fits in well with exploratory data analysis or other interactive workflows. It can be installed by adding the `IJulia.jl` package.

`Pluto.jl` is a newer tool, which adds reactivity and interactivity. It is also more amenable to version control than Jupyter notebooks because notebooks are saved as plain Julia scripts. Pluto is unique to Julia because of the language’s ability to introspect and analyze dependencies in its own code. Pluto also has built-in package/environment management, meaning that Pluto notebooks contains all the code needed to reproduce results (as long as Julia and Pluto are installed).

23. The Julia Ecosystem Today

A tour of relevant available packages as of 2023.

The Julia ecosystem favors composability and interoperability, enabled by multiple dispatch. In other words, because it's easy to automatically specialize functionality based on the type of data being used, there's much less need to bundle a lot of features within a single package.

As you'll see, Julia packages tend to be less vertically integrated because it's easier to pass data around. Counterexamples of this in Python and R:

- Numpy-compatible packages that are designed to work with a subset of numerically fast libraries in Python
- special functions in Pandas to read CSV, JSON, database connections, etc.
- The Tidyverse in R has a tightly coupled set of packages that works well together but has limitations with some other R packages

Julia is not perfect in this regard, but it's neat to see how frequently things *just work*. It's not magic, but because of Julia features outside the scope of this article it's easy for package developers (and you!) to do this.

Julia also has language-level support for documentation, so packages can follow a consistent style of help-text and have the docs be auto-generated into web pages available locally or online.

The following highlighted packages were chosen for their relevance to typical actuarial work, with a bias towards those used regularly by the authors. This is a small sampling of the over 6000 registered Julia Packages^[2]

23.0.1. Data

Julia offers a rich data ecosystem with a multitude of available packages. Perhaps at the center of the data ecosystem are `CSV.jl` and `DataFrames.jl`. `CSV.jl` is for reading and writing files text files (namely CSVs) and offers top-class read and write performance. `DataFrames.jl` is a mature package for working with dataframes, comparable to Pandas or dplyr.

Other notable packages include `ODBC.jl`, which lets you connect to any database (given you have the right drivers installed), and `Arrow.jl` which implements the Apache Arrow standard in Julia.

Worth mentioning also is `Dates`, a built-in package making date manipulation straightforward and robust.

Check out [JuliaData.org](https://juliadata.org) for more packages and information.

23.0.2. Plotting

`Plots.jl` is a meta-package providing an interface to consistently work with several plotting backends, depending if you are trying to emphasize interactivity on the web or print-quality output. You can very easily add animations or change almost any feature of a plot.

`StatsPlots.jl` extends `Plots.jl` with a focus on data visualization and compatibility with dataframes.

`Makie.jl` supports GPU-accelerated plotting and can create very rich, beautiful visualizations, but its main downside is that it has not yet been optimized to minimize the time-to-first-plot.

23.0.3. Statistics

Julia has first-class support for missing values, which follows the rules of three-valued logic so other packages don't need to do anything special to incorporate missing values.

`StatsBase.jl` and `Distributions.jl` are essentials for a range of statistics functions and probability distributions respectively.

Others include:

- `Turing.jl`, a probabilistic programming (Bayesian statistics) library, which is outstanding in its combination of clear model syntax with performance.
- `GLM.jl` for any type of linear modeling (mimicking R's `glm` functionality).
- `LsqFit.jl` for fitting data to non-linear models.
- `MultivariateStats.jl` for multivariate statistics, such as PCA.

You can find more packages and learn about them [here](#).

23.0.4. Machine Learning

`Flux`, `Gen`, `Knet`, and `MLJ` are all very popular machine learning libraries. There are also packages for `PyTorch`, `Tensorflow`, and `SciKitML` available. One advantage for users is that the Julia packages are written in Julia, so it can be easier to adapt or see what's going on in the entire stack. In contrast to this design, `PyTorch` and `Tensorflow` are built primarily with C++.

Another advantage is that the Julia libraries can use automatic differentiation to optimize on a wider range of data and functions than those built into libraries in other languages.

23.0.5. Differentiable Programming

Sensitivity testing is very common in actuarial workflows: essentially, it's understanding the change in one variable in relation to another. In other words, the derivative!

Julia has unique capabilities where almost across the entire language and ecosystem, you can take the derivative of entire functions or scripts. For example, the following is real Julia code to automatically calculate the sensitivity of the ending account value with respect to the inputs:

23. The Julia Ecosystem Today

```
julia> using Zygote

julia> function policy_av(pol)
    COIs = [0.00319, 0.00345, 0.0038, 0.00419, 0.0047, 0.00532]
    av = 0.0
    for (i,coi) in enumerate(COIs)
        av += av * pol.credit_rate
        av += pol.annual_premium
        av -= pol.face * coi
    end
    return av                # return the final account value
end

julia> pol = (annual_premium = 1000, face = 100_000, credit_rate = 0.06)

julia> policy_av(pol)        # the ending account value
4048.08

julia> policy_av'(pol)       # the derivative of the account value with
                             # respect to the parameters
(annual_premium = 6.802, face = -0.0275, credit_rate = 10972.52)
```

When executing the code above, Julia isn't just adding a small amount and calculating the finite difference. Differentiation is applied to entire programs through extensive use of basic derivatives and the chain rule. **Automatic differentiation**, has uses in optimization, machine learning, sensitivity testing, and risk analysis. You can read more about Julia's autodiff ecosystem [here](#).

23.0.6. Utilities


There are also a lot of quality-of-life packages, like `Revise.jl` which lets you edit code on the fly without needing to re-run entire scripts.

`BenchmarkTools.jl` makes it incredibly easy to benchmark your code - simply add `@benchmark` in front of what you want to test, and you will be presented with detailed statistics. For example:


```
julia> using ActuaryUtilities, BenchmarkTools

julia> @benchmark present_value(0.05,[10,10,10])

BenchmarkTools.Trial: 10000 samples with 994 evaluations.
Range (min ... max): 33.492 ns ... 829.015 ns | GC (min ... max): 0.00% ... 95.40%
Time (median): 34.708 ns | GC (median): 0.00%
Time (mean ± σ): 36.599 ns ± 33.686 ns | GC (mean ± σ): 4.40% ± 4.55%
```



33.5 ns Histogram: log(frequency) by time 45.6 ns <

Memory estimate: 112 bytes, allocs estimate: 1.

Test is a built-in package for performing testsets, while Documenter.jl will build high-quality documentation based on your inline documentation.

ClipData.jl lets you copy and paste from spreadsheets to Julia sessions.

23.0.7. Other packages

Julia is a general-purpose language, so you will find packages for web development, graphics, game development, audio production, and much more. You can explore packages (and their dependencies) at <https://juliahub.com/>.

23.0.8. Actuarial packages

Saving the best for last, the next article in the series will dive deeper into actuarial packages, such as those published by JuliaActuary for easy mortality table manipulation, common actuarial functions, financial math, and experience analysis.

References

