Brian Lee Yung Rowe

Modeling Data With Functional Programming In R

Preface

This book is about programming. Not just any programming, but programming for data science and numerical systems. This type of programming usually starts as a mathematical modeling problem that needs to be translated into computer code. With functional programming, the same reasoning used for the mathematical model can be used for the software model. This not only reduces the impedance mismatch between model and code, it also makes code easier to understand, maintain, change, and reuse. Functional programming is the conceptual force that binds the these two models together. Most books that cover numerical and/or quantitative methods focus primarily on the mathematics of the model. Once this model is established the computational algorithms are presented without fanfare as imperative, step-by-step, algorithms. These detailed steps are designed for machines. It is often difficult to reason about the algorithm in a way that can meaningfully leverage the properties of the mathematical model. This is a shame because mathematical models are often quite beautiful and elegant yet are transformed into ugly and cumbersome software. This unfortunate outcome is exacerbated by the real world, which is messy: data does not always behave as desired; data sources change; computational power is not always as great as we wish; reporting and validation workflows complicate model implementations. Most theoretical books ignore the practical aspects of working in the field. The need for a theoretical and structured bridge from the quantitative methods to programming has grown as data science and the computational sciences become more pervasive.

The goal is to re-establish the intimate relationship between mathematics and computer science. By the end of the book, readers will be able to write computer programs that clearly convey the underlying mathematical model, while being easy to modify and maintain. This is possible in R by leveraging the functional programming features built into the language. Functional programming has a long history within academia and its popularity in industry has recently been rising. Two prominent reasons for this upswing are the growing computational needs commensurate with large data sets and the data-centric computational model that is consistent with the analytical model. The superior modularity of functional programs isolates data management from model development, which keeps the model clean and reduces development overhead. Derived from a formal mathematical language, functional programs can be reasoned about with a level of clarity and certainty

not possible in other programming paradigms. Hence the same level of rigor applied to the quantitative model can be applied to the software model.

Divided into three parts, foundations are first established that uses the world of mathematics as an introduction to functional programming concepts (Chapter 2). Topics discussed include basic concepts in set theory, statistics, linear algebra, and calculus. As core tools for data scientists, this material should be accessible to most practitioners, graduate students, and even upper class undergraduates. The idea is to show you that you already know most of the concepts used in functional programming. Writing code using functional programming extends this knowledge to operations beyond the mathematical model. This chapter also shows counter examples using other programming styles, some in other languages. The point isn't to necessarily criticize other implementation approaches, but rather make the differences in styles tangible to the reader. After establishing some initial familiarity, Chapter 3 dives into functions, detailing the various properties and behaviors they have. Some important features include higher-order functions, first-class functions, and closures. This chapter gives you a formal vocabulary for talking about functional programs in R. This distinction is important as other functional languages have plenty more terms and theory ignored in this book. Again, the goal is how to leverage functional programming ideas to be a better data scientist. Finally, Chapter 4 reviews the various packages in R that provide functionality related to functional programming. These packages include some built-in implementations, paradigms for parallel computing, a subset of the so-called Hadleyverse, and my own `lambda.r` package, which offers a more comprehensive approach to writing functional programs.

While the first part provides a working overview of functional programming in R, Part II takes it a step further. This part is for readers that want to exploit functional programming principles to their fullest. Many topics in Part I reference deeper discussions in Part II. This canon begins by exploring the nature of vectorization in Chapter 5. Often taken for granted, we look at how colloquial vectorization conflates a few concepts. This chapter unravels the concepts, showing you what can be done and what to expect with each type of vectorization. Three primary higher-order functions *map*, *fold*, and *filter* follow. Chapter 6 shows how the concept of *map* appears throughout R, particularly in the `apply` family of functions. I show how to reason about data structures with respect to the ordinals (or index) of the data structure. This approach can simplify code by enabling the separation of data structures, so long as an explicit ordinal mapping exists. While *fold* is more fundamental than *map*, its use is less frequent in R. Discussed in Chapter 7, *fold* provides a common structure for repeated function application. Optimization and many iterative methods, as well as stochastic systems make heavy use of repeated function application, but this is usually implemented as a loop. With *fold*, the same function used to implement a single step can be used for multiple steps, reducing code complexity and making it easier to test. The final function of the canon is *filter*, which creates subsets using a predicate. This concept is so

integral to R that the notation is deeply integrated into the language. Chapter 8 shows how the native R syntax is tied to this higher-order function, which can be useful when data structures are more complex. Understanding this connection also simplifies porting code to or from R when the other language doesn't have native syntax for these operations.

Programming languages can't do much without data structures. Chapter 9 shows how to use native R data structures to implement numerous algorithms as well as emulate other data structures. For example, lists can be used to emulate trees, while environments can emulate hash tables.

The last part focuses on applications and advanced topics. Part III can act as a reference for implementing various algorithms in a functional style. This provides readers with tangible examples of using functional programming concepts for algorithm development. This part begins by proposing a simple model development pipeline in Chapter 11. The intention is to provide some reusable structure that can be tailored to each reader's needs. For example, the process of backtesting is often implemented in reverse. Loops become integral to the implementation, which makes it difficult to test individual update steps in an algorithm. This chapter also shows some basic organizational tricks to simplify model development. A handful of machine learning models, such as random forest, are also presented in Chapter 11. Remarkably, many of these algorithms can be implemented in less than 30 lines fo code when using functional programming techniques. Optimization methods, such as Newton's method, linear programming, and dynamic follow in Chapter 13. These methods are usually iterative and therefore benefit from functional programming. State-based systems are explored in Chapter 12. Ranging from iterative function systems to context-free grammars, state is central to many models and simlations. This chapter shows how functional programming can simplify these models. This part also discusses two case studies (Chapter 14 and Chapter 15) for implementing more complete systems. In essence, Part III shows the reader how to apply the concepts presented in the book to real-world problems.

Each chapter presents a number of exercises to test what you've learned. By the end of the book, you will know how to leverage functional programming to improve your life as a data scientist. Not only will you be able to quickly and easily implement mathematical ideas, you'll be able to incrementally change your exploratory model into a production model, possibly scaling to multiple cores for big data. Doing so also facilitates repeatable research since others can review and modify your work more easily.

Contents

I Foundation 1

1 Introduction 3
- 1.1 The model development workflow . . . 4
- 1.2 Language paradigms . . . 6
- 1.3 Imperative versus declarative algorithms . . . 7
- 1.4 Elements of a functional programming language . . . 9
- 1.5 Review of object-oriented programming . . . 13
- 1.6 Syntax, notation, and style . . . 17
- 1.7 Examples and package dependencies . . . 18

2 The functional programming language called mathematics 19
- 2.1 The declarative nature of set theory . . . 20
- 2.2 Statistics . . . 21
  - 2.2.1 The mean and imperative loops . . . 21
  - 2.2.2 Covariance and declarative functions . . . 23
  - 2.2.3 Symbolic conveniences in linear regression . . . 26
  - 2.2.4 Probability distributions and taxonomies . . . 27
- 2.3 Linear algebra . . . 31
  - 2.3.1 Dispatching the dot product . . . 32
  - 2.3.2 Matrix multiplication as object-oriented programming 34
  - 2.3.3 Matrix factorization and collaboration . . . 36
  - 2.3.4 The determinant and recursion . . . 38
- 2.4 Calculus . . . 39
  - 2.4.1 Transforms as higher-order functions . . . 39
  - 2.4.2 Numerical integration and first-class functions . . . 42
- 2.5 Summary . . . 43
- 2.6 Exercises . . . 44

3 Functions as a lingua franca 45
- 3.1 Vectorization . . . 48
- 3.2 First-Class Functions . . . 50
- 3.3 Closures . . . 53
- 3.4 Functions as factories . . . 57
- 3.5 Mediating iteration . . . 58
- 3.6 Interface compatibility . . . 61

3.7 Codifying behavioral changes 63
3.8 Inversion of control via callbacks 65
3.9 State representation 68
3.10 Mutable state 72
3.11 Summary 75
3.12 Exercises 75

4 Alternate functional paradigms **77**
4.1 The built-in functional programming canon 78
4.2 Infix ”pipe” notation 80
4.3 The `lambda.r` syntax and type system 83
4.3.1 Pattern matching 83
4.3.2 Guard statements 86
4.3.3 Types and type constraints 88
4.3.4 Type variables 94
4.4 The MapReduce paradigm 96
4.5 The split-apply-combine paradigm 103
4.6 The `tidyverse` canon 107
4.7 Summary 109
4.8 Exercises 109

II The Canon **111**

5 Vector Mechanics **113**
5.1 Vectors as a polymorphic data type 115
5.1.1 Vector construction 117
5.1.2 Scalars 118
5.1.3 Atomic types 120
5.1.4 Coercion 121
5.1.5 Concatenation 122
5.2 Set theory 127
5.2.1 The empty set 127
5.2.2 Set membership 129
5.2.3 Set comprehensions and logic operations 130
5.2.4 Set complements 131
5.3 Indexing and subsequences 133
5.3.1 Named indices 134
5.3.2 Logical indexing 136
5.3.3 Ordinal mappings 138
5.3.4 Sorting 139
5.4 Recycling 139
5.5 Exercises 141

6 Map Vectorization **143**
6.1 A motivation for *map* . . . 143
6.1.1 Map implementations . . . 145
6.2 Preservation of cardinality . . . 146
6.2.1 Functions as relations . . . 146
6.2.2 Demystifying `sapply` . . . 148
6.2.3 Computing cardinality . . . 149
6.2.4 Idempotency of vectorized functions . . . 150
6.2.5 Identifying *map* operations . . . 152
6.3 Order invariance . . . 153
6.4 Function composition . . . 156
6.4.1 *Map* as a linear transform . . . 158
6.4.2 Multivariate *map* . . . 159

7 Fold vectorization **161**
7.1 A motivation for *fold* . . . 161
7.1.1 Initial values and the identity . . . 163
7.1.2 *Fold* implementations . . . 167
7.1.3 Ordinal maps . . . 169
7.1.4 Data structure preservation . . . 170
7.1.5 Identifying and correcting bad data . . . 171
7.2 Merging data frames . . . 175
7.2.1 Column-based merges . . . 176
7.2.2 Row-based merges . . . 180
7.3 Sequences, series, and closures . . . 182
7.3.1 Constructing the Maclaurin series . . . 183
7.3.2 Multiplication of power series . . . 185
7.3.3 Taylor series approximations . . . 186

8 Filter **189**
8.1 Subsetting notation . . . 190
8.2 Predicates . . . 190
8.2.1 Partitions . . . 192
8.2.2 Using filters . . . 192
8.3 Mapping to ordinals . . . 193
8.3.1 Using `which` to extract ordinals . . . 194
8.4 Context-aware data fills . . . 194
8.5 Exercises . . . 195

9 Canonical Data Structures **197**
9.1 Primitive operations . . . 198
9.1.1 The list constructor . . . 198
9.1.2 Raw element access . . . 200
9.1.3 Selecting subsets of a list . . . 202
9.1.4 Replacing elements in a list . . . 203

9.1.5 Removing elements from a list 204
9.1.6 Lists and NULLs 204
9.1.7 Concatenation 205
9.2 Comparing lists 206
9.2.1 Equality 207
9.2.2 Orderings 208
9.2.3 Metric spaces, distances, and similarity 209
9.2.4 Comparing other spaces 212
9.3 Map operations on lists 213
9.3.1 Applying multiple functions to the same data 214
9.3.2 Cardinality and null values 215
9.3.3 Hierarchical data structures 216
9.4 Fold operations on lists 219
9.4.1 Abstraction of function composition 219
9.4.2 Merging data 220
9.5 Function application with do.call 220
9.6 Emulating trees and graphs with lists 223
9.6.1 Modeling the binomial asset pricing model using trees 225
9.7 Data Frames 227
9.7.1 Concatenation in 2 dimensions 228
9.8 Map processes 229
9.9 Fold processes 229
9.10 Exercises 229

10 Advanced Functional Programming 231
10.1 Recursion 231
10.2 Fixed point combinators 233
10.3 Continuations 235
10.4 Lazy Evaluation 237
10.5 Monads 237
10.6 Computational Graphs 237
10.7 Summary 237
10.8 Exercises 237

III Application 239

11 Machine Learning Algorithms 241
11.0.1 Modeling random forest with trees 241

12 State-Based Systems 247
12.1 Using Closures for State Management 248
12.1.1 Generators 249
12.1.2 Memoization 251
12.2 Deterministic systems 253
12.2.1 Iterative function systems 254
12.2.2 Conway's Game of Life 255

12.2.3 Context-free grammars 261
12.3 Finite State Machines 267
12.4 Probabilistic Systems 274
12.4.1 Markov chains 274
12.4.2 The Chinese restaurant process 277
12.4.3 Kalman filters 282
12.5 Exercises 282

13 Optimization Methods 283
13.1 Root finding with Newton-Raphson optimization 283
13.2 The power method 284
13.3 Linear programming 284
13.4 Dynamic programming and the Bellman operator 284
13.5 Markov decision processes 284
13.6 The gradient and Hessian 284

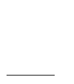
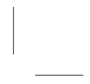
14 Case Study: The `optim` function 285

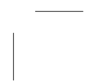
15 Case Study: Ebola Situation Report ETL 287
15.1 Standardized transformations 287
15.1.1 Data extraction 288
15.1.2 Data normalization 293
15.2 Cleaning data 300
15.2.1 Fixing syntactic and typographical errors 302
15.2.2 Identifying and filling missing data 306
15.3 Validation 310
15.3.1 Internal consistency 310
15.3.2 Spot checks 312
15.3.3 Configuration management 314

16 Epilogue 317

A A lambda calculus primer 319
A.1 Reducible expressions 320
A.2 Church numerals 323

Bibliography 327



List of Figures

1.1 An idealized model development process based on Box's role of statistics in scientific learning. 5

2.1 The *map* operation maps a function to a set of values 25
2.2 The *zip* operation constructs a single vector of tuples from a set of vectors. 26
2.3 Java class hierarchy for distributions. `Object` is the root class for all Java objects. Dashed boxes indicate abstract classes that cannot be instantiated. 28
2.4 Java class hierarchy for different statistics. In OOP, even simple functions and metrics are forced into a class hierarchy, which ultimately have no relationship with the underlying mathematical concepts. In this example, the `AbstractStorelessUnivariateStatistic` is an implementation artifact unrelated to univariate statistics. 29
2.5 Comparison of operations for two interfaces in the Apache Commons Math library. The primary difference is that types change from `int` to `double`. 30
2.6 Convolution as a computational graph 40
2.7 Regression analysis with Box-Cox transform as a computational graph. To predict a value, the model is applied to new data X' followed by an inverse Box-Cox transformation. . . . 41
2.8 Comparison of residuals before (left) and after (right) a Box-Cox transformation. New predictions on the model must have inverse transform applied to get correct value. 42

3.1 Simple functions versus a closure. Left: A standard function only has access to variables within its own scope. Any references to variables outside the function scope will be searched recursively until the global environment is reached. Right: Closures reference variables in their enclosing scope. This environment is bound to the function, even after the enclosing function has been executed. With lexical scoping, any variable not in the current scope will be recursively searched in the parent scope until the global environment is reached. 53

3.2 Relationship between the call stack frames and lexical scopes. Both are accessible from the current frame. The lexical scope assumes this code is bundled in a package. 55
3.3 Precision-recall curve for the `adult` dataset 64
3.4 The `model_pipeline` as an example of inversion of control . . 66
3.5 Simulated time series with corresponding exponential moving average (dotted line) 74

4.1 A visual depiction of a layered approach to function development. The outer layers provide convenient interfaces for each domain case, while the inner layers are pure mathematical functions. 91
4.2 The MapReduce method parallelizes data processing over multiple *map* and $fold$ stages. Data is first partitioned and then sent to multiple *map* processors. The results are collected and grouped by key. Then these keys are partitioned and sent to a set of $fold$ jobs. 97
4.3 Two approaches to applying a function to a set 104
4.4 Batting averages for each player classification 105

5.1 Partial type coercion hierarchy for concatenation 121
5.2 Panel data as the partition of a set. The partition is defined based on the value of the `group` column. 128
5.3 Convert a time into a decimal hour 140

6.1 The graph of f over a set X 144
6.3 Comparing two 'graphs' of the same function 154
6.4 Zip converts column-major data into a row-major structure. Column-major data structures give fast sequential access along column indices. A row-major structure optimizes for row access. 159

7.1 How $fold$ operates on a vector input x. The result of each application of f becomes an operand in the subsequent application 162
7.2 Iterated application of *union* over X 165
7.3 Comparing the alignment of a derived time series 168
7.4 Cumulative death values are inconsistent with daily dead totals and need to be fixed. 172
7.5 Two approaches to combining tabular data together. Adding new features to existing samples is a join, while increasing the sample for the same features is a union. 176
7.6 A set-theoretic interpretation of join operations based on table indices. A full join (not shown) is a combination of both a left and right outer join. 177
7.7 The `parse_nation` function modeled as a graph. 182
7.8 The Maclaurin series approximation of e^x about 0. 184

7.9 Approximation of $cos(x)$ about $x = 2$ 187

8.1 A predicate partitions a set into two subsets 192

9.1 Excerpt from *A Tree For Me* 211
9.2 Using a list, multiple functions can be applied to the same object via `lapply` 214
9.3 When cardinality is lost, the ordinals are also lost 216
9.4 An excerpt of a JSON structure from the `govtrack.us` API . . 218
9.5 A cartoon tree 224

12.1 A small subset of the World Cities dataset 250
12.2 The Heighway dragon in paper 254
12.3 Four iterations of the Heighway dragon 256
12.4 Final state of Game of Life 257
12.5 Two matrices showing a shift ”East” and how that encodes the state of a particular neighbor. Shifting the board matrix in each of eight directions effectively creates a tensor. The sum over the tensor coordinates is equal to the neighbor count at each cell. 259
12.6 A cartoon CFG that parses ”a man walks in the park with a dog” 262
12.7 The parse tree for ”a man walks in the park with a dog” 264
12.8 A simulated price series 268
12.9 A trading strategy represented by a FSM 269
12.10 A trading strategy represented as a set of state transitions . . . 270
12.11 Generating events based on channel location 271
12.12 Trading signals for a time series 273
12.13 A random realization of text based on The Sun Also Rises . . . 275
12.14 A realization of the Chinese restaurant process. Each point represents the table that a given patron is seated. 279
12.15 The probabilities associated with the 11th patron sitting at existing tables 1-4 or new table 5. As dictated by exchangeability, all the probabilities are the same for each congruent table permutation. 281

15.1 Ebola situation report table 288
15.2 Portion of the Liberia Ministry of Health situation report web page 289
15.3 How `xpathSApply` transforms an XML document. Boxes represent functions, and arrows represent input/output. 290
15.4 Modeling conditional blocks as trees 294
15.5 Mapping a partition to a configuration space simplifies transformations 295
15.6 Raw extract of PDF situation report. Numerous words are split across lines due to formatting in the source PDF. 296

15.7 Constructing table boundaries by shifting a vector. The first $n-1$ indices represent the start index, while the last n-1 indices are used as corresponding stop indices. 297
15.8 Parsed data.frame from unstructured text 301
15.9 Common syntax errors in Liberian situation reports 302
15.10 The control flow of a sequence of if-else blocks 304
15.11 Using `ifelse` to simplify control flow 305
15.12 Histogram of patients lost in follow up in Nimba county . . . 308
15.13 Using ordinals to map functions to columns 310
15.14 A table in the Liberia Situation Report containing cumulative counts 311
15.15 A transformation chain mapping file names to configurations 315

Listings

2.1 Naive implementation of the mean using a loop 22
2.2 A declarative implementation of the mean using `fold` 22
2.3 A loop implementation of covariance 23
2.4 Covariance using native R functions 24
2.5 An attempt at declarative notation with a loop 24
2.6 Using functional primitives to implement covariance 25
2.7 Implementation of *zip* 25
R/math_as_fp.R 25
3.1 A rudimentary model processing pipeline 48
3.2 A function to summarize model performance 50
3.3 Implementation of Winsorization 54
3.4 Adding Winsorization to the `income_pipeline` 56
3.5 Winsorization as a single function 56
3.6 Function to normalize a number of columns in the `adult` dataset 58
3.7 An implementation of stochastic gradient descent for logistic regression 60
3.8 Facade for training over multiple epochs with logistic regression 60
3.9 An implementation of *k*-fold cross-validation 63
3.10 Compute and plot the precision-recall curve based on a set of cutoff points 65
3.11 L_2 regularization for SGD 67
3.12 A resource management function 69
3.13 Using a closure to manage external resources 71
3.14 Logistic regression using global assignment operator 73
3.15 A *map* implementation of an EMA uses the gloabl assignment operator to append each incremental value to `s`. This code is unsafe, since it can lead to side effects. 74
3.16 In a $fold$ implementation of an EMA, the closure only reads variables. Read operations have no side effects and are therefore safe. 74
R/funs_lingua_franca.R 75
4.1 EMA using `Reduce` 79
4.2 Get position without a loop 80
4.3 The algae L-system produces a sequence of characters whose length corresponds to the Fibonacci sequence. 86

4.4 A new function clause for algae provides the wiring to automatically iterate over the L-system. The %isa% operator tests whether an object is an instance of a particular type. 87
4.5 A generalized using implementation with a custom exit handler 93
R/using.R 93
R/using.R 94
R/using.R 94
4.6 Remove punctuation from sentences 98
R/wordcount.R 98
R/wordcount.R 98
4.7 Implementation of a key-value data structure 99
R/mapreduce.R 99
4.8 Intermediate higher-order function that manages iteration based on job length. The function processes each partition in succession. In the end, the lists are concatenated into a single list. 100
R/mapreduce.R 100
R/mapreduce.R 101
R/baseball.R 102
R/baseball.R 102
R/baseball.R 106
5.1 Loading the diabetes dataset 126
R/combinator.R 233
R/combinator.R 233
R/combinator.R 234
R/combinator.R 234
R/combinator.R 234
R/combinator.R 235
11.1 The make_tree function for a random forests implementation 244
12.1 Initial implementation of the rules of Life 258
R/gol.R 260
R/gol.R 261
12.2 A function to apply productions to a sequence of tokens. Unlike true shift-reduce parsers, passes of the parser hold the size of the token buffer constant. 263
R/cfg.R 265
12.3 Parse a complete sentence 266
12.4 Apply unit productions 267
R/trading_fsm.R 269
R/trading_fsm.csv 270
R/trading_fsm.R 270
R/trading_events.csv 271
R/trading_fsm.R 271
R/trading_fsm.R 272

R/trading_fsm.R 272
R/trading_fsm.R 272
R/trading_fsm.R 273
R/markov_chain.R 275
R/markov_chain.R 276
R/markov_chain.R 276
R/markov_chain.R 277
R/chinese.R 278
R/chinese.R 278
R/chinese.R 279
R/chinese.R 279
R/chinese.R 280
R/chinese.R 282
R/math_as_fp.R 283

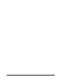
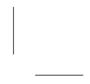
List of Tables

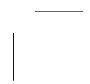
1.1 Primitive types across different programming languages. In Python and Java, primitives types are scalars. 14
1.2 Packages used in the book 18

3.1 In-sample performance of logistic regression 61

5.1 Logical operators given input(s) of length n 131

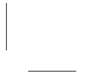
12.1 Time to read and select the same country subset from World Cities. Without memoization, the time increases linearly based on the number of iterations. Memoization incurs the cost of reading from the file system once per country, which is then amortized across all subsequent calls. 252

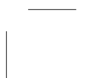





Part I

Foundation





1

Introduction

As data scientists, computational scientists, and quantitative analysts (herein just data scientist) it's easy to think of modeling as a strictly mathematical exercise. It's easy to focus on this part and ignore the fact that we spend most of our time programming. Famously called "the sexiest job of the 21st century" [16], it's particularly easy to ignore the decidedly unsexy 80% of data science: writing programs to clean, transform, and move data around. That means the majority of a data scientist's time is spent programming. And the cruel reality is that the worse you are at programming, the more time you spend doing it. Given this perverse imbalance, it's useful investing some time honing our programming skills to even out the time spent modeling versus the time spent writing code. A good way to do this is by writing programs with functional programming (FP) in mind. This is the practical motivation for this book.

It's likely that most of what you know about programming and software design doesn't apply to data science. Many academic programs focus on web development or mobile app development. These curricula tend to focus on object-oriented programming, which is well suited for GUI applications that deal with digital analogues to real world phenomena. Even server side systems may use classes to organize groups of operations and maintain state. While this approach can be effective, it works less well for systems built for data science. To be clear, I am talking about model pipelines, batch processing jobs, and even real-time predictive models built by data scientists. This distinction is important because systems built by professional programmers have design goals different from computational systems, which may warrant the use of object-oriented programming.

What makes programming for data science different from systems or app development? As hinted above, the entities modeled are significantly different. The reason is that much of data science programming is manipulating data and transforming it with mathematical operations. It's quite possible that a data science script or program can be reduced to a long sequence of repeated function application or extended function composition. Unlike a GUI window that needs to know what is inside it, a linear operator doesn't need to know anything about it's operands to function. This is different from object-oriented programming, where associated data structures and operations must be defined in advance. Object-oriented design (OOD) requires up-front planning of how a program or system will be used. Without this

design stage, many object-oriented systems end up being fragile, difficult to understand, and difficult to change.

Software design constitutes part of the software development process. But data scientists aren't professional programmers, just like most of us aren't professional drivers. Like driving, programming is a skill used to accomplish another task. And like driving, there's a minimum competency we need to have to avoid unnecessary risk and expense. Poorly designed software is expensive to operate and change. Therefore, we need to write code good enough to make our lives (and those around us) better, while staying focused on our work as data scientists.

One aspect of this philosophy is to understand the model development process and how it differs from conventional software development. Usually, a project starts as an idea or hypothesis that warrants investigation. This first phase is more exploratory in nature. If the analysis yields a promising result, more time and energy might be invested. At this stage, data is probably stored in flat files with few external resources used. Eventually the model might mature into something that runs in a production environment. Model development thus begins more ad hoc than a typical software development project. Rather than designing software at the beginning, it's more likely that it will happen further in the process once the model shows promise. Hence, early on it's more important to plan for change rather than design a system architecture.

Another significant difference between data science programming and typical programming is around the programming lifecycle. Professional software development focuses on automated operation of systems. The system is designed with this intention from the start, so attention is paid to how bits of code are organized. Data science programs typically start as an exploratory exercise. Many experiments will result in nothing, so the code will be thrown away. Some survive the gauntlet and emerge as the vehicle for repeatable science. Others reach a higher level of transcedence, becoming operational code that runs in a production environment. Hence data science programs often have a changing raison d'être depending on the research stage. How we write the code determines how long it takes for code to metamorphose from one stage to the next. Of course, this pupa feeds on our time to fully transform, which can be quite costly.

1.1 The model development workflow

Let's look at the model development process in more detail. Borrowing the iterative learning process described by Box [9], Figure 1.1 depicts an iterative model development process that starts with ideas or an initial theory based on current knowledge. In the first cycle, model design is minimal and instead the

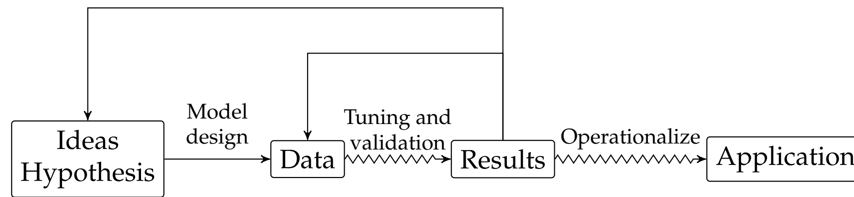


FIGURE 1.1: An idealized model development process based on Box's role of statistics in scientific learning.

focus is on understanding the data. This leads to formulating a hypothesis, which leads to model design and feature engineering. The model is then trained, tuned, and validated producing some results. Until the results are satisfactory, the cycle repeats, with each pass bringing more maturity and stability to the process. As a project matures, not only does the code mature, but data sources also become more formal. During exploration and initial model development, data are usually stored in flat files, parameters are hard coded, and there is just one way to run a model. As we explore additional features and other models, the processing pipeline essentially stays the same, but steps in the pipeline are swapped. Once good enough, the model will be promoted. The next chapter of its life will be as part of an application or run as a production system on its own. During this phase, the model must be *operationalized* to withstand the demands of a production system. The cost to operationalize and integrate a model can be great, so this investment should occur only after the model has proven itself. The approved models take one of two paths. Either they are maintained by data scientists themselves or are handed off to a technology team that (possibly) ports it to another language and prepares it for production. Either choice leads to a period of clean-up to make the original code easier to understand, more robust, and possibly support alternate data sources. Poorly written code leads to questions and bugs, both of which are expensive to address. Unit tests and other techniques borrowed from software engineering are employed to ensure that the process works as expected. The amount of effort is commensurate with how clean the original code is. This can seem overwhelming when functions are long and interrelated bits of logic are strewn about the function. Functional programming breaks apart these functions into bite-sized pieces. By virtue of their size, small functions can usually do just one thing. Hence, small functions help isolate different types of logic, further clarifying the purpose of each function.

Like survival of the fittest, though, not all models survive. If results do not improve after a few cycles, the best choice may be to abandon a project. In this case, hopefully parts of the model or code can be re-purposed for a different project. This is only practical if the logic and functions associated with the model are self-contained and easy to reuse. The fact that model

development and science in general can lead to many dead-ends clearly changes the way software is written. The optimal workflow is also different and must emphasize ease of readability and reuse over performance and stability.

Most software today is developed iteratively, a few features at a time. At the beginning, a so-called minimally viable product is built, which is a stripped down version of some future software vision. This system is meant to be immediately used by people, so it needs to have a certain level of robustness and reliability. The cycle usually encompasses a handful of activities: gather requirements, design, implement, test, deploy, collect feedback/data. This process also repeats itself. Over time, the software gradually improves converging to what the end user wants. Throughout the process, each iteration produces working (i.e. usable) software. For this process to be effective, a bit of design precedes each implementation.

In sum, both type of software development must deal with change. Traditional software design places more emphasis on organization, stability, and structure than model development. The onus is thus on the data scientist to effectively manage change and prepare for eventual operationalization. One key to managing this evolving code is simplicity. It's well understood that complex systems are more brittle and harder to change than simple systems. The same is true of models and code. Following the advice of Einstein, code should be as simple as possible but no simpler. Using functional programming techniques can help simplify code and promote reuse by facilitating modularity and reducing dependencies across functions.

1.2 Language paradigms

Two related concepts are discussed throughout the book. The first is how algorithms are written. Imperative algorithms are most common and explicitly describe control structures and variable manipulation. On the other hand, declarative algorithms describe what operations to perform (the results you want) instead of how to perform the operations. Take for example calculating the mean of the elements in a list. We normally think of the algorithm along the lines of Algorithm 1.2.1. This type of algorithm details how each variable is used to arrive at a final computed result.

Algorithm 1.2.1: MEAN(x)

$total \leftarrow 0$
for xi in x
  do $\{total \leftarrow total + xi$
return $(total/length(x))$

Usually though, we're thinking of it in terms of Algorithm 1.2.2. Yes, the second algorithm is just mathematical notation, which is the point. Historically, many algorithms were imperative out of necessity. Now, programming languages are far more expressive and can represent more complex operations without having to detail the movement of every bit of memory in the computer. Consequently, our algorithms should follow suit and utilize more declarative notation in their definitions.

Algorithm 1.2.2: MEAN(x)

return $(\sum x_i / |x|)$

One argument for imperative algorithms is that it's easy to determine the time complexity of an algorithm. This is useful. Algorithmic complexity of declarative algorithms is also easy to work out. In some cases it can be easier since the operations occurring within iterations is so concise. Furthermore, with big data solutions often hiding the implementation of algorithms in the name of computational performance, the value of big O calculations on local code is minimized. Platforms like TensorFlow, Ufora, and H2O all follow this trend.

Related to algorithm style is the programming language paradigm, which focuses on how code is structured. Common paradigms include procedural programming, object-oriented programming, and functional programming. Procedural programs are traditional programs that use procedures as the core organizational unit. Languages suh as C, Pascal, and Fortran are procedural. Object-oriented programming structures code into classes from which objects are instantiated. Java, C++, and Python are object-oriented, though Python supports multiple language paradigms. Functional programming is like procedural programming in that functions are the base building block. However, functional programming makes heavy use of concepts rooted in the lambda calculus [13] and has a more declarative structure. Other features like closures, are prevalent in functional programming but rare in procedural languages.

1.3 Imperative versus declarative algorithms

Algorithms not only contribute heavily to the data scientist's arsenal but are now a large part of contemporary life. Algorithms today are far more sophisticated than in the past. Despite this prevalence and advancement, how we describe algorithms is still quite primitive. Look at any paper describing a numerical method, and the algorithm is described imperatively. One thesis of this book is that describing algorithms imperatively conditions us into writing code imperatively. This seems strange, since much of math is declarative.

Even programming languages are becoming more expressive and declarative, thanks to plentiful and cheap computing resources. This trend is not new. Ever since punchcards and assembly language, computer scientists have strived to make programming languages more expressive. Advances in just-in-time compilation enable programmers to focus on application logic instead of algorithm efficiency since code is optimized on the fly. The significance of compiler improvements is that more programs can be described declaratively, with the actual operations determined by the compiler. These observations point to a world where imperative algorithms become increasingly obsolete.

Mathematical notation relies extensively on symbolic logic, which is declarative in nature. Operations are often described symbolically, where the implementation is assumed. For example, the expected value of a variable X is $E[X] = \sum_i p(x_i)x_i$ for all $x_i \in X$. We do not need to describe all the operations associated with $\sum$ for people to know what this operation signifies. Symbolic notation is declarative in that it describes what the operation does, but not how to achieve the result. Without declarative notation, many operations we take for granted become complicated. For example, suppose every time we reference matrix inversion we need to explicitly detail the steps associated with the operation. Clearly that would be annoying and possibly even hide the value of the work we are doing. Declarative notation enables us to move beyond these details to focus on the essence of mathematics. The same is true of its software counterpart.

How we describe algorithms is not just a cosmetic concern. As modelers, we are natives of mathematics. It is part of our DNA and nature. Much of our work is representing real-world problems as mathematical problems. By extension our solution is mathematical, so why not the algorithm? When computers had limited resources (or when computers were people), excruciatingly detailed step-by-step algorithms were required out of necessity. Algorithms like matrix factorization are quite mechanical, while others are more declarative. For example, the power method for finding the eigenvector associated with the dominant eigenvalue can be found by repeatedly applying the recurrence relation $x_n = Ax_{n-1}$ until it converges. A typical algorithm describing this method looks like Algorithm 1.3.1. In this first version, the algorithm details the moving around of data between variables. This is characteristic of imperative loops. The problem with algorithms described in this manner is that they rely heavily on manipulating variables and their indices. This has nothing to do with math and is instead a function of how the computer accesses and manipulates data.

Algorithm 1.3.1: POWER($A, x_0, tolerance$)

$error \leftarrow \infty$
while $error > tolerance$
do $\begin{cases} x_1 \leftarrow Ax_0 \\ x_1 \leftarrow \frac{x_1}{\|x_1\|} \\ error \leftarrow x_1 - x_0 \\ x_0 \leftarrow x_1 \end{cases}$
return (x_0)

An alternative representation is Algorithm 1.3.2, which uses recursion to model the recurrence relation. This has a simplifying effect since each iteration of the function is self-contained. Since the variables are scoped for a single iteration only, there is no need to manage variables explicitly. The temporary variable x_1 is used as a convenience but isn't necessary. Notice how the body of this algorithm essentially reduces to a chain of repeated function application.

Algorithm 1.3.2: POWER($A, x_0, error, tolerance$)

if $error \leq tolerance$
  then return (x_0)
$x_1 \leftarrow Ax_0$
return (POWER($A, \frac{x_1}{\|x_1\|}, x_1 - x_0$))

Data science is filled with iterative methods like this. From transforming all records in a dataset, to optimizing functions, training a model over multiple epochs, iteration is central to it all. Throughout the book we'll see how declarative algorithms simplify the translation of mathematical ideas into code and solve numerous other problems along the way.

1.4 Elements of a functional programming language

We've established that writing good programs makes pragmatic sense. A more philosophical perspective is that algorithms have always been a part of mathematics, so writing good programs is good math. Going further, one result of the lambda calculus is the Curry-Howard Isomorphism, which shows that programs are equivalent to proofs. [46] Most of the functions we write probably won't satisfy the criteria for this to work. Even so, the idea is attractive because it hints that we can reason about our programs in a way that gets us to absolute certainty. This perspective can help change the way we look at the role of code in our models.

Functional programming uses plain old functions as the base building

block. The simplicity of this approach compared to object-oriented programming induces many profound changes in program design and structure. Most significant is that FP enforces modularity and isolation, promoting not just reuse but parallelization. In the age of big data, parallelizing code is a necessity. Writing programs in a functional style enables a seamless transition between a single processing node and multiple processing nodes. Another benefit of the FP style is that there is a clear separation between data and functions. In OOP systems, a class defines both the data and the methods (aka functions) that operate on it. Modularity exists via class hierarchies and design patterns but at the cost of interdependence. These dependencies make it difficult to reuse a class in a different context or in isolation (apart from the rest of the class). This again becomes a challenge when distributing work across multiple compute nodes. From a portability perspective, it is easier to just send data as opposed to a language-specific object containing data, state, and functions. Since data and functions are discrete first-class entities in functional programming, there is more flexibility in how each are used. This independence may seem subtle but it echoes the ethos of mathematics, where functions operate on specific mathematical entities irrespective of the application or domain. For example, an eigenvalue decomposition works on any square matrix whether this matrix represents a graph, a utility matrix, or a covariance matrix of asset returns. The same is true in functional programming.

Functions serve many purposes, and depending on how they are used, they are called something different. When functions are treated as values and passed as arguments to other functions, they are called first-class. If a function is created without binding it to a variable, it is known to be anonymous. Functions that retain state from surrounding scopes are called closures. Finally, functions that take other functions as arguments or return functions are known as higher-order functions. [1] Like stem cells, functions in R have maximum potential and can take on any of these traits. It's not uncommon for higher-order functions to return a closure. Functions like `aggregate` routinely take anonymous functions as an argument. Such heavy use of functions leads to the idea that functions help define atomic units of work. This leads to the idea of modularity.

A natural artifact of functional programming, modular programs comprise many small functions that can easily be rearranged and reused. This works because higher-order functions provide much of the wiring to accomplish common tasks, such as iteration, transformation, and grouping. In R this is seen most clearly with functions like `apply` and `aggregate`. Both of these functions are variations on the canonical functional *map*, detailed in Chapter 6. As the name suggests, *map* implements a mapping from an input to an output. This concept is firmly seated in the mathematical idea of a function, where a function maps one set to another set. In a vectorized language like

[1] Also known as a functional. We use these two terms interchangeablely.

R, many functions behave this way natively, such as the algebraic operators. When native vectorization is not available, *map* provides the wiring to quickly endow vector semantics to any function. Due to complex data structures, this is surprisingly often, and the battery of `apply` functions is testament to the variations occurring in R.

A more primitive functional is *fold*, from which *map* can be implemented. Discussed in Chapter 7, *fold* implements iterated function application. Like a swiss-army knife, *fold* can be used in nearly any situation. A general pattern revealed in the book is how *map* is used to transform individual datum, while *fold* is used to transform systems of data. Many mathematical systems or processes can be modeled as a *fold* operation, though we don't always look at them from this perspective. Just like *map*, *fold* has roots in mathematics, this time via inductive processes. The incremental value of a series can be defined in terms of the previous incremental value. An arbitrary term in a series can be modeled as $x_i = f(i, x_{i-1})$. This representation is valid for both summation and product series, as well as stochastic processes and iterative systems like the Kalman filter. For example, one series expansion of $\frac{pi}{4}$ is defined as the series $\sum_{i=0}^{\infty} \frac{(-1)^i}{2i+1}$. Each term in the series can be defined in terms of the previous term: $x_i = x_{i-1} + \frac{(-1)^i}{2i+1}$. It's trivial to then tie this back to an iterated function representation. Another powerful application of *fold* is aggregation, which can be considered a special case of an inductive process. Instead of retaining all elements of a sequence, only the final value is kept. This is similar to the difference between `cumsum` and `sum`. One consequence is that many aggregate statistics like variance can be reformulated as an inductive *fold* process.

Extending induction further, state-based systems can be modeled using *fold* operations. The general formulation is that the current state of an arbitrary system X_t is a function of its prior state X_{t-1}. In other words, $X_t = f(X_{t-1})$. This implies that given some initial state X_0, the state at X_n is an n-fold function composition $f^{(n)}(X_0)$. This simple observation demonstrates that any iterative system can be represented as a *fold* operation. It's easy to convince yourself that simple deterministic systems like iterative function systems (IFS) fit within the *fold* model, but what about parsers and machine learning models? To answer that, let's look at another simple state-based system. Markov chains are like an IFS since they don't take external input and evolve based on a set of rules. The difference is that the rules are probabilistic instead of fixed. Finite state machines also evolve over time, although based on external input. Their representation is $X_t = f(a_t, X_{t-1})$, which is the standard form of *fold*. That's why this formula is also used to represent an infinite series. Context-free grammars also rely on state. Both parsing and language realization evolve according to their internal state. How these systems utilize state are detailed in Chapter 12. Machine learning models are all state-based systems. The process of training a model is no more than updating a state until it reasonably describes data. Chapter 11 deconstructs and reformulates the random forest algorithm in this manner.

The last canonical higher-order function is *filter*. As the name suggests, *filter* removes elements from a vector or list based on specified criteria. While it operates on individual elements, unlike *map* the output cardinality is not equal to the input. Hence, *filter* sits somewhere in between *map* and *fold*. The *filter* concept, detailed in Chapter 8, is deeply integrated into R via subsetting notation. This notation is a syntactic convenience, which is generalized by the use of a predicate. In fact, a *filter* process is equivalent to using a predicate with *map*. Recognizing this equivalence can simplify code and present reuse opportunities. With these three functionals, all other iterative operations can be easily implemented.

Native vector operations are a key feature of R. Most general purpose languages define scalars as primitives. However, languages (or libraries) designed for numerical analysis often use vectors as primitives. It turns out that vector operations implicitly use functional programming concepts since the mechanics of iteration are implied in the operations. This makes R particularly well-suited for a FP style since so many operations already fit within this paradigm. Chapter 5 explores the behavior of this foundational structure, connecting the semantics of vector operations with functional programming. You'll find that fluency in functional programming concepts is a short stroll from native R semantics. Embracing the FP features of R will vastly simplify your code, freeing you up to do more of the science in data science.

Vectors only support values of the same mode. Hence, a logical vector only supports `TRUE`, `FALSE`, and the logical `NA`. These cannot be used in a numeric vector. When all elements are of the same type, using a vector is appropriate. Accessing arbitrary elements within a vector is fast, as are native vector operations. Despite this performance improvement, there are times when we want to hold values of different types in the same structure. Mathematically, a tuple usually represents this concept, while in R this is implemented by the `list` type. [2] Chapter 9 shows that lists behave similarly to vectors. The main difference is that subsetting is different from element access, so additional syntax exists to differentiate between the two operations. Since lists can hold arbitrary objects (including other lists), lists are also used to pack function arguments. The most common case is when using `do.call`, discussed in Section 9.5. This technique is used when a function is called dynamically, or when arguments to a function are constructed dynamically. Lists act as the building block for creating more complex data structures. To illustrate this idea, Section 9.6 shows how to implement trees and graph structures using lists.

Another core data structure in R is the data frame, which represents tabular data. Chapter 9 discusses how the subsetting concepts for vectors and lists are extended into two dimensions. The same approach works for matrices and higher-dimensional arrays that represent tensors. Iteration also needs to

[2] In many functional languages, the list is a generic container that holds multiple elements. Typically it is implemented as a linked list, where each element contains the actual value plus a pointer to the next element in the list.

be modified to handle an extra dimension. R approaches this numerous ways, based on different `apply` functions. The eponymous `apply` function operates on rows or columns of a data frame, where as `sapply` operates on an index and `mapply` operates on a slice of a data frame. Knowing when to use each function depends on style but also the types contained in the data frame.

1.5 Review of object-oriented programming

Much of the personal computer revolution began at Xerox PARC. Object-oriented programming also has roots at PARC, specifically with the Smalltalk language [24]. While not the first object-oriented language, Smalltalk was the first one to heavily incorporate GUIs into the programming model. This original form of object-oriented programming was based on message passing between objects. The point was how to enable self-contained modules to communicate with each other, as opposed to worrying about how to manage the internal properties and behaviors of the modules [23]. The idea was to delegate the dispatching of a concrete function to an object that had its own context and knew which function made the most sense. With GUIs, this is quite compelling, since events can occur on one part of an interface that can affect objects elsewhere on the screen. The object triggering the event often has no idea how other objects will handle the event, providing a compelling use case for embedding functions directly with the state they use.

Messages in Smalltalk were simply identifiers followed by additional parameters. The recipient would then decide how to react to the message. In other words, message passing is a way to delegate control to other software components. These days this is a common practice when interacting with web-based APIs: we send a message describing what we want, plus a payload of supporting details and get something back. How the service constructs the rsponse isn't our concern, so long as it's correct. This sounds awfully similar to how we've been characterizing declarative programs. This concept even makes its way into R via S3 and S4 generics. The delegation occurs through the use of special functions (e.g. `UseMethod` for S3) that are responsible for dispatching to the correct function.

Contemporary object-oriented programming extends these concepts to create a different beast altogether. Languages like Python, Java, C++ [3] use the same conceptual model with minor differences between them. To begin, there is a concept of a class hierarchy. Classes represent a template defining members of the class, including properties and methods (operations) on the member. This structure resembles the Linnaeun taxonomy, where all living creatures belong in the hierarchy. Common properties, such as the number

[3] I'm explicitly omitting C# since most data scientists don't use it.

R	Python	Java
numeric vector	int	int
numeric vector	float	float
numeric vector	double	double
character vector	char	char
character vector	string	java.lang.String
logical vector	bool	boolean
N/A	tuple	N/A
list	list	Array
environment	dict	java.util.HashMap
matrix	N/A	N/A
array	N/A	N/A
data.frame	N/A	N/A

TABLE 1.1: Primitive types across different programming languages. In Python and Java, primitives types are scalars.

of legs, are grouped together in a parent class. As an example, the order Crocodilia is a member of the class Repitilia. One commonality of reptiles is that they have four legs, [4] so this property might be set in Repitilia to avoid repetition in subclasses.

In software, this taxonomy defines your universe of types, built on top of the primitive types in the language. Table 1.1 compares primitive types in a few languages. Irrespective of the language, variables can only be created based on existing types. In *dynamically typed* languages, this is done behind the scenes, whereas in *statically typed* languages, it is explicit. In strict [5] object-oriented languages like Java, variables can only be *instantiated* from classes defined in the hierarchy. This variable (or object) is called an instance of the class. For example, creating a software crocodile looks like

```
CrocodylusPorosus crocodile = new CrocodylusPorosus();
```

in Java. Python is less verbose, partly due to dynamic typing.

```
crocodile = CrocodylusPorosus()
```

You might wonder if it's possible to create instances higher up in the taxonomy. For example, is it possible to instantiate an object of type Repitilia? Clearly, this class is a man-made fabrication, and it would be meaningless to create such an "animal". Classes that cannot be instantiated are called *abstract*. In Java, the compiler ensures that any class marked as abstract cannot be instantiated. In Python the same behavior can be emulated using the `abc` module.

[4] Snakes are an exception, though they descend from four legged reptiles.

[5] The technical term is "pure", but technically Java is not pure, so I use "strict" as a compromise to distinguish it from Python.

New object instances are created via a special method called the constructor. This method is defined inside the class and details how to create objects of the given class. In Python, this function is named `__init__` and takes at a minimum a parameter named `self` [6], which represents the new object. To illustrate, suppose gender must be given when calling the `CrocodylusPorosus` constructor. This property is assigned to `self` in the constructor.

```
class CrocodylusPorosus:
  def __init__(self, gender):
    self.gender = gender
    self.energy = 1
```

The newly instantiated object will have this attribute attached to it. Object properties can represent attributes of the object or its state. The number of legs, gender, and eye color are all attributes of a crocodile while the last time she ate is part of her internal state. All of these properties are specific to each crocodile and are accessed directly or via a method. The former is popular in Python and is known as *inspection*, while the latter is standard in Java and is called *invocation*. Invocation is considered safer, since a class can define appropriate default values. For example, suppose some crocodiles are tagged with a unique id to understand their behavior. Among a group of crocodiles some have tags and some don't. In Python, we can simply attach this property to the tagged crocodiles.

```
crocodile.id = 32423
```

This flexibilty is convenient but presents a problem when we attempt to access the id of an untagged crocodile.

```
>>> crocodile_1 = CrocodylusPorosus('female')
>>> crocodile_1.id
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: CrocodylusPorosus instance has no attribute 'id'
```

In Java, this approach is not allowed and results in a compiler error. If one class is used to represent all crocodiles, a tag id must be added as a property for all crocodiles. Otherwise, a new class must be created that *extends* the original class and introduces the new property. This new class is a subclass of the parent, inheriting its properties and methods. The same approach works in Python.

```
class TaggedCrocodylusPorosus(CrocodylusPorosus):
  def __init__(self, gender, tag):
    CrocodylusPorosus.__init__(self, gender)
    self.tag = tag
```

The methods of a class follow a similar design process. Suppose we want to know how hungry a crocodile is. We can define a method `is_hungry` that

[6] This name is not enforced by the interpreter, but it is a standard convention.

returns true or false based on the amount of energy the crocodile has. Thanks to inheritance, instances of `TaggedCrocodylusPorosus` will also have this method defined.

```
class CrocodylusPorosus:
  def __init__(self, gender):
    self.gender = gender
    self.energy = 1
  def is_hungry():
    if self.energy < 6: return true
    return false
```

This *instance method* only knows about the energy associated with each specific instance. For simulations, this can be quite useful since numerous agents can be creatd where each is responsible for managing its own state. [7]

Continuing this process, it's easy to see how class hierarchies are born. Over time, properties and methods migrate between classes until they find some equilibrium between generalization and specificity. As more classes are created, each expects to operate on specific other classes, creating an ever expanding web of dependencies. Hence, any change to one class may affect all other classes that depend on it. This is why design is so important in object-oriented systems.

Let's return to message passing. Earlier, I said that S3 generics resemble message passing. S3 is a dispatching system that connects methods with different classes. A function is first registered as being an S3 method by delegating the dispatching to the `UseMethod` function. A concrete implementation is based on the type of the first argument. The `UseMethod` function uses this information to dispatch the underlying function implementation. For example, we can create crocodiles using this approach.

```
CrocodylusPorosus ← function(gender)
  UseMethod("CrocodylusPorosus")

CrocodylusPorosus.character ← function(gender) {
  o ← list(gender=gender, energy=1)
  class(o) ← 'CrocodylusPorosus'
  o
}
```

Calling this function produces objects with the class `CrocodylusPorosus`.

Suppose we want to implement a method to enable our digital crocodile to eat. Let's call this function `eat`. Let's also register this function as an S3 method.

```
eat ← function(animal, food) UseMethod("eat")
```

This setup allows us to treat `eat` as a message to `animal`. In essence we are

[7] Indeed, this was one of the driving motivations for the original object-oriented programming language, SIMULA.

saying we want the recipient of the message to handle eating the way that is most appropriate for it. Importantly, as the calling function (message passer), we don't know what that entails.

S3 solves this by defining separate methods for each recipient type. For crocodiles we can define it as

```
eat.CrocodylusPosorus ← function(animal, food) {
  prey ← c("Tapirus indicus", "Tragulus napu", "Homo Sapiens")
  if (! class(food) %in% prey) stop(sprintf("A %s is inedible", class(food)))
  add_energy(animal, get_energy(food))
  die(food)
  animal
}
```

Notice that after some rearranging, this is semantically similar to the Smalltalk message passing syntax `animal eat: prey`. This also looks eerily similar to infix notation. We'll see in Chapter 4 that extending the message passing concept to multiple arguments is equivalent to pipe notation. One argument of this book is that object-oriented programming is largely unnecessary for data science. Through numerous examples, we'll see how dats science problems can be solved simply and quickly using FP principles.

1.6 Syntax, notation, and style

As the centerpiece of the book, functions can be described in various ways. From a mathematical perspective, a function is generally characterized by its domain and range (co-domain). For a function f, it is described in the abstract as $f : X \rightarrow Y$, meaning f maps elements of X to elements of Y. If a function has multiple inputs, the input domains are separated by the $\times$ symbol. For example, $+ : \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}$ describes the addition operator over the reals. Functions that take vector arguments use a superscript to indicate the length of the vector. The summation operator can be characterized as $\sum : \mathbb{R}^n \rightarrow \mathbb{R}$, indicating that the length of the input is arbitrary. From a computer science perspective, this description of a function is essentially its signature, less any variable names.

To improve readability, different fonts are used to represent different entities within the text. When referring to the language, R is typeset using a sans serif font. Code listings use a `fixed width` font. References to variables, functions, and packages within the prose also use a fixed width font. Mathematical expressions and objects are *italicized*. When switching between mathematical objects and their code counterparts, I maintain font consistency to help distinguish the context. In general, code is provided inline, as this is more conversational. That said, breaking up a function into too many little pieces can impede understanding. Thankfully, functions are meant to be

Package	Purpose
caret	Model developent and tuning
dplyr	Data manipulation
futile.logger	Logging
lambda.r	Functional programming dispatching and type system
lambda.tools	Collection of functions for functional programming
magrittr	Infix pipe notation
plyr	Data manipulation
purrr	Collection of functions for standardized development
rmr2	MapReduce implementation for RHadoop project
zeallot	Pattern matching for assignment

TABLE 1.2: Packages used in the book

short, so they can usually be listed in toto. Code examples may also depict the result of a command in the R console. In these situations, executed code is preceded by a > character, which represents the console prompt. One last note about code listings is that a ligature is used for the assignment operator. Hence, instead of <−, the symbol ← is used. Not only is this more readable, but it also serves a pedagogical purpose. Anecdotally, I've seen that programming concepts are better retained when manually typed versus simply copying and pasting code. Readers that want complete, executable listings can always reference the examples online [8].

The canonical higher order functions are always italicized. This helps avoid confusion with other meanings for these words. It also reinforces the canonical nature of these higher-order functions.

1.7 Examples and package dependencies

A number of packages are referenced in this book. A handful are written by me to support my own model development. Table 1.2 lists packages referenced and their purpose. Readers should be able to install and load libraries. If not, a number of introductory texts on R are available.

[8]An open source project with all code listings is available at **??**

2

The functional programming language called mathematics

We hinted that mathematical notation has a lot in common with functional programming. The connection is far deeper, extending all the way to the lambda calculus, which is a formal mathematical system for defining mathenatics itself, all based on functions. This book will stay closer to the surface, showing how functional programming concepts aid the model development process. The more you appreciate the connection between math and functional programming, the easier it will be to implement mathematical ideas. This chapter works through a number of examples in various disciplines to highlight this connection. Where appropriate we'll touch on the related FP concepts and point the reader to the chapter where it's discussed in more detail. We'll also occasionally juxtapose functional implementations with plain imperative code or object-oriented code. These counterexamples will aid in internalizing the differences between the approaches. One conclusion is that mathematical structures are not as compatible with the object-oriented approach. This stems from the fact that functions and objects are distinct entities (though functions are objects), just like in functional programming. In contemporary OOP, classes are structures that contain both data and operations on the data. This makes for some strange operations as we'll see in this chapter.

As we work through comparisons between language paradigms, keep in mind that there are two scenarios we're balancing. The first is as the user of an existing library or code that is written in a functional style. Understanding functional programming concepts will help you utilize these functions to their fullest. The second is writing your own algorithms or libraries. Here you also need to consider other users of your package, so the details of the implementation are important not just to you but to your users that want to understand what your function or model is doing. You want to optimize readability of the code with ease of use and maintainability. Another goal is to limit the requirements necessary to use your package. Box said that "Almost never is an experimental result put to use in the circumstances in which it was obtained" [9]. We can say the same about mathematics and mathematical software. This approach is most open and can lead to surprising

and wonderful new uses. [1] The less opinionated your package, the more possibilities you create.

2.1 The declarative nature of set theory

Controversial when first introduced by Georg Cantor, set theory is now so ubiquitous that it is often taken for granted. Even business students get a taste of set theory via Venn diagrams. Much of set theory is described declaratively. Given two sets X and Y along with an entity x, simple operations like set membership $x \in X$, intersection $X \wedge Y$, and union $X \vee Y$ are all represented declaratively. Construction of sets is also declarative, most commonly seen in set builder notation. For example, the squares can quickly be defined $\{x^2 : x \in \mathbb{W}\}$. This notation is flexible and can contain predicates, another aspect of functional programming. Discussed in Chapter 8, predicates are functions that return a logical value. Subsetting notation makes heavy use of predicates. A set that can be defined with predicates is the set of prime numbers. This set is described as all numbers whose only positive divisors are one and itself. We define the set of prime numbers as $\mathbb{P} = \{p \in \mathbb{N} \mid p \text{ is prime}\}$. This set can be used without necessarily constructing it, which is good since it's infinite. Of course, many sets are infinite. The combination of symbolic and declarative notation allows us to make use of such sets even though it's impossible to completely itemize them.

A sequence is a function that maps ordinals to a set. Many sequences are defined inductively, which is declarative, since the actual mechanics of constructing the set are implied. The well known Fibonacci numbers are defined by the following equations.

$$
\begin{aligned}
F_0 &= 1 \\
F_1 &= 1 \\
F_n &= F_{n-1} + F_{n-2}
\end{aligned}
$$

If a number x is a member of this set, we denote it as $x \in F$. If x also happens to be a prime number, we say $x \in F \wedge x$ prime. With this concise description, conceptually we know what the elements of this set are, without having to construct it nor detail the algorithm to calculate its members. This is central to the difference between declarative algorithms that describe *what* the result of an operation is versus imperative algorithms that describe *how* to manipulate

[1] This happened to me, with my `tawny` package. Initially designed to clean covariance matrices of asset returns via random matrix theory and shrinkage estimation, the interface expected a data frame of asset returns. It turned out that numerous practitioners in computational psychology wanted to use the same methods, which led to removing those data type constraints to expand the utility of the package.

data structures to achieve a specific result. In R this same expression is denoted `x %in% F &&is.prime(x)`. Most programming languages support logical expressions that are quite similar to their mathematical counterparts. R takes this concept a step further by utilizing the results of logical expressions as predicates to other operations.

Subsetting uses this concept extensively. For example, suppose we want to describe the set of all prime Fibonacci numbers. There are numerous ways to do this. Using our definition of primes from before, this is simply $F \wedge \mathbb{P}$. Alteratively, set builder notation allows us to describe each element explicitly: $\{x \mid x \in F \wedge x \text{ is prime}\}$. For this second approach, the R syntax again mimics this notation: `x[x %in% F &is.prime(x)]`. When reading the above expression, did you stop to consider how the primes are computed? If not, then you experienced the benefit of declarative code. Expressions like these are easy to understand because they focus on describing what the set is as opposed to the implementation details of how to construct the set within the programming language.

2.2 Statistics

Descriptive statistics are a staple of data science and often one of the first tools used to characterize data. Rarely do we need to implement these functions from scratch. In this section that's precisely what we'll do since their simplicity reveals the connection between mathematical expressions and functional programs more clearly. Seeing this relationship is a necessary ingredient for quickly translating mathematical ideas into code reality.

2.2.1 The mean and imperative loops

Let's start by examining the mean of some univariate data. Assuming zero knowledge of R idioms, how might we implement the mean of a vector? The mean is typically defined

$$\frac{1}{n}\sum_{i}^{n} x_i$$

for a vector of length n. A common approach is to initialize a temporary variable and loop through each element, adding it to the temporary variable. The final step divides this sum by the number of elements. The algorithm might be codified as in Algorithm 1.2.1. A working implementation is nearly verbatim. This coding style is called *imperative* because each step of the algorithm is explicit. Imperative algorithms tell the computer how to manipulate variables and memory to achieve a specific result. The hallmark of imper-

ative code are loops and repeated variable assignment. Usually temporary variables are updated based on some repeated operation.

```
mean.i ← function(x) {
  sum ← 0
  for (xi in x) sum ← sum + xi
  sum / length(x)
}
```

LISTING 2.1: Naive implementation of the mean using a loop

Both procedural and object-oriented paradigms are imperative in nature. The difference between these paradigms is structural, in terms of how code is organized, as opposed to how algorithms are implemented.

In contrast, declarative programs express what a computation should do as opposed to how. Function implementations appear to be more symbolic, closer to mathematical notation. Functions are also used more frequently to group units of computation together. For example, the declarative implementation of mean uses $fold$ and a closure[2] to mediate the loop operation.

```
mean.d ← function(x) {
  fold(x, function(xi,sum) xi + sum, 0) / length(x)
}
```

LISTING 2.2: A declarative implementation of the mean using `fold`

Fold abstracts iterated function application, adding the next element in the sequence to the accumulated result. In other words,

```
function(xi, sum) xi + sum
```

is applied to each element of x along with the accumulated value or state. At the end of the iteration, the final value is returned. In this expression, $fold$ is computing the sum of x. This works by expanding $\sum_i^n x_i = ((((0 + x_1) + x_2) + ...) + x_n)$. Coming from an imperative background, this may seem like a strange way to implement the summation operator. Chapter 7 devotes time to discussing the motivation behind this approach. For now, let's focus on encapsulating the implementation in its own function.

```
sum.d ← function(x) fold(x, function(a,b) a + b, 0)
```

The mean can now be written

```
mean.d ← function(x) sum.d(x) / length(x),
```

which is notationally similar to the underlying mathematical expression.

The function $fold$ is known as a higher-order function, or functional, because it operates on functions. Higher order functions can also return functions. The many uses of these special functions are discussed in Chapter 3.

[2]For now, a closure can be thought of as an anonymous function. The semantics of closures will be discussed in more detail in Chapter 3.

Functional languages typically have a small set of standard higher order functions that act as building blocks for more complex operations. We'll explore the canonical set of functions in Chapters 6 - 8.

Back to the implementation, notice that no variables are overwritten. Variables in functional programs tend to have a scope so limited that updating the value of a variable is unnecessary. Some pure functional languages do not even allow multiple assignemnt to variables, which provides a strong guarantee on deterministic behavior. The benefit is that functions are easier to understand since the variables have limited scope to change. Hence, there is no need to scan lines and lines of code to determine the behavior of a function.

2.2.2 Covariance and declarative functions

In the previous section, we pretended that we were ignorant of the native vectorization in R. This momentary suspension of disbelief allowed us to distinguish between imperative and declarative algorithms that operate on vectors. Building on this knowledge, let's implement the covariance of two variables X and Y. Recall that covariance is defined

$$Cov(X,Y) = \frac{1}{n}\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y}).$$

for two random variables X and Y of equal length. The implementation is similar to `mean.i` since there is a summation involved in the calculation. There's more setup, though, since we need to subtract the means and take the product of each element-wise pair (x_i, y_i).

```
cov.ia ← function(x,y) {
  sum ← 0
  ux ← mean(x)
  uy ← mean(y)
  for (i in 1:length(x)) {
    sum ← sum + (x[i] - ux) * (y[i] - uy)
  }
  sum / length(x)
}
```

LISTING 2.3: A loop implementation of covariance

The summation operator is a frequent actor in mathematical expressions, but its meaning can get buried in the implementation, particularly as the summand gets more complicated. When embedded in a loop, another implication is that each implementation of the summation loop must be tested each time. A better approach is to encapsulate the summation operator so it takes any vector expression and sums the elements. A native R approach does just this. Syntactically, this version looks closer to the mathematical expression than the imperative algorithm.

```
cov.r ← function(x,y)
  sum((x - mean(x)) * (y - mean(y))) / length(x)
```

LISTING 2.4: Covariance using native R functions

The similarity is more than cosmetic. Our expression is now declarative, where the implementation is a detail as opposed to the main focus.

Using vectorization may seem like cheating, but as we'll see in this book, vectorization is actually functional programming in disguise. Functional programming gives us a framework for applying the same vectorization concepts to data structures and functions that are not natively vectorized. Does that mean we can simply use vectorized functions in imperative code to make it declarative? For example, how can we transform the imperative implementation to leverage `sum`? To use `sum`, we need to pass it a vector. Instead of updating a temporary variable, now we need to create a temporary vector. At each iteration, the intermediate term is appended to this vector. Finally, this vector is passed to `sum` to produce the desired result.

```
cov.ib ← function(x,y) {
  z ← c()
  ux ← mean(x)
  uy ← mean(y)
  for (i in 1:length(x)) {
    z ← c(z, (x[i] - ux) * (y[i] - uy))
  }
  sum(z) / length(x)
}
```

LISTING 2.5: An attempt at declarative notation with a loop

But that didn't do anything to simplify the code! We simply replaced one temporary variable for another. The reason the code is still imperative is that we're still using loops and explicitly interacting with each individual vector element. Code written this way is the programming version of micromanagement. What's better is to describe the transformations you want to make on data and allow the interpreter to handle the details. This is how SQL queries work, where you specify the data you want, and the query engine figures out how to efficiently retrieve the results. It's only when the query engine gets confused that you care about the implementation. We already saw how $fold$ can be used to mediate a loop. Another approach is to use *map*, which applies a closure to every element in a vector, or $map(f, X) = (f(x_1), f(x_2), ..., f(x_n))$ for x_i in X. The idea is that each pair of elements in X and Y are passed to the closure f and their product of differences computed, just like in the updated imperative implementation. The only wrinkle with this approach is that *map* operates on a univariate closure. If we want to pass more than one argument to f, we need to wrap them up in a tuple, which is what *zip* does. Shown in Listing 2.7, this function is a staple of functional

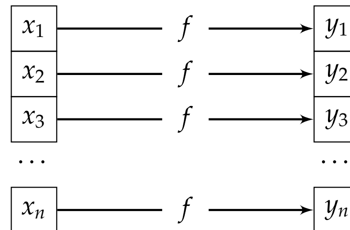


FIGURE 2.1: The *map* operation maps a function to a set of values

languages and has the signature $zip : X^n \times Y^n \rightarrow (X,Y)^n$. Putting it all together (and using `sapply` as our *map* implementation), we have

```
cov.da ← function(x,y) {
  ux ← mean.d(x)
  uy ← mean.d(y)
  sum(sapply(zip(x,y),
    function(xy) (xy[1] - ux) * (xy[2] - uy))) / length(x)
}
```

LISTING 2.6: Using functional primitives to implement covariance

This implementation avoids loops and reassignment. Indexes are used to grab the x and y values in the "tuple". One thing to note is that using array indices is not anathema to functional programming. However, the variable being indexed has a limited scope, namely within the closure. This is a form of protection since the indexing is independent of the source variables X and Y, and are not affected if they change.

```
zip ← function(x,y) {
  lapply(1:length(x), function(i) c(x[i], y[i]))
```

LISTING 2.7: Implementation of *zip*

It turns out that *zip* isn't used much in R because `sapply` isn't the only *map* implementation. The function `mapply` is designed to handle multivariate data, so it can be used directly to simplify the implementation.

```
cov.db ← function(x,y) {
  ux ← mean.d(x)
  uy ← mean.d(y)
  sum(mapply(function(a,b) (a-ux) * (b-uy), x,y)) / length(x)
}
```

The vast array of *map*-like functions in the `apply` family can be daunting at first. It's important to remember that R is a pragmatic language, and these variations are designed to minimize the amount of code necessary to create your models. The drawback is that it can be more challenging to learn. Chapter

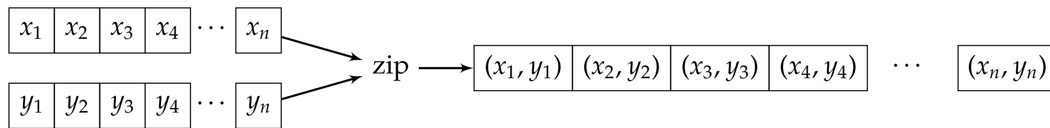


FIGURE 2.2: The *zip* operation constructs a single vector of tuples from a set of vectors.

6 details the various *map* implementations and discusses when to use one over another.

2.2.3 Symbolic conveniences in linear regression

Consider the `swiss` dataset that comes bundled with R. This dataset lists the standardized fertility measure and socio-economic indicators for the French-speaking provinces of Switzerland, circa 1888 [18]. We can use a multiple regression to explore how these socio-economic factors influence the fertility rate. This relationship can be described as

$$\text{fertility} = \alpha + \beta_a \text{ agriculture} + \beta_e \text{ education} + \beta_c \text{ catholic} + \epsilon$$

Conveniently, R provides the so-called formula notation for specifying models, which closely maps to the mathematical expression. Formula notation derives from [55], originally appearing in older statistical computing systems Genstat and GLIM.

```
lm(Fertility ~ Agriculture + Education + Catholic, data=swiss)
```

For sake of comparison, the Python package `scikit-learn` uses a more traditional object-oriented interface. [10] The approach mirrors most object-oriented libraries: create an object representing the model, and then call methods on that instance.

```
model = linear_model.LinearRegression()
model.fit(X_train, y_train)
```

On the surface it's not so different, except two steps must be taken instead of one. An object of class `LinearRegression` must first be constructed and then the model fit to the data. Also, the data needs to be partitioned explicitly between predictors and response variables. The predictors are exactly the contents of `X_train`, as opposed to a specified subset. In this example it's not too troubling, but in cases where you want to consider different variables as the responses, it can be annoying to perform inline slicing to extract a specific subset. In general, a 4-tuple must be defined that encompasses the training and test sets for the response and predictor variables.

To see how the object-oriented paradigm begins to break down, let's complicate things by a hair. Suppose we want to ignore the intercept term. In R we specify it in the formula by adding a 0 term.

```
lm(Fertility ~ 0 + Agriculture + Education + Catholic, data=swiss)
```

With `scikit-learn`, you might think that you specify this option in the `fit` method. This is wrong, as it happens to be an argument of the constructor:

```
model = linear_model.LinearRegression(fit_intercept=False)
model.fit(X_train, y_train).
```

Surprises like this typically occur in the name of consistency. One of the challenges with object-oriented programming is deciding which variables are properties of an object versus arguments to a function. In this case, `LinearRegression` is a subclass of `LinearModel` which defines the `fit(X,y)` method. For the sake of interface consistency, all subclasses must conform to this signature. The difficulty is that models don't really conform to a taxonomy based on their specification. Models certainly have precedent and descend from other methods, but this doesn't necessarily dictate what parameters a model requires. While it might make sense to include `fit_intercept` in this method, any parameters specific to `LinearRegression` need to find a different home. The next best location is in the model object. On the other hand, the declarative approach has no such issue since there is no distinction between the model and the fit. Hence, there is no explicit object constructor in the first place. More importantly, the formula interface accommodates details specific to different models without cluttering up the function signature. We also leave the implementation details of extracting the response and explanatory variables from the dataset to the function. Hence, the only thing we need to focus on is how to specify the model!

2.2.4 Probability distributions and taxonomies

Say we have a random variable X that is normally distributed according to $N(0,1)$. The general specification for a Gaussian variable is indistinguishable from a function signature: $N(\mu,\sigma)$. If we want a binomially distributed variable instead, we denote this as $B(n,p)$. Again, the notation looks like a function signature. Painfully obvious, yet not all software representations are consistent with this notation.

Users of R know that sampling from a distribution follows the naming convention `"r", distribution` in EBNF [3], such as `rbinom` for the binomial distribution. For example, generating 100 samples from a 10 trial binomial distribution with $P(success) = 0.5$ looks like `x ←rbinom(100, 10, .5)`. Aside from random sampling, other properties of a probability distribution are obtained by using a different prefix in the function name. For each distribution, the density function starts with ″d″, the distribution function uses ″p″, while ″q″ is used for the quantile function. This convention is easy to understand and easy to remember.

In an object-oriented world, naming conventions are often eschewed in fa-

[3] Extended Backus-Naur Form is a common syntax for describing syntax.

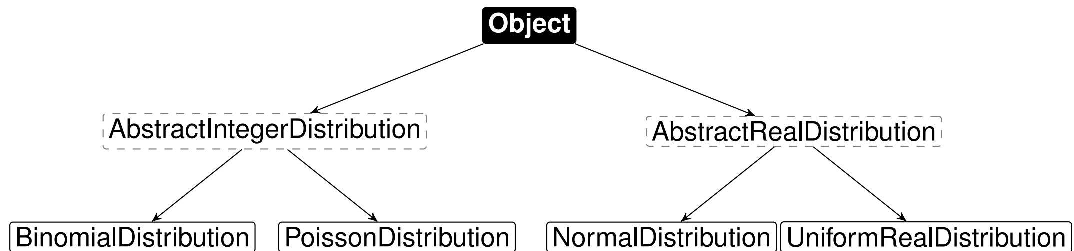


FIGURE 2.3: Java class hierarchy for distributions. `Object` is the root class for all Java objects. Dashed boxes indicate abstract classes that cannot be instantiated.

vor of explicit class hierarchies. This is the case with Java, where the binomial distribution is part of a larger hierarchy (see Figure 2.3) that includes both interfaces and abstract base classes across separate branches for discrete and continuous distributions. Interfaces declare a set of operations that classes must implement, while abstract base classes implement common operations. This class hierarchy can give a sense of order, but like other man-made taxonomies it's dangerous to treat it as an absolute truth. Class hierarchies are supposed to facilitate modularity and reusability, particularly around method implementations. One can argue that the mean is common to all distributions, so it can be implemented in the abstract base class. But why does the mean need to be associated with a distribution? Can't the mean be calculated against any vector of data, not just those associated with a distribution?

Ultimately, this is a core problem of class hierarchies: their imposition of structure incorrectly presume an optimal design. Taxonomies can work well when all elements to be classified are known in advance. Each element must also cleanly belong to a group in the taxonomy. But things quickly get messy when exceptional cases are encountered. The chimeric platypus wreaked havoc in the Linnaeun taxonomy for decades [22]. In software these taxonomies can complicate the design of software by imposing artificial boundaries between sets of properties and operations. Like the platypus, mathematical objects end up getting tangled up in these taxonomies with properties strewn across multiple branches. The consequence is that more time is spent fixing the taxonomy than doing data science.

Earlier we showed how to sample from the binomial distribution. Let's turn our attention to the normal distribution and summarize the empirical distribution for a given realization of a random variable from $N(0, 1)$. The R approach is to apply a chain of functions to transform a simple model specification into the summary object.

```
> summary(ecdf(rnorm(100)))
Empirical CDF:    100 unique values with summary
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
-2.8350 -0.3155  0.2129  0.1376  0.7431  2.0880
```

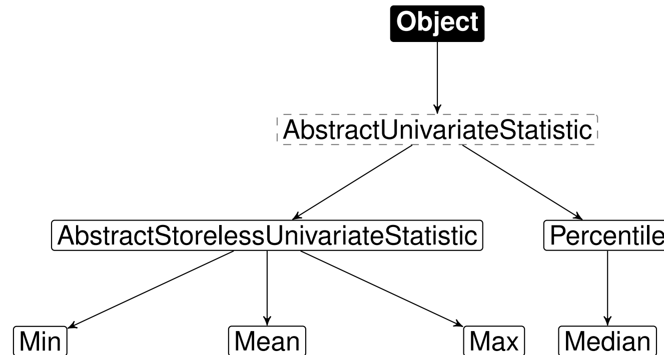


FIGURE 2.4: Java class hierarchy for different statistics. In OOP, even simple functions and metrics are forced into a class hierarchy, which ultimately have no relationship with the underlying mathematical concepts. In this example, the `AbstractStorelessUnivariateStatistic` is an implementation artifact unrelated to univariate statistics.

The equivalent Java code is

```
NormalDistribution n = new NormalDistribution();
double[] samples = n.sample(100);
EmpiricalDistribution d = new EmpiricalDistribution();
d.load(samples);
d.sampleStats();
```

Clearly the Java version requires more work. To be fair some of the work is associated with static typing, [4] but most of the code is related to creating and manipulating objects. The resulting algorithm is thus quite different from the original mathematical description. The question to ask yourself is: does the extra structure aids in modeling and memory retention or is it simply an artifice that impedes the translation of ideas into code?

Another wrinkle with the Java version versus the R summary is that the computed statistics don't include the median. The median is a curious statistic because it's calculation cannot be described directly as an algebraic expression. Instead, the median m is defined declaratively, insomuch that the implementation is implied:

$$P(X \leq m) \geq \frac{1}{2} \text{ and } P(X \geq m) \geq \frac{1}{2}$$

In R the median is given by an eponymous function. The Java approach is to introduce another class hierarchy [2] depicted in Figure 2.4. Computing the median requires creating an object that represents the median and then calling

[4] With static typing, the type of a variable is specified explicitly, which enables the compiler or interpreter to check for type compatibility. In functional languages like ML and Haskell, this information can be inferred via Hindley-Milner type inference.

IntegerDistribution
double cumulativeProbability(int x)
double getNumericalMean()
double getNumericalVariance()
int sample()
int[] sample(int sampleSize)

RealDistribution
double cumulativeProbability(double x)
double density(double x)
double getNumericalMean()
double getNumericalVariance()
double sample()
double[] sample(int sampleSize)

FIGURE 2.5: Comparison of operations for two interfaces in the Apache Commons Math library. The primary difference is that types change from `int` to `double`.

a method to compute it on a set of data. From an ontological perspective, there might be reason to organize these different statistics into a taxonomy. However, what is the value when you simply want to compute a statistic on data?

```
Median median = new Median();
median.evaluate(samples);
```

Notice how much extra structure is required in the OOP approach. For a single statistic it might seem insignificant, but imagine doubling the number of lines in your complete codebase! One implication is that it takes more effort to change the design if new requirements crop up. We've already established that the beginning of data science projects are more uncertain than comparable software projects, so the likelihood that an initial design will persist through the code lifecycle is low.

Why does Java need such a complex taxonomy of classes? A common theme in software development is the DRY principle, or don't repeat yourself. This is another way of saying that a software design should maximize modularity and reuse. If your starting point is a class hierarchy, then common attributes and methods naturally group together, while bespoke elements appear as terminals in the hierarchy. It's easy to see that as the number of types increase, the depth of the hierarchy will increase to capture shared attributes. One complication with Java is that primitive types are not objects but are distinct entities outside the class hierarchy. For mathematical functions this means that a concept like a random number generator will have a different signature if it produces continuous numbers versus integer values. We see this in Figure 2.5, where two parallel class hierarchies exist to support continuous and discrete distributions. Even though `sample()` is associated with both the `IntegerDistribution` and `RealDistribution`, they are distinct methods. Indeed, while the operations are conceptually the same, most of the methods defined by these interfaces have different signatures.

If this all seems overwhelming and a bit of a distraction from data science, that's the point I'm making. The difference between a professional programmer versus a data scientist is that programmers are paid to build systems, whereas data scientists are paid to build models. Same tools, different goals. Data scientists still need to write good code, but for different reasons. Object-

oriented languages are designed for system development and GUIs, where behaviors and operations are planned in advance. The scientific method is much messier and often requires changing approaches, comparing methods, and running the same model with varying parameters or different sets of data. For the data scientist, good code is quick to write, easy to modify, and easy to tie back to the underlying mathematical model. This long-winded discussion about Java and OOP is a siren song for those looking to create complex class hierarchies in R.

2.3 Linear algebra

The last section discussed how object-oriented programming introduces unnecessary structures that contradict the structures inherent in mathematics. This isn't to say that all structure is bad. The world of mathematics is replete with interesting and distinct structures. Many fields define specialized entities, or *types*, for which certain operations apply. Linear algebra is one such field where properties and operations exist only for specific types. [5] For example, the determinant is undefined for vectors. It seems reasonable to prevent functions from operating on invalid types. The object-oriented approach uses a class hierarchy to define valid operations on specific types with optional static type checking to ensure compile-time argument compatibility.

All that is really needed to achieve type safety is for operations to be aware of the types they can operate on. Focusing just on type awareness means we can decouple operations from classes. The S3 class system works this way. Described as an object-oriented type system, how S3 *dispatches* functions actually resembles functional programming. For example, the `determinant` is implemented only for objects of class `matrix`. This type awareness is instilled in `UseMethod`, which searches for a specially named function based on the type of the first argument. Given a function `determinant` and an argument `x`, the evaluated function is `"determinant", class(x)` in EBNF, where `class(x)` is the class, or type, of the object. If `determinant` is applied to a variable whose class isn't `matrix`, an error will result.

```
> determinant(1:3)
Error in UseMethod("determinant") :
  no applicable method for 'determinant' applied to
  an object of class "c('integer', 'numeric')"
```

In many ways, dispatching dictates how functions are defined and used. R has many dispatching systems that tend to be conflated with class/type systems, including S3, S4, and ReferenceClasses, all of which are built-in.

[5] I use type and class interchangeably, though "class" is typically reserved for object-oriented programming.

We'll touch on a few of these dispatching approaches in this section as well as compare them to the `pandas` package [35], which implements data frames in Python.

2.3.1 Dispatching the dot product

For two vectors **a** and **b**, their inner product is $\mathbf{a} \cdot \mathbf{b} = \sum_i a_i b_i$. The mathematical representation is unencumbered by the mechanics of a programming language. We usually don't even consider the algorithmic details of the dot product. Rather, we just *know* that $\mathbf{a} \cdot \mathbf{b}$ is the dot product of two vectors and yields a scalar. Since the operator is only defined for vectors, using anything else is non-sensical. When implementing the dot product in a programming language, this knowledge must be codified to ensure proper behavior. Not only do we need to consider how the function will handle different arguments, but how to call the function as well.

In R, the dot product is represented by `%*%`, where any function that begins and ends with `%` acts as an infix operator. This syntax produces an R expression very similar to the mathematical expression: `a %*%b`. How does R know what function to call? [6] For native R functions, the mechanics of dispatching consist of the following steps: lookup the function, and if it exists, call it. To illustrate let's implement an R function that simply uses the `%*%` operator.

```
dot ← function(a,b) a %*%b
```

Suppose we want to prevent invalid arguments from calling the underlying function. In the age of web services and distributed computing, this situation is common. We may write an R wrapper to exexute a job on a virtual cluster in the cloud. Doing so not only costs money but takes time, and the last thing you want is to kick off a big job only to discover the process died due to some silly data issue *the next morning*. In this toy example two common problems can arise. First, the vectors may not be numeric. This is an example of having a type mismatch. The second scenario is when the vectors have incompatible lengths. Neither of these situations occur in our pure mathematical utopia, but the real world is riddled with such issues. To solve the first problem, we can explicitly check the type of each argument.

```
dot ← function(a,b) {
  if (!(class(a) %in% 'numeric' && class(b) %in% 'numeric'))
    stop("Both arguments must be numeric vectors")
  a %*% b
}
```

This is slightly different from the S3 approach, where only the type of the first argument can be matched.

```
dot ← function(a,b) UseMethod("dot")
```

[6] The `%*%` operator is a primitive function that calls Fortran code. Thus the details are slightly different from a native R function, but conceptually the mechanics are the same.

```
dot.numeric ← function(a,b) a %*%b
```

The dispatching is handled by `UseMethod`, which maps a function to a type. This can be seen by looking at the result of

```
> methods(dot)
[1] dot.numeric
```

The output indicates that whenever `x` is a numeric vector, `dot(x,y)` will call `dot.numeric(x,y)`. But this doesn't protect against the type of the second argument. Errors won't be discovered until the underlying implementation is executed. Languages that use dynamic typing are usually weakly typed and have this issue including R and Python. The implication is that type safety must be implemented explicitly in a preamble to the function. This isn't necessarily bad, since it balances fast prototyping with robustness later on, when a program is ready for production.

With the S4 class system, the types of all function arguments can be specified, emulating *strong typing*. In contrast to weak typing, strongly typed languages disallow any type mismatches. However, the price paid for this explicit type safety is unintuitive syntax.

```
setGeneric("dot.s4", function(a, b) {
  standardGeneric("dot.s4")
})

setMethod("dot.s4", signature(a="numeric", b="numeric"),
  function(a, b) { a %*% b })
```

With S4, we can prevent calls with incorrect argument types from executing the function.

```
> dot.s4(1:3, letters[1:3])
Error in (function (classes, fdef, mtable)  :
  unable to find an inherited method for function 'dot.s4'
  for signature '"integer", "character"'
```

Checking whether inputs are well-formed is common in many native R functions. One advantage of static type checking is that the compiler or interpreter does this for you. In R, many functions have a de facto preamble devoted to type checking. These preambles often muddy the function implementation. Type checking logic quickly mixes with dispatching logic and even model logic, so it's unclear what the purpose of a function is. The built-in `optim` function suffers from such an identity crisis. In Chapter 14 we'll critique this function to explore other issues in its implementation. The biggest drawback of strong typing is that if you commit to it too early in the model development process, you may end up giving yourself more work. Until you know exactly how the model will run as a repeatable process, it's guaranteed that functions and components of the model will change. Adding type constraints this early on can often break code in places where weak typing is more permissive.

Type checking cannot protect against all data integrity issues. It is powerless when two vectors have incompatible lengths. Type checking also won't prevent you from attempting to invert a singular matrix. Situations like this require additional constraints on the variables, typically added as extra conditional expressions in the function preamble. For the dot product, the lengths of the two arguments must be equal.

```
dot ← function(a,b) {
  if (length(a) != length(b)) stop("Incompatible lengths")
  a %*% b
}
```

In other cases, the lengths of vectors don't necessarily need to be equal. Rather, they need to be compatible according to vector recycling rules. [7] One way to check this is by computing the greatest common divisor of the lengths, which must be greater than one.

A dispatching system consistent with functional programming is provided by `lambda.r`, which is discussed more thoroughly in Chapter 4. This library introduces new syntax for optionally specifying type constraints that dictates what arguments are supported by the function. In this way, function authors can decide if they want the flexibility and simplicity of a standard function or strong typing similar to what is provided by S4 or ReferenceClasses. The philosophy is that model logic should be segregated from data manipulation logic. `Lambda.r` focuses on transforming common data integrity logic into declarative statements. This approach frees data scientists from needlessly worrying about data integrity details of algorithms.

```
dot(a,b) %::% numeric : numeric : numeric
dot(a,b) %as% a %*%b
```

This type constraint is similar to the conditional expression, except it is declarative in nature. It also encompasses the return type of the function, which is useful not only in type inference systems, but as self-documenting code. Another benefit is that the type constraint can be added *as necessary* without affecting the body of the code. When we implement models, we need to think about both the correctness of the implementation and also how people interact with our functions. Even if you are not developing a formal package for publication, these considerations are relevant.

2.3.2 Matrix multiplication as object-oriented programming

The mechanics of matrix multiplication extend the dot product for vectors. Given two matrices A and B with product $C = AB$, each element of C is defined $C_{i,j} = A_{i,*} \cdot B_{*,j}$. In other words, each element $C_{i,j}$ is the dot product of the ith row vector of A with the jth column vector of B. The notation is

[7] By default, R will repeat the elements of a smaller vector to match the length of a longer vector if the shorter length divides the longer length.

consistent with what we expect multiplication to look like. In R the notation is the same as with the dot product.

```
> a ← matrix(1:6, ncol=3)
> b ← matrix(6:1, nrow=3)
> a %*% b
     [,1] [,2]
[1,]   41   14
[2,]   56   20
```

How does a typical object-oriented framework represent this operation? The answer is dictated by the mechanics of dispatching, which we can see from the Python package `pandas`. Recall that object-oriented programming typically defines classes with associated methods, or bound functions. When using an instance of a class, the first argument to a method is the object itself. This poses an ontological dilemma: how should binary operators be represented? Purists of OOP tend to favor internal consistency over consistency with mathematical notation. For two `DataFrame`s `a` and `b` that represent matrices, the answer is `a.dot(b)`. [8] But this is inconsistent with mathematical notation. Worse, algebraic operations are conflated in the class definition of `DataFrame` with operations for loading and exporting the data contained in the structure. This organizational strategy contributes to the ambiguity of what a class is supposed to represent. Is it meant to represent a pure mathematical computation, a container of data, or both?

It's possible to model matrices this way with ReferenceClasses as well. ReferenceClasses implement object references for instance variables while largely preserving S4 syntax for defining classes [**?**].

```
Matrix ← setRefClass("Matrix",
  fields="data",
  methods=list(
    initialize=function(x, ncol=NULL, nrow=NULL) {
      if (is.null(ncol) && is.null(nrow))
        stop("Either ncol or nrow must be set.")
      if (is.null(ncol)) ncol ← length(x) / nrow
      if (is.null(nrow)) nrow ← length(x) / ncol
      if (nrow * ncol != length(x)) stop("Incorrect dimensions")
      data ↞ matrix(x, ncol=ncol, nrow=nrow)
    },
    dot=function(y) data %*% y$data
  )
)
```

Using this class looks similar to the `pandas` formulation.

```
> a ← Matrix$new(1:6, ncol=3)
> b ← Matrix$new(6:1, nrow=3)
```

[8] One can parry that it ”reads” the same, but we've seen elsewhere the negative structural impact of this approach.

```
> a$dot(b)
     [,1] [,2]
[1,]   41   14
[2,]   56   20
```

What this example shows is that programming languages allow us to model computations however we want. It's up to the data scientist to ensure that the approach is efficient and faithful to the underlying mathematical representation.

For completeness, here is the same implementation using the `R6` package [12]. This package attempts to create a syntax for class definition that is more intuitive and similar to the syntax of Java or Python.

```
Matrix ← R6Class("Matrix",
  public=list(
    data=NULL,
    ncol=NULL,
    nrow=NULL,
    initialize=function(x, ncol=NULL, nrow=NULL) {
      if (is.null(ncol) && is.null(nrow))
        stop("Either ncol or nrow must be set.")
      if (is.null(ncol)) ncol ← length(x) / nrow
      if (is.null(nrow)) nrow ← length(x) / ncol
      if (nrow * ncol != length(x)) stop("Incorrect dimensions")
      self$data ← matrix(x, ncol=ncol, nrow=nrow)
    },
    dot=function(y) self$data %*% y$data
  )
)
```

While the mechanics of class definition is different, the usage is identical.

```
> a ← Matrix$new(1:6, ncol=3)
> b ← Matrix$new(6:1, nrow=3)
> a$dot(b)
     [,1] [,2]
[1,]   41   14
[2,]   56   20
```

Both of these approaches gives us a syntax in R that feels more like object-oriented code. You might protest and ask why you would do such a thing in the first place. Again, that is precisely the point. These counterexamples are meant for you to question their utility in terms of implementing mathematical ideas. In the right places, object-oriented designs can add value. It's important though to know when such an approach is counterproductive.

2.3.3 Matrix factorization and collaboration

It is often said that data science is a collaborative effort. What's interesting about collaboration is that it demands coordination. Working alone is simple

from a process perspective. Adding a second person immediately requires effort to coordinate work and manage changes to models and code. Functional programming is useful in this context since functions are discrete entities. In object-oriented programming the atomic unit is the class, since it cannot be divided into smaller constituents. To see why, what happens if you separate an instance variable from a class? Obviously it's no longer a part of the class, which means it cannot be removed. The more properties and methods in the class, the harder it is to collaborate. The reason is that methods tend to operate on the same shared internal state of the class. Changes to an instance property can propagate to multiple methods. Hence, when multiple people work on the same class, coordination issues quickly arise. Java addresses this issue by using interfaces to separate properties and method implementations that are class specific. At other times, class hierarchies attempt to create modularity. Python, on the other hand, has the concept of mix-in to help solve this problem.

In systems development, an implementation is singular, insomuch that only one algorithm is created. Over time the algorithm may change, but effectively there is only one at any given time. In computer science, mathematics, and data science, multiple algorithms exist to solve the same problem. For example, there are numerous algorithms to sort a list. The same is true of matrix inversion. There is no one right way to invert a matrix. Depending on the problem, the LU decomposition might be the best approach, whereas for a sparse matrix, another method might be suitable. In cases where a matrix is singular but you still want an inverse, a pseudoinverse can be appropriate.

Example 2.1. Suppose Alice is the author of our matrix class and chose to use the LU decomposition to invert the matrix. Bob comes along and wants to use Newton's method. Where does this go? Bob might propose adding a new method `inverseNRM` that implements the algorithm. Bob might also suggest adding a method parameter to the current inverse to choose the algorithm. This could work, but Alice notes that Newton-Raphson is an approximate method, so an extra argument is required to specify the number of iterations or tolerance to use in the calculation. The ellipsis argument can be used to pack these arguments together, but Alice doesn't want to update the package every time someone wants a new matrix inversion method. Bob agrees and wonders if a subclass is better. He could name it `MatrixNRM` but doesn't like creating a whole new subclass just to overwrite one method. Besides, the same thing could happen with a linear solver, which would lead to a mess of classes.

Alice has a flash of inspiration and decides to define a new class that represents the LU decomposition. Then users pass their matrix object into the constructor and get the inverse from this new object. This approach handles any number of matrix factorizations and algorithms for finding the inverse. Now the problem is how to communicate this change to all the users of her package. This is the approach of the Apache Commons Math package. By creating a separate class for the LU decomposition, the decomposition

is decoupled from the matrix. The object-oriented way decouples functions from classes, though new classes must be created to house the orphaned function!

□

2.3.4 The determinant and recursion

Recursion is a common theme of mathematics and computer science. Functional programs tend to use recursion more often than imperative programs that tend to favor loops. From a data science perspective, recursive functions are useful because they can simplify certain types of algorithms. Iterative algorithms, can often be simplified by implementing a single iteration step first. A second, higher level function can mediate the sequence of steps. This description essentially describes the iterated function application that $fold$ mediates. The Fibonacci sequence, the Newton-Raphson method for root finding, along with the pseudoinverse of a matrix can be formulated this way.

The determinant is a property of a matrix that among other things tells us whether a matrix A is invertible. Unlike the other quantities discussed, the determinant $\det(A)$ is usually not defined symbolically. Instead the determinant is typically described algorithmically. For example, given a matrix
$A = \begin{pmatrix} a & b & c \\ d & e & f \\ g & h & k \end{pmatrix}$
the calculation of the determinant is described thus [30]:

> We go across the first row of the matrix A, $(a\,b\,c)$. We multiply each entry by the determinant of the 2×2 matrix we get from A by crossing out the row and column containing that entry. Then we add and subtract the resulting terms, alternating signs (add the a-term, subtract the b-term, add the c-term).

In symbolic form, it's easy to see the self-referential nature of the definition. Like the Fibonacci sequence, each term refers to itself multiple times.

$$\det(A) = a \det \begin{bmatrix} e & f \\ h & k \end{bmatrix} - b \det \begin{bmatrix} d & f \\ g & k \end{bmatrix} + c \det \begin{bmatrix} d & e \\ g & h \end{bmatrix}$$

Computing the determinant is a classic recursive definition despite not being described that way. In essence, the determinant can be defined

$$\det(A) = \sum_i -1^{i+1} i \det(A_{-1,-i}),$$

where $A_{-i,-j}$ is a matrix formed by removing row i and column j. For this definition notice that native vectorization can't really help us much. That's okay because this definition is easily implemented using $fold$.

```
det ← function(A) {
  fold(1:ncol(A), function(i,s) s + (-1)^(i+1) * i * det(A[-1,-i]))
}
```

Alternatively we can use the sum function in conjunction with *map*. We'll see in Chapter 7 why this is an equivalent expression.

```
det ← function(A) {
  sum(sapply(1:ncol(A), function(i) (-1)^(i+1) * i * det(A[-1,-i])))
}
```

In the case of the determinant, the halting criterion is dictated by the size of the matrix. For algorithms like Newton-Raphson, additional information must be provided to tell the algorithm when to stop. A common approach is to iterate over an explicit sequence of ordinals. This can be combined with a tolerance to terminate early once an acceptable level of convergence has been achieved.

2.4 Calculus

So far we've seen mathematical functions that operate on specific types of objects. Full of surprises, the mathematical world has a bewildering array of functions and operators, including higher-order functions. Recall that these functions operate on other functions. A related concept is the first-class function, which means a function can be treated as a value. The derivative, integral, and their variants are all higher-order functions since they take functions as arguments and also return functions. Consequently, the argument to these functions is a first-class function.

2.4.1 Transforms as higher-order functions

The derivative and integral both take functions as operands so conceptually these functions are being treated as data. The Leibniz notation for the derivative hints at this concept: $\frac{df}{dx} \equiv \frac{d}{dx}f$. Conceptually this is no different from writing the derivative as a function $d(f) \equiv \frac{d}{dx}f$ for the univariate case. The derivative not only takes a function as input but returns a function. Take for example the polynomial function $f(x) = ax^3 - bx + 4$. When referring to this function as f, we are implictly treating it like a first-class entity. This becomes explicit when we apply the derivative to f:

$$f' = \frac{d}{dx}f.$$

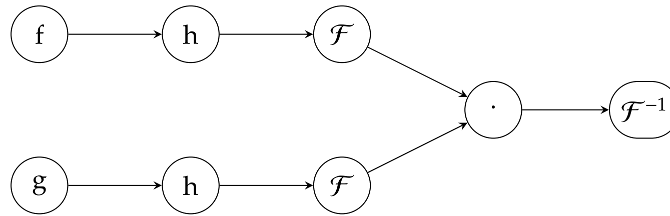


FIGURE 2.6: Convolution as a computational graph

The indefinite integral is like the derivative since it both takes a function as input and returns a function. From calculus we know that the integral is the inverse of the derivative and vice versa. We can think of this pair of functions as a transform and its inverse transform. This concept is useful in numerous fields, including data science, computer science, and electrical engineering. It is often easier to operate on an equation in a transformed space, so you need the inverse transform to recover the result in the correct space. Convolution is an example, where convolution in the time domain becomes multiplication in the frequency domain, and vice versa. More formally, for a Fourier transform $\mathcal{F}$,

$$\mathcal{F}\{f \otimes g\} = \mathcal{F}\{f\} \cdot \mathcal{F}\{g\},$$

and

$$\mathcal{F}\{f \cdot g\} = \mathcal{F}\{f\} \otimes \mathcal{F}\{g\}.$$

The same holds for the Laplace transform.

Suppose there's additional work to do in the frequency domain, which is embodied by a function h. Our expression becomes

$$\mathcal{F}\{h \circ f \otimes h \circ g\} = \mathcal{F}\{h \circ f\} \cdot \mathcal{F}\{h \circ g\}.$$

To recover the transformed convolution simply requires applying the inverse transform on both sides to get

$$h \circ f \otimes h \circ g = \mathcal{F}^{-1}\{\mathcal{F}\{h \circ f\} \cdot \mathcal{F}\{h \circ g\}\}.$$

This sequence of operations boils down to multiple function composition, which can be expressed as a simple computational graph (see Figure 2.6).

In data science, a common transform is the Box-Cox transform. When data does not satisfy normality assumptions, this transform can help reshape the data to minimize skew and produce normal residuals. A transformed response is then fit to the predictors. The resulting model is thus in the transformed space, so when applied to new data, the result must be transformed with an appropriate inverse transformation. Figure 2.7 depicts the overall process. To make this more concrete, let's analyze the built-in `trees` dataset. We want to fit a linear model Volume $\sim \beta_0 + \beta_1$Height $+ \beta_2$Girth $+ \epsilon$. Doing

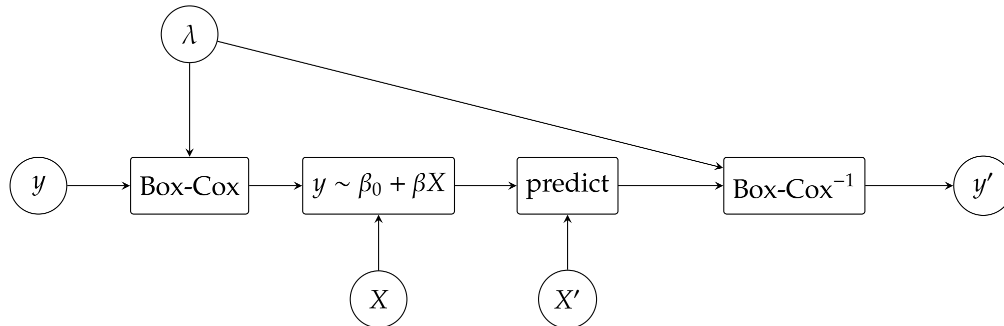


FIGURE 2.7: Regression analysis with Box-Cox transform as a computational graph. To predict a value, the model is applied to new data X' followed by an inverse Box-Cox transformation.

so yields non-normal residuals, suggesting that a Box-Cox transformation is appropriate (see Figure 2.8). The Box-Cox transform is defined

$$BC(y;\lambda) = \begin{cases} \frac{y^\lambda - 1}{\lambda} & \text{if } x \neq 0 \\ \ln y & \text{if } x = 0 \end{cases}$$

Ignoring the piecewise definition at 0, a naive version of the Box-Cox transform is

```
bc_xform ← function(y, lambda) (y^lambda - 1) / lambda.
```

We also need the inverse transform.

```
bc_iform ← function(y, lambda) (lambda * y + 1)^(1/lambda)
```

To get the actual value, we find the maximum log likelihood using

```
boxcox(Volume ~ log(Height) + log(Girth), data=trees,
  lambda=seq(-0.25, 0.25, length=10)).
```

A simple way to use the Box-Cox transformation applied to volume is to create a new variable.

```
trees$V1 ← bc_xform(trees$Volume, -0.08333333)
```

Now fit the model.

```
model.bc ← lm(V1 ~ log(Height) + log(Girth), data = trees)
```

As expected, the residuals of the new model (Figure 2.8) look much better. In a real analysis we would be more rigorous in verifying their normality. Finally, let's create some new data and predict the tree volume.

```
> nd ← data.frame(Girth=10, Height=70)
> bc_iform(predict(model.bc, newdata=nd), -0.08333333)
[1] 14.63051
```

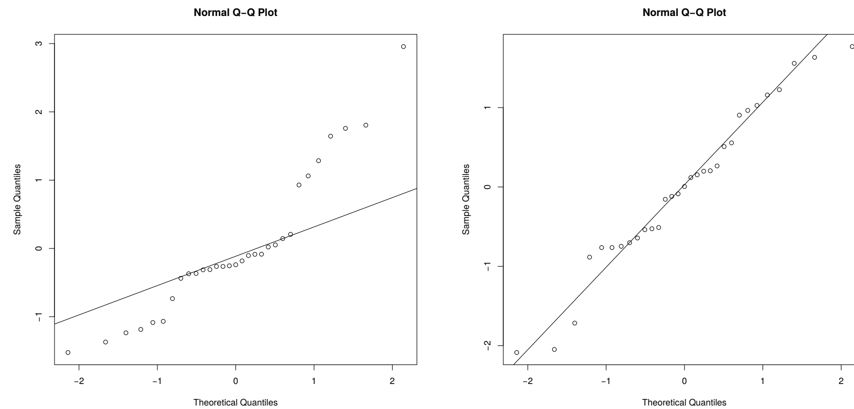


FIGURE 2.8: Comparison of residuals before (left) and after (right) a Box-Cox transformation. New predictions on the model must have inverse transform applied to get correct value.

2.4.2 Numerical integration and first-class functions

A well known application of Monte Carlo simulation is numerical integration. By randomly throwing darts on a grid, the integral of a function is the proportion of darts under the curve (represented by the function) versus the total thrown. This approach predates computers and was first explored by Buffon and his eponymous needle to estimate π. Buffon's method was to draw parallel lines on a plane separated by a constant distance d. Using a needle of length $l < d$, Buffon determined the probability of the needle crossing 0 or 1 lines as

$$p_0 = 1 - 2r\theta$$
$$p_1 = 2r\theta$$

where $r = l/d$ and $\theta = 1/\pi$ [45]. Solving for θ and then substituting, we get $\hat{\pi} = \frac{2r}{p_1}$. Hence, the value of π can be determined by counting the number of times the needle crosses a line. As the number of trials increases, the estimate converges to the true value of π.

As a thought experiment Buffon's needle is quite compelling, but conducting the experiment ourself seems a bit tedious. Thankfully we have computers to do our bidding, and they can run the simulation for us in a fraction of the time. Instead of using a needle, we can randomly simulate coordinates on a plane. Then we can estimate π by counting the points inside the unit circle. The area of a circle is πr^2. By carefully choosing random values in the range $[0, 1]$, the area reduces to $\frac{\pi}{4}$. This can be implemented using

```
g ← function(k) {
```

```
  n ← 10^k
  f ← function(x,y) sqrt(x^2 + y^2) <= 1
  z ← f(runif(n),runif(n))
  length(z[z]) / n.
}
```

We can get successively better approximations by iterating over a sequence of exponents.

```
a ← sapply(3:8, g)
> a*4
[1] 3.188000 3.126800 3.131800 3.143452 3.141161 3.141364
```

No matter the function we want to integrate, the algorithm is essentially the same. We can generalize the function by passing the predicate as an argument. Doing so takes advantage of an anonymous function to simplify the logic.

```
g ← function(k, f) {
  n ← 10^k
  z ← f(runif(n),runif(n))
  length(z[z]) / n
}
```

As we increase the number of samples, our estimates begin to converge towards the true value of π.

```
a ← sapply(3:8, function(k) g(k, function(x,y) sqrt(x^2 + y^2) <= 1))
> a*4
[1] 3.224000 3.152800 3.145080 3.140936 3.142540 3.141622
```

2.5 Summary

Much of functional programming you already know from mathematics. Using these concepts in your code keeps the implementation closer to the mathematical expression. This makes the code easier to understand. Extending these concepts to the rest of your code improves modularity and reliability of the code.

Benefits aside, programs shouldn't blindly be implemented in a FP style. Code designed for their side effects often have a more intuitive implementation using object-oriented code, such as file descriptors or GUIs.

2.6 Exercises

Exercise 2.1. Implement standard deviation using a declarative approach

Exercise 2.2. Implement the dot product using a declarative approach

Exercise 2.3. Implement the determinant using an imperative approach. What did you notice is different about the implementation?

Exercise 2.4. Implement the Newton-Raphson algorithm for finding the pseudoinverse of a matrix.

3

Functions as a lingua franca

It is no secret that modeling data involves a lot of data processing. The mental effort is colloquially split 20% to 80% between data processing and modeling. Program code is curiously the opposite, with data processing taking up the majority of the effort and lines of code. This is due to the steps involved and the inherently messy nature of data versus the pure and ideal world of mathematical models. As data moves between libraries, components, and systems, the formats and data structures are often incompatible. Coercing these disparate pieces of software to cooperate requires ad hoc data transformation to mold data into the correct structure. For simple exercises this isn't much of a problem. More ambitious projects have greater complexity and thus presents a greater challenge. Collaboration is often necessary to complete these larger data science initiatives. Good code structure and modularity are no longer nice to have but are essential to project success. Performance tuning also requires modular code, making it easier to isolate bottlenecks in a model. With functional programming, just a few concepts solve the bulk of application design problems seen in data analysis. This small kernel of concepts and techniques is fast to learn. Model implementations are usually streamlined in the process. The end result is more time doing science and less time wrangling data.

As a paradigm, functional programming is not language-specific. Rather, functional programming is a philosophy for structuring programs based on function composition. In addition to function composition, functional programming comprises a (mostly) standard set of syntactic and semantic features. Many of these concepts originate from the lambda calculus, a mathematical framework for describing computation via functions. While each functional language supports a slightly different set of features, there is a minimal set of overlapping concepts that we can consider to form the basis of functional programming. This set consists of first-class functions, closures, and higher-order functions. [1] Once these concepts are mastered, it is easy to identify situations that can benefit from their use in any language. In principle this is the same as learning the syntax of a new language: you begin by looking for the delimiter for statements, expressions, and blocks as well as how to create variables and call functions. These conceptual building blocks

[1] Other concepts include lazy evaluation, currying, partial application, pattern matching, and tail recursion. Readers interested in exploring these topics are encouraged to read [].

of a language act as a lingua franca irrespective of the specific language in question. The same is true within a language paradigm. The semantics of classes and objects in an object-oriented language act as a lingua franca in the world of object-oriented programming, while the function exclusively serves this purpose in a functional programming paradigm.

Loosely the dual of the previous chapter, this chapter focuses on functional programming concepts and less on the related mathematics. Since iteration is so fundamental to data science, a brief overview of vectorization is presented in Section 3.1. Vectorized operations implicitly use *map* and $fold$ on their argument(s) and is a core piece of functional programming in R. Our everyday use of vectorization is what makes adopting additional functional programming concepts so compelling. Without native vectoriztion, a higher-order function must be used explicitly in conjunction with a first-class function to achieve the same vectorized properties. Any scalar function can become vectorized using a combination of the three canonical functionals.

Section 3.2 introduces the computer science concept of a first-class citizen, which is an essential ingredient to functional programming. Having first-class status is what allows functions to be passed as arguments to a higher-order function or as a return value. Functions in R are also referred to as closures, but not all functions are closures. Described in Section 3.3, these special functions reference variables in their enclosing scope. Closures often act as disposable adapters between two different function signatures. Every higher-order function defines its own specific signature for its function argument. These signatures are designed to be as general as possible, resulting in incompatibilities with many existing functions. For example, functions for *map* must be univariate, while functions to $fold$ must be bivariate. But functions don't always have signatures compatible with *map* and $fold$. Even if a function has a signature matching a particular higher-order function, it's rare that it also satisfies the signature of another higher-order function. Functions with a different signature need to be wrapped in another function that has the correct signature. Hence, closures act as the glue between arbitrary first-class functions and the specific function signatures that functionals expect.

Closures are the multi-function tool of functional programming. Usually a one or two line closure can solve most interface compatibility issues. Closures can also help generalize a function, by exposing a callback function to control certain aspects of a transformation. But without their higher-order function counterpart, closures aren't nearly as exciting. There's a special symbiosis between these two types of functions, and they must be used in conjunction to realize their full value. A *higher-order function*, also called a *functional* is any function that takes a function as an operand, returns a function, or both. Any generalized function becomes a higher-order function by virtue of adding a function to its signature. We've already seen how higher-order functions can mediate iteration and also transform functions. Higher-order functions can also act as function factories, producing the closures that another functional requires. This pattern is discussed in Section 3.4. Functions can have an array

of options to finely control their behavior. The more parameters, the less likely the defaults will suit your needs. Section 3.7 shows how to use a closure to codify these behavioral changes.

In general higher-order functions provide the machinery for transforming data in a repeatable way. Since data analysis involves many individual records having the same general structure (e.g. vectors or table-like structures), it is beneficial to divide data processing into a function that is responsible for manipulating a single record at a time, and a function that is responsible for iterating over all records. The first function is a first-class function passed to the second function, which is a higher-order function. This separation of concerns appears in Section 3.2 with `mean` and `apply`, respectively. Data can be partitioned in various ways, so R conveniently provides numerous *map*-like functions to mediate the iteration.

With a core set of semantic constructs, it is unnecessary to learn additional patterns and frameworks. And since functional programming concepts transcend specific programming languages, the same approaches can be used irrespective of the language. This is all the more important as state of the art machine learning algorithms are being developed in various languages. The end result is more time spent modeling and less on the dirty work of data transformation. To illustrate their use, we'll use a logistic regression as a backdrop. The `adult` dataset [33] assigns income as a binary class explained by numerous factors. For each person in the dataset, numerous demographic variables such as age, gender, education, race, native country are collected. Logistic regression is a good starting point for two class prediction problems. For pedagogical reasons, we'll use stochastic gradient descent to fit the regression instead of the closed form solution. Creating this model will show how functions serve so many different purposes.

The higher level lesson of the chapter is that functional programming facilitates change. Unlike pure software development, data science begins as an unstructured, exploratory exercise. Over time as hypotheses are validated, a process that starts off ad hoc becomes more structured. Once an initial model shows promise, changes are focused on tuning. Model inputs stabilize as do the corresponding transformations. At this point it can make sense to wrap the whole processing chain into a single pipeline function. Once the full pipeline is automated and repeatable, it's easy to create variations to tune the model. The whole model pipeline will itself be encapsulated in a function `income_pipeline` as shown in Listing 3.1. At its most basic, this function is responsible for loading data and fitting the model and evaluating its performance. As a first cut, we're just measuring in-sample performance. Over the course of the chapter, we'll fill in the details of the functions and improve the pipeline as our needs change and grow.

```
income_pipeline ← function(...) {
  df ← load_income()
  m ← logistic(income ~ age + education.num + hours.per.week,
      df, ...)
  p ← do_predict(m, df)
  list(model=m, performance=performance(p, df$income))
}
```

LISTING 3.1: A rudimentary model processing pipeline

3.1 Vectorization

A common description of R is that it is a vectorized language. Primitive operators and functions natively work on vectors instead of scalars. The result is compact code that more closely resembles the notation of vector math. In the general univariate case, many vectorized functions take the form $f : X^n \rightarrow Y^n$, where X represents an arbitrary input domain, and Y is an arbitrary range, possibly the same as X. Compare this signature to a generic unary scalar functions, which have the form $g : X \rightarrow Y$. Operating on a vector of values requires applying the function to each input element often as a loop. Leveraging vectorization is central to writing good R code. Loops are often unnecessary since they are implied in each operation. Like mathematical vectors, R vectors have specific behaviors and properties that can be exploited. This section highlights some basic concepts to facilitate the discussion for the remainder of the chapter. a full treatment of the mechanics of vectorization in given in Chapter 5.

We saw in Section 2.2 that statistics like the mean and covariance typically assume vector arguments (in the guise of random variables). These statistics take the form $f : X^n \rightarrow Y$, since they aggregate a set of values into a single value. Metrics like distance behave the same way, except they are bivariate and look like $f : X^n \times X^n \rightarrow Y$. Iterative methods, such as optimization and simulation, also rely heavily on sequence data types. The most obvious effect of vectorization is that algebraic operations become simpler. For example, normalizing a dataset x can be simply accomplished using `(x - mean(x)) /sd(x)`. Notice how this function behaves the same for all x, so long as $|x| > 1$. Compare this to a non-vectorized approach that must explicitly subtract the mean and divide by the standard deviation for each value in x. This is similar to how a covariance matrix is simpler when described in matrix notation over individual elements. In element-wise, or scalar, form the covariance of two vectors is

$$\Sigma_{i,j} = \frac{1}{n-1} \sum_k (x_{i,k} - \mu_i)(x_{j,k} - \mu_j).$$

Constructing the complete covariance matrix requires iterating over all the rows and columns of the matrix. For a set of vectors, matrix notation describes the complete covariance matrix as

$$\Sigma = \frac{1}{n-1} X^T Q X,$$

where Q is a projection operator that subtracts the mean of X from each series. This second definition is clearly more compact and efficient in conveying the meaning of the operation. We are trying to achieve this same compactness and clarity in our code.

Vectorization isn't limited to mathematical operations. Indeed, most built-in functions in R have vector semantics, extending this notational efficiency throughout the language. Consider parsing operations that frequently use regular expressions to find elements that satisfy a specific pattern. In languages like Python and PERL, regular expressions operate on single strings. To operate on a list of strings requires looping over each element. In R regular expressions are applied to a vector, so the same task can be accomplished in a single operation. A toy example is extracting all elements of the `education` column of the `adult` dataset that begin with 'B' or 'M' using `grep`, such as

```
> grep('^[BM]', adult$education).
```

The first argument is a scalar that represents the regular expression. This pattern is applied to each element of the input vector, which in this case contains entries like "10th", "Bachelors", and "Some-college". The result is a vector of ordinals pointing to all elements that match the pattern. This signature is different from the earlier algebraic functions and looks like $grep : string \times string^n \rightarrow \mathbb{N}^m$. Even with these simple examples, it's clear that iteration is at the heart of data science. The canon of higher-order functions all focus on specific ways to iterate over data. When a function is not natively vectorized nor preserve vectorization, these higher-order functions effectively transform it into a vectorized function. In other words, combining *map* and a scalar function yields $map : (g : X \rightarrow Y) \times X^n \rightarrow Y^n$.

Example 3.1. Before developing our model to predict income, it's useful to think about how we're going to evaluate the model. Some common measures include overall accuracy, precision, and recall. With these metrics we can also compute the F1 score and the ROC curve. These metrics can all be computed based on the confusion matrix. Vectorization simplifies the calculations, making them almost trivial. The approach is to compute the contingency table first. Then the different measures simply sum different elements of the matrix and divides by a different sum of cells. Hence, the whole operation uses *map* and *fold* implicitly. `sum` is an example of a *fold* computation since it is a binary operator with the signature $sum : \mathbb{R}^n \rightarrow \mathbb{R}$. Vector division is a binary *map* operation and looks like $/ : \mathbb{R}^n \times \mathbb{R}^n \rightarrow \mathbb{R}^n$. This simplifies into a univariate *map* operation by wrapping the two operands in a tuple. The signature thus

```
performance ← function(pred, obs) {
  cm ← table(pred, obs)
  list(confusion=cm,
    accuracy=sum(diag(cm)) / sum(cm),
    precision=cm[2,2] / sum(cm[2,]),
    recall=cm[2,2] / sum(cm[,2]))
}
```

LISTING 3.2: A function to summarize model performance

becomes $/ : \mathbb{R}^{n \times n} \rightarrow \mathbb{R}^n$. Coincidentally, this new signature is the same as the `diag` function.

□

3.2 First-Class Functions

A cornerstone of computer science, the lambda calculus defines functions as so-called *lambda abstractions* [13, 6], which is synonymous with anonymous functions. The identity function $I : X \rightarrow X$ is defined $\lambda x.x$. This lambda abstraction is equivalent to `function(x) x` in R. Function application works via β-reduction, which replaces variables in a function definition with concrete values. In the application $\lambda x.x[x = 2]$, all occurrences of x are replaced by 2 in the function body, yielding 2. The same works in R using parentheses to delineate expressions:

```
> (function(x) x)(x=2)
2
```

Anonymous functions are convenient for throw-away glue code. Other times, we want functions to stick around longer. An alternative is to name the function. A simple way to name a function is to assign the lambda abstraction to a variable, as in $I = \lambda x.x$. This is normally how we work with functions in R.

```
I ← function(x) x
```

Once this function is assigned to a variable we can use it like any other value. For instance, we can pass the function as an argument to another function, and we can store the function in a data structure, like a list. Functions that satisfy these properties are called *first-class*.

Both the lambda calculus and Turing machines treat functions like any other data that can be manipulated by a program. In the definition of the

lambda calculus, Λ consists of lambda terms that are either lambda abstractions (functions) or variables [6]. Consequently, functional languages treat everything as data. It's thus easy to take this for granted in R, but in languages like Java, functions are not first-class. The reason is one of design, as opposed to functionality, since even at the hardware level, data and instructions are ultimately both represented as a sequence of bits (data). Fundamentally, there is not much to distinguish functions from data. In Chapter 2 we saw that mathematics has a similar point of view. First-class functions are not only consistent with mathematics, they streamline model development by enabling the creation of ad hoc interfaces and connectors between operations in a pipeline.

Let's explore these three properties in more detail. All functions are first-class in R. [39] To see how functions behave like values, we define a univariate function that increments its argument as $\lambda x.x+1$. Also known as the successor function, this function is used to produce the Peano numbers. In mechanical terms, we are assigning a function to the variable named `succ`.

```
> succ ← function(x) x + 1
```

We've declared the variable `succ` and assigned a function as its value. This function can now be used like any other variable. As mentioned above, one requirement for being first-class is that a data structure must be able to store the function as an element. Both built-in functions and user-defined functions are first-class and can be stored in a list.

```
> some.funs ← list(sum, succ)
> some.funs
[[1]]
function (..., na.rm = FALSE)  .Primitive("sum")

[[2]]
function (x)
x + 1
```

Our list `some.funs` contains two functions as elements. We can extract one of the functions and assign it to a new variable. The new variable points to a function and can be applied to an input like the original function.

```
> increment ← some.funs[[2]]
> increment(4)
[1] 5
```

Functions can also be passed as arguments to other functions. This is common practice in R, and most users first encounter this with one of the `apply` functions, such as `sapply`, `mapply`, or `tapply`. The eponymous `apply` iteratively processes each element in an `array`, `matrix`, or `data.frame`. In two dimensions, an element is meant to be a row (column) of the table-like structure. The function passed to `apply` is sequentially applied to each row (column) in the data structure. When operating on columns, `apply` has the

mathematical signature $apply : X^{n \times m} \times \Lambda \rightarrow Y^m$, where Λ denotes a lambda abstraction (i.e. function). The actual signature of `apply` takes a data structure, the margin, which controls whether the iteration is along rows or columns, and a function that is applied to each row or column. Therefore, the function is treated as a value that is passed into `apply`, which satisfies one of the requirements for being first-class.

Let's turn our attention to the primary case study of the chapter. Suppose we want to predict income from the `adult` dataset [33], which is divided into two classes: less than or greater than $50k per annum. This dataset is a mix of categorical and continuous variables. For sake of simplicity, we'll focus on three continuous variables only: age, education level, and hours worked per week. A plausible first step in the analysis is examining some summary statistics. We can compute the mean of each variable using `apply`. Each argument passed to `mean` is thus a column of the `adult` dataset.

```
> cols ← c('age','education.num','hours.per.week')
> apply(adult[,cols], 2, mean)
          age  education.num hours.per.week
    0.2956388      0.6053726      0.4024232
```

Like its canonical brethren, the purpose of `apply` is to provide the machinery around iteration, which is a generalized operation. This implies that given constant data, the structure of the result of `apply` is the same, irrespective of the function used. The only requirement is that the functions must all have the same signature. The first-class function argument only needs to know how to process a single element (in this case a vector) instead of a set of elements. Segregating the mechanics of iteration from the model logic means that the same function can be used for a single vector or multiple vectors without modification or ceremony. As with *map* the first-class function argument is meant to be univariate. If the function takes more than one variable, a closure can be used to match the signatures. Section 3.6 elaborates on this technique. For now, let's see how the output structure is preserved by replacing the mean with another statistic, like standard deviation.

```
> apply(adult[,cols], 2, sd)
          age  education.num hours.per.week
    0.1868581      0.1715139      0.1259961
```

Both `mean` and `sd` have the same signature, namely $f : \mathbb{R}^n \rightarrow \mathbb{R}$. Any function with this signature will yield the same structure through `apply`, whereas a function with a different signature will produce a different output structure. For example, the successor function has the signature $succ : \mathbb{N} \rightarrow \mathbb{N}$. Using this function results in $apply : \mathbb{R}^{n \times m} \times \Lambda \rightarrow \mathbb{R}^{n \times m}$, instead of $apply : \mathbb{R}^{n \times m} \rightarrow \mathbb{R}^m$.

Data-dependent return types can be confusing to newcomers. One advantage of functional programming is that we can deduce output types by applying mathematical reasoning to our programs. Without writing any code, we can deduce the types and shape of data moving through functions. Reasoning about matrix dimensions is a similar exercise, except that now the operations

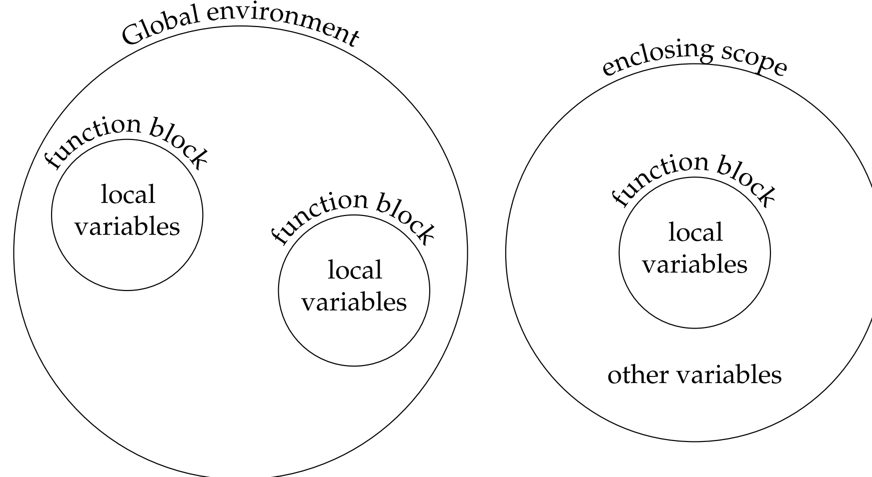


FIGURE 3.1: Simple functions versus a closure. Left: A standard function only has access to variables within its own scope. Any references to variables outside the function scope will be searched recursively until the global environment is reached. Right: Closures reference variables in their enclosing scope. This environment is bound to the function, even after the enclosing function has been executed. With lexical scoping, any variable not in the current scope will be recursively searched in the parent scope until the global environment is reached.

span more than just matrices. This concept is discussed more thoroughly in Part II.

3.3 Closures

A closure differs from a basic function by having an associated external scope bound to the function (see Figure 3.1). This means that variables can be referenced outside the function scope and accessed as immutable values. The significance is that the closure provides a way to track interstitial state strictly within the context of the function in question. This is analogous to instance objects holding state in object-oriented programming. In pure languages these variables are immutable and within the closure the values are guaranteed to be constant. This property is essential for deterministic behavior and local reasoning of a program. In R, the default behavior is that variables in the enclosed scope *appear* immutable. Attempting to reassign the value of one of these variables results in a new variable that masks the variable in the

enclosing scope. Outside of the inner function block, the original value is preserved. R is not a pure functional language, and Section 3.10 shows how to override this default behavior with a special global assignment operator.

Consider our descriptive statistics for the income classification. The number of hours worked per week has extreme values such as 1 and 99. Extreme values are known to hamper models, and Winsorization is one way to improve their robustness. [29] Implementing and tuning the Winsorization threshold can be accomplished with a closure, which is described in Listing 3.3. The idea is that we construct a function that applies Winsorization to our data. In other words, we want to return a closure $winsorize : \mathbb{R} \rightarrow X^n$ that Winsorizes the w most extreme values. This closure doesn't take `x` as an input and must get it from the higher-order function $winsorizer : X^n \rightarrow \Lambda$. [2]

```
winsorizer ← function(x) {
  function(w, scale=TRUE) {
    bounds ← quantile(x, c(w/2, 1-w/2))
    x[x < bounds[1]] ← bounds[1]
    x[x > bounds[2]] ← bounds[2]
    if (scale) x ← scale(x)
    x
  }
}
```

LISTING 3.3: Implementation of Winsorization

The returned function is a closure since it references the vector `x` defined outside its body. This can be easily verified by debugging a call to the function. Inside the function scope (environment in R terminology), only the variables defined in the function signature exist.

```
> winsorize ← winsorizer(adult$hours.per.week)
> debug(winsorize)
> winsorize(.1)
Browse[2]> ls()
[1] "w"     "scale"
```

So where is `x` defined? To find the answer requires investigating the mechanics of the R interpreter. When debugging code, `ls` is used to list the elements within the current scope. Within a function, only the variables defined in the function are visible, which is why only `w` and `scale` appear in the output above. From the debugged function, external variables can be accessed two different ways. One is via the call stack, which keeps track of the chain of functions called to get to the current function. When variables along this hierarchy are automatically visible, it is called dynamic scoping. Both Bash and Lisp use dynamic scopes. Searching the call stack to find `x` is fruitless though. Lexical scopes provide another path to variables based on the code

[2]The implementation below adds an optional argument `scale` discussed in Exercise 3.2.

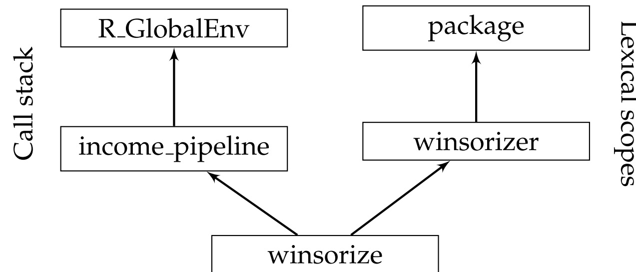


FIGURE 3.2: Relationship between the call stack frames and lexical scopes. Both are accessible from the current frame. The lexical scope assumes this code is bundled in a package.

structure. To retrieve the contents of the enclosing scope requires moving up the hierarchy of lexical blocks. This is done using the `parent.env` function.

```
Browse[2]> ls(envir=parent.env(as.environment(-1)))
[1] "x"
```

Variables are scoped lexically in R, which means the structure of the source code determines where in the source a variable is visible. Code is visually and syntactically divided into blocks (surrounded by curly braces { and }). Each block has its own scope. Variables created in the block are destroyed at the end of the block. Lexical scoping allows variables to be seen within a nested block. If a variable is referenced but not found in the current scope, then the enclosing scope is searched for the variable. This continues until the global environment `R_GlobalEnv` is reached, at which point an error is printed if the variable is still not found. Variables defined in unrelated scopes are still not visible to each other. For example, variables defined in two independent functions cannot be referenced by the other.

In the case of Winsorization, how does a closure benefit us? The censoring will likely be codified in a pipeline in the final analysis. Once a threshold is chosen, the code will likely be forgotten about and not modified. Creating a closure to support multiple thresholds thus seems superfluous. During data exploration and model development, though, the story is different. Before codifying the censoring process, we need to choose the optimal threshold w for Winsorization. Doing so requires holding all other parameters constant while varying w. Winsorization isn't particularly difficult to implement, but it still takes up a few lines. Code that forms a single, atomic operation should ideally be placed in its own function. If not, it can be difficult to know where one operation ends and another begins. It's also easier to replace one implementation with another when the code is already isolated in its own function. Using a closure wraps up all this logic and data into a function that can be called on demand.

The Winsorization threshold w can be tuned using this approach. The

same function `w.hpw` is used in each iteration. Thanks to immutability, we know that the column value is constant at the beginning of each call.

```
income_pipeline ← function(...) {
  df ← load_income()
  w.hpw ← winsorizer(df$hours.per.week)
  lapply(c(.05,.1,.15), function(w) {
    df$hours.per.week ← w.hpw(w)
    m ← logistic(income ~ age + education.num + hours.per.week, df, ...)
    p ← do_predict(m, df)
    list(model=m, performance=performance(p, df$income))
  })
}
```

LISTING 3.4: Adding Winsorization to the `income_pipeline`

Tuning the threshold merely requires iterating over some values with `lapply` and calling and evaluating the model each time. This gives us a set of model parameters and their corresponding in-sample performance. At a later stage, we can compare the models to find which w works the best.

For sake of argument, let's collapse the higher-order function and closure into a single function. What does the function look like if we don't return a closure? Two changes are evident. First, the return value is no longer a function but the return value of the closure instead. Also, the new function signature is the union of the higher-order function and closure signatures.

```
winsorize ← function(x, w, scale=TRUE) {
  bounds ← quantile(x, c(w/2, 1-w/2))
  x[x < bounds[1]] ← bounds[1]
  x[x > bounds[2]] ← bounds[2]
  if (scale) x ← scale(x)
  x
}
```

LISTING 3.5: Winsorization as a single function

This insight hints at the inverse process of dividing an existing function into a higher-order function that returns a closure. Creating such a higher-order function involves partitioning the function signature into two sets, one for the functional and the other for the closure. The body of the closure usually comprises the minimum logic required to process its arguments. From a computational performance perspective, this minimizes the amount of redundant work performed, since calculations occurring in the functional are effectively cached when the closure is executed.

3.4 Functions as factories

As we saw in the previous section, it can be beneficial to split a direct function call into two calls, one returning a function and the other calling the returned function. The rationale is that by having two functions, it is easier to understand the purpose of both via explicit separation of concerns. The outer function acts as a constructor, or *function factory*, initializing certain values of the returned function. The general signature of a function factory is $factory : X \rightarrow \Lambda$. The signature of the returned function in Λ is clean and concise. Function factories can be useful when you want a function to be called from a different context. In some other context, the data required to call a function might not be easily accessible. Rather than modifying a whole bunch of function signatures to accommodate a new parameter, it can be simpler to just use a closure to codify this new behavior. This concept is explored further in Section 3.7.

Function factories are particularly useful when a function must be called multiple times and the cost of initializing data for the function is high. Other times we want to break the dependence between conditioning data and using the data. At times it can be impractical to carry around multiple sets of model parameters to use at a later time. Function factories provide a convenient way to handle these cases. A common scenario is when you want to control graphical parameters to a function deep within a pipeline. Another example is building Winsorization into a generic preprocessing step. Furthermore, we want to test the model with and without Winsorization. A common approach is to manipulate the data frame for each model run. This requires resetting the data back to its original state. A naive approach might reload the data from an external data source after each model run, which can be costly from a time perspective. A better approach is to use a closure that retains a copy of the original, unaltered dataset. Each call with a different configuration results in only the desired transformations and not the cumulative effects. Our initial version of the `winsorizer` takes this approach. However, it only handles a single column. It's better to extend the interface to accept a complete data frame and specify the columns to transform in the call to the closure. A new `normalizer` function is responsible for applying Winsorization to multiple columns in the data frame. It also scales each column to $[0, 1]$ in preparation of the logistic regression.
For a single data configuration, the call looks like

```
> n.fn ← normalizer(x)
> w ← c(hours.per.week=.1, age=.2)
> df ← n.fn(w)
```

This approach makes it easy to iterate over these different data representations. In this implementation, the Winsorizer takes the complete data frame instead of a column (see Exercise 3.3). The original Winsorizer only took a sin-

```
normalizer ← function(x) {
  wf ← winsorizer(x)
  cols ← c("age","education.num","hours.per.week")
  function(w, col=names(w)) {
    if (length(w) > 0) {
      x ← fold(col, function(n, acc) { acc[,n] ← wf(n, w[n]);
          acc }, x)
    }
    fold(cols, function(col,acc) { acc[,col] ← scale(acc[,col
        ]); acc },x)
  }
}
```

LISTING 3.6: Function to normalize a number of columns in the `adult` dataset

gle vector, whereas with the `normalizer` we want to specify all the columns that require Winsorization in a single call. We can retain the whole data frame in the `winsorizer`, since then we can wrap the set of transformations into a *fold* operation. A named vector conveniently specifies all the columns to be Winsorized. Different configurations can be held in a list, one for each parameter set.

```
> configs ← list(c(),
+   c(hours.per.week=.1),
+    c(hours.per.week=.1, age=.2)
+ )
```

Finally, we use `lapply` to run the model against each set of parameters .

```
> n.fn ← normalizer(x)
> dfs ← lapply(configs, function(cfg) n.fn(cfg))
```

When variables have different types, a list must be used instead of a vector. This opens up the possibility of using `do.call` to directly unpack the arguments from the list. This technique is described in Section 9.5.

3.5 Mediating iteration

The canonical higher-order functions provide simple frameworks for managing iterative processes. These functions take care of the mechanics of iteration and leave the element-level transformation to the closure argument. Many machine learning techniques are iterative in nature and benefit greatly from these functions. To illustrate, let's classify the income groups of the `adult` dataset using logistic regression. Our simple model is $income = \theta_0 + \theta_1\, age + \theta_2\, education + \theta_3\, hours\ per\ week + \epsilon$.

We can implement this ourselves using stochastic gradient descent (SGD). [8, 56] Convergence is generally faster with SGD than normal gradient descent at the cost of stability. An iterative method, SGD updates the coefficients θ after each iteration based on the gradient of the cost function. Alternatively, θ can be updated by maximizing the gradient of the likelihood function. The update rule is given by $\theta := \theta + \alpha \nabla_\theta \ell(\theta)$, where ℓ is the log likelihood. Given the current sample x and response y, each element of θ is updated by the simplified expression

$$\theta_j := \theta_j + \alpha(y - h_{\theta_j}(x))x_j, \tag{3.1}$$

where $h_{\theta_j}(x)$ is the model. The learning process over the whole dataset is described in Algorithm 3.5.1. The algorithm starts by computing the error of the estimate. Next is a function to compute the update step. This function takes the input and output along with the estimated coefficients, applying the gradient to all dimensions simultaneously. One epoch of SGD can be described as a $fold$ operation over the per element update rule.

Algorithm 3.5.1: LogisticSGD$((X, Y), \theta_0, \alpha)$

$error \leftarrow \lambda y, x, \theta.\text{SIGMOID}(\theta^T x) - y)$
$step \leftarrow \lambda y, x, (\theta, \epsilon).\theta - \alpha\ \text{ERROR}(y, x, \theta)x$
$\text{FOLD}(\lambda r, m.\text{STEP}(r_y, r_x, m), (X, Y), (\theta_0, \epsilon))$

Iterative algorithms like SGD typically describe all iterations as loops, which then get translated into code. This is rather inefficient since it's easier to think about a single iteration first and then implement the wiring for multiple iterations. Functional programming gives us a programming paradigm that is consistent with our thought process. As an added benefit, having an explicit function at the incremental level makes it easier to test code, since the function is scoped to a single atomic operation.

The implementation in Listing 3.7 is slightly different from the algorithm. The error is computed directly instead of being wrapped in a function. In this case it's computationally more efficient since the gradient is computed in two parts. There's also initialization code that is present. we want to collect the errors from each iteration. This is added to the list, which acts as a 3-tuple. The point is that implementations do not need to match algorithms verbatim. Performance and other pragmatic reasons often act as wedges that separate an algorithm from its code counterpart. This function represents the pure mathematical portion of the implementation. The gradient is computed based on the data, and then a new estimate is returned. It's completely ignorant of anything related to the `adult` dataset and the wiring to implement formula notation. As the saying goes, this ignorance is bliss because the code is self contained and free of dependencies. Hence it's easy to test this function and also reuse it in other scenarios.

Now, if `logistic_sgd` is to remain blissfully ignorant, some other function needs to shoulder the burden of wiring the model specification with the

```
logistic_sgd ← function(X,Y, init=NULL, alpha=.1) {
  step ← function(x,y,fit) {
    error ← sigmoid(x %*% fit$w + fit$b) - y
    fit$w ← as.vector(fit$w - alpha * error * x)
    fit$b ← as.vector(fit$b - alpha * error)
    fit$error ← c(fit$error,error)
    fit
  }

  if (is.null(init)) init ← list(b=0, w=rep(0, ncol(X)))
  init$error ← NULL
  fold(t(cbind(Y,X)), function(r,m) step(r[-1], r[1], m), init)
}
```

LISTING 3.7: An implementation of stochastic gradient descent for logistic regression

SGD implementation. Training typically occurs over multiple epochs, and `logistic_sgd` only provides a single epoch. Another function is necessary for managing epochs. We'll call this function `logistic`, and place the responsibility of mapping a user friendly interface to it. This function also prints the sum of squared errors for each training epoch.

```
logistic ← function(formula, data, alpha=0.1, epochs=10) {
  tm ← terms(formula)
  response ← rownames(attributes(tm)$factors)[1]
  predictors ← attributes(tm)$term.labels
  fold(1:epochs, function(i, acc) {
    errors ← acc$error
    fit ← logistic_sgd(data[,predictors],data[,response], acc,
         alpha)
    fit$error ← c(errors, sum(fit$error^2))
    flog.info("[%s] error=%s", i, fit$error[length(fit$error)
        ])
    fit
  }, NULL)
}
```

LISTING 3.8: Facade for training over multiple epochs with logistic regression

By dividing the code into multiple small functions, it's easy to rearrange the pieces. If we ever want to use a different iterative optimization method, we just swap out `logistic_sgd` with something else, like `logistic_nr`, for Newton-Raphson optimization. Exercise 3.10 explores the changes required to support arbitrary optimization algorithms.

The model pipeline in Listing 3.1 sequences all the steps from loading data to fitting the model to evaluating performance. Calling the function

Model	Accuracy	Precision	Recall
SGD	0.7902948	0.6374491	0.2995791
Built-in	0.7815111	0.6718676	0.1812269

TABLE 3.1: In-sample performance of logistic regression

`m ←income_pipeline(epochs=5)` yields both the model parameters and performance metrics. It compares well with the stock logistic regression implementation. Both results are summarized in Table 3.1. For the comparison, we use `glm` to train the model. Then we get the in-sample predictions by using .5 as the cutoff between classes.

```
> adult ← load_income()
> m2 ← glm(income ~ age + education.num + hours.per.week, adult,
+   family=binomial)
> p2 ← ifelse(predict(m2, adult) >= 0.5, 1,0)
> performance(p2, adult$income)
```

We implemented logistic regression in two functions with about 10 lines each. A production quality implementation will require more code to handle categorical data and multiple classes. Nonetheless, the conciseness of our toy version shows how most iteration problems can quickly be solved using the canonical higher-order functions. Over time, code tends to become more modular since functions naturally isolate different computing tasks.

3.6 Interface compatibility

Typical model development partitions a dataset into a training set and a test set. A separate test set is a better indication of model generalizability versus in-sample testing. However, for small datasets, it can be problematic to set aside a portion of the data for testing, since that data could be used for training. Cross-validation gets around this issue by partitioning the training data into k disjoint folds. Each fold is reserved as a test set against the remainder, which acts as the training set. The model is thus trained k times and the parameters averaged across folds to produce final model parameters. In other words, cross-validation both regularizes the model in addition to enabling use of the full dataset for training.

Most algorithms for k-fold cross validation start with a loop. This seems natural since there are k iterations of the training process. However, such algorithms tend to get bogged down in the implementation. As usual, it's hard to see how the algorithm works when it's obscured by the thick underbrush of programming language mechanics. An algorithm based on functional programming takes advantage of set theory and lambda abstractions to simplify

it. Algorithm 3.6.1 describes the k-fold cross-validation in just four steps, where the emphasis is on the partitioning of the data and evaluating the model. Loops, intermediate variables, and initialization are all unnecessary.

Algorithm 3.6.1: $\text{XVALIDATE}(model, data, k)$

$X \leftarrow \text{PARTITION}(data, k)$
$\Theta \leftarrow \text{MAP}(\lambda x.\text{MODEL}(data - x), X)$
$Y \leftarrow \text{MAP}(\lambda x, \theta.\text{PREDICT}(x, \theta), zip(X, \Theta))$
$P \leftarrow \text{MAP}(\lambda x, y.\text{PERFORMANCE}(x, y), zip(X, Y))$
return $(\bar{\Theta}, P)$

One reason why this algorithm is so concise is that it leverages the implied ordinals between variables (see Section 6.3). Since the output of *map* is consistent with the order of its input, the ordinals are identical. Hence, it's easy to combine these vectors together in other operations. We could have created a one-off cross validation implementation, but there's a clear reason for making it general. Doing so is straightforward, since all that is required is defining the signature of the closure as $f : X^{m \times n} \times \Theta^k$. The input is simply a data frame or matrix of observations, while the idealized output is a vector of model parameters. Irrespective of the model being used, it's possible to create a closure with the appropriate signature. This guarantees that our function is universal.

All function arguments have an implied signature based on how they are called. How do we call this generic version of cross-validation? We need to match the function call with the logistic regression. Any time a higher-order function specifies a function signature that is different from the signature of the function we want to pass to it, a closure is used to bridge the gap in signatures. The key is that the signature of the closure must always match the expected signature, while the higher-order function generating the closure can be arbitrary. Once again the corresponding R code in Listing 3.9 deviates from the idealized algorithm. The main reasons are related to logging and creating compatible data structures that are ignored in the algorithm. Another difference is the partition operation. Instead of partitioning the data explicitly, it returns sets of indices. This simplification makes it easy to use `tapply` to mediate both the partitioning of data and the iteration. If we want, we can use a similar approach as `logistic` and create a separate function to manage each epoch. In this case it manages an individual fold of the cross validation. The drawback of not using a closure is that additional arguments must be added to the signature. This is the cost of independence and should be weighed carefully.

To use this function, we can simply swap it into the `income_pipeline` and re-run. This gives us a new model and corresponding performance.

```
xvalidate ← function(model, data, k) {
  part ← (1:nrow(data)) %% k
  tuple ← tapply(1:nrow(data), sample(part,length(part)),
      function(idx) {
    flog.info("Train on fold leaving out %s samples", length(
        idx))
    th ← model(data[-idx,])
    flog.info("Predict on out-of-sample data")
    y ← do_predict(th, data[idx,])
    names(y) ← idx
    list(model=th, pred=y)
  })

  flog.info("Merge out-of-sample predictions")
  pred ← do.call(rbind, lapply(tuple,
    function(tpl) data.frame(idx=as.numeric(names(tpl$pred)),
        y=tpl$pred)))
  data$pred ← pred$y[order(pred$idx)]
  perf ← performance(data$pred, data$income)

  flog.info("Construct final model")
  model ← colMeans(do.call(rbind, lapply(tuple, function(tpl)
      {
    c(tpl$model$b, tpl$model$w)
  })))

  list(model=model, performance=perf)
}
```

LISTING 3.9: An implementation of *k*-fold cross-validation

3.7 Codifying behavioral changes

Functions don't need to be higher order to be useful. Plain old functions are effective at changing the behavior of existing functions. Some common examples are include handling `NA`s in a function or overriding default values of model parameters. For example, the default behavior of `mean` is to return `NA` if any `NA`s exist in the vector. At times it can be useful to override this behavior. When the `mean` is used as an argument to a higher-order function, though, the behavior is fixed according to the function defaults. Achieving a different behavior requires wrapping the target function in another function, such as

`function(x) mean(x, na.rm=FALSE).`

The advantage of using first-class functions is that the possibilities are infinite,

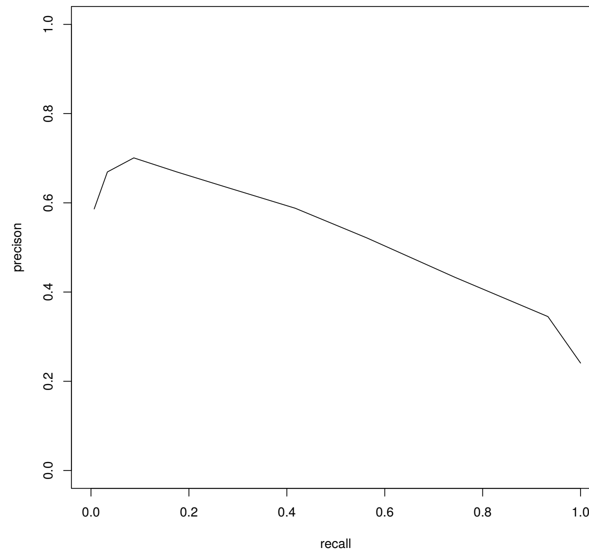


FIGURE 3.3: Precision-recall curve for the `adult` dataset

so the author of a function does not have to guess at which implementations to provide. Instead, a package author can focus on the *ideal* interface, knowing that a user of the package can use functional programming concepts to conform any function interface to match the package interface.

Suppose we want to use a different set of performance metrics during cross-validation. The current implementation uses the `performance` function, implemented in Example 3.1. What if we want to produce the precision-recall curve for our model, as shown in Figure 3.3? Two things need to happen. Currently, the prediction returns the classes and instead needs to return the class probabilities. We also need to add a performance function argument to the interface, so it's configurable. But notice that `xvalidate` must assume an explicit signature for the new function. The obvious choice is to use the same signature as what the current `performance` function uses. This is great for backwards compatibility, but other functions might not have the same signature. what information does our precision-recall function require? At a minimum it needs the class probabilities and a vector of cutoff points. We may also want to plot the curve. For convenience we'll provide default cutoff values at intervals of 0.1 and assume the user wants to plot the result, similar to `hist`. One implementation is shown in Listing 3.10. This time an anonymous closure is passed directly to `sapply`, since it's so specific to the *map* operation.

To use `pr_curve`, the prediction step needs to know to return the class

```
pr_curve ← function(prob, obs, cutoff=(0:10)/10, plot=TRUE) {
  pr.mat ← t(sapply(cutoff, function(ct) {
    pred ← ifelse(prob <= ct, 0,1)
    perf ← performance(pred,obs)
    c(recall=perf$recall, precison=perf$precision)
  }))
  if (plot) plot(pr.mat, type='l', xlim=c(0,1), ylim=c(0,1))
  invisible(pr.mat)
}
```

LISTING 3.10: Compute and plot the precision-recall curve based on a set of cutoff points

probabilities instead of the output classes. With this small change, `pr_curve` replaces the raw `performance` function directly.

```
> adult ← load_income()
> m ← logistic(income ~ age + education.num + hours.per.week, adult)
> p ← do_predict(m, adult, cutoff=NA)
> pr_curve(p, adult$income))
```

When integrating this function into our pipeline, we don't want to plot the curve for each epoch. We also might want to include additional cutoff points for greater resolution. The simplest way to do that is to create a closure that wraps the call. This new function has the exact interface we require and also the exact behavior we desire.

```
perf.fn ← function(prob, obs) {
  pr_curve(prob, obs, (0:20)/20, FALSE)
}
```

Functions don't always behave how we want. Despite this petulance, most functions are flexible enough to accommodate our needs. A combination of options, pre-processing, and post-processing, are usually sufficient in this regard. It's good practice to bundle these steps along with the function call in its own function. It can then be easily passed as an argument to any higher-order function.

3.8 Inversion of control via callbacks

Many R scripts are procedural in nature. They specify a sequence of tasks, possibly organized by function. For ad hoc analysis or data processing, this is usually sufficient. In these sorts of scripts, the control flow is described explicitly by the script. When projects grow in complexity, or common tasks need to

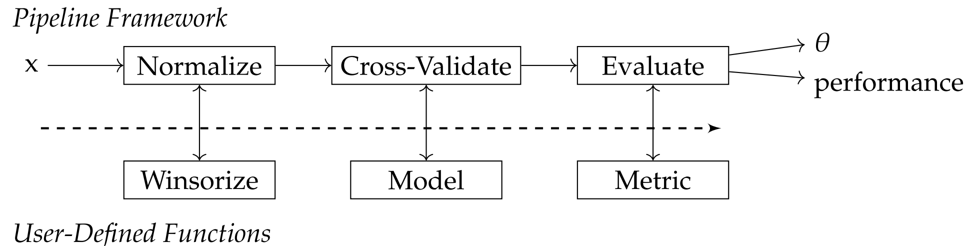


FIGURE 3.4: The `model_pipeline` as an example of inversion of control

be executed efficiently, it can make sense to delegate control flow to a library or framework. This is known as *inversion of control* (IoC), since the user code is no longer managing the flow of control in the program. In this paradigm user code focuses on logic specific to the data scientist's model or application, delegating the generic workflow to the framework. Web server frameworks follow this pattern, where the mechanics of responding to the HTTP protocol is handled by the framework. The user only needs to fill in her custom application logic. In modern object-oriented programming languages, inversion of control is often handled using decorations or annotations. In functional programming or languages that don't support annotations, callback functions are used for the same purpose.

Inversion of control can appear at varying degrees of granularity. Most server frameworks have absolute control over the whole program flow, whereas *map*, $fold$, and $filter$ are microcosms of this pattern. As we've seen, these higher-order functions mediate various forms of iteration. The control flow associated with iteration is the underlying loop along with any variable initialization and management. This is all governed by the higher-order function, which calls the first-class function at each step in the iteration. On a larger scale, model processing pipelines have a similar structure. Frameworks that implement specific workflows lay somewhere in between. An example of this is creating a framework for tuning models, similar to the `caret` package [31]. With `caret`, optimizing tuning parameters and cross validation are managed by the package. While control isn't completely commandeered, a significant portion of the workflow is handed off to `caret`.

```
> control ← trainControl("cv", 10, savePredictions=T)
> adult$income ← as.factor(adult$income)
> fit ← train(income ~ age + education.num + hours.per.week,
+   data=adult, method="glm", family=binomial, trControl=control)
```

Our model pipeline for the `adult` dataset can also use IoC. We can make it a generic framework by specifying additional arguments in the signature. At a minimum we need to add arguments for the data and the model to use. We can be a little ambitious and also add an argument to include a regularization step. L_2 regularization is a common regularization method that adds the L_2

```
L2 ← function(lambda) function(w) lambda * w
```

LISTING 3.11: L_2 regularization for SGD

norm as a penalty term when minimizing the loss. The general form for a model f with loss function V is

$$min_f \sum_i^n V(f(\hat{x}_i), \hat{y}_i) + \rho R(f), \tag{3.2}$$

where R is the regularization function. Notice that R is treated as a functional taking the model function as input. In the case of L_2 regularization on linear models of the form $f(x) = \mathbf{w} \cdot \mathbf{x}$, equation 3.2 simplifies to

$$min_f \sum_i^n V(\mathbf{w} \cdot \hat{\mathbf{x}}), \hat{y}_i) + \rho \sum_i w_i^2, \tag{3.3}$$

Once we introduce regularization into the processing flow, ideally we go all in. Otherwise, we end up with esoteric bits of conditional logic that muddy the code. It's better to exploit mathematical properties to achieve the same goal. We can use $\rho = 0$ to suppress the effect of regularization. From an interface perspective supporting regularization requires adding two extra arguments to the pipeline signature: the coefficient ρ and the regularization function. This isn't great, since the two variables must be consistent for regularization to work properly. For example, what is the correct behavior when $\rho = 0$ but R is non null? Using a function interface is useful though, since it's easy to swap out one implementation for another. If we want to use L_1 regularization instead, we simply replace R. Keeping R as part of the interface is thus mandatory. However, we can include ρ in the regularization function and remove it from the higher level interface. This is equivalent to creating a new function $\lambda x.\rho R(x)$. Alternatively, we can create a function factory that takes the weight and returns the regularization function, which is $\lambda \rho.\lambda x.\rho R(x)$, for some regularizer R.

At this point `income_pipeline` is fast becoming a generic framework that orchestrates the model fitting process. A name that reflects this generality is `model_pipeline`, which we'll use going forward. Users of this function simply provide the data and model they want to use, and the framework takes care of the rest. To fit a model to the `adult` dataset with 6-fold cross-validation of logistic regression with L_2 regularization looks like

```
> model_pipeline(income ~ age + education.num + hours.per.week,
+   logistic, adult, reg.fn=L2(.2))
```

Example 3.2. Consider the `baseball` salary dataset. [51] This dataset compares player salaries with their performance statistics. For compatibility rea-

sons, we'll flip some of the variables around and predict whether a player was a free agent in the 1992 season based on offensive performance and salary.

```
> uri ← "http://www4.stat.ncsu.edu/~boos/var.select/baseball.txt"
> bb ← read.csv(uri, header=TRUE)
> model_pipeline(x14 ~ x1 + x2 + x4 + x8 + y, logistic, bb,
+   scale.fn=scale.bb)
```

□

The beauty of a framework is that it handles all the details of encoding a repeatable workflow. But as hard as they try, frameworks do not apply in all situations. One solution is to declare your framework "opinionated", which is an obtuse way of saying "do it my way". An alternative approach is to provide sufficient hooks to enable users of the framework to customize the behavior as they see fit.

3.9 State representation

In certain cases, retaining state over disparate operations is appropriate, particularly for representing external resources. These resources are often singletons in the physical world (or in the operating system environment), so modeling them as a single shared object with state makes sense. An I/O connection is a common example, where a resource is opened, read, and finally closed. Here a file descriptor represents the state of the file and must be managed accordingly. Making multiple connections to the same file can be problematic and can result in consistency errors. Object-oriented paradigms are often heralded for their ability to manage state. In an object-oriented paradigm a class represents a generic file, and an instance of the class is a specific file. This file object can then be opened, read, and closed. The power of the object-oriented approach is that all resources, variables, and operations associated with the file are encapsulated within the class definition. The challenge is that each resource and method returns arbitrary instances of other classes. The pathological case results in an exceptionally granular class hierarchy representing each and every concept within the language. Knowing when to stop modeling the class hierarchy is one of the hardest problems in designing object-oriented systems as one must balance reusability with ease of use. Highly granular class libraries are good for reuse, but it leads to exceptionally verbose implementations that are difficult to learn. In Java, there are distinct classes for files, connections, streams, and buffers. Loading a file in Java requires interacting with objects from each of these classes, which means understanding how a file system is modeled along with their individual APIs, in addition to the implicit state machines embedded within the

```
using ← function(resource, handler, exit=close) {
  on.exit(exit(resource))
  tryCatch(handler(resource), error=stop)
}
```

LISTING 3.12: A resource management function

class. An example of this are connections that must be closed after opening. When resources aren't properly closed, it can lead to memory leaks as well as running out of operating system resources. [3] Despite all this granularity, you still have to manually manage the actual resources being modeled. The saving grace is that all of the machinery for managing a resource can be encapsulated in a single class, which limits the hunt for documentation. On the other hand, languages that favor monolithic classes (like Objective-C or the `pandas` library in Python) are also difficult to learn because so many permutations exist for performing an operation that it isn't immediately obvious which one to use. This is similar to the interface problem in Section 3.8 where two arguments must be consistent for regularization to work properly.

So the benefit of object-oriented programming comes at the cost of complexity. Not surprisingly, functional programming provides a liberating alternative to the tyranny of all-encompassing class hierarchies. Indeed, to implement a generic model pipeline requires exactly zero new types. Rather than attempting to optimize an interface for the most common use cases, functional programming interfaces are restricted in quantity. Since closures are so easy to create (and their resources managed efficiently), it is often trivial to conform two interfaces together on an ad hoc basis. The implication is that flexibility is no longer a design decision for the package developer but a simple compatibility problem for the package user. This approach preserves a simple and clear interface for functions while avoiding the slippery slope of optimal interface design.

In terms of state management, closures can provide the same encapsulation as a class can. The key difference is that creating a closure does not require a lot of ceremony defining classes and instantiating objects. Closures can be created ad hoc as an anonymous function or more formally via a function factory. Any resources defined in the closure can be automatically garbage collected once all references to the closure are gone. The result is a simpler code base since there are fewer formal type/class definitions.

To illustrate a functional programming approach to state management, let's implement a function that is inspired by the `with` keyword in Python. A `with` statement automatically manages resources within the scope of a block. When the end of the block is encountered or an error is encountered,

[3] This is true of most programming languages. In R, unused connections are eventually garbage collected to prevent memory leaks.

the specified resource is automatically closed. [4] Since R defines `with` as a technique to access objects in environments, we'll call our version `using`, which is shown in Listing 3.12. In the `adult` analysis, we didn't say how to get the data. Some data is available online. Due to authentication requirements, `read.csv` cannot parse them directly. In these cases it's necessary to open a connection explicitly, passing the connection object to `read.csv`. Afterward the resource must be closed. Socket connections work similarly and need to be closed after use. Compressed files can be read directly by `read.csv`. For sake of argument, we'll pretend that compressed files must be read the same way as other connections. [5] Compression is often used to minimize network transfer. Once downloaded, the file is opened using `gzfile`. The connection is then passed to `read.csv`, which reads the data. Finally the connection is closed. The `using` function bundles this all together into a single function call. Any handler can be passed to `using`, such as

```
df ← using(gzfile('data/adult.data.gz'), load_income).
```

The value of a function like this is that any errors in the handler will automatically close the resource. When code is meant to run automated, without human intervention, handling resources and errors properly becomes critical. If not, memory leaks can overwhelm the system or mysterious errors may surface in data at some later point. It's better to detect errors quickly and halt processing altogether. For massive jobs this may be impractical. In these cases it may be better to skip bad records/jobs and let the rest of the processing continue.

One difference between `using` and Python's `with` statement is that `using` is a function call, whereas `with` is a language control structure. Depending on your philosophical leanings this is either a benefit or drawback. One argument for a control structure approach is that blocks of code can be expressed directly. Our implementation wraps blocks up in a function, which can add a few keystrokes. But this is superficial. With lazy evaluation (see Section 10.4), literal blocks can be passed to a function.

Example 3.3. Exploratory analysis usually begins by visually inspecting data to look for any obvious patterns or trends. For seasonal data, it can be useful to plot the data various ways to get a complete picture. Sometimes a function needs to change these parameters to display a custom plot. A good citizen will ensure that the original parameters are restored once the function exits. A typical implementation looks like

```
plot_result ← function(x) {
  opar ← par(mfrow=c(2,2), ...)
```

[4] In Python, `with` operates on a callable object that has a `__enter__` and `__exit__` function defined. This interface simplifies closing resources, since the connection object is responsible for both operations. It's less direct in R, since this protocol doesn't exist. In Section **??**, we'll see how types can provide a similar system in R.

[5] This is simpler than setting up a protected server.

```
open_resource ← function(resource, exit=close) {
  function(handler, destroy=FALSE) {
    if (destroy) return(exit(resource))
    tryCatch(handler(resource),
      error=function(e) { exit(resource); stop(e) })
  }
}
```

LISTING 3.13: Using a closure to manage external resources

```
  on.exit(par(opar))

  models ← model_pipeline()
  lapply(models, function(model) pr_curve(model$prob, model$obs)
}
```

The use of `on.exit` is required to properly account for errors that may arise in the function. Without this inlcusion, the parameters will not be restored properly if an error is encountered. This approach works well but is easily overlooked. The same can be accomplished with `using`. [6]

```
plot_result ← function(x) {
  models ← model_pipeline()
  using(par(mfrow=c(2,2)), function() lapply(models,
    function(model) pr_curve(model$prob, model$obs), par)
}
```

Notice how this approach cleanly separates the mechanics of managing the state of the graphics environment from the visualization code.

□

In the above cases no closure is required because the handler operation is effectively atomic. What if the resource must stay open for an indefinite period of time? Here a closure can be used to manage the resource. While the above technique is useful for a fixed set of operations, it doesn't work well for arbitrary operations in disconnected control sequences. Taking a cue from Javascript, we can overload a function with multiple behaviors to achieve the desired behavior. Named parameters makes this a simple and safe exercise as seen in Listing 3.13. The general approach is to define the default operation as the primary interface for the signature. Other operations are then controlled by optional arguments to the function.

Working with this function involves naming the returned function and calling this in lieu of `using`. The advantage of `open_resource` is that all resources can be managed consistently irrespective of the resource in question. This reduces the number of idiosyncratic details to be remembered in the language.

[6] The removal of the `par` lines in `plot_handler` is implied.

```
> cat("distribution,x\n", file="example.data")
> using.resource ← open_resource(file("example.data"))
> using.resource(readLines)
[1] "distribution,x"

> cat(sprintf("normal,%s\n",rnorm(2)), file="example.data")
> using.resource(readLines)
[1] "normal,1.35652438112218"  " normal,1.19550937688085"

> using.resource(destroy=TRUE)
```

The recurring theme of separation of concerns is yet again the main benefit. Clearly thinking about what is model logic versus general software machinery provides an opportunity to cleanly implement models according to the mathematical sequence of function composition. Once this distinction in code purpose is made, it also becomes clear that much of the data management machinery is general and can be easily reused at a level of sophistication that exceeds granular functions. This is because we have encoded a process workflow within a higher-order function and closure as opposed to a single operation.

3.10 Mutable state

In the previous section, the state being managed was static. Once a file resource is opened, the resultant connection object doesn't change state until it's closed. Some algorithms and systems have state that update throughout the life of a process. The SGD algorithm is like this, where the state consists of the vector of parameters Θ. Typically variables retained in a closure are immutable, but with the special global assignment operator (or double arrow operator) ↞, it is possible to change the value of a variable in an enclosing scope.

Recurrence relations can befuddle data scientists new to functional programming. These algorithms require updating values incrementally. While most intermediate R users know to avoid loops, How to use a higher-order function to solve the problem can be unclear to them. Most users reach for a variant of `apply` and attempt to update the state of a variable at each time step. We know that external variables referenced within a closure are immutable, so the update will fail. It's at this point where desperation sets in, the glass cover broken, and the double arrow operator pulled into action.

In the case of the logistic regression, a typical solution is shown in Listing 3.14. The global assignment operator restricts modularity, since variable references follow the lexical chain. If the step function were to be placed outside the logistic function, the variables `b` and `w` would be created and written

```
logistic ← function(formula, data, alpha=0.1) {
  tm ← terms(formula)
  response ← rownames(attributes(tm)$factors)[1]
  predictors ← attributes(tm)$term.labels
  b ← 0
  w ← rep(0, length(predictors))
  step ← function(x,y) {
    error ← sigmoid(x %*% w + b) - y
    w ↞ w - alpha * error * x
    b ↞ b - alpha * error
    error
  }
  es ← apply(data[,c(response,predictors)], 1,
    function(row) step(row[predictors], row[response]))
  list(b=as.vector(b), w=w, error=cumsum(es^2))
}
```

LISTING 3.14: Logistic regression using global assignment operator

to in the global environment! the operator will continue to access enclosing environments until a matching variable is found. If the global environment is reached and no variable is found, one is created. Careless usage can therefore result in variables being created in the global environment. This behavior is not safe and is why the global assignment operator needs to be used sparingly and with care.

Example 3.4. Exponential moving averages (EMA) can be used to help quantify the trend associated with a non-stationary time series. Unlike a simple moving average (SMA), EMAs require a complete history to be computed correctly. [7] They are defined recursively based on a linear combination of the current value and its previous value. Let x_t be the value of the series at time t. The EMA s_t is

$$s_1 = x_1 \tag{3.4}$$

$$s_t = \alpha x_t + (1 - \alpha)s_{t-1}, \tag{3.5}$$

which points to the fact that its calculation requires a state. A closure can be used to compute the value of the EMA over time.

```
> set.seed(235813)
> r ← rnorm(250, sd=.01)
> x ← 100 * cumprod(1+r)
```

A *map* implementation requires the global assignment operator, while a *fold* implementation treats the accumulator as the state.

[7] The EMA can be considered a simple state-based system. Additional state-based systems are discussed in Chapter 12.

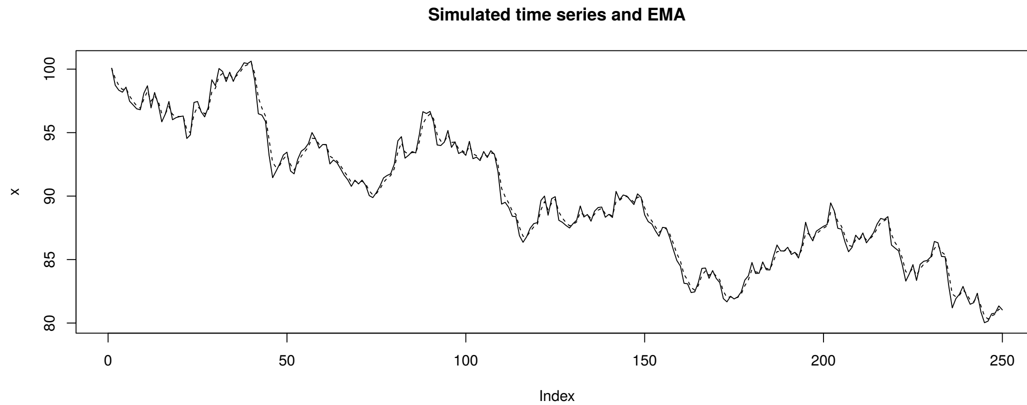


FIGURE 3.5: Simulated time series with corresponding exponential moving average (dotted line)

```
ema_map ← function(y, alpha) {
  s ← y[1]
  lapply(y[-1], function(yi) {
    si ← alpha * yi + (1-alpha) * s[length(s)]
    s ↞ c(s,si)
  })
  s
}
```

LISTING 3.15: A *map* implementation of an EMA uses the gloabl assignment operator to append each incremental value to `s`. This code is unsafe, since it can lead to side effects.

Thankfully the double arrow operator is often unnecessary. State that updates over time is more safely handled as a *fold* operation than *map*.

```
ema ← function(y, alpha) {
  fold(y[-1], function(yi, s)
    c(s, alpha * yi + (1-alpha) * s[length(s)]), y[1])
}
```

LISTING 3.16: In a *fold* implementation of an EMA, the closure only reads variables. Read operations have no side effects and are therefore safe.

□

3.11 Summary

The benefits of functional programming are legion, and this chapter highlighted many of these benefits. The catalyst is the simple idea that functions are first-class, like any other data. Higher-order functions are a natural consequence of this idea. This simple toolkit can be applied to virtually any situation offering a clean separation of concerns between model logic, data management logic, and application logic. The end result is a modular program with a clear delineation between reusable pieces of data logic and model-specific ad hoc pieces.

We also explored the mathematical connection with functional programming concepts, which will facilitate model development in subsequent chapters. The brief introduction to the lambda calculus provides a formal framework for understanding function transforms within code, which can simplify model implementation as well as provide insights into the model itself.

3.12 Exercises

Exercise 3.1. Describe the mathematical signature of the `table` function for 1 vector input and also for 2 vector inputs. Is there a general form for the signature?

Exercise 3.2. The scale function is defined

```
scale.bb ← function(df) {
  cols ← c('y','x3','x4','x5','x6','x7')
  df[cols] ← lapply(cols, function(col) scale(df[,col]))
}
```

Identify the equivalent *map* and *fold* operations in the function.

Exercise 3.3. Modify the `winsorizer` function to take the full data frame. The new signature should match $winsorizer : X^{n \times m} \times \Lambda$.

Exercise 3.4. The `scale` parameter in the `winsorize` function assumes a specific scaling algorithm. Change the interface to take a function instead. What should the signature of the closure be?

Exercise 3.5. Example 3.1 discusses various methods of measuring model performance. Implied in this discussion is a cutoff point of 0.5, but plots like the ROC curve and the precision-recall curve require computing these metrics over a number of cutoff points. Modify the `performance` function to support a cutoff point.

Exercise 3.6. Continuing with Exercise 3.5, write a function that calculates the ROC curve based on the output of the `performance` function.

Exercise 3.7. An alternative approach is to use `lapply` instead of *fold* in the `normalizer`. We can still replace all specified columns at once, though the usage changes slightly.

```
n.fn ← normalizer(x)
df[,names(w)] ← n.fn(w)
```

Modify the `normalizer` to use `lapply` instead of *fold*. How much difference is there in the structure of the code? Which approach do you prefer? Why?

Exercise 3.8. The `normalizer` in Listing 3.6 is specific to the `adult` dataset. Update the function to be generic.

Exercise 3.9. Add sampling to `logistic_sgd`, so the training set is ordered randomly.

Exercise 3.10. The `logistic` function uses `logistic_sgd` to find the optimal parameters of the logistic regression. Modify the function to support an arbitrary optimization function, such as Newton-Raphson. What else is needed to make the function backwards compatible?

Exercise 3.11. In our version of cross validation, we call a function named `performance` to evaluate the model. This simply returns the confusion matrix. What if we want to compute some other statistic instead? The simplest option is to add that function to the signature so that it becomes parameterized. To preserve the original behavior, simply use the original function as the default value. Anyone that wants different behavior provides their own function to evaluate the performance.

4

Alternate functional paradigms

The R language provides a strong foundation for writing functional programs. Somewhat surprising is that R has also become a platform for introducing alternate functional programming paradigms. This is possible due to its reflection capabilities and also an active user community focused on improving the usability of the language. Many of these alternate paradigms are borrowed from other languages and repurposed to be consistent with R idioms. Some approaches are more obviously derived from functional programming while others hide this relationship beneath layers of syntax. This chapter reveals the connection between these alternate paradigms and functional programming. Understanding this relationship not only highlights how these packages work but also sheds light on when they are appropriate to use.

Starting off, Section 4.1 briefly examines the built-in collection of higher-order functions. These functions are borrowed from Lisp and include `Map`, `Reduce`, `Filter`, `Find`, `Position`, and `Negate`. The first three functions appear in the canon described in Part II, illustrating how fundamental these higher-order functions are to functional programming. Through language extensions, R can appear more like other functional programming languages. More than a facade, packages that introduce new syntax also introduce new semantics. The `magrittr` package [48] discussed in Section 4.2 introduces pipe notation. Related to arrows and monads in Haskell [], the syntax also hints at computational graphs. Another language extension, `lambda.r` [43], [1] provides an alternate method for defining functions and types. Reviewed in Section 4.3, this package aims to facilitate model development by replacing the various object-oriented class and dispatching systems with a simpler, FP-based system.

Functional programming concepts are also introduced via protocols or patterns. Instead of new syntax, new functions are introduced, coupled with a methodology. For example, the older `plyr` framework advocates the so-called split-apply-combine approach for data manipulation. This approach attempts to standardize data aggregation and is discussed in Section 4.5. It's similar to a GROUP BY query in SQL and `by`, `aggregate`, and `tapply` in base R. These operations implicitly rely on function composition and *map*-vectorization. The newer `dplyr` framework focuses exclusively on transfor-

[1] created by the author

mations of data frames using magrittr to mediate function composition. On the other hand, purr is like plyr, and introduces a set of function primitives to be used exclusively. These are meant to provide deterministic return types to functions. These packages are discussed in Section 4.6. Another paradigm targeting data processing workflows is the so-called MapReduce framework discussed in Section 4.4. The framework provides a distributed version of *map* and *fold* designed to process hundreds of gigabytes of data.

Moving on from the adult dataset, this chapter uses the baseball dataset from the plyr package as the primary example. This dataset is different from the baseball datsset described in Example 3.2. Instead of looking at salaries, this dataset provides player statistics for each year in a player's career. Our analysis will begin by grouping the players into rookie, normal, and veteran. For each group, we'll calculate some aggregate statistics such as batting average.

4.1 The built-in functional programming canon

Despite being part of the standard R library, Map and Reduce live in relative obscurity, alongside their Lisp-inspired brethren Filter, Position, Find, Negate [39]. For Map, the reason is fairly obvious, since the apply family provides so many options for applying a function to a set. While prevalent in mathematics, the *fold*/reduce concept is less understood outside of the functional programming community. In concept Reduce works the same as fold, except it provides options for iterating from either end of a vector and whether to accumulate incremental results. The cost of this convenience is complexity: Reduce requires 68 lines of code, whereas fold has 8 lines spread over two function clauses. Numerous embedded conditional blocks are required to handle two additional arguments. For an end user, does this verbosity in the function matter? One argument for clean implementations is debugging. When you get unexpected results it's necessary to debug the code to determine the cause. The more complicated the code is, the more difficult it is to debug. This observation echoes Kernighan [28], who remarked "everyone knows that debugging is twice as hard as writing a program in the first place. So if you're as clever as you can be when you write it, how will you ever debug it?" Simplicity is thus preferred over complexity in most cases. In other words, complexity is a cost to be minimized. To justify complexity, the utility derived from it should be greater than its cost. For example, there are two simpler alternatives to including an option for left or right iteration for Reduce. One is to implement a reverse function that can be composed with Reduce. Another is to provide left and right versions (which appear in other languages). Both approaches simplify the implementation and doesn't require extra effort on the part of the end user.

Whether to use one or the other is a matter of taste and philosophy. Some prefer the swiss army knife approach, where a function has something for everyone. This approach maximizes flexibility. Others prefer simplicity and conciseness. This approach maximizes comprehension and flow, since fewer options and concepts need to be mastered.

Example 4.1. Revisiting Example 3.4, the EMA can be implemented using `Reduce` instead of `fold`. As expected, there aren't too many differences besides cosmetic changes.

```
ema_red ← function(y, alpha) {
  Reduce(function(s, yi)
    c(s, alpha * yi + (1-alpha) * s[length(s)]), y)
}
```

LISTING 4.1: EMA using `Reduce`

The most significant difference is that `Reduce` uses the first two elements in the first operation (when no initial value is provided), while `fold` makes this explicit. This is a convenience but can hide the mechanics of the operation.

□

The built-in canon of higher-order functions include four additional functions. Filter provides wiring for removing elements from a set. This function is part of the canon written in this book and is discussed in Chapter 8. The last two functions are designed for searching data structures. Find returns the first element that satisfies the given predicate. On the otherhand Position returns the associated index. Datasets are often loaded in toto in data science and are partitioned in memory. This differs from database interactions, where a query is executed on a server to provide the exact subset requested. From R, these subsets are extracted using subsetting notation. The Filter concept uses functions instead of inline expressions to partition the data. Filtering data is so ubiquitous that we'll see a similar function in `dplyr`. Suppose we have a portfolio worth $100. Each day the net asset value (NAV) of the portfolio moves around depending on the performance of the assets in the portfolio. Using the NAV series from Example 3.4, we want to know the first time our portfolio drops 5% in value. The `Position` function can do this. It returns the index of the first value in a vector that satisfies a predicate.

```
> Position(function(a) a < 95, x)
[1] 22
```

Behind the scenes `Position` uses a loop to mediate the iteration. A more faithful implementation can be accomplished using `lapply` along with a continuation to exit early from `lapply`. Continuations are discussed further in Section 10.3.

```
position ← function(f, x, nomatch=NA_integer_) {
  callCC(function(k) lapply(seq_along(x), function(i) {
    if (f(x[i])) k(i)
    nomatch
  }))[1]
}
```

LISTING 4.2: Get position without a loop

4.2 Infix "pipe" notation

Like a mischievous deity, function composition appears in many forms. Even in pure mathematical notation, function composition has two distinct syntactic forms: as chained function application and as an infix operator. In either form, the function precedence is right to left. This means that the right-most function is applied to the operand, followed by the application of the second right-most function, and so on. It happens that function application syntactically defines its arguments to the right of its name and is typically vocalized as "f of x" or "f applied to x". This arrangement has the unintended consequence of right-to-left precedence for function composition. Some people find this notation counterintuitive and would rather use left-to-right precedence. While the difference is mostly syntactic, it's worth exploring it further to appreciate the implications of the style. Suppose we are fitting an error, trend, seasonal (ETS) state space model to a time series x and want to apply a Box-Cox transformation first. In standard function notation this operation can be codified as $ets(boxcox(x))$. If this were a common operation we could assign it to a new function h using the infix $\circ$ operator, such as $h = ets \circ boxcox$ or inline as $(ets \circ boxcox)(x)$. Note that this is still right-to-left precedence since *boxcox* is applied to x first and then *ets* applied to that result.

Left-to-right precedence switches the order of operations so that operands to the left are applied first. One example of left-to-right precedence comes in the form of the pipe operator, derived from UNIX, where pipes direct the standard output (stdout) of one command to the standard input (stdin) of another. Pipe notation can be considered equivalent to function application via the relationship $f(x) = x|f$.[2] UNIX shells have supported pipes for decades [26], but the concept is relatively new to programming languages outside of the shell. It may not be immediately obvious how pipe syntax represents function composition, so let's see what happens when a second function is

[2] In UNIX shells the correct syntax is actually `echo $x | f`, which ensures that the contents of `x` are directed to stdout.

added. This results in

$$\begin{aligned} x \mid g \mid f &= (x \mid g) \mid f \\ &= g(x) \mid f \\ &= f(g(x)). \end{aligned}$$

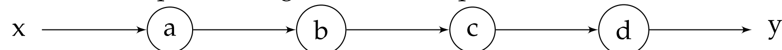
Syntactically the order of the functions is reversed, but due to precedence rules the operation is the same. Let's pretend that our Box-Cox transformation and ETS fit are the UNIX commands `boxcox` and `ets`, respectively. The notation for fitting the model becomes

```
echo $x | boxcox | ets.
```

To parse this syntax we read left-to-right so that the contents of `x` are first sent to standard out via the `echo` command and then routed to the Box-Cox transformation, and finally to the ETS model for fitting.

One benefit of pipe syntax is that it is consistent with how we model data flow. Since we read (English) language and write code left-to-right, it's natural to think about time moving left to right across the page or screen. This left-to-right precedence also appears in object-oriented programming under the guise of "method chaining", where successive methods are called on the return value of the previous method call. For example, assuming the corresponding method exists on each return value, a popular representation of function composition is `x.boxcox().ets()`.[3] The common theme is that additional operations are appended to the end of the previous command instead of explicitly passed as an argument to each successive operation.

The `magrittr` package implements pipe syntax in R with the infix operator `%>%` [48]. Aside from the visual consistency with chronological ordering, why might you use this syntax over standard function composition? One argument is that long chains of traditional function composition can be difficult to read. Infix notation attempts to make these transformation chains more legible. For example, we might model a sequence of transformations as



which is equivalent to $y = d(c(b(a(x))))$. With the magrittr infix notation the chain looks similar to the illustration: `y ←x %>% a %>% b %>% c %>% d`. Another way to preserve visual chronological order is with an interstitial variable, such as

```
o ← a(x)
o ← b(o)
o ← c(o)
y ← d(o).
```

In this case we take advantage of top-to-bottom program flow to visually

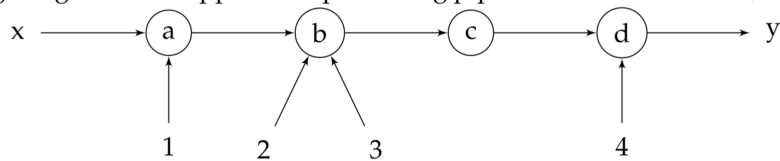
[3]This syntax can seem convenient, but it is difficult to simultaneously generalize and maintain consistency across object types.

indicate that a precedes b. This function composition can be implemented with $fold$ as well. The trick is to iterate over the set of functions so they are passed in order as operands to the first-class function argument.

```
fs ← list(a,b,c,d)
y ← fold(fs, function(f,xx) f(xx), x)
```

Up until now the vector argument to $fold$ has been a vector or list of values. In this example we are passing a sequence of functions as the argument. This is perfectly valid since in functional programming, functions are first-class. Doing so opens up yet more types of calculations that $fold$ can represent. Choosing one representation over another becomes a matter of clarity and style. Regardless of the approach taken what's important is the equivalence of the two representations.

Pipe notation shines when chaining numerous functions with arbitrary length signatures. Suppose our processing pipeline is more involved, such as



or $y = d(c(b(a, 1), 2, 3)), 4)$ in function form. Some functions in the chain take additional parameters besides the primary data argument. The corresponding `magrittr` syntax is `y ←x %>% a(1) %>% b(2,3) %>% c() %>% d(4)`. This construction works by inserting the result of the previous function as the first argument to the current function. In other words, `x %>% a(1)` ≡ `a(x, 1)`. This is semantically equivalent to using $fold$ with a data structure that holds both the functions and their respective arguments. To illustrate this equivalence, we create a function `chain` that mimics pipe notation. This function takes a list of lists, where each inner list element represents a function call using the specified function and arguments. As with pipe syntax the first argument is not specified in the configuration since it gets passed as the result of the previous function call.

```
chain ← function(fs,x) fold(fs,
  function(tpl,xx) do.call(tpl[[1]], c(list(xx),tpl[-1])), x).
```

Executing the series of functions requires encoding the functions in the list structure and then calling each function in succession.

```
fs ← list(list(a,1), list(b,2,3), list(c), list(d,4))
chain(fs,x).
```

Here the pipe notation is more readable than a comparable $fold$ implementation since the functions are not abstracted in a data structure. The drawback with chaining functions this way is that an implicit function signature is defined for each chained function. Every function must be defined consistently so that the output of each function is compatible with the first argument of

other functions. Hence, `magrittr` works well within a particular package where functions are designed with pipe notation in mind. Functions that don't have a compatible function signature need to be wrapped in a closure to rearrange the function arguments. This is similar to how the functions are wired together with *fold*, so there is less benefit over other approaches outside of the proscribed use case.

Another area where infix pipe notation is less successful is when the function composition is dynamic. Section 12.2 implements Conway's game of Life is implemented. One approach to the `epoch` function encodes the rules of Life as functions. The actual rules can then be specified at simulation time. To implement this with `magrittr` requires wrapping the whole chained call in a closure. For this type of dynamic function composition, a *fold* approach can be more appropriate.

4.3 The `lambda.r` syntax and type system

`magrittr` provides syntax for chaining function calls together. How these functions are identified and called still relies on the traditional dispatching approaches of R, which are closely linked to the type systems in the language. R has numerous dispatching/type systems including the built-in (and foundational) S3 and S4 systems, a pass-by-reference system built atop S4 called ReferenceClasses [11], plus a newer object system R6 built atop ReferenceClasses [12]. Each of these systems are based on object-oriented programming concepts. The main idea with modern OOP is that variables and functions are bound to objects as a way to organize code. Applications are written by composing objects built from a taxonomy of classes where each class defines their associated variables and functions. While meant to limit redundant structures and promote reuse, poorly designed OOP systems can actually increase interdependencies that inhibit reuse. This emphasis on OOP is a shame since so many language features in R hail from functional programming, as we've seen. To fill this void, `lambda.r` was written as a dispatching and type system based on functional programming principles and concepts, including pattern matching, multipart functions, guard expressions, and type constraints.

4.3.1 Pattern matching

The foundation of `lambda.r` is function definition. Instead of explicitly assigning a variable to a function object, `lambda.r` introduces the `%as%` operator to declaratively define functions. This is similar to how mathematical functions are defined. [4] Let's return to the Fibonacci sequence and define it using

[4] outside of the lambda calculus

function notation, so that

$$\begin{aligned} fib(0) &= 1 \\ fib(1) &= 1 \\ fib(n) &= fib(n-1) + fib(n-2), \text{when } n \geq 2 \end{aligned}$$

for $n \in \mathbb{N}$. With `lambda.r` the definition is virtually the same aside from the use of `%as%` for the assignment operator and braces to handle scoping rules. [5]

```
fib(0) %as% 1
fib(1) %as% 1
fib(n) %as% { fib(n-1) + fib(n-2) }
```

This simple example showcases a number of features of `lambda.r` including multipart function definitions and pattern matching. Multipart functions generalize the concept of piecewise functions, that have multiple definitions depending on the input criteria. In the Fibonacci sequence example, when fib is applied to 0, the first function clause will match, yielding a result of 1. When the argument is 1, the first function clause is skipped since it doesn't match the input. However, the second does, again resulting in 1. The last function defines the argument as a variable n. This acts as a catch all definition for all $n \notin \{0, 1\}$.

For a given function name, functions are evaluated lexically in the order they are defined until a matching function is found or the set of available functions is exhausted. If a function is defined early and has a liberal function signature matching many combinations of parameters, it will take precedence over variants defined later even if they have more restrictive signatures that match the parameters. To illustrate, suppose the Fibonacci sequence implemented [6]

```
fib(n) %as% { fib(n-1) + fib(n-2) }
fib(0) %as% 1
fib(1) %as% 1.
```

The first clause `fib(n)` will always match, so the clauses `fib(0)` and `fib(1)` will never be called.

Example 4.2. Multipart functions appear frequently in mathematics, particularly to get around discontinuities in a function. For example, the *sinc* function is discontinuous at 0 and is defined in two parts:

$$\text{sinc}(x) = \begin{cases} 1, & \text{when } x = 0 \\ \frac{\sin(x)}{x}, & \text{otherwise.} \end{cases}$$

[5] Function bodies typically need to be surrounded in curly braces, with the exception of scalar values and simple function calls.

[6] If following along in the R interpreter, the existing `fib` definition must be removed. When defining `lambda.r` functions, the lexical order is maintained for each function clause. Signatures that match existing function clauses will overwrite the previous definition. Whenever the situation is ambiguous, a new clause is added to the function definition.

The `lambda.r` counterpart is similar:

```
sinc(0) %as% 1
sinc(x) %as% { sin(x)/x },
```

which demonstrates how conditional expressions are naturally expressed with pattern matching and declarative syntax. Conditional blocks are a tried and true approach but also well known for being difficult to comprehend [34]. While complexity is minor in trivial examples like this, real world functions like `Reduce` and `optim` are difficult to understand, due to many nested conditional blocks. This difficulty is typically quantified by cyclomatic complexity, which is a graph-theoretic measure of software complexity [34]. This measure shows how loops and conditional blocks add to the complexity of a program. Nested conditional blocks are even worse as the cyclomatic complexity grows. Guard statements help to reduce code complexity since the code paths are isolated.

□

Strings can also be pattern matched within function definitions. Numerous built-in functions switch logic on string arguments. For example `optim` and `glm` use a string argument to indicate what algorithm is used. This choice determines which other arguments are required, along with the allowable values for the arguments. In `optim`, different blocks of code are executed based on the method specified. It's not always clear what is going on in such a function, making it difficult to debug. As a user of such a function, it is also difficult to know which combinations of arguments are allowed. The complexity of the code can be minimized by separating the implementation based on a string argument, like the optimization method. Not only does each function clause define it's own required arguments, it is also easier to understand what the function is doing since there are fewer code paths. The downside is that there may be some redundant code, but this is often outweighed by the increased clarity of the definition. Section 14 rewrites `optim` as a case study.

Example 4.3. Formal grammars have this form as well. Chapter 12 explores many such systems. To whet our appetite, L-systems are simple grammars used to simulate natural looking plants and vegetation. L-systems begin with an axiom, followed by a set of productions, or re-write rules. Consider a system for "growing" algae, defined by the axiom A along with two productions: $A \rightarrow AB$ and $B \rightarrow A$. This can be implemented using pattern matching as shown in Listing 4.3. Whenever the character `"A"` is encountered, the first clause is triggered, returning `c("A", "B")`. If `algae` is called with `"B"`, then `"A"` is returned. Any other input will result in an error.

The actual sequence is generated with a call to $fold$. Within each $fold$ iteration is a call to `lapply`, which maps each token in the input to a sequence. The `unlist` function collapses the list structure into a vector.

```
algae("A") %as% c("A", "B")
algae("B") %as% "A"
```

LISTING 4.3: The algae L-system produces a sequence of characters whose length corresponds to the Fibonacci sequence.

```
> fold(1:3, function(i,x) unlist(lapply(x, algae)), "A")
[1] "A" "B" "A" "A" "B"
```

These operations preserve order, so the sequence remains equivalent, despite the change in data structure. This preservation, or order invariance, is a useful property of R operations and is discussed in Section 6.3. The $fold$ operation applies this transformation 3 times, using the output of one iteration as the input to the next. While the algae system is remarkably simple, it exhibits a fascinating property. The length of each production corresponds to a number in the Fibonacci sequence. For example, three successive iterations correspond to the Fibonacci number 5, while four iterations results in a sequence of length 8.

□

4.3.2 Guard statements

Executing different function variants within a multipart function sometimes requires more detail than simple pattern matching. For example, what happens if `fib` is applied to a negative number? Guard statements extend the pattern matching concept to support any logical predicate that state the conditions for execution. They are defined in an additional clause of the function definition specifying one or more boolean guard expressions. Guards are analogous to pre-conditions in languages like Eiffel [36]. [7] They also behave like assertions in Python, except that they are evaluated before the function is executed and their scope is segregated from the main function body. To illustrate, the first two clauses of the Fibonacci sequence definition can be defined as a single clause with a single guard expression.

```
fib(n) %when% { n < 2 } %as% 1
```

A guard expression can also make function clauses independent of the lexical evaluation order by placing a more restrictive constraint around the acceptable input for a given clause.

```
fib(n) %when% { n >= 2 } %as% { fib(n-1) + fib(n-2) }
```

Defining `fib` with the above two function clauses, the order of the clauses no longer matters.

[7] `lambda.r` also supports post-conditions with the `%must%` clause. See [43] for more details.

Guard statements support an arbitrary number of guard expressions, separated by either a new line or a semi-colon. These predicates must all return true if and only if the given function clause executes. This implies that multiple guard expressions are logically combined with a conjunction. Hence, the two guard expressions in the following definition of `fib` are evaluated as $n \geq 2 \wedge n \in \mathbb{N}$.

```
fib(0) %as% 1
fib(1) %as% 1
fib(n) %when% {
  n >= 2
  is.numeric(n)
} %as% { fib(n-1) + fib(n-2) }
```

The value of this approach is that it cleanly separates different types of logic. Logic that ensures all arguments are well-formed goes into the guard statement, while logic related to the actual computation goes in the function body. Without the explicit separation of concerns, commingled logic quickly becomes difficult to decipher and a burden to maintain. When no function clauses match the arguments, an error is produced. Halting the processing prevents errors from propagating through the model undetected.

```
> fib(-1)
Error in UseFunction(fib, "fib", ...) : No valid function for 'fib(-1)'
```

Example 4.4. The Lindemayer system used an external *fold* to mediate the iteration. This can be added to `algae` to make the function more convenient to use. The end user mostly wants to specify the number of iterations and possibly change the initial conditions. With an extra function clause and a guard statement, this high-level interface is juxtaposed with the production rules.

```
algae(n, init="A") %when% {
  n %isa% numeric
  n > 0
} %as% {
  fold(1:n, function(i,x) unlist(lapply(x, algae)), init)
}
```

LISTING 4.4: A new function clause for algae provides the wiring to automatically iterate over the L-system. The `%isa%` operator tests whether an object is an instance of a particular type.

Now we just call `algae(3)` to get the character sequence corresponding to three iterations of the system.

```
> algae(3)
[1] "A" "B" "A" "A" "B"
```

□

Example 4.5. Section 12.3 describes a simple stock trading system. The model includes a location function

$$\mathscr{L}(x_t) = \begin{cases} 0, & \text{if } b^- > b^+ \\ -1, & \text{if } x_t < b^- \\ 1, & \text{if } x_t > b^+ \\ 0, & \text{otherwise} \end{cases}$$

that specifies the location of the current price vis-a-vis a channel. The channel is a tuple (b_-, b_+) and envelops the price series representing the lower and upper bounds. This function returns four values, based on specific logical conditions. A typical solution uses conditional blocks within the body of the code.

```
location ← function(x,b) {
  if (b$lower > b$upper) 0
  else if (x < b$lower) -1
  else if (x > b$upper) 1
  else 0
}
```

This approach is sufficient but also establishes a poor foundation. Starting with conditional blocks is no better than a straw house: good for basic shelter but isn't sturdy enough to build higher. Multipart functions cleanly separate scope between clauses, encouraging separation of concerns from the ground up. Using guard statements in four distinct clauses conveniently captures the logic distinguishing the different function clauses.

```
location(x,b) %when% { b$lower > b$upper } %as% 0
location(x,b) %when% { x < b$lower } %as% -1
location(x,b) %when% { x > b$upper } %as% 1
location(x,b) %as% 0
```

□

4.3.3 Types and type constraints

Most of the concepts in this book can be derived from the untyped lambda calculus. In practice, the typed lambda calculus is preferred since it is more general and provides the foundation for modern functional programming languages. Types provide context to data. Consider data stored in a vector. Structurally, a vector is a sequence of values. But what this sequence represents is unknown. A type provides this information, so two vectors can be distinguished programmatically. Functions can use type information to restrict arguments to specific sets of values and also for dispatching a specific function according to the type of each argument. For example we've seen how `apply` requires an array, an integer, and a function as its arguments. If

these type requirements are not satisfied, the call to `apply` will fail. This is one of the few areas where R behaves like a strongly typed language, but this type checking is usually function specific.

R provides a number of built-in types such as primitive vector types like `character`, `numeric`, `logical` and their descendants `array` and `matrix` as well as more general `list` and `data.frame` types. Another common type that is less well known is the `environment`, which is the only built-in hash table in R. It's also possible to create user-defined types. Base R provides the S3 and S4 object systems for defining types (aka classes). The beauty of S3 is its simplicity. The concept of type is just an attribute on an object. This `class` attribute is used by the S3 system to call functions specific to the type. When calling an S3 function, say `f(x)`, the function `f` actually dispatches to another function having a name equivalent to `sprintf("%s.%s", deparse(substitute(f)), class(x))`. For example, if `x` is a matrix, then the function `f.matrix` will be called. If no such function is defined an error results. S4 is more robust but suffers from being cumbersome to use. [**?**] The newer ReferenceClasses attempts to address some of the warts of S4 and provide property access via a pointer reference [11], making objects mutable. `lambda.r` introduces its own type system that integrates with the previously described function dispatching system. The goal of the `lambda.r` type system is to provide a syntax that is easy to use and consistent with functional programming principles.

Types in `lambda.r` are defined via a constructor. This function knows how to create objects of a given type. All values returned by the type constructor are automatically assigned the proper type. A trivial example is creating an `Integer` type that simply returns the value provided to the constructor.

```
> Integer(x) %as% x
```

Instances of the `Integer` type are created by calling this constructor. In the following example, `x` will have type `Integer`.

```
> x ← Integer(5)
```

Behind the scenes, the returned type is simply an S3 object, so the same `class` function can be used to inspect the type. One difference between S3 and `lambda.r` is that types in `lambda.r` must start with a capital letter. This convention is used by `lambda.r` to automatically anoint capitalized functions as type constructors. It also visually separates type constructors from other functions, making it easier to distinguish them in code.

Types in isolation aren't particularly useful. That changes when they are used to dispatch functions via type constraints. These optional declarations precede a function definition and specify the type for each argument as well as the return type. When the function is called, the type of each argument is compared with the type constraint. If the types don't match, an error is

raised. [8] Similarly, if the type of the return value doesn't match the stated type, an error is raised. Type constraints start like a function definition but instead of the `%as%` operator, the `%::%` operator is used, followed by a list of colon-separated types. Built-in types are supported just like custom `lambda.r` types. Type constraints must be declared prior to the function definition. Once declared, the constraint is *greedy* and will retain scope until another type declaration with the same number of parameters is declared or an incompatible signature is encountered.

Example 4.6. The Fibonacci sequence can be reimplemented using the `Integer` type. The reasoning is that the original `fib` function accepts more than integers, which can result in infinite recursion. We previously protected ourselves from this situation with a guard statement.

```
fib(n) %::% Integer : Integer
fib(0) %as% Integer(1)
fib(1) %as% Integer(1)
fib(n) %as% { fib(n-1) + fib(n-2) }
```

The call `fib(1)` will fail because `1` is not of type `Integer`.

```
> fib(1)
Error in UseFunction("fib", ...) : No valid function for 'fib(1)'
```

Properly typing the argument by calling `fib(Integer(1))` will give the correct output. Note that pattern matching works even with the custom type.

```
> fib(Integer(1))
[1] 1
attr(,"class")
[1] "Integer" "numeric"
```

□

The benefit of type constraints is that they provide explicit strong typing to functions *when needed*. For interactive languages like R, over-eager type checking can be a drag on productivity. Optional type constraints provide extra safety or more granular function dispatching only as necessary. Unlike standard type checking in R, which is embedded in the function body, type constraints are self-documenting. Just by looking at the function signature, it's clear what types a function expects so that guesswork is kept to a minimum.

Example 4.7. Typing in the name of a `lambda.r` function into the console prints information about the function, including any type constraints and associated function clauses. For example, entering `fib` results in

[8] unless there are multiple type constraints. In this case all type constraints with compatible function signatures will be evaluated before an error is raised.

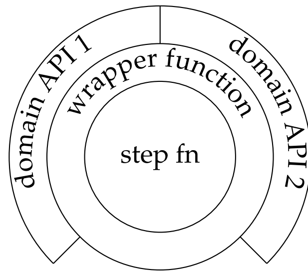


FIGURE 4.1: A visual depiction of a layered approach to function development. The outer layers provide convenient interfaces for each domain case, while the inner layers are pure mathematical functions.

```
> fib
<function>
[[1]]
fib(n) %::% Integer:Integer
fib(0) %as% ...
[[2]]
fib(n) %::% Integer:Integer
fib(1) %as% ...
[[3]]
fib(n) %::% Integer:Integer
fib(n) %as% ...,
```

which shows how the type constraint propagates to all compatible functions. Hence, every clause of `fib` requires an argument of type `Integer`. The `describe` function is used to show the actual function body.

In addition to the function, the clause number specified by the printed output must be provided.

```
> describe(fib, 2)
function(.lambda_1) { Integer ( 1 ) }
attr(,"topenv")
[1] "<environment: R_GlobalEnv> "
attr(,"name")
[1] "fib"
```

□

A common use for type constraints is working with arguments of different types. From a portability perspective, functions should be written with the most granular arguments possible. For models, this usually means pure mathematical vectors along with model parameters. These act like the greatest common factors when considering reuse. Mathematical models are context agnostic, while composite data structures are often domain specific. In contrast, convenience often dictates using composite structures. Which is the

better approach? Type constraints are great for the indecisive because you get the best of both worlds: a granular interface that's portable and broadly useful, plus a version specific to your use case. Figure 4.1 shows this layered approach to model design. At the center is the pure mathematical model, which is likely defined at the individual step or epoch level. Surrounding this is usually a wrapper function that has a more convenient interface (see the algae L-system as an example). A final layer provides domain-specific conveniences, such as data loading, parsing, and transformation logic to ensure the inputs are properly formed.

Example 4.8. Building on the Fibonacci sequence example, a new user might not be familiar with the `Integer` type and may try to call `fib` using a numeric value. From a usability perspective, adding a separate function clause to handle this scenario is beneficial.

```
fib(n) %::% numeric : Integer
fib(n) %when% { is.integer(n) } %as% fib(Integer(n))
```

Now any built-in numeric value is converted to `Integer` automatically. This new clause is the high-level interface, while the earlier definitions represent the pure model. Multiple purposeful function interfaces provide a clean separation of concerns between the mathematical logic of the function and the data manipulation logic.

□

Type constraints are compatible with all native types. One slight exception is the `function` type. Due to the precedence rules of the parser, functions cannot syntactically be declared in a type constraint. `lambda.r` addresses this issue by defining the `Function` alias to represent the function type. This approach is consistent with the convention that types in `lambda.r` are capitalized. An example of this usage is in the `lambda.tools` package [44], which defines a number of useful higher-order functions. The familiar *fold* functional defines a type constraint that uses the `Function` type.

```
fold(x, fn, acc, ...) %::% . : Function : . : ... : .
```

This type constraint spans two separate function clauses, one for one-dimensional objects and another for tabular data. It also introduces two other symbols within the type constraint. The first is the `.` (dot) type, which effectively means "don't care" or "ignore". In cases where the type of an argument cannot be known in advance, the dot type tells `lambda.r` to ignore this argument in the type constraint. Similar to the dot, the `...` (ellipsis) type tells `lambda.r` that a particular argument corresponds to the ellipsis in the function signature. Functions making use of the ellipsis have a curious signature because the cardinality of the argument list is not fixed. It is bounded from below but unbounded from above. Since type constraints match explicit typed function signatures with an argument list, the ellipsis adds unwelcome wrinkles to the computational model. The corresponding ellipsis type smooths

out these wrinkles and tells lambda.r to just match additional arguments to the ellipsis without further type checking.

Example 4.9. In Section 3.9, the using function simplified opening and closing resources. Unfortunately, R has no consistent way to handle external resources, so this function doesn't scale beyond file-like connections. For example, functions to open images, like png, don't return a connection object. These functions are called for their *side effects*. In this case, graphics output that normally goes to the screen are redirected to the newly created resource. To close the resource, a call to dev.off() is made instead of close. Inconsistencies like this can plague permissive languages like R, blocking the path to mastery. Custom types help tame the wildness of the language by consolidating the various resources in a simple type hierarchy. With this approach all resources can be conveniently opened and closed using using.

The strategy is to reimplement using so it can close any resource. Instead of close, we'll define a homonymous replacement klose. This function will use type constraints to specify different behaviors for different objects, like files or graphics devices. There's no need to reimplement file-like connections, so klose just needs to be backwards compatible. However, we do need to create types for all graphics devices, such as Png to wrap png. We'll also create a base type GraphicsDevice that these devices will inherit, which simplifies our implementation later on.

Let's look at our new version of using first in Listing **??**. Two small changes are all we need. The first is changing the default exit function to klose as mentioned above. The other change is more subtle. What is the point of having resource as an explicit statement? The reason is connected to *lazy evaluation*. Including this line ensures that the resource function is evaluated when we need it. Lazy evaluation is an advanced programming concept and is discussed in Chapter 10.

```
using ← function(resource, handler, exit=klose) {
  on.exit(exit(resource))
  resource
  tryCatch(handler(resource), error=stop)
}
```

LISTING 4.5: A generalized using implementation with a custom exit handler

Before creating the png wrapper, let's create the base GraphicsDevice type first. This type is *abstract*, since it does not correspond to any actual graphics devices. It's definition is simple and uses the ellipsis argument to create a list. The net effect is that calls to this function produce a list with type GraphicsDevice. Most base classes follow this pattern if they are only providing taxonomic structure and not functionality.

```
GraphicsDevice(...) %as% list(...)
```

The Png type extends GraphicsDevice by calling the type constructor at

the end of its body. This ensures that the returned object is also of type `GraphicsDevice`. The connection with R's graphics system is through the `png` call, which creates the actual graphics device. Any subsequent plotting in the `handler` will now be captured.

```
Png(path, width, height, ...) %as% {
  png(path, width, height, ...)
  GraphicsDevice(path=path, width=width, height=height)
}
```

Finally, the `klose` function closes the resource after the handler finishes. This multipart function has two clauses for the two types of objects it supports. The first are graphics devices, which extend our custom abstract type and are closed by `dev.off`. File-like objects use the system's `close` function, since there's no reason to re-write the default behavior. Type constraints are self-documenting and simplify code using lexical structure in place of nested conditional blocks.

```
klose(object) %::% GraphicsDevice : .
klose(object) %as% dev.off()

klose(object) %::% connection : .
klose(object) %as% close(object)
```

This simple framework comprises two type constructors and two functions. In all it's 13 lines of code. Its petiteness belies its power. Our resource management framework provides a consistent way to interact with arbitrary resources in R. To see how this eases working with devices, let's render the simulated time series of Example 3.4 to a PNG.

```
using(Png("ts.png", 800,600), function(o) plot(x))
```

We can also write the raw data to a file, using the same syntax. Now there is only one syntax to remember irrespective of the type of resource. More importantly, we're guaranteed that resources are properly closed after their use.

```
using(file("ts.data"), function(o) write(x,o))
```

This framework easily extends to support not only additional graphics devices (see Exercise 4.7) but also other I/O like database connections.

□

4.3.4 Type variables

Type constraints are a powerful component of `lambda.r`. The use of type constraints make R code strongly typed and more self-documenting. However, blanket use of strong typing comes at the cost of slower development.

Selective use of the dot type can balance type safety with ease of use. But the dot type can also be too liberal at times. In these situations, a *type variable* can be an effective compromise between ample structure and flexibility. [9] These special variables act as placeholders within a type constraint. Any single lowercase letter appearing in a type constraint is treated as a type variable. Type variables enforce type consistency across the function argument(s) and return value, while allowing the actual type to be arbitrary. Mathematically these functions have the form $f : X \rightarrow X$ in the univariate case, where X is arbitrary. For example, in the Fibonacci sequence example, the type constraint can be re-written as `fib(n) %::% a : a`. The type variable `a` indicates that the actual types can be arbitrary so long as the input and the return value have the same type.

Example 4.10. Suppose we want a more general way to specify Lindenmayer systems. An end user wants to generate a sequence based on a model and the number of iterations. One approach is to dump all the model parameters into the function signature, like `lsystem(n, axioms, productions, constants)`. This seems simple enough, but some systems have additional model parameters, like an angle for rotating a drawing head. There are a few ways types can model this system. The most application-specific is to define an `Algae` type. This type can hold all model parameters. The matching signature is `lsystem(n, model)`, where `model` ∈ `LSystem`. This function is responsible for wiring calls over multiple iterations.

```
lsystem(n, model) %::% numeric : LSystem : character
lsystem(n, model) %as% {
  fold(1:n, function(i,x) render_step(x, model), model$init)
}
```

A second, lower level function manages each step of the system. For algae, this is nothing more than calling the `algae` function. For other systems it might involve translating symbols into drawing commands.

```
render_step(x, model) %::% Algae : character
render_step(x, model) %as% {
  algae(1,x)
}
```

Notice that now we need to create types for each model, which seems excessive. This libertine approach will eventually catch up with us, manifesting as a rigid, yet brittle code base. The reason is that we used a function to encode the production rules. In this case we lose generalization, whereas using a data structure to represent the rules is more general. The current type constraint also only works for character-based L-systems. Replacing this with a type constraint quickly extends the function to support numeric L-systems.

Eschewing a large type hierarchy, we can use a single type to represent

[9] Type variables in `lambda.r` are not as theoretically rigorous as those found in Haskell.

the L-system. The lsystem signature is largely the same except we ignore the return type. Type variables can be added to `render_step` indicating that the input and output must share a type. Hence, the function only enforces type consistency between input and output.

```
lsystem(n, model) %::% numeric : LSystem : .
lsystem(n, model) %as% {
  fold(1:n, function(i,x) render_step(x, model), model$init)
}

render_step(x, model) %::% a : LSystem : a
render_step(x, model) %when% {
  ! model %hasa% angle
} %as% {
  unlist(lapply(x, function(xx) model$productions[xx]))
}
```

□

`lambda.r` is a comprehensive framework for writing numerical models and systems in a functional style. Many of the features are exclusive to `lambda.r` and offer a simpler and more convenient alternative to the object-oriented approaches within the language.

4.4 The MapReduce paradigm

A popular technique for processing data is colloquially known as MapReduce [17, 41]. This computing paradigm can process hundreds of gigabytes of data across thousands of independent compute nodes As the name suggests, it composes arbitrary *map* operations with reduce (aka *fold*) operations. The simplest formulation is a single map stage followed by a reduce stage. We already saw an example of this style of computation in the implementation of the stochastic gradient method for logistic regression in Listing 3.2. Many data processing problems can be transformed into a MapReduce problem. MapReduce systems like Hadoop are known for requiring numerous dedicated IT staff to manage them. These systems are also limited to batch processing since the performance overhead is relatively large. So what is the value of using MapReduce as a way to model a calculation? It turns out that there are numerous problems where the cost of separating the data and parallelizing the calculation is smaller than the cost of each incremental calculation in the *map* stage. Since *map* operations are by definition independent of each other, these calculations can easily be parallelized. Vector operations that can be modeled by *map* are known as "embarassingly parallel" since there is no special logic or methodology needed to make these operations parallel. This

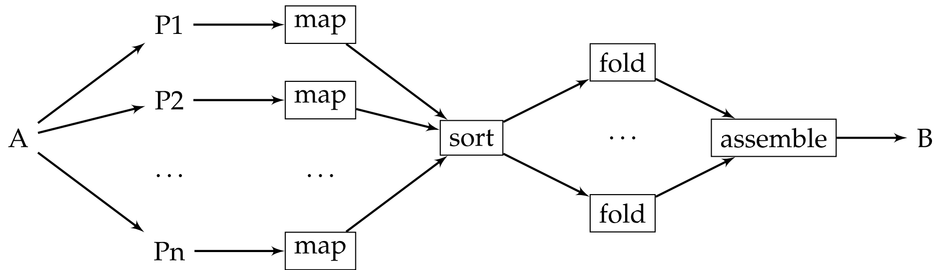


FIGURE 4.2: The MapReduce method parallelizes data processing over multiple *map* and $fold$ stages. Data is first partitioned and then sent to multiple *map* processors. The results are collected and grouped by key. Then these keys are partitioned and sent to a set of $fold$ jobs.

is in contrast to hard parallelization problems, like matrix factorization, that require effort to parallelize. Algorithm 4.4.1 shows how conceptually simple MapReduce is when viewed through the lens of functional programming.

To understand how easy it is to distribute *map* processes, let's consider a set A. We know that we can partition A into any number of disjoint subsets $P_1, \cdots, P_n$, where $A = \bigcup^n P_i$. Now think of some function $f : A \rightarrow B$ that operates on elements of A where $B = f(A)$ is the graph of A. Suppose each partition of A is on a separate compute node. Then $B = \bigcup^n f(P_i)$. This result shows that a partition of size n can be distributed to n independent compute nodes and the answer will be the same as though it were computed on a single node. What's different about the set-theoretic reasoning and the real world? When we replace sets with vectors, the only significant difference is that duplicate values can exist. These duplicates don't impact the independence of the function application and thus don't affect the partitioning scheme. One other detail is that to construct the original image, ordinality must be preserved. This is an important detail that may not matter for abstract data processing but is significant if ordinals are used across objects and calculations.

Algorithm 4.4.1: $\text{MapReduce}(x, f_{map}, f_{reduce}, jobs_m, jobs_r)$

$partition_m \leftarrow \text{partition}(x, jobs_m)$
$result_m \leftarrow \text{map}(f_{map}, partition_m)$
$group_r \leftarrow \text{map}(\lambda\, k.\{(k, \text{val}(result_m)) : \text{key}(result_m) = k\}, \text{unique}(\text{key}(result_m)))$
$partition_r \leftarrow \text{partition}(group_r, jobs_r)$
$reduce_r \leftarrow \text{map}(f_{reduce}, partition_r)$
return $(reduce_r)$

While it is easy to distribute *map* processes, alas, the same is not true of $fold$ processes. A serial dependence exists in $fold$ operations, preventing them from being trivially separated. So what role do they play in distributed

computing? For a moment let's pretend only *map* processes exist. What types of operations are possible if our world is limited to *map*? Quite frankly, our world is dull because all we can do is continually transform a given set on a per element basis. If we add in *filter*, now we can selectively transform a set and also control cardinality. But this is still limited. The inclusion of *fold* enables aggregation similar to a `GROUP BY` clause in SQL. Without *fold* there is no way to count elements within the MapReduce pipeline. So even though a reduce stage provides no distribution of work, it does provide functionality that is unavailable with *map* alone. MapReduce frameworks also include an implied sorting operation that groups keys together to pass to the reduce stage. This is a key step that makes this aggregation possible.

Example 4.11. The word count algorithm is considered the "Hello, World" equivalent for MapReduce. We'll use the `mapreduce` function provided by the `rmr2` package [5]. The basic idea is to take some document and count the word frequency for each word appearing in the document. We'll use an SEC 10-K filing for Amazon as the document. Before executing the MapReduce job, we clean the document by removing non-word characters that can confuse word count algorithms. [10].

```
clean ← function(x) {
  gsub('[,.;:\'"()]','',tolower(x))
}
```

LISTING 4.6: Remove punctuation from sentences

Let's read in the 10-K filing and split on white space. To simplify the operation, we concatenate the list of character vectors into a single character vector.

```
words ← using(file('../data/amazon_10q.txt'),
  function(rsc) do.call(c, strsplit(clean(readLines(rsc)), '\\s')))
```

Next we initiate the pipeline by calling `mapreduce`. To count the word occurrences, the *map* stage simply returns a key-value pair where the key is the word itself and the value is 1. Once all words are processed, they are sorted and aggregated by key. Each reduce job gets a vector of 1s that represent all the occurrences of the given key. Hence, all the reduce function needs to do is sum the 1s to obtain the actual count. The output is a list containing key-value pairs that represent a word and the number of occurrences in the document.

```
o ← mapreduce(words,
  map=function(k,v) keyval(v,1),
  reduce=function(k,v) keyval(k,sum(v))
)
```

In native R counting words is a trivial exercise: `table(words)`. Using MapReduce, we need to let the framework manage the iteration and data management. This is an example of inversion of control, (see Section 3.8) where the

[10]Blindly stripping all symbols from sentences also removes semantic information. This is not recommended practice in most real-world situations

end user only provides the domain-specific logic and leaves the rest to the `mapreduce` framework. [11]

□

To better understand how MapReduce works, we can implement an in-memory version of the framework. We'll replicate the interface of the `mapreduce` function from the `rmr2` package so any functions written for our emulator can work in an actual MapReduce environment like Hadoop. MapReduce jobs can operate at the individual record level and also at the block level, depending on how the input and algorithms are designed. Our implementation follows Algorithm 4.4.1, processing data at the individual record level, so each row of the data frame is treated as a separate record. A preliminary step is implementing the `keyval` list type used by `mapreduce` as a standard object container. The keyval function creates simple key-value pairs. This type wraps all data objects passing through the pipeline, providing a dimension for grouping data.

```
keyval ← function(k,v) {
  kv ← list(v)
  names(kv) ← k
  kv
}
```

LISTING 4.7: Implementation of a key-value data structure

Our in-memory `mapreduce` implementation leverages `lambda.r` to organize the code. Different function signatures are syntactically separated, making it easier to understand what each function clause is doing. The first clause converts data frames into a compatible list structure. Just as easily, we could have used a single function clause and called `as.list` on all input `x`. This is a safe type coercion on lists, since they act as a fixed point of the function. This observation is true for all target types and their corresponding type coercion functions.

```
mapreduce(x, map, reduce, mjobs, rjobs) %::% data.frame :
    Function : Function : a : a : .
mapreduce(x, map=NULL, reduce=NULL, mjobs=10, rjobs=2) %as% {
  mapreduce(as.list(x), map, reduce, mjobs, rjobs)
}
```

The `mjobs` and `rjobs` specify the number of distributed workers that operate on the map or reduce stage, respectively. Each worker gets a subset of the data to process. Our in-memory version only uses a single thread, so these parameters serve only to partition the dataset. But hold on, earlier we claimed that *fold* processes cannot be distributed. What then is the purpose

[11] This implementation actually differs from standard implementations as blocks of data are treated as a single object instead of record by record. For now, we'll conveniently ignore this semantic difference.

of multiple reduce jobs? Recall that there is an implied sorting step prior to initiating the reduce stage. This serves to group values by key. Typically multiple keys will exist, so that each key has a set of values associated with it. Each pair of key and set of values corresponds to a single $fold$ operation. Hence, for k unique keys, there will be k reduce operations. These operations are independent and can thus be partitioned similar to the *map* operations.

Before discussing the actual implementation, we first need to define a helper function `iter_fn`, which manages applying a function to each input partition. Most of this function is dedicated to identifying degenerate partitions and handling them in a logically consistent manner. The number of partitions is dictated by the number of jobs per stage. By creating a separate higher-order function, we guarantee that the user-defined function can operate on the elements within a partition without knowing anything about the partitions themselves. In the map stage, this function is used to partition the input based on the number of map jobs.

```
iter_fn ← function(f, x, job.length)
  function(i) {
    idx ← (job.length * (i-1) + 1) : (job.length * i)
    do.call(c, lapply(x[idx], function(a) {
      if (is.null(a)) return()
      if (is.na(a)[1]) return()
      if ("character" %in% class(a) && nchar(a)[1] == 0)
          return()
      f(names(a),a)
    }))
  }
```

LISTING 4.8: Intermediate higher-order function that manages iteration based on job length. The function processes each partition in succession. In the end, the lists are concatenated into a single list.

The first part of this implementation mediates the map job. The work is predominantly tied to partitioning the data and ensuring results are well formed. The end result is collected in `mresult`, which is a list of tuples $R = (K, V)$ whose original cardinals and ordinals are preserved.

```
mapreduce(x, map, reduce, mjobs, rjobs) %::% . : Function :
    Function : a : a : .
mapreduce(x, map=NULL, reduce=NULL, mjobs=10, rjobs=2) %as% {
  # Map stage
  if (!is.null(map)) {
    flog.info("Start map phase")
    mlength ← ceiling(length(x) / mjobs)
    mresult ← do.call(c, lapply(1:mjobs, iter_fn(map, x,
        mlength)))
    names(mresult) ← NULL
    mresult ← unlist(mresult, recursive=FALSE)
    mresult ← mresult[!sapply(mresult, is.null)]
```

```
    # Sort
    flog.info("Start sort")
    x ← tapply(mresult, names(mresult),I)
  }
```

Once the output set is reconstructed from the partitions, the results are grouped by key, using `tapply` and the identity function. Thus, all values $V_k \subset V$ associated with a given key k are collected into a list so that $r_k = (k, V_k) \in \tilde{R}$. This simplified set of results $\tilde{R}$ is passed to the reduce stage.

```
  # Reduce stage
  if (!is.null(reduce)) {
    flog.info("Start reduce phase")
    rlength ← ceiling(length(x) / rjobs)
    rresult ← do.call(c, lapply(1:rjobs, iter_fn(reduce, x, rlength)))

    x ← do.call(c, rresult)
    x ← x[order(names(x))]
  }
  x
}
```

As with the map stage, the input set is partitioned and then the user-defined `reduce` function applied to each key-value pair r_k.

One detail worth exploring is the fact that the reduce stage doesn't make use of $fold$. How can this be given our claim that the two are equivalent? The answer is that the wiring provided by the reduce stage operates at the batch level. Each r_k in $\tilde{R}$ is independent, whereas the values associated with a given key are interrelated. That means the set $\tilde{R}$ can be partitioned just like the *map* jobs. It is up to the `reduce` function passed as an argument to `mapreduce` to operate appropriately. Unlike arguments to $fold$, this function has the simplified signature $f : V^n \rightarrow W$.

Example 4.12. A calculation that uses multiple key groups is determining the batting average of baseball players according to their player class. This problem extends an example in [52] that computes the career year of baseball players. We define three player classes: rookies, who are playing in their first year, lame ducks that are in their final year, and the rest as being in their prime. The initial task is to append the career year y_c and player classification ξ to each player record. With this information we can calculate the batting average for each group. This information doesn't exist explicitly, so we add these variables to each player partition. The career year for a given player is defined $y_c = y_i - y_0 + 1$, where $y_0 = \min y$ is the first year a player played.

The player classification function uses this information to partition the

career years into 0 (rookies), 1 (prime), and 2 (lame ducks) and is defined

$$\xi(y_c) = \begin{cases} 0, & \text{when } y_c = 1 \\ 2, & \text{when } y_c = \max y_c \\ 1, & \text{otherwise.} \end{cases}$$

The implementation uses two vectorized `ifelse` expressions for performance purposes.

```
player_class ← function(year) {
  ifelse(year==1, 'rookie',
    ifelse(year==max(year), 'lame duck', 'prime'))
}
```

The MapReduce version constructs a processing pipeline via `mapreduce`. Data and functions need to conform to the MapReduce specification. The map stage is responsible for computing both the career year and player class for each record. It's also responsible for assigning the player class as the key used to sort records prior to the reduce stage. The reduce stage simply takes each of these partitions and computes the aggregate batting average for each player class.

```
batting_avg_mr ← function(baseball) {
  bb ← by(baseball, baseball$id, I)
  mapreduce(bb,
    map=function(k,v) {
      v$cyear ← v$year - min(v$year)
      v$class ← player_class(v$cyear)
      lapply(1:nrow(v), function(i) keyval(v$class[i], v[i,]))
    },
    reduce=function(k,v) {
      flog.info("Operate on %s",k[1])
      v ← do.call(rbind, v)
      keyval(k[1], with(v[v$ab>0,], mean(h/ab)))
    }
  )
}
```

For comparison, the same calculation can be performed using a few applications of the built-in `tapply`. The career year is first added as a new column to the baseball data frame. Not only does `tapply` mediate the map stage, but it also serves the same function as the reduce stage, where the user-defined `reduce` function is `mean`.

```
baseball$cyear ← do.call(c,
  tapply(baseball$year, baseball$id, function(y) y - min(y) + 1))
```

Computing the player class follows a similar pattern. The index to `tapply` is again the player id.

```
baseball$class ← do.call(c, tapply(baseball$cyear, baseball$id,
  player_class))
```

This function can be defined independently and called per player. Finally, `tapply` uses the player class as the new index variable to compute the mean batting average of each class.

```
with(baseball[baseball$ab>0,], tapply(h/ab, class, mean))
```

□

MapReduce systems can be intimidating due to their size and complexity. Under the hood, there's no magic and is simply a parallelized abstraction of the standard set of functional programming primitives. Code written according to FP principles can be trivially migrated from a single-threaded approach to a distributed approach using MapReduce.

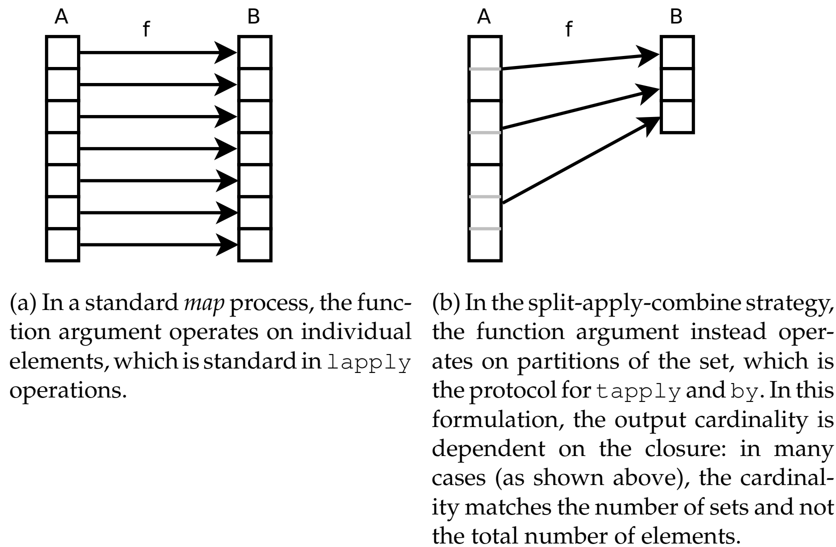
4.5 The split-apply-combine paradigm

Another data processing workflow is known as the split-apply-combine paradigm and was popularized by the `plyr` package [52]. The basic idea is to split a data frame (or other data structure) into separate groups, apply a function to each group, and then reconstruct a final data frame. The same general approach appears in `tapply`, `mapply`, and `by`, all of which partition sets based on some index vector(s). This is equivalent to creating a partition and applying a function to each partition in a *map* process. Recall the discussion in Section 2.2 regarding the mechanics of *map* vectorization. [12] Figure 2.1 describes a function $f : A \rightarrow B$ that produces the formal map from A to B when applying $map\,f\,a$ for all $a \in A$. The split-apply process is similar except the processed elements $A_i \subset A$ are sets created by a partition function $p : X \rightarrow \mathbb{N}_n$. Figure 4.3 illustrates the difference between the two approaches. Notice that by constructing p so that $n = |A|$ yields a standard *map* process!

For data scientists, partitions are often synonymous with panels. Panel data is segregated based on some variable or combination of variables, such as gender and race. An index variable is convenient because it is a simplified special case of the more general partition function. Indeed, given an ordinal i and an index variable k, the corresponding partition function $\lambda i.k[i]$ can be constructed.

To illustrate the relationship between indexes and partition functions, let's use the `baseball` dataset and calculate the career year again, as described in Example 4.12. In the `plyr` formulation, indexing is specified using the

[12] See Chapter 6 for a complete treatment of *map* vectorization.



(a) In a standard *map* process, the function argument operates on individual elements, which is standard in `lapply` operations.

(b) In the split-apply-combine strategy, the function argument instead operates on partitions of the set, which is the protocol for `tapply` and `by`. In this formulation, the output cardinality is dependent on the closure: in many cases (as shown above), the cardinality matches the number of sets and not the total number of elements.

FIGURE 4.3: Two approaches to applying a function to a set

special . quoting function[13] that wraps the indexing columns to force delayed evaluation. These columns partition the rows of the data frame by player. Each subset is passed to the built-in `transform` function that attaches a new column `cyear` to each subset, which represents the career year of a player for a given year.

```
baseball ← ddply(baseball, .(id), transform, cyear=year - min(year) + 1)
baseball ← ddply(baseball, .(cyear), transform, class=player_class(cyear))
```

The same augmentation can be implemented using the built-in `by` function:

```
baseball ← do.call(rbind,
  by(baseball, baseball$id, transform, cyear=year - min(year) + 1))
```

The primary difference is that `by` returns a list of data frames. To get a single data frame requires a manual `rbind`-ing of the subsets. So `plyr` saves us a few characters, which is always welcome. That said, it is also useful to understand the foundational idioms of R, as reflected in the `by` syntax. This example also illustrates how certain syntactic conveniences detract from standard computer science principles. For example, many stock higher-order functions utilize the ellipsis to pass additional parameters to the function argument. It's clearer using a closure to indicate the purpose of arguments.

```
baseball ← do.call(rbind,
  by(baseball, baseball$id, function(bb) {
```

[13]not to be confused with the lambda.r dot type constraint

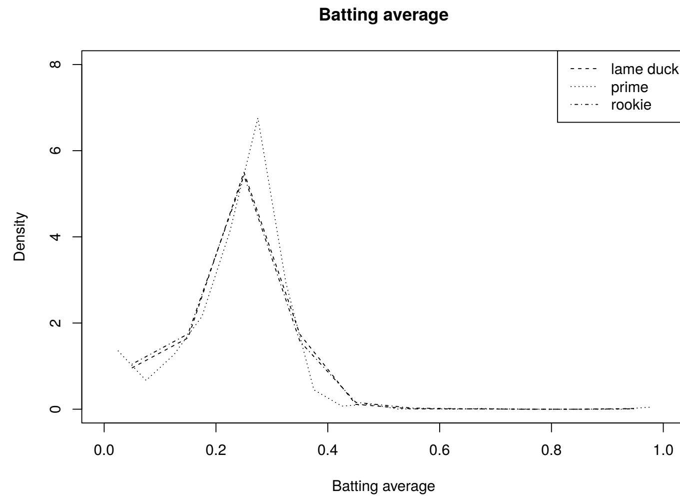


FIGURE 4.4: Batting averages for each player classification

```
    bb$cyear ← year - min(year) + 1
    bb$class ← player_class(bb$cyear)
    bb
  }))
```

If we want to use a more flexible partition function, it's necessary to use `by` to mediate the construction of an index via the partition function. An example of this is calculating the performance of different player classes.

Moving on to the analysis, we measure player performance in terms of batting average. A first pass simply asks whether rookies and lame ducks play better or worse than prime players. We can answer this question by computing the batting average in each player class, defined as μ_b = hits/at bats. For this statistic, we only expect one value per class, so the `daply` function is used. Initially we attached the player class to the baseball data frame and then computed the statistic based on that. But what exactly is the player class? Put simply, it is a partition function and can be used directly in the calculation of the batting average.

```
by(baseball, player_class(baseball), function(b) mean(b$h/b$ab)),
```

which looks a lot like

```
daply(baseball, .(class), function(b) mean(b$h/b$ab)).
```

Unfortunately some players have 0 at bats for a given year, resulting in `NA` for the mean. The quick and dirty solution is to specify `na.rm=TRUE` in the call to `mean`, but this can hide valid data issues. A better approach is to remove all years with 0 at bats *before* computing the mean.

```
daply(baseball[baseball$ab>0,], .(class), function(df) mean(df$h/df$ab))
```

Adding the pre-filtering step removes some of the syntactic convenience of `plyr`. At this point it's actually simpler and cleaner to use native R idioms.

```
with(baseball[baseball$ab>0,], tapply(h/ab, class, mean))
```

Finishing off this example, let's explore the distribution of batting averages per player classification. In addition to plotting each distribution, it would be nice to write code general enough that it can be used, even if the number of player classifications change. Another player class worth exploring is the set of players playing while injured, and also the set of players playing a full year after a debilitating injury. The following function does this by taking advantage of a generator discussed in Section 12.1.1 to create a set of ordinals on the fly. These ordinals can then be used however we see fit. In this case we use them to define unique line types for each player class. Without the generator it would be necessary to manually mange an index variable iterating over the ordinals and explicitly constructing subsets based on the index. Alternatively, we can partition the data frame and then iterate over the partition ordinals. Neither approach is as clean as using a generator to manage the ordinals.

```
plot_batting_average ← function(baseball) {
  plot(c(0,1),c(0,8), type='n',
    main='Player performance  by player class',
    xlab='Batting average', ylab='Density')
  lty ← seq.gen(1)
  lgd ← with(baseball[baseball$ab>0,], tapply(h/ab, class,
     function(x) {
    l ← lty()
    h ← hist(x, plot=FALSE)
    lines(h$mids, h$density, lty=l)
    l
  }))
  legend('topright', names(lgd), lty=lgd)
}
```

□

The split-apply-combine approach has more than a striking resemblance to MapReduce. How do these two techniques differ or are the differences cosmetic? In our original description of MapReduce, the map function operates on each element of the input vector. Oftentimes this behavior is modified so that $f(a), a \in A$ is replaced with $f(P_i)$, where $P_i \subset A$. Using this construction, each partition is passed to a map job, to be reconstructed by a succeeding reduce job. There isn't much difference between the two paradigms aside from split-apply-combine being designed for a single node while MapReduce is designed for multiple nodes. But wait, a second look reveals that

split-apply-combine describes a single type of computation. MapReduce is more general and built atop the higher-order functions *map* and *fold*. Consequently, MapReduce is not limited to a map job followed by a reduce job. More complex pipelines can be constructed comprising multiple map and reduce jobs.

4.6 The `tidyverse` canon

The so-called "tidyverse" is a collection of packages designed to simplify and standardize model development in R [7, 53]. This comprehensive approach begins with a replacement for data frames called "tibbles" and a set of packages that define a set of transformations on them. In some ways this is an extension of the standardized transformations defined in the `plyr` package both in the workflow (i.e. split-apply-combine) and in the function naming conventions. The `dplyr` package introduces six functions: `filter`, `select`, `arrange`, `mutate`, `summarize`, and `group_by`, which expands the operations available in `plyr`. It also uses `magrittr` for sequencing chains of function composition. We can recreate the batting averages across player class once again. The `%<>%` operator included in `magrittr` indicates that baseball is both the input and output of the transformation chain. The first operation partitions the data by player id, followed by the addition of the career year variable. The second transformation chain repeats the process to add the player class.

```
baseball %<>%
  group_by(id) %>%
  mutate(cyear=year - min( year) + 1) %>%
  mutate(class=ifelse(cyear==1, 0, ifelse(cyear==max(cyear), 2, 1)))
```

The same technique filters out the bad records and computes the batting average for each player class.

```
baseball %<>%
  filter(ab>0) %>%
  group_by(class) %>%
  mutate(bat.avg=mean(h/ab))
```

Data transformation follows a structured, sequential process with `dplyr`. Like MapReduce, the idea is that most problems can be framed according to the semantics defined by the package, in this case six transformation functions, or verbs. Once you've learned their semantics, in theory the time spent writing data processing code is minimized. Debugging code should also be easier, since the number of operations is intentionally limited. There are two drawbacks to this approach. Packages with strict semantics shine when problems fit within their structure. For problems that don't fit neatly, it can require significant mental gymnastics to reformulate a problem to be compatible. Strict

adherence to this workflow also leaves some strange syntactic artifacts that are difficult to decipher. For example, certain operations require the piped object, so the `.` is used to represent this value. To simplify the creation of lambda abstractions, the ~ character is repurposed from formula notation. This overloading of operators can be confusing to newcomers. Furthermore, when users learn packages with numerous innovations, the emphasis tends to be on the framework mechanics and less on transferable computer science concepts. This leads to vendor reliance and ultimately to lock-in that limits future options.

An argument for using base R features is that they are stable. The R core team is known for being conservative in changing the language as backwards compatibility is considered sacred. Package development is typically less conservative and dependent on the philosophy of the package maintainer. Consequently, bugs can arise more frequently in addition to interfaces that can change at inopportune moments with little advance warning. An example of this is the `pandas` library in Python that has had many breaking changes over it's development.

The goal of the `purrr` package [54] is to guarantee constant output types from *map* operations. Function composition requires that the output type from one function must match the expected input type of the next function. To this end, numerous *map* implementations are defined, where the name hints at the output type. Suppose we predict batting average for each player class based on other performance statistics. Inspired by the purrr documentation, let's assess the R^2 of each model.

```
baseball %>%
  split(.$class) %>%
  map(~ lm(bat.avg ~ g + rbi + hr, data=.)) %>%
  map(summary) %>%
  map_dbl("r.squared")
```

Using a naming convention to indicate the argument types is reminiscent of the naming conventions used in `plyr`. While admirable, this explicitness is largely unnecessary. As we saw in Section 3.2, types can be deduced based on the analysis of signatures. This process is similar to working out the final dimensions of a matrix operation. For *map* processes, type stability can be achieved with `lapply` or by setting `simplify=FALSE` with other members of the `apply` family. On the other hand, the output type of $fold$ operations is csse-specific. If types need to be guaranteed, then a `lambda.r` type constraint can be used on the wrapper function. Another approach is to construct a monad to manage strong typing. This latter approach is more advanced and is discussed in Section 10.5.

4.7 Summary

It's an exciting time for functional programming. Big data and distributed computing have created an environment where functional programming methods are essential to the data science curriculum. R has enjoyed the fruits of this renaissance. Numerous packages have endowed R with additional FP capabilities. Packages like `magrittr` and `lambda.r` add new semantics to the language, while others like MapReduce and `dplyr` define structured approaches to computing based on FP principles. When choosing a framework the question is whether the benefit outweighs the cost of adoption. There is also the question of what long term effect simplifying frameworks have on programming skill. On the one hand, performing data science in R should be accessible. On the other, data scientists should be encouraged to become better programmers and not rely on crutches. The choice is largely a matter of philosophy and out of scope for this book.

4.8 Exercises

Exercise 4.1. Rewrite `Reduce` using `lambda.r`. How many clauses are appropriate? Is it better to use pattern matching or not?

Exercise 4.2. The Kronecker delta is defined $\delta(i, j) = \begin{cases} 0, & \text{if } i \neq j \\ 1, & \text{otherwise} \end{cases}$. Implement this function using guard expressions in `lambda.r`.

Exercise 4.3. Implement covariance of two variables as a MapReduce algorithm.

Exercise 4.4. The astute reader might notice that although the `fib` function is now restricted to an argument of `Integer` type, the `Integer` type has no such type restriction. So all we've done is created the appearance of type safety. Use a type constraint and/or guard statement to add type safety to the `Integer` type constructor.

Exercise 4.5. Implement covariance of two variables using `dplyr`. How does it compare to the MapReduce version?

Exercise 4.6. Assume `lapply` is a `lambda.r` function. Write the least permissive but most general type constraint for `lapply`. Are there any cases that won't work with your implementation?

Exercise 4.7. Extend the resource management framework in Example 4.9 by adding a `Pdf` type.

Exercise 4.8. Implement the `dplyr` function `group_by` as a *fold* operation. What's different about the implementation than the one in the package?

Exercise 4.9. Under what circumstances would it make sense to create custom types for the baseball analysis?