Essential Computational Thinking

Computer Science from Scratch

Ricky J. Sethi
Essential Computational Thinking

Computer Science from Scratch

Ricky J. Sethi
For Rohan, Megan, Surrinder, Harindar, and especially the team of the two Harindars.
Acknowledgments

I would like to thank my current and former students for the feedback and help in crafting a clearer, more concise book than my meandering initial drafts. In particular, thanks to Kabir Chug and Bryce “Texas” Shurts for assisting with the reviews and administration of the text. I’d also like to acknowledge my Fitchburg State University colleagues for their feedback and encouragement, and particularly Kevin B. Austin, Natasha Kurtonina, Nadimpalli V.R. Mahadev, and John Russo for going the extra mile.

I’m grateful to all of the reviewers, especially Dmitry Zinoviev of Suffolk University, Frank J. Kachurak of Pennsylvania State University, and Siva Jasthi of Metropolitan State University, who so generously donated their time and expertise to help produce a much better and more correct book with their detailed and insightful feedback.

This book depends on quite a few open source resources and I’d like to acknowledge some that were indispensable to its development like LaTeXTemplates.com, www.pexels.com, www.pixabay.com, venngage.com, htmltidy.net, repl.it, and generatedata.com.

And last, but certainly not least, I’d like to thank my family, especially my mother for the constant encouragement and positive reinforcement; my father, for reading endless drafts and commenting thoughtfully and proudly on each one; my son, for letting me bounce ideas off him and making life a joy, in general; and my beautiful wife, for being a constant support and always helping edit drafts in a topic that’s not only outside her area of expertise but also includes occasional sentences full of equations, the bane of her existence.

I should add that any errors which remain, and I’m sure there will be quite a few, are my sole responsibility. The book resource page, available at http://research.sethi.org/ricky/book/, will also contain the reams of errata as they’re discovered.

Ricky J. Sethi
http://research.sethi.org/ricky/
January 2020
Preface

Why a book on CS0 from scratch?

Most of us write the books that we would have wanted to read and this is no different. As such, it follows a normal CS0 breadth-first approach but lays particular emphasis on computational thinking and related topics in data science.

My hope is this book will help build a theoretical and practical foundation for learning computer science from the ground up. It’s not quite from first principles as this is not a rigorous text. But, following William of Occam’s Razor, it is placed on a firm mathematical footing that starts with the fewest assumptions.

And so we delve into defining elementary ideas like data and information; along the way, we quantify these ideas and eke out some of their links to fundamental physics in an effort to show the connection between computer science and the universe itself. In fact, all the laws of physics can be represented as computable functions, and we can, in a very real sense, think of computer science as the most basic representation of the physical universe.

The eminent physicist John A. Wheeler went so far as to say, “…the universe is made of information; matter and energy are only incidental.” I view the role of this book, at its theoretical core, to be to help explore, elucidate, and communicate this deep and perhaps surprising relationship between physics, information, and computation.

Target Audience

This book would be most appropriate for highly-motivated undergraduates who have a significant technical bent and a genuine interest in computer science. It is also appropriate for non-CS graduate students or non-CS professionals who are interested in learning more about computer science and programming for either professional or personal development.

So what kind of a background do you need to appreciate these various connections and learn some of the more advanced material? Although you should be quantitatively inclined, you don’t need any specific background in mathematics. Whatever math we need, we’ll derive or explain along the way. Things might look heinous at times but all the tools you need should be in here.
Book Organization

Computer science is a diverse field and, in Part 1, we explore some of its theoretical bases in a connected but wide-ranging manner with an emphasis on breadth, not depth. Some chapters might still pack quite a punch but, if students are interested in computer science, they should get an overview of the field in large measure, not necessarily in its entirety. Not only is computer science an enormously broad area but this book, to paraphrase Thoreau, is limited by the narrowness of my experience and vision to those areas that caught my eye.

For that matter, this book is not a comprehensive introduction to a particular programming language, either. By necessity, we are constrained to only meeting those parts of Python or Java that support our understanding of fundamental computational ideas. In general, computer scientists tend to be a language agnostic bunch who pick the most appropriate language for a particular problem and that’s the approach in this book, as well. But we should still get a sense of the large variety of topics that underly our study of computer science, from the abstract mathematical bases to the physical underpinnings.

Once the theoretical foundation is laid in Part 1, we shift gears to a purely pragmatic exploration of computing principles. So, in Part 2, we learn the basics of computation and how to use our greatest tool: programming. These chapters should be short and approachable. Students will hopefully find them digestible in brief sittings so they can immediately apply the ideas they’ve learned to start writing substantial programs, or computational solutions. Computer science is an inherently empirical science and few sciences can offer this unique opportunity to let students actually implement and execute theoretical ideas they have developed.

In Part 3, we’ll explore some relatively sophisticated computational ideas. We’ll both meet powerful programming concepts as well as investigate the limits of computational problem solving. We’ll also increase the tools in our toolkit by learning about object-oriented programming, machine learning, data science, and some of the underlying principles of software engineering used in industry. In addition, online supplementary chapters will address topics such as P vs. NP, Big O notation, GUI development, etc.

How to use this book

Depending upon student or instructor interests, Part 1 can be considered completely optional and skipped entirely. The most important aspects of Part 1 are summarized in Ch. 3 and, if you don’t want to emphasize the theoretical underpinnings, you can safely choose to start directly with Part 2.

Similarly, in Part 3, you can pick and choose any chapters that interest you as each chapter in Part 3 is independent of the other chapters. For a typical semester-length course, I spend about 1-2 weeks on Part 1, cover all of Part 2, and pick topics of interest from Part 3 depending on the particular course.

You can also access supplementary information for the text at http://research.sethi.org/ricky/book/, including complementary videos and auto-graded problem sets.

Nota Bene: This book uses standard bibliographical notation as is common in computer science literature so references are indicated by a number in brackets, like [XX]. It also uses bold and italics to define phrases and emphasize concepts inline. Finally, it contains call out boxes like IDEA boxes, NOTE boxes, etc., which are important points that should be highlighted separately or extend a concept without breaking the main narrative.¹

Okay, that’s enough build-up... strap yourself in and let’s get started on our adventure!

¹The same applies to footnotes, of course.
## Contents

### Theory: What Is Computer Science?

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>On the Road to Computation</td>
<td>3</td>
</tr>
<tr>
<td>0.1</td>
<td>What Is Knowledge?</td>
<td>3</td>
</tr>
<tr>
<td>0.1.1</td>
<td>Step 1: Gather the Facts</td>
<td>3</td>
</tr>
<tr>
<td>0.1.2</td>
<td>Step 2: Contextualize the Data</td>
<td>4</td>
</tr>
<tr>
<td>0.1.3</td>
<td>Step 3: Take Some Action!</td>
<td>5</td>
</tr>
<tr>
<td>0.2</td>
<td>Declarative vs Imperative Knowledge</td>
<td>6</td>
</tr>
<tr>
<td>0.3</td>
<td>The Key to Science</td>
<td>7</td>
</tr>
<tr>
<td>0.4</td>
<td>Computer Science: The Study of Computation</td>
<td>8</td>
</tr>
<tr>
<td>0.5</td>
<td>A Review of Functions</td>
<td>8</td>
</tr>
<tr>
<td>0.5.1</td>
<td>Graphical Representation</td>
<td>9</td>
</tr>
<tr>
<td>0.5.2</td>
<td>Relational Representation</td>
<td>10</td>
</tr>
<tr>
<td>0.5.3</td>
<td>Functional Representation</td>
<td>11</td>
</tr>
<tr>
<td>0.6</td>
<td>Computable Functions</td>
<td>12</td>
</tr>
<tr>
<td>0.6.1</td>
<td>Computation and Algorithms</td>
<td>13</td>
</tr>
<tr>
<td>0.7</td>
<td>Talking in Tongues: Programming Languages</td>
<td>14</td>
</tr>
<tr>
<td>0.7.1</td>
<td>Going to Church-Turing</td>
<td>16</td>
</tr>
<tr>
<td>1</td>
<td>Computational Thinking and Information Theory</td>
<td>21</td>
</tr>
<tr>
<td>1.1</td>
<td>What Is Thinking?</td>
<td>21</td>
</tr>
</tbody>
</table>

Contact adopt@cognella.com, 800.200.3908, for more information © COGNELLA

DO NOT DUPLICATE, DISTRIBUTE, OR POST
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.2</td>
<td>Deductive vs Inductive Thinking</td>
<td>22</td>
</tr>
<tr>
<td>1.3</td>
<td>Thinking About Probabilities</td>
<td>23</td>
</tr>
<tr>
<td>1.3.1</td>
<td>Calculating the Probabilities</td>
<td>24</td>
</tr>
<tr>
<td>1.3.2</td>
<td>Samples and Populations in Statistics</td>
<td>24</td>
</tr>
<tr>
<td>1.3.3</td>
<td>Conditional Probabilities and Sample Spaces</td>
<td>25</td>
</tr>
<tr>
<td>1.4</td>
<td>Logical Thinking</td>
<td>25</td>
</tr>
<tr>
<td>1.4.1</td>
<td>Origins of Formal Logic</td>
<td>25</td>
</tr>
<tr>
<td>1.4.2</td>
<td>Propositional Logic and Argumentation</td>
<td>26</td>
</tr>
<tr>
<td>1.4.3</td>
<td>Declarative vs Imperative Statements</td>
<td>28</td>
</tr>
<tr>
<td>1.5</td>
<td>Computational Thinking and Computational Solutions</td>
<td>29</td>
</tr>
<tr>
<td>1.5.1</td>
<td>Computational Thinking Overview</td>
<td>30</td>
</tr>
<tr>
<td>1.6</td>
<td>Two Fundamental Models of Programming</td>
<td>32</td>
</tr>
<tr>
<td>1.6.1</td>
<td>Declarative vs Imperative Programming Languages</td>
<td>33</td>
</tr>
<tr>
<td>1.7</td>
<td>Pseudocode</td>
<td>34</td>
</tr>
<tr>
<td>1.7.1</td>
<td>Pseudocode Expanded: Now with Twice the Pseudocode</td>
<td>35</td>
</tr>
<tr>
<td>1.8</td>
<td>Functional and Imperative Models of Computation</td>
<td>37</td>
</tr>
<tr>
<td>1.8.1</td>
<td>Computation in General</td>
<td>38</td>
</tr>
<tr>
<td>1.9</td>
<td>Information Theory</td>
<td>38</td>
</tr>
<tr>
<td>1.9.1</td>
<td>The Birth of Information Theory</td>
<td>39</td>
</tr>
<tr>
<td>1.9.2</td>
<td>First Rule of Communication Club: There Is No Meaning</td>
<td>39</td>
</tr>
<tr>
<td>1.9.3</td>
<td>Shannon Information</td>
<td>42</td>
</tr>
<tr>
<td>1.9.4</td>
<td>Information, Uncertainty, and... Surprise!</td>
<td>44</td>
</tr>
<tr>
<td>1.9.5</td>
<td>Data → Information → Knowledge Revisited</td>
<td>45</td>
</tr>
<tr>
<td>1.10</td>
<td>Shannon’s Information Entropy</td>
<td>46</td>
</tr>
<tr>
<td>1.10.1</td>
<td>Entropy in Physics</td>
<td>47</td>
</tr>
<tr>
<td>1.10.2</td>
<td>Information Entropy and Physical Entropy</td>
<td>50</td>
</tr>
<tr>
<td>1.10.3</td>
<td>Demons Arise!</td>
<td>52</td>
</tr>
<tr>
<td>1.10.4</td>
<td>Connecting Information Entropy and Thermodynamic Entropy</td>
<td>54</td>
</tr>
<tr>
<td>1.10.5</td>
<td>The Universe Itself Is Informational</td>
<td>56</td>
</tr>
<tr>
<td>2.0</td>
<td>Computational Problem Solving</td>
<td>59</td>
</tr>
<tr>
<td>2.1</td>
<td>What Is a Model?</td>
<td>59</td>
</tr>
<tr>
<td>2.1.1</td>
<td>Models and Abstractions</td>
<td>60</td>
</tr>
<tr>
<td>2.2</td>
<td>Data Representations</td>
<td>62</td>
</tr>
<tr>
<td>2.2.1</td>
<td>Data Structures</td>
<td>63</td>
</tr>
<tr>
<td>2.3</td>
<td>Number Representations</td>
<td>64</td>
</tr>
<tr>
<td>2.3.1</td>
<td>Positional Number Representations</td>
<td>64</td>
</tr>
<tr>
<td>2.3.2</td>
<td>Computational Representations</td>
<td>65</td>
</tr>
<tr>
<td>2.3.3</td>
<td>Physical Representations</td>
<td>65</td>
</tr>
<tr>
<td>2.4</td>
<td>Digital Representations</td>
<td>67</td>
</tr>
<tr>
<td>2.5</td>
<td>Boolean Algebra</td>
<td>68</td>
</tr>
<tr>
<td>2.5.1</td>
<td>Boolean Algebra and Digital Circuits</td>
<td>69</td>
</tr>
</tbody>
</table>
II Basics: Algorithmic Expression

3 Computational Thinking and Structured Programming .......................... 91
3.1 Review of Computation .......................... 91
3.1.1 Modern Digital Computers .......................... 91
3.1.2 Evolution of Computers .......................... 92
3.1.3 What Is a Computer Program? .......................... 95
3.2 Computational Thinking Basics .......................... 96
3.3 Minimal Instruction Set .......................... 96
3.4 Getting Started with Python .......................... 98
3.4.1 What Is Python and Where Do I Get It? .......................... 98
3.4.2 “Hello World” in Python .......................... 99
3.4.3 Error, Error... Does Not Compute .......................... 100
3.5 Syntax, Semantic, or Logic Errors .......................... 101
3.5.1 Debugging .......................... 102
3.6 State of a Computational System .......................... 103
3.7 Natural vs Formal Languages .......................... 104
3.8 Translating Your Programs .......................... 105
3.9 Playing with Python .......................... 107
3.9.1 Python and Mathematics .......................... 108
3.10 An Example Using Computational Thinking .......................... 109

4 Data Types and Variables .................................................. 111
4.1 Different Types of Data .................................................. 111
4.2 Data Type = Values + Operations .................................................. 112
Advanced: Data and Computation

8  Object-Oriented Programming (OOP) ............................................. 169
8.1  Zen and the Art of Object-Oriented Programming .......................... 169
8.1.1  Installing Java ........................................................................ 170
8.2  The Road to OOP ........................................................................ 173
8.2.1  Designing the Objects ................................................................. 175
8.3  Solving Problems with Software Objects ...................................... 178
8.3.1  A Detailed Example in Java ......................................................... 180
8.3.2  Solving Computational Problems Using Objects ...................... 183
8.3.3  OOP Fundamentals .................................................................... 185
8.4  What Is Java? ............................................................................... 186
8.4.1  Class Structure .......................................................................... 186
8.4.2  Object-Oriented Concepts and Computational Thinking Principles 190
8.4.3  OOP Advantages and Motivations ............................................ 194
8.5  Data Structures and I/O in Java .................................................. 195
8.5.1  Arrays and ArrayLists ................................................................. 197
8.5.2  Exception Handling in Java ......................................................... 198
8.5.3  I/O in Java ................................................................................. 200

9  Databases and MDM ....................................................................... 203
9.1  Mind Your Data ............................................................................ 203
9.1.1  Data Abstraction ....................................................................... 204
9.1.2  Data Models and Master Data Management ............................... 206
9.1.3  Three LEVELS of Data Model Abstractions .............................. 208
9.2  Database Management .................................................................. 208
9.2.1  DataBase Life Cycle (DBLC) PHASES .................................... 210
9.2.2  LAYERS of Database Abstraction ............................................ 213
9.2.3  Client-Server Database Application TlERS ............................... 214
9.3  Relational Database Model .......................................................... 214
9.3.1  Data Models and Database Models .......................................... 215
9.4  Database Modeling and Querying ................................................ 218
9.4.1  Databases, Data Warehouses, and Data Lakes ......................... 218
9.4.2  Structured Query Language (SQL) ......................................... 219
9.4.3  Embedded SQL ......................................................................... 220
9.5  Normalization .............................................................................. 220
9.5.1  1st Normal Form (1NF) ............................................................... 223
9.5.2  2NF and 3NF ............................................................................ 224
9.5.3  Denormalization ........................................................................ 224
10 Machine Learning and Data Science .............................................. 227
10.1 Computational Thinking and Artificial Intelligence .......................... 227
  10.1.1 What Is Machine Learning? ........................................... 229
10.2 Getting Started with Machine Learning ........................................ 229
  10.2.1 Movie Night with the Family ....................................... 231
  10.2.2 Building the Generalized Movie Night Model ...................... 232
  10.2.3 Selecting a Movie Night Algorithm .................................. 234
  10.2.4 Implementing the Movie Night Decision Trees ...................... 235
  10.2.5 Ensemble Methods .................................................. 239
  10.2.6 Miss the Random Forests for the Decision Trees ................. 240
  10.2.7 Generalized Classification Task .................................... 241
  10.2.8 Unsupervised Learning via k-Means ................................ 243
10.3 Elements of Machine Learning .................................................. 247
  10.3.1 Machine Learning Writ Large ....................................... 249
  10.3.2 ML Definition: Tasks ............................................... 250
  10.3.3 ML Definition: Models ............................................. 251
  10.3.4 ML Definition: Performance Measures ............................ 253
  10.3.5 Model Building and Assessment .................................... 253
10.4 Data Science and Data Analytics .............................................. 255
  10.4.1 Business Intelligence/Business Analytics ........................... 256
  10.4.2 The Jataka Analytics .............................................. 257
  10.4.3 Is Data Science a Science? ....................................... 259
10.5 Bayesian Inference and Hypothesis Testing ................................ 260
  10.5.1 Null Hypothesis Significance Testing .............................. 260
  10.5.2 Hypothesis Testing .................................................. 262
  10.5.3 Bayes Theorem ..................................................... 264
  10.5.4 Bayesian Hypothesis Testing ....................................... 265
  10.5.5 Bayesian Inference and Statistics .................................. 266
  10.5.6 Supervised Learning with Naive Bayes ............................ 267
  10.5.7 Common Statistical Metrics ........................................ 268
10.6 The Entropy Strikes Back ......................................................... 271
  10.6.1 Random Variables ................................................... 272
  10.6.2 Shannon Information, Shannon Information Entropy, and Information Gain ............................................. 275
  10.6.3 Entropy and Information in Terms of Number of States of Physical Systems ........................................ 277
  10.6.4 Entropy for Classification ......................................... 278
  10.6.5 Entropy for Decision Trees ....................................... 279
10.7 Learning in Decision Trees ...................................................... 279
  10.7.1 Sources of Error in Machine Learning .............................. 282
10.8 Machine Learning, Data Science, and Computational Thinking ............. 283

Bibliography ..................................................................................... 285

Index ................................................................................................. 287
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>Collection of sensor temperature values</td>
<td>4</td>
</tr>
<tr>
<td>0.2</td>
<td>Plot of Temperature vs. Time</td>
<td>5</td>
</tr>
<tr>
<td>0.3</td>
<td>Transformation of Data to Information to Knowledge</td>
<td>5</td>
</tr>
<tr>
<td>0.4</td>
<td>A Function is a Correspondence</td>
<td>9</td>
</tr>
<tr>
<td>0.5</td>
<td>One-to-One and Many-to-One Mappings</td>
<td>10</td>
</tr>
<tr>
<td>0.6</td>
<td>Graphical Representations of the two functions</td>
<td>10</td>
</tr>
<tr>
<td>0.7</td>
<td>A function can be represented as a black box</td>
<td>11</td>
</tr>
<tr>
<td>0.8</td>
<td>Some essential ideas in the theory of computation</td>
<td>20</td>
</tr>
<tr>
<td>1.1</td>
<td>Computational Thinking Steps</td>
<td>31</td>
</tr>
<tr>
<td>1.2</td>
<td>Signal degradation image directly from Hartley’s paper</td>
<td>40</td>
</tr>
<tr>
<td>1.3</td>
<td>Maxwell’s Demon at work</td>
<td>53</td>
</tr>
<tr>
<td>1.4</td>
<td>Data to Information to Knowledge</td>
<td>57</td>
</tr>
<tr>
<td>2.1</td>
<td>Accuracy vs Precision</td>
<td>60</td>
</tr>
<tr>
<td>2.2</td>
<td>Values, Data, and Data Representation</td>
<td>63</td>
</tr>
<tr>
<td>2.3</td>
<td>Positional Number Sysems</td>
<td>64</td>
</tr>
<tr>
<td>2.4</td>
<td>8-fingered Simpsons</td>
<td>66</td>
</tr>
<tr>
<td>2.5</td>
<td>A simple electrical switch example</td>
<td>67</td>
</tr>
<tr>
<td>2.6</td>
<td>AND and OR circuits</td>
<td>67</td>
</tr>
<tr>
<td>2.7</td>
<td>Shannon and Boolean Circuits</td>
<td>70</td>
</tr>
<tr>
<td>2.8</td>
<td>Full Adder circuit based only on NAND gates.</td>
<td>71</td>
</tr>
<tr>
<td>FIGURES</td>
<td>Page</td>
<td></td>
</tr>
<tr>
<td>-------------</td>
<td>------</td>
<td></td>
</tr>
<tr>
<td>2.9 Information Processing Workflow</td>
<td>79</td>
<td></td>
</tr>
<tr>
<td>2.10 Data Processing Workflow for Temperature Problem from Section 0.1</td>
<td>86</td>
<td></td>
</tr>
<tr>
<td>2.11 Difference between CS, CIS, and IT</td>
<td>87</td>
<td></td>
</tr>
<tr>
<td>3.1 The Fetch-Execute Cycle in the CPU</td>
<td>95</td>
<td></td>
</tr>
<tr>
<td>3.2 NetBeans IDE</td>
<td>99</td>
<td></td>
</tr>
<tr>
<td>3.3 Python IDLE Shell and Text Editor</td>
<td>100</td>
<td></td>
</tr>
<tr>
<td>3.4 Python Virtual Machine and Compiler</td>
<td>106</td>
<td></td>
</tr>
<tr>
<td>3.5 Syntax, Semantic, and Logical Errors</td>
<td>110</td>
<td></td>
</tr>
<tr>
<td>4.1 An Informal Model of Fundamental Program Elements</td>
<td>120</td>
<td></td>
</tr>
<tr>
<td>5.1 Syntax of the selection control structure in Python</td>
<td>126</td>
<td></td>
</tr>
<tr>
<td>5.2 Gauss’s Summation Trick</td>
<td>129</td>
<td></td>
</tr>
<tr>
<td>6.1 Abstract Fastener Type</td>
<td>136</td>
<td></td>
</tr>
<tr>
<td>6.2 Numeric Abstract Data Type</td>
<td>137</td>
<td></td>
</tr>
<tr>
<td>6.3 Data Types and Representations</td>
<td>145</td>
<td></td>
</tr>
<tr>
<td>7.1 Coffee Making Metaphor</td>
<td>147</td>
<td></td>
</tr>
<tr>
<td>7.2 Coffee Maker Metaphor</td>
<td>148</td>
<td></td>
</tr>
<tr>
<td>7.3 Coffee Maker Function Metaphor</td>
<td>149</td>
<td></td>
</tr>
<tr>
<td>7.4 Exception Handling</td>
<td>166</td>
<td></td>
</tr>
<tr>
<td>8.1 NetBeans IDE</td>
<td>171</td>
<td></td>
</tr>
<tr>
<td>8.2 Java Application Outline</td>
<td>172</td>
<td></td>
</tr>
<tr>
<td>8.3 Modular Programming Representation</td>
<td>175</td>
<td></td>
</tr>
<tr>
<td>8.4 Class Blueprint and Two Human Objects: Joe and Hannah</td>
<td>178</td>
<td></td>
</tr>
<tr>
<td>8.5 Object Reference Variable vs Primitive Variable</td>
<td>185</td>
<td></td>
</tr>
<tr>
<td>8.6 Java programs, classes, and methods</td>
<td>187</td>
<td></td>
</tr>
<tr>
<td>8.7 Association, Aggregation, and Composition</td>
<td>193</td>
<td></td>
</tr>
<tr>
<td>9.1 The raw notes for the home business transformed into structured data</td>
<td>204</td>
<td></td>
</tr>
<tr>
<td>9.2 The Data → Information → Knowledge data processing workflow</td>
<td>204</td>
<td></td>
</tr>
<tr>
<td>9.3 The idea of abstraction in the context of a map</td>
<td>205</td>
<td></td>
</tr>
<tr>
<td>9.4 The data processing workflow for the Contacts “database”</td>
<td>206</td>
<td></td>
</tr>
<tr>
<td>9.5 Database Life Cycle (DBLC)</td>
<td>211</td>
<td></td>
</tr>
<tr>
<td>9.6 ERD for Customers</td>
<td>212</td>
<td></td>
</tr>
<tr>
<td>9.7 Application Development Tiers and Layers of Database Abstraction</td>
<td>215</td>
<td></td>
</tr>
<tr>
<td>9.8 Database Normalization</td>
<td>222</td>
<td></td>
</tr>
<tr>
<td>10.1 Machine Learning Definition</td>
<td>230</td>
<td></td>
</tr>
<tr>
<td>10.2 Plot of Cats and Dogs by Features</td>
<td>231</td>
<td></td>
</tr>
<tr>
<td>FIGURES</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-----------------</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10.3 Machine Learning Model</td>
<td>232</td>
<td></td>
</tr>
<tr>
<td>10.4 Decision Tree Classification Example</td>
<td>234</td>
<td></td>
</tr>
<tr>
<td>10.5 Movie Night Decision Tree Image</td>
<td>238</td>
<td></td>
</tr>
<tr>
<td>10.6 Ensemble Method: Stacking.</td>
<td>240</td>
<td></td>
</tr>
<tr>
<td>10.7 Ensemble Method: Bagging.</td>
<td>240</td>
<td></td>
</tr>
<tr>
<td>10.8 Ensemble Method: Boosting.</td>
<td>241</td>
<td></td>
</tr>
<tr>
<td>10.9 Machine Learning Model Detail</td>
<td>242</td>
<td></td>
</tr>
<tr>
<td>10.10 Supervised learning approach for a general predictive model</td>
<td>244</td>
<td></td>
</tr>
<tr>
<td>10.11 Iris Dataset Scatterplot</td>
<td>246</td>
<td></td>
</tr>
<tr>
<td>10.12 Structured, Un-Structured, and Semi-Structured Data for the Iris Dataset.</td>
<td>248</td>
<td></td>
</tr>
<tr>
<td>10.13 How Artificial Intelligence (AI) and Machine Learning (ML) are related.</td>
<td>249</td>
<td></td>
</tr>
<tr>
<td>10.14 Eight Common ML Approaches</td>
<td>250</td>
<td></td>
</tr>
<tr>
<td>10.15 Traditional Computer Science vs Machine Learning paradigms</td>
<td>251</td>
<td></td>
</tr>
<tr>
<td>10.16 Machine Learning Categories</td>
<td>251</td>
<td></td>
</tr>
<tr>
<td>10.17 Elements of Classical Machine Learning</td>
<td>252</td>
<td></td>
</tr>
<tr>
<td>10.18 Splitting the full dataset for supervised learning.</td>
<td>253</td>
<td></td>
</tr>
<tr>
<td>10.19 Training Testing Data</td>
<td>254</td>
<td></td>
</tr>
<tr>
<td>10.20 Hypothesis Testing, Two-Tailed Z Test, and Probability Tree</td>
<td>263</td>
<td></td>
</tr>
<tr>
<td>10.21 Confusion Tables</td>
<td>269</td>
<td></td>
</tr>
<tr>
<td>10.22 Rényi Entropy</td>
<td>278</td>
<td></td>
</tr>
<tr>
<td>10.23 Decision Tree Example with Entropies</td>
<td>281</td>
<td></td>
</tr>
</tbody>
</table>
## Tables

<table>
<thead>
<tr>
<th>Section Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1 Organized Sensor Values</td>
<td>4</td>
</tr>
<tr>
<td>1.1 List of Variables and Control Structures in Algorithm 1.1</td>
<td>34</td>
</tr>
<tr>
<td>2.1 State Table for PB&amp;J System</td>
<td>82</td>
</tr>
<tr>
<td>2.2 State Transition Table for a simplified Turing Machine</td>
<td>84</td>
</tr>
<tr>
<td>4.1 Simple symbol table for first two statements</td>
<td>116</td>
</tr>
<tr>
<td>4.2 Final symbol table after the third statement</td>
<td>117</td>
</tr>
<tr>
<td>4.3 Some Python Operators</td>
<td>117</td>
</tr>
<tr>
<td>5.1 Boolean and Relational Operators in Python</td>
<td>125</td>
</tr>
<tr>
<td>5.2 ASCII Art for a backslash made out of asterisks</td>
<td>132</td>
</tr>
<tr>
<td>6.1 Two Kinds of Fasteners: Two Fastener Structures</td>
<td>136</td>
</tr>
<tr>
<td>6.2 Two Python Numeric Data Structures</td>
<td>137</td>
</tr>
<tr>
<td>8.1 Class Blueprint for Human Objects</td>
<td>176</td>
</tr>
<tr>
<td>8.2 Physical, conceptual, and software objects</td>
<td>177</td>
</tr>
<tr>
<td>9.1 Different levels of abstraction</td>
<td>207</td>
</tr>
<tr>
<td>9.2 Conceptual, Logical, and Physical Data Models</td>
<td>209</td>
</tr>
<tr>
<td>9.3 Database models extend data models by adding in constraints.</td>
<td>216</td>
</tr>
<tr>
<td>9.4 Customer Table in the DBMS</td>
<td>217</td>
</tr>
<tr>
<td>Section</td>
<td>Page</td>
</tr>
<tr>
<td>----------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>9.5 Orders Table in the DBMS</td>
<td>217</td>
</tr>
<tr>
<td>9.6 Logical-Physical Relational Terminology</td>
<td>218</td>
</tr>
<tr>
<td>9.7 Table showing the three anomalies</td>
<td>221</td>
</tr>
<tr>
<td>9.8 Emails Repeating Group</td>
<td>223</td>
</tr>
<tr>
<td>9.9 Multi-Valued Attribute in Email column</td>
<td>223</td>
</tr>
<tr>
<td>9.10 New Email-Student Table</td>
<td>223</td>
</tr>
<tr>
<td>9.11 2NF and 3NF Issues</td>
<td>224</td>
</tr>
<tr>
<td>10.1 Show Preferences</td>
<td>233</td>
</tr>
<tr>
<td>10.2 Decision tree dataset for the family movie night</td>
<td>235</td>
</tr>
</tbody>
</table>
## Algorithms

<table>
<thead>
<tr>
<th>Section</th>
<th>Algorithm Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>Algorithm for doing dishes</td>
<td>14</td>
</tr>
<tr>
<td>1.1</td>
<td>Algorithm for calculating average temperature</td>
<td>35</td>
</tr>
<tr>
<td>1.2</td>
<td>Algorithm for calculating average temperature using Procedures</td>
<td>36</td>
</tr>
<tr>
<td>2.1</td>
<td>Algorithm for adding $29 + 14 = 43$ using Roman numerals</td>
<td>62</td>
</tr>
<tr>
<td>5.1</td>
<td>Algorithm for printing an ASCII art picture of a backslash</td>
<td>132</td>
</tr>
<tr>
<td>6.1</td>
<td>Algorithm for computing the Chinese zodiac animal for a particular year</td>
<td>143</td>
</tr>
<tr>
<td>7.1</td>
<td>Algorithm for making coffee</td>
<td>148</td>
</tr>
<tr>
<td>7.2</td>
<td>Algorithm for the morning coffee routine using Functions</td>
<td>150</td>
</tr>
<tr>
<td>7.3</td>
<td>Algorithm for computing distance between two Cartesian points</td>
<td>165</td>
</tr>
</tbody>
</table>
Code Listings

0.1  Canonical “Hello World!” program in Java ......................................................... 15
0.2  Canonical “Hello World!” program in Python ..................................................... 16
3.1  “Hello World!” program for the Python shell ...................................................... 99
3.2  “Hello World!” program for the Python Text Editor .......................................... 100
3.3  First attempt at computing the average temperature .......................................... 109
4.1  Reserved words or keywords in Python ............................................................. 112
4.2  Operators and Operands in Python ..................................................................... 116
4.3  Compute area of circle for crayon of given length ................................................ 121
5.1  Final algorithm for normal students’ approach to summing numbers ...................... 130
5.2  Print an ASCII art picture of a backslash made out of asterisks ............................. 133
6.1  Compute Chinese zodiac sign for a given year .................................................... 143
7.1  Compute Cartesian Distance ................................................................................ 165
8.1  “Hello World!” program in Java for NetBeans .................................................... 171
8.2  Java application to create a BankAccount for Joe and make some deposits .......... 181
9.1  Example of SQL statements .............................................................................. 219
9.2  Example of Embedded SQL in Java .................................................................. 221
10.1 Decision Tree Classifier for the movie night dataset .......................................... 237
10.2 k-Means clustering on the Iris Dataset .............................................................. 245
10.3 Gaussian Naive Bayes classifier on the Iris Dataset ............................................ 268
0 On the Road to Computation ............ 3
0.1 What Is Knowledge?
0.2 Declarative vs Imperative Knowledge
0.3 The Key to Science
0.4 Computer Science: The Study of Computation
0.5 A Review of Functions
0.6 Computable Functions
0.7 Talking in Tongues: Programming Languages

1 Computational Thinking and Information Theory ..................... 21
1.1 What Is Thinking?
1.2 Deductive vs Inductive Thinking
1.3 Thinking About Probabilities
1.4 Logical Thinking
1.5 Computational Thinking and Computational Solutions
1.6 Two Fundamental Models of Programming
1.7 Pseudocode
1.8 Functional and Imperative Models of Computation
1.9 Information Theory
1.10 Shannon’s Information Entropy

2 Computational Problem Solving ...... 59
2.1 What Is a Model?
2.2 Data Representations
2.3 Number Representations
2.4 Digital Representations
2.5 Boolean Algebra
2.6 What Is Information Processing?
2.7 What Is Computer Information Systems (CIS)?
2.8 Programming Languages
2.9 Computational Thinking Defined
2.10 Problem Space and System State
2.11 Computational Thinking in Action
0. On the Road to Computation

In general we look for a new law by the following process. First we guess it. Then we compute the consequences of the guess to see what would be implied if this law that we guessed is right. Then we compare the result of the computation to nature, with experiment or experience, compare it directly with observation, to see if it works. If it disagrees with experiment it is wrong. In that simple statement is the key to science.

– Richard P. Feynman, The Character of Physical Law

0.1 What Is Knowledge?

If you’re like me, you’re usually rushing out the door in the morning. Wouldn’t it be nice to know if you need a jacket before you dash to your next class? You could check the weather channel but that doesn’t give you the exact temperature in your particular neck of the woods, which can often vary significantly from the city center. So maybe you recruit some of your friends and place some temperature sensors all around campus to get a more accurate gauge of the temperature. This isn’t a very realistic scenario but let’s play along for a bit.

0.1.1 Step 1: Gather the Facts

The sensors you distribute might record temperature, humidity, and their GPS location. Suppose your friends go around every hour and record these values for you. How could you use the mass of notes your friends pile up at your doorstep every day, as shown in Figure 0.1?

---

1With the implied caveat that the initial guess you make is somehow falsifiable, as defined by Karl Popper. You can see Feynman’s video of this quote at his Messenger Lectures on “The Character of Physical Law” at Cornell University in 1964 here: http://www.cornell.edu/video/richard-feynman-messenger-lecture-7-seeking-new-laws
As a first step, you might decide to organize all these disparate values in order to make sense of them. How should you go about organizing them? You could put them in a table or spreadsheet format. Let’s say you decide to organize the values for each sensor based on the time your friends collected them.

<table>
<thead>
<tr>
<th>Timestamp</th>
<th>Temperature</th>
<th>Humidity</th>
<th>GPS Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>4am</td>
<td>68°F</td>
<td>30%</td>
<td>Library</td>
</tr>
<tr>
<td>5am</td>
<td>71°F</td>
<td>31%</td>
<td>Library</td>
</tr>
<tr>
<td>6am</td>
<td>73°F</td>
<td>35%</td>
<td>Science Building</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Table 0.1: Organized Sensor Values

You would end up with something like the table shown in Table 0.1. After organizing these values, you can already notice trends in the Temperature and Humidity columns: both seem to be increasing as the day wears on.

Now we’re getting somewhere! Let’s pause and see what we’ve done so far. We started by recording some values, which are the quantitative observations of the physical world.\(^2\) Initially, we had a jumbled mess of values but, after organizing these values into a more appropriate structure, we end up with data that we can use to conduct some kind of analysis. We’ll define this idea of data more precisely later but, for now, let’s think about how we might use the data, which are the structured representations of the raw values.

### 0.1.2 Step 2: Contextualize the Data

We could, for instance, graph this data that we’ve collected in order to see if we can discover any trends over time in the dataset. If we plot the Temperature versus Time, we can clearly see a trend as shown in Figure 0.2.

We could then make a determination about whether it’ll be cold enough for a jacket or not: if the predicted temperature at 8am is below a certain value, say 70°F, you might conclude that it is cold. Conversely, if the temperature is above that certain threshold of 70°F, you would conclude that it is not cold and, in fact, it’s decidedly warm. This threshold is very subjective: 70°F might be hot for you but still considered freezing for your friend!

What have we accomplished so far? First and foremost, we have now succeeded in transforming

\(^2\)We’ll have more to say about a precise definition of the term value in Section 4.2.
the observed values into data by finding a suitable representation for those values (in this case, as a table) and then using that data with a rule (in this case, a threshold) to get some insight about the world you’re observing (in this case, that it’s hot if the temperature value is above the threshold at 8am).

Giving the data a context like this transforms it into usable information. We’ll once again hold off on quantifying this idea of information until later, when we meet Claude Shannon’s theory of information and uncertainty and Leo Szilard’s relating of information and energy. Nevertheless, we can continue on our adventure using just this intuitive concept of information as data that has been given a context of some sort to help reduce uncertainty and thus reveal meaning.

0.1.3 Step 3: Take Some Action!

The transformation of data into information makes it useful. You can now use that information to make actionable decisions. In this instance, we can use the trend in our graph from Figure 0.2 to predict that the temperature on campus would likely be above 70°F at 8am and conclude, using our threshold, that it’s hot outside and, based on this information, decide to not wear a jacket.

We have thus managed to transform our information into actionable knowledge, the final decision to skip the jacket. This is an example of rule-based knowledge that we have codified into action. We can represent this process, the transformation of data to information to knowledge, as shown in Figure 0.3.

Figure 0.2: Plot of Temperature vs. Time

Figure 0.3: Transformation of Data to Information to Knowledge
**Knowledge**

As structured or organized information that can lead to action in problem solving [1]–[3]. In the simplest terms, you can think of it as a collection of facts about some topic. This collection of information is often referred to as a *knowledge base* and is modeled or represented as a set of sentences.

The sentences in a knowledge base are usually expressed in a formal knowledge representation language in which each sentence represents some assertion about the world and the sentences can be used in logic-based reasoning systems to draw conclusions about the world. The logic-based systems can even incorporate fuzzy logic where the thresholds are not such stark dividers.

---

**Knowledge Generation Process**

Our simple rule-based example above demonstrates this process: we collected raw observations of the world via sensors. We then transformed those raw values into data by structuring them into a suitable representation (e.g., a table). Next, we gave that data some context by using a threshold to draw a conclusion and predict whether each temperature value was hot or cold. We organized these facts about the warmth and time of day into a graph and used that to gain actionable knowledge which we used to make a decision about whether or not to wear a jacket at that time.

---

**0.2 Declarative vs Imperative Knowledge**

The knowledge used in such knowledge-based systems is curated using *knowledge management* techniques that define two types of knowledge: explicit and tacit. Explicit knowledge, also called know-what knowledge, is formalized and codified, e.g., in books and tables. Tacit knowledge, also called know-how knowledge, is less formal and refers to experience-based knowledge that can often be intuitive and hard to define formally. We'll see alternative representations of these in Section 10.1.1, as well.

These two types of knowledge, know-what (explicit) and know-how (tacit), can more generally be classified as either declarative or procedural. **Declarative knowledge** involves knowing what is true and is composed of statements of fact. For example, stating that the average temperature on campus at 7am was 75°F is a declaration of a fact but does not give us any information about how to actually compute that average.

On the other hand, **imperative knowledge**, also called procedural knowledge, involves knowing how to do something and is normally referred to as how-to knowledge which usually involves methods or recipes for deducing information. In this case, procedural knowledge would be the specification of a method or procedure for computing the average temperature on campus; for example, one approach might be:

1. Count the number of sensors that reported a temperature value at 7am
2. Add up temperature values reported from each sensor at 7am
3. Divide the sum of temperatures by the number of sensors

---

Footnote:

3 The field of knowledge representation further formalizes and organizes the information just as a data representation formalizes and organizes observed values. These representations are used in knowledge engineering to build expert systems using problem-solving models that are built into modules called inference procedures or inference engines [4].
0.3 The Key to Science

On the road so far, we’ve taken our quantifiable observations, the raw facts, and represented them in some structure. These structured observations are our data, in our case the table of temperature values. Information can be thought of as data which has been further placed in a useful context so as to reveal its meaning\(^4\), in this case the determination of whether it was cold or hot. We used that conclusion to gain actionable knowledge and make a decision about whether or not we should take a jacket. That decision reflects our falsifiable prediction about the world; e.g., if we don’t take the jacket and the temperature suddenly drops precipitously, our decision, based on our prediction of the temperature, would be falsified.

A Quick Recap

So giving the raw facts, the observations of nature, some structure transforms values into data. Then, giving some context to that data, or reaching some conclusion with that data, gives us information. That information reflects our prediction about the world. We can organize that information into a knowledge base and use some logic-based reasoning to derive a decision about the future, the actionable knowledge, from that information which we can subsequently falsify.

At its core, all fundamental science is about making predictions in the form of experiments: precise, quantifiable, falsifiable predictions. As Richard P. Feynman put it,

“The fundamental principle of science, the definition almost, is this: the sole test of the validity of any idea is experiment.”

So if science is about making predictions, how is it different from the predictions that astrologers make? The core distinction is in the kinds of predictions each makes. Most horoscopes, for example, will give you general predictions. These horoscopes will usually say things like, “you’ll have a great day today.” Scientific predictions, on the other hand, are precise, quantitative predictions; they don’t say you’ll have a nice day but instead they say you’ll step outside your door at precisely 3:07pm and get struck by a meteor!

Admittedly, most predictions are not quite so dire and are usually more prosaic in nature. But prediction itself, which we can also call experimentation, is at the heart of basic science. We can think of basic science as that which focuses on the most fundamental aspects of the universe, matter, and energy. The examination of basic science has revealed an intimate relationship between such fundamental science and computation. In fact, computer science can be thought of as the study of computation so the first step might be to define computation in order to explore the relationship between computation and fundamental science.

\(^4\) Raw facts or values based on observations of the manifestations of the universe are already a representation of sorts but here we have added a further level of abstraction in the structured representation of values as data. If we’re so inclined, we can think of the data as a meta-representation that can be used to construct or describe or reason about the “lower-level” representation of the raw observations of nature.
The transformation of data ⇒ information ⇒ knowledge is essential for making actionable decisions. But if we step back and examine these transformations, a natural question arises: what the heck are these transformations or processes or procedures that manipulate and munge our representations of the world? We can start to think of these transformations of our representations of the observable universe as computations.

In fact, we can think of a computation as some process that transforms one representation of the world, our input, to some other representation of the world, our output. For example, the input might be a list of temperatures at each of the sensors distributed around campus at 6am and the output would be the average of all the sensors’ temperature values. The process or method that transforms the input to the output would be the calculation of the average itself. We might thus say that the problem of computing the average of temperatures of all the sensors at 6am is a computable problem.

As it happens, an important component of fundamental science deals with those problems in the universe that are computable, a term we’ll be defining precisely in just a bit. As David Deutsch says [5], “the laws of physics refer only to computable functions.” This implies that, in a very real sense, all the laws of physics belong to this set of computable functions, even though computable functions themselves are only a small subset of all possible mathematical functions!

As a first step, let’s think about precisely what we mean by the phrase “computable function”. We can break that term up and take each in turn, starting by first defining a function.

### A Review of Functions

Let’s review exactly what the term function means. If you’re already on an intimate basis with functions, please feel free to skip this section entirely. If functions are odd, nebulous entities for you, follow along below!

A function can be thought of as a mapping or a correspondence from one set of values, often called the domain, to another set of values, called the range (selected from a possibly larger set of possible numbers called the codomain), as seen in Figure 0.4. A set, for our purposes, is just a group of objects that is represented as a single unit.⁶

---

⁶ In one sense, we could say computer science is about making predictions that are computable and tractable while physics is about making predictions that are also physically realizable. Information could then be thought of as the connection...
A Function is a Correspondence, or Mapping, between a Domain and a Range

Figure 0.4: A Function is a Correspondence, or Mapping, between a Domain and a Range

The key to this correspondence is that each member of the domain is matched with one, and only one, member of the range, as shown in Figure 0.4. This correspondence is also called, quite unimaginatively, a one-to-one mapping since each value in the domain is paired with a single unique value in the range, as seen in Figure 0.5a.

In our temperature example, for a specific sensor, you can think of the Time values (e.g., 4am, 5am, 6am, etc.) as the domain and the Temperature values (e.g., 68°F, 69°F, 70°F, etc.) as the range. So for each Time value in the domain (let’s say, 4am), you will have a specific value associated with it in the range (i.e., 68°F, as per Table 0.1). Since we won’t always be dealing with Temperatures and Times, I’m going to switch to a more abstract example with simpler numbers as seen in Figure 0.5 so we can continue to explore functions as a more general idea.

A function can also have a many-to-one mapping, where more than one value in the domain might be mapped to the same unique value in the range, as shown in Figure 0.5b, where both 3 and –3 in the domain are mapped to the same unique value, 9, in the range. Please note, even in this mapping, each value in the domain is still matched only with a single value in the range; e.g., –3 is mapped only to 9, and not to both 9 and 4. However, different values in the domain may be matched to the same value in the range; e.g., –3 and 3 in the domain both map to 9 in the range shown in Figure 0.5b.

We can represent relationships between the domain and the range in a few ways: diagrammatically (in a cartoon as seen above or in a graph, as we’ll see in the graphical representation below), as a relation (a set of ordered pairs), or in functional notation (using symbols).

0.5.1 Graphical Representation

A graphical representation of these two mappings is shown below in Figure 0.6, where we see the familiar Cartesian coordinate system with the abscissa labelled x (for the domain) and the ordinate labelled y (for the range). Information Theory is in fact central to physics, biology, etc. [6], [7]

In mathematics, a set can most generally be thought of as an object that contains other objects. We can also think of a set as a collection of unique elements. An ordered set gives the elements an ordering, or a mapping to natural numbers, usually cardinal numbers; this ordering can involve a relation between every pair of elements, a full ordering, or just some pairs of elements, partially ordered sets. A sequence is an ordered set that allows for repeating elements while a series is the summation of an infinite sequence of numbers. A vector can then be represented as either a finite ordered set of indices (e.g., (x₁, x₂, x₃, ...)) or a finite sequence of values (e.g., (3, 2, 3, ...)), which can also be called an n-tuple, with the strong disclaimer that a vector and n-tuple are not the same thing. As you can see, none of these statements are precise mathematical definitions and there are subtleties, some the size of buses, that should be taken into account for any formal consideration of these terms.
labelled $y$ (for the codomain).

Figure 0.6a shows the graph corresponding to the same one-to-one mapping from Figure 0.5a while Figure 0.6b shows the graph corresponding to the same many-to-one relationship in Figure 0.5b.

### 0.5.2 Relational Representation

A relation is a rule that associates each element of the domain with at least one element of the range. It can be represented, somewhat confusingly, as a set of ordered pairs as shown below for the first and
second relations:

\[ (-3, 3), (-2, 2), (-1, 1), (0, 0), (1, -1), (2, -2), (3, -3) \]  

\[ (-3, 9), (-2, 4), (-1, 1), (0, 0), (1, 1), (2, 4), (3, 9) \]  

Once again, Equation (1a) corresponds to the one-to-one mapping from Figure 0.5a while Equation (1b) shows the relation corresponding to the many-to-one relationship in Figure 0.5b. Each term in these two equations is an ordered pair, like \((-3, 9)\) in Equation (1b), where the first element, \(-3\), is from the domain in Figure 0.5b and the second element, 9, is the corresponding value from the range, also in Figure 0.5b.

Not all mathematical constructs are functions; in fact, you can have relations other than one-to-one or many-to-one! For example, you might have a relation in which one element of the domain maps to more than one element of the range in an equally unimaginatively named one-to-many relationship. Any relationship that is not one-to-one, though, is not a function. In fact, the badly named constructs called multi-valued functions, which are many-to-many relations, are actually reverse/inverse mappings and not functions.

### 0.5.3 Functional Representation

In addition to the graphical and relational representations, we can express these relationships in **functional notation**, in which we think of the function, or relation, as an opaque black-box, wherein you can’t look inside the box to see the inner workings of how it’s doing its processing, as shown in Figure 0.7.

![Figure 0.7: A function \( f \) can be represented as a black box with input \( x \) and output \( y \)](image_url)

In this approach, we think of the **input**, represented by the letter \( x \), as a value from the domain and the result or **output**, represented by the letter \( y \), as the corresponding number from the range. The function, or black box, is given a symbolic name; in this case, we called it \( f \) but we could just as well have called it \( MyFunction \) or \( \phi \).

In the same way, we gave a symbolic name to the input, \( x \), and the output, \( y \). These symbols, \( x \) and \( y \), will be used by the function, \( f \). Since their values can vary (i.e., we can provide anything as input and

---

\(^{7}\) Often, you might hear the term **predicate**, as well. A predicate is a relation or function that returns a **true** or **false** value, also called a Boolean value, as opposed to a generic function which can return any kind of value. We’ll meet Boolean values in a later chapter and explore this distinction in more detail then but the term predicate can be somewhat confusing as it has other meanings in math and computer science, as well.
the output might change depending upon that specific input), these placeholders are called variables. Since the variable we decided to call \( x \) represents the input, it is often called the independent variable as the input isn’t controlled by the function. The input is, however, changed or transformed by the function, \( f \), and the symbolic name for the output, the variable we decided to call \( y \), is referred to as our dependent variable since its value depends on the input. We’ll have quite a bit to say about variables later on.

Since the function, \( f \), is the black box that transforms the input, \( x \), we can also represent this process as \( f(x) \), which essentially means “function of \( x \).” The parentheses, “(“ and “)”, are used to demarcate the name of the function, \( f \), from its input, \( x \). So the entire process consists of some input plus some transformation, which, in anticipation of the next chapter, we could write as:

\[
\text{Process} = \text{Transformation of the } + \text{Input} \tag{2}
\]

Using the above symbolic representations, we can write our two relations in functional notation as:

\[
y_1 = f_1(x) \tag{3a}
\]

\[
y_2 = f_2(x) \tag{3b}
\]

or, more familiarly, as:

\[
y_1 = f_1(x) = -x \tag{4a}
\]

\[
y_2 = f_2(x) = x^2 \tag{4b}
\]

Here, \( f_1(x) \) represents the mapping in Figure 0.5a and \( f_2(x) \) represents the relationship in Figure 0.5b. But we cheated a bit and actually peeked inside the black box to also define the two functions (Equations (3a) and (3b)) as \(-x\) and \(x^2\), respectively, as shown in Equations (4). Please note that we decided to add a subscript to both the functions (the \( f \)’s) and the outputs (the \( y \)’s) as we’ve been dealing with two different relations. However, we decided to not add a subscript to the input as we’ll later use this device to distinguish between the terms parameter and argument as used specifically in computer science (as opposed to mathematics, in general).

0.6 Computable Functions

Now that we have a handle on functions, we can start to think about computable functions and what the term computable, or computation, might mean. Computation can be thought of as a method or procedure that transforms some input information into some output information. In the classical theory

---

8More precisely, we could say a variable is any symbol used to represent an unspecified element chosen from some set of elements.
of computation, we generally deal with processing some input string of generic symbols into some output string of generic symbols (these strings are technically a finite ordered sequence of symbols from a finite set of such symbols). Some of these transformations can be accomplished by using a function because that’s what functions do: they transform some input into some output as we’ve seen above!\(^9\)

Putting these two terms together, a function is said to be computable if, for some given input, the corresponding output can be calculated by using a finite mechanical procedure.\(^10\) What is this finite, mechanical procedure? It’s any effective procedure, defined as some method or approach that has a finite number of unambiguous, finite instructions; in addition, it must always halt after a finite number of steps and, when applied to a certain class of problems, produce the correct answer.

0.6.1 Computation and Algorithms

These effective procedures can be carried out by any agent, often called a computing agent; sometimes the computing agent is a human, sometimes a digital machine, and sometimes the universe itself.\(^11\) When an effective procedure, also called an effective method or effectively computable operation, can calculate all the values of a function, it is called an algorithm for that function.

For example, we could have an algorithm for computing the average of any set of numbers. We can express this algorithm as simple instructions: add all the numbers and divide by the total number of numbers. We could also express this mathematically as \(\sum_i \frac{T_i}{N}\). We applied this algorithm for solving the general problem of calculating averages to the specific case of calculating the average temperature of our temperature sensors in Section 0.1.3.

At a simpler level, you can think of an algorithm as a recipe or a sequence of step-by-step instructions. As can be seen above in the definition of an effective procedure:
1. each of the algorithm’s steps should be finite and exact
2. the number of instructions should be finite, and
3. when the algorithm finishes, it should solve the problem

The word algorithm is a corruption of the last name of Abu Ja’far Muhammad ibn-Musa Al-Khwarizmi, a 9th century Persian mathematician. He wrote a book called Kitab al jabr [8], whose name was Latinized to Algebra since al jabr means a reduction of some sort. Al-Khwarizmi wrote another book in which he described a formal, step-by-step procedure for doing arithmetic operations, like addition and multiplication, on numbers that were written in the new base-10 positional number

\(^9\)Computable functions are a subset of all the possible functions in mathematics. However, all laws of physics correspond only to computable functions, except for possible hypercomputational systems which posit there are some natural phenomena that cannot be simulated computationally. We will also look at this some more when we discuss the difference between Polynomial (P) and Non-Deterministic Polynomial (NP) problems.

\(^10\)Computer science deals with mathematically computable functions but the term function is used slightly differently in programming, as we’ll see in later chapters.

\(^11\)An idealized computing agent should be able to accept an instruction, store and retrieve information from some memory, carry out the actions required by the instructions, and produce an output.
system that had just been developed in India. Over time, references to his work started to be called by his name which, in Latin characters, was written as Algoritmi. Eventually, his name was corrupted to Algorismi, and then to algorism, and then finally to algorithm, which is the word now used for those formalized, step-by-step procedures.\footnote{In fact, even today, Al-Khwarizmi’s name is spelled with many variations, including Al-Khowarizmi or Al-Khuwarizmi.}

\textbf{Example 0.1} One example of a simple algorithm might be doing dishes. If you were to describe the process of cleaning a pile of dirty dishes to a 5-year-old, you might describe the step-by-step process, or algorithm, as shown in Algorithm 0.1.

\begin{algorithm}
\caption{Algorithm for doing dishes}
\begin{algorithmic}[1]
\State Grab a wash cloth
\State Add detergent to the washcloth
\State Turn on water
\While{there are still dirty dishes left on the pile of dirty dishes}
\State Grab a dish
\State Clean it with your wash cloth
\State Rinse it off
\State Dry the dish
\State Put it on the drying rack
\EndWhile
\State Turn off water
\end{algorithmic}
\end{algorithm}

Most algorithms are more complex, and precise, than this simplistic approach to cleaning dishes. Broadly speaking, an \textit{algorithm} can be said to be a general method consisting of a finite sequence of steps that are performed in order to reach a desired result. In fact, an algorithm can be thought of as a general method to solve a whole family, or \textit{class}, of related questions, as opposed to only one specific problem; e.g., we can have an algorithm for computing averages (of \textit{any} set of numbers) and use that algorithm to compute the average in the specific case of the Temperature values gathered by our sensors. An \textit{algorithm}, in this sense, \textit{defines} what might be considered to be a \textit{computation}.

\textbf{Problem 0.1} Can you come up with an algorithm for a hungry friend instructing them exactly how to make a peanut butter and jelly sandwich? How would your algorithm be different if you were to draft it for your hungry 5-year-old nephew instead?

\section*{0.7 Talking in Tongues: Programming Languages}

So far, we’ve figured out that \textit{computation} deals with the \textit{transformation} of input information into output information by an \textit{effective procedure}. A \textit{computable function}, we found, uses an \textit{effective procedure} to calculate the output value. This effective procedure, called an \textit{algorithm}, is expressed as a sequence of finite instructions that are carried out by some \textit{computing agent}.

If the task at hand is physical, as in the cleaning dishes example, the computing agent that carries out the process can be a human. But if we were going to use a digital machine, like a modern personal computer, as the computing agent to help solve some of the more abstract problems, we’d need a way to express the algorithm, or \textit{sequence of precise instructions}, in the language of the computer.
Some formal definitions

1. **Effective Procedure**: a procedure is said to be effectively computable if it can be carried out by a *computing agent* and consists of a finite number of exact, finite instructions or steps which always finish after a finite number of steps and produce the correct answer
   - An effective procedure solves some category of problems and can consist of multiple steps or just a single step
2. **Computable Function**: a function whose *output* can be calculated using an effective procedure on some given *input*
3. **Algorithm**: A well-ordered sequence of effectively computable, clearly defined steps that can be understood and carried out by a computing agent which can complete the operations successfully in a finite amount of time
   - Also known as a *computation*: an effective procedure that calculates the values of a function is called an *algorithm*

An effective procedure can be expressed in different forms, or representations. Each expression of the effective procedure is an algorithm if it calculates the value of some function. Some algorithmic expressions, or representations, are intended for for human computing agents while other representations are intended for digital computing agents.

Just as a function can be expressed as either a graph or a relation, different algorithms can represent the same effective procedure for different computing agents. In one sense, you can think of an algorithm as a way to *translate* the effective procedure for a particular computing agent.

Most modern-day computers express instructions for their central processing unit (CPU) in a particular *programming language*. These programming languages tend to be much more specific, precise, and limited in scope when compared to natural languages like English. Like natural languages, they have a specific syntax and structure which must be followed precisely in order to direct the computer to carry out, or *execute*, the instructions in the algorithm. An *ordered set of instructions* in a particular programming language is called a *program* or *source code* or simply *code*.

We’ll be talking more about the essential aspects of these programming languages later as we explore Java and Python, two popular programming languages. By way of preview, let’s write a simple program in the Java programming language that just prints the phrase, “Hello World!” This Java program is shown in Listing 0.1.13

```java
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello World!");
    }
}
```

Listing 0.1: Canonical “Hello World!” program in Java

---

13This is an example of the so-called canonical “Hello World!” program which is the traditional first program you might write when you first learn a new programming language. If your program successfully prints out this phrase, that means you’ve mastered the mechanics of writing and executing, or running, a program in that language. And, if you’re trying to impress your parents, you can now pad your résumé by listing this as a language you’re “familiar with,” although it would probably be more accurate to still say, “aware of”.
Chapter 0. On the Road to Computation

This same “Hello World!” program can be written in Python, as well, as shown in Listing 0.2. As you can see, these different programming languages take different approaches to specifying a particular computational solution. Thus, one language might be more appropriate for a specific problem or perhaps a specific programmer’s style. In the case of printing out the string, “Hello World!”, the Python approach is much simpler and I’d likely pick that. But if I had to write an Android app, I’d probably pick Java.

```python
print("Hello World!")
```

Listing 0.2: Canonical “Hello World!” program in Python

0.7.1 Going to Church-Turing

In the early 20th century, a lot of great minds were struggling with defining the notion of computation in a mathematical sense. People like Alonzo Church, Kurt Gödel, Emil Post, Alan Turing, and Haskell Curry all came up with independent derivations or approaches for this concept.

One of the fathers of computer science, Alan Turing, described a hypothetical computing agent in 1936 that later came to be known as the Universal Turing Machine (Turing himself much more modestly called it a Logical Computing Machine or Universal Machine). This imaginary machine followed the model of a ticker tape; a ticker tape was a machine that would print stock market information on paper tape. These ticker tape machines would generate reams and reams of paper tape; so much so, in fact, that people would toss the shredded paper tape out their windows during parades, thus giving us the ticker tape parade.

The theoretical Universal Turing Machine consisted of a simple tape head that could move over an infinite length of such a tape; the head could only carry out simple, primitive instructions like moving along, reading symbols from, and writing symbols to the tape. This straightforward machine was an idealized computing agent and model for mathematical calculations which formalized the notion of

---

14 Curry credited and built upon work by Moses Schönfinkel and, implicitly, Friedrich Frege

15 A Universal Turing Machine is simply a Turing Machine which can simulate another Turing Machine. Turing had originally used this formal approach to reformulate Gödel’s results on the limits of proof and mathematical computation. The simpler Universal Turing Machine approach was able to show the same results as the universal, arithmetic based formal language used by Gödel.
algorithmic computation, systematic procedures that produce a solution to a problem in a finite number of steps.\textsuperscript{16}

Up to this point, we had only met computation in an informal sense as a sequence of steps or process for manipulating data according to simple, finite rules. Turing formalized this notion of computation by providing a mathematical model for it: the Universal Turing Machine. Alonzo Church took a somewhat different approach to get at the idea of computation.

Instead of looking at “functions as graphs” (the way we did earlier when we thought of a function as a way to describe a relationship that maps a value from the domain to the co-domain), he looked at functions in the older sense of “functions as rules”.\textsuperscript{17} Defining a function in this paradigm meant specifying the rules for how that function should be calculated. Church used a version of this in which he expressed “functions as formulae,” a system for manipulating functions as expressions. When preparing the manuscript, his typesetter accidentally used the Greek symbol \( \lambda \) in his equations and so this approach came to be called Lambda calculus, which is also the basis for functional programming.

People soon realized that all these different approaches to computation were computationally equivalent in that each approach could encode and simulate the other approaches and produce the same results. Church thought all of these equivalent formalisms embodied the same general notion of computability and his thesis was generalized to the Church-Turing thesis: if there exists an effective procedure or method for obtaining the value of a mathematical function, that function can be computed by a Universal Turing Machine.

Let’s just pause for a bit and consider what this might be saying because, upon reflection, it might sound a little nutty: the Church-Turing Thesis says that every function that is computable, i.e., every computation and every single thing in the universe that is computable, can be described by a set of operations (an algorithm) expressed in some language for an appropriate computing agent.

The universality of this idea simply cannot be emphasized strongly enough: it essentially says that everything in the universe that is possible to compute can be computed by a Universal Turing Machine (or any of the equivalent formalisms for the idea of computation and computability).

In large measure, the entire theory of computation and, by extension, computer science, deals with the Church-Turing Thesis and effectively computable functions and finding the limits of which functions are computable. There have been several offshoots of this theory; one such variation, the Physical Church-Turing Thesis, addresses what can physically be realized by a computer in our universe; another,

\textsuperscript{16}Turing had posited that if an effective procedure existed for obtaining the values of a mathematical function, that function could be computed by a Universal Turing Machine.

\textsuperscript{17}We will revisit these ideas later when we discuss the difference between the functional and the imperative approach.
the Complexity-Theoretic Church-Turing Thesis, considers what can be efficiently computed. Indeed, the application of the Church-Turing Thesis has wide-ranging implications for the universe itself and implies that the universe might be equivalent to a Universal Turing Machine, as examined in digital physics, or perhaps a hypercomputer!

Transforming Information via Computation

Fundamental science is about making falsifiable predictions that are verified by experiment. We start with observations of nature; giving these raw facts some structure transforms values into data. Giving some context to that data, or reaching some conclusion with it, transforms data into information. That information reflects our prediction about the world. We can organize information into a knowledge base and use logic-based reasoning to derive a decision about the future, which is our actionable knowledge that we can subsequently falsify. Computation is the process by which we transform data into information and then into knowledge.

Uncomputable... that's what you are...

There are, of course, other possibilities, as well, and the bad news is that not all problems have computational solutions. The most famous of these is the halting problem designed by Turing. In this scenario, Turing considered a program that would print true if, and only if, it runs forever and showed that it is impossible to write such a program.

Computation, or computability, as we’ve built up so far, means the process of solving any problem that can be solved by some algorithm which manipulates symbols in the same way as a Universal Turing Machine. Our definition of an algorithm, however, still isn’t completely formalized as we describe precisely, but not axiomatically, what we mean by an algorithm; even the term effectively computable depends upon the capabilities of a computing agent so if a computing agent is able to understand and execute an algorithm in a finite amount of time, then that algorithm is effectively computable for that computing agent.18

This means the Church-Turing Thesis can never be proved mathematically! Because it has not been formally proved, it is referred to as a thesis and not a theorem. The good news is that it has been extensively validated as there is an equivalent Universal Turing Machine for every algorithm that exists for all tasks that can be solved algorithmically. In addition, all other models for computing agents and algorithms have been proved to be equivalent to a Universal Turing Machine; i.e., the Universal

---

18 This hearkens back to some of the attempts to formalize the notion of functions and computability in the 20th century, a la Kurt Gödel’s formal definition of a class of general recursive functions, Alonzo Church’s definition of functions on natural numbers in terms of the λ-calculus, and Alan Turing’s definition of functions on natural numbers in terms of his abstract model of a simple machine. The Church-Turing Thesis states that these three formally defined classes of computable functions are equivalent and assumes these models are also equivalent to the informal notion of effectively calculable, a term used to describe functions that are computable by mechanical means. As a side note, the idea of relaxing the constraint of discreteness in these approaches is also tied to the idea of hypercomputation, as discussed later.
Turing Machine can do everything done by these other computational models upon processing their algorithms.

Conversely, this means that if we can find a problem that can be expressed as a symbol manipulation task which we can prove cannot be solved by a Universal Turing Machine, then the Church-Turing Thesis necessarily implies that the problem is uncomputable or unsolvable. To prove the problem cannot be solved by a Universal Turing Machine means that such a machine cannot exist and that no one will ever find one as nature itself prohibits its existence.

This might seem like a hyperbolic claim but, as Kurt Gödel showed in his incompleteness theorems, there are true statements about a formal system that cannot be proved to be true using that system. Although Gödel did this in the context of a formal system for ordinary arithmetic of natural numbers, his theorem motivated mathematicians to find some method for showing which statements are unprovable in a generalized formal system. These methods would require some computational procedure to process and recognize such statements.

So when mathematicians like Church, Post, Turing, etc., started to propose many different models for these computational procedures, along with an accompanying computing agent to carry out these procedures, their goal was to help provide a formal basis for mathematical proofs as that might help guarantee the correctness of a proof without relying upon ambiguous or unproven or subjective statements. The main advantage of formalizing this process of generating mathematical proofs is that it would allow for automating theorem-proving by providing a simple set of rules for some computing agent to follow.

In fact, complexity theory analyzes the classes of problems that are computable and the ones that are un-computable. We could, in a way, think of physics as dealing with computable functions and think of mathematics as dealing with all functions, computable or otherwise. Computer science can then be thought to be the discipline that’s concerned with determining which functions are computable or not and sits at the intersection of the other two disciplines. We’ll pick this topic up again when we deal with the P vs NP problem later on. For now, though, let’s continue our adventure by exploring the idea of computational thinking in detail in the next chapter!
A computation is an effective procedure that transforms input information into output information. An effective procedure is a finite set of instructions for carrying out some unambiguous symbolic manipulation or transformation. If an effective procedure can calculate all the values of a function, it is called an algorithm for that function and that function is then considered a computable function. Algorithms, ordered sets of instructions to solve some class of problems, and computable functions are the basic objects in the theory of computation. A Turing Machine is a mathematical description of any algorithm in terms of a hypothetical machine with a simple read-write head and a tape of paper; a Turing Machine is one formalization of the general idea of an algorithm. In fact, computation itself can then be thought of as the study of algorithms or such formal models of algorithms.
1. Computational Thinking and Information Theory

We need to do away with the myth that computer science is about computers. Computer science is no more about computers than astronomy is about telescopes.

– Michael R. Fellows

1.1 What Is Thinking?

The history of formal approaches to thinking in the Western world is usually traced back to the Hellenic culture. One of the most influential thinkers in the ancient Greek world was Socrates, the first moral philosopher. Socrates favoured using critical thinking to reason about questions using a dialectic method of inquiry in conversations.

So when you asked Socrates a question, rather than directly answering you, he’d ask you questions to lead you to the answers; often, he’d ask questions that led you to ask the question he was interested in discussing, as well. This approach, called the Socratic Method, was so irritating to some that it led the Athenians to sentence Socrates to death.

Socrates was interested in finding out the true nature of moral and ethical ideas by looking at many examples of that idea and then generalizing from those examples. For example, if he was interested in figuring out the essential nature of a chair (he really wasn’t), he might use this type of inductive reasoning to look at many examples of chairs and find those characteristics that are the same across all chairs.

1 This quote is sometimes also attributed to Edsger Dijkstra and Ian Parberry; please see https://en.wikiquote.org/wiki/Computer_science for the juicy details.

2 As you might suspect, this wasn’t the real reason; Socrates was officially tried for “refusing to recognize the gods recognized by the state” and for “corrupting the youth” but I’m convinced, based on how students respond to my own implementation, it was due to the Socratic method.
Once you start to think about all the various things we might call chairs (stools? tripods? beanbags?), you might end up not coming to any real conclusion about the essence, or essential Idea, that was common to all chairs. Socrates’ reasoning was similarly inconclusive but it did inspire his pupil, Plato, to start thinking more deeply about such Ideas or idealized Forms.

Plato decided that all these examples of chairs were just pale imitations of some idealized chair that existed on another plane of existence. In fact, he posited that there was an eternal world of Ideas and Forms that exists separate from our material world and every material object (like our ever-present chair), was just an imperfect shadow of some Ideal Chair that existed in that other plane of existence. That Ideal Chair is perfect and eternal and every chair in our physical plane of existence is a pale imitation of that Ideal Chair.

Plato used this type of deductive reasoning from these other-worldly Forms to gain insight about their reflections in our physical world. In his imagining, just as in Socrates’, we are all born with an innate knowledge of these Ideas or Forms and can use critical reasoning to discover their reflections or implications.

Plato’s student, Aristotle, wasn’t as interested in a metaphysical plane of existence with idealized Forms. He strongly favoured using experience and physical perception to reason about the world. He used both inductive reasoning to gain knowledge about universal characteristics of objects by examining many examples (like Socrates) and then used deductive reasoning to take these derived universal principles and examine particular objects of that kind (like Plato).

**Philosophy... it’s the talk on a serial port...**

Greek philosophy started around 610 BCE with Thales and Anaximander of Miletus and flourished through the Hellenistic period which ended around 30 BCE. It covered a wide range of human experience and thought but, arguably, culminated with three of the most influential ancient philosophers: a decorated soldier, a wealthy wrestler, and a dictator’s son-in-law. They’re usually better known as Socrates, Plato, and Aristotle, respectively.

Socrates was interested in logic and ethics and used the dialectic approach, in which two people with different opinions try to find the truth using argumentation. The Socratic Method in particular used questions to disprove the other person’s point in order to highlight what they didn’t know or what was not known. This approach involved using techniques like using questions to summarize what the other person was saying, asking for evidence, and challenging their assumptions or finding exceptions to their argument.

This kind of evidence-based reasoning not only informed Socrates’ pupils, including Plato and Aristotle, but laid the foundation for Western thought, in general. This tradition continues in the process of finding computational solutions to (computable) problems.

1.2 Deductive vs Inductive Thinking

Finding a computational solution requires you to think computationally. In order to get a clear idea of computational thinking, we’ll need to address both words that make up this term. We already know

---

3 Despite my attempt to use a consistent example, I should add the disclaimer that the ancient Greeks were not obsessed with seating implements.
Thinking. In the context of computational problem solving, can be considered to be the ability to rationally draw inferences from statements, observations, and data using deductive, inductive, or probabilistic analyses [9]. That’s quite a mouthful so let’s take it one at a time.

Deductive reasoning involves drawing conclusions which necessarily follow from some body of evidence. It often includes constructing an argument such that if the premises are true, then the conclusion must necessarily be true.4 We can also reason backward from this definition to infer, for example, that if the conclusion is false, then at least one of the premises must also be false.

Inductive reasoning is similar but an inductive argument goes beyond the available evidence to reach conclusions that are likely to be true but are not guaranteed to be true. Thus, in inductive inference, the conclusions are likely or plausible but (often) not certain to be true. This is because, in inductive arguments, the truth of the premises does not guarantee the truth of the conclusion. Often, inductive reasoning is used to build hypotheses which can then be checked via deductive reasoning.

Probabilistic inference, also incorporated in abductive reasoning, utilizes probabilities to assess the likelihood of the conclusion. It is based on syllogisms5, a form of deductive reasoning, but incorporates probabilities in an intermediate step, as well.6

1.3 Thinking About Probabilities

As we can see, probability has snuck into our journey and, it turns out, probability is a central aspect of almost all investigations of nature. Let’s talk a little bit about probability since it will be so germane to much of what we do later.

Probability can be thought of as a kind of uncertainty about or likelihood of some event occurring. Probability is one of the simplest mathematical theories and only has three basic rules known as Kolmogorov’s Axioms of Probability.7 In their delineation below, please feel free to ignore, for now, the mathematical formulas for probabilities, like $P(A)$, $P(A | B)$, etc., as we’ll have plenty of time to familiarize ourselves with that mathematical notation later:

1. The probability itself is simply a value between 0.0 and 1.0 (inclusive)
   - This means probability is just a real number like 0.1 or 0.35. It’s usually expressed as a percentage by multiplying this number by 100 so that 0.1 becomes 10% and 0.35 becomes 35% so you might say the probability of event A happening is 35%

   $$0 \leq P(A) \leq 1$$ and

2. The total probability is always 1.0 (or, if multiplied by 100, 100%)
   - This just means that if you’re looking at some event, either it will happen or it will not happen; one of those things is guaranteed to be true. Put another way, the probability of A

---

4For our purposes, we can think of an argument as having mathematically well-defined procedures for reaching conclusions with certainty or necessity, based on the evidence that is presented.

5The most famous syllogism is, “All men are mortal. Socrates is a man. Therefore, Socrates is mortal.” In probabilistic reasoning, you might change that to, “I believe all men are mortal. I think Socrates is a man. Therefore, in all likelihood, Socrates is probably mortal.” Abductive reasoning is usually less formal and, given some observations, is called, “inference to the best explanation.”

6Inductive reasoning also incorporates some aspect of probability as the truth of the premises does not guarantee the truth of the conclusion. This is why we can say the conclusion of a deductive argument is true if the premises are true but, for an inductive argument, we say it’s strong (weak) if the probability of its conclusion given its premises is high (low).

7We could probably add in Countable Additivity or replace Additivity with it if the events are mutually exclusive, I suppose.
Chapter 1. Computational Thinking and Information Theory

happening OR of A not happening is 1
\[ P(A \cup \neg A) = 1 \]

3. The probability of an event A OR event B happening is (the probability of A + the probability of B - the probability of A AND B happening together)
\[ \Rightarrow P(A \cup B) = P(A) + P(B) - P(A \cap B) \]

Kolmogorov’s Axioms of Probability

1. The probability is a value between 0.0 and 1.0
2. The total probability is always 1.0
3. (The probability of event A OR event B happening) = (the probability of A) + (the probability of B) - (the probability of A AND B happening together)

1.3.1 Calculating the Probabilities

Probabilities are sometimes calculated by looking at frequencies. Suppose I’m interested in figuring out the probability of getting snake-eyes (two 1’s) at the Craps table in Las Vegas. One way to do it would be to roll the two dice a whole bunch of times.

As you roll more and more dice, the frequency of snake-eyes in those rolls (i.e., how many times you roll a 2 compared to how many times you roll a 3, 4, 5, 6, 7, 8, 9, 10, 11, or 12) will start to come close to the probability of getting a 2 if you roll any two random, unbiased dice thanks to the “Law of Large Numbers”, which says that if you make a large number of observations, the results should be close to the expected value.

But there’s a caveat: the frequency over a small number of rolls of the dice may be very different from the actual probability; the law of large numbers, unsurprisingly, doesn’t apply when the numbers are small. Also, some events only happen once, like the 2016 election in the USA.

1.3.2 Samples and Populations in Statistics

These rolls of dice that we collected in estimating the probabilities can be considered to be a sample. In general, when we’re reasoning or thinking computationally about a problem, we’ll usually be examining a statistical population, which consists of all possible measurements or outcomes that are of interest in the problem context. A random sample is a portion of the population that is unbiased and representative of that population. A random sample is one in which every member of the population has an equal chance of being selected. When picking members of the population to include in the sample, we can use many sampling methods like Random Sampling, Stratified Sampling, Sampling With Replacement and Without Replacement, etc.

One of the goals of computational problem solving is to infer statistical properties of the population from the statistical properties of the sample. For example, we might want to determine the number of Independents in the population of all eligible voters in the USA. We might select a random sample of 1,000 eligible voters and find that 300 of them were Independents. From the sample, we might then infer that 30% of the entire electorate are also Independents.

\[ ^8 \text{They can also be calculating using a Bayesian approach which we’ll discuss later.} \]
\[ ^9 \text{If you’re unfamiliar with Craps, don’t worry; it’s just a game played with dice and is one of the easiest ways to make and lose money in Vegas. For our purposes, the most important part is that it’s played with two die, each of which can have a value between 1 and 6.} \]
We often make claims about the population by providing a confidence interval, which is the range of values that contain the true value with a certain probability, called the confidence level. This confidence level is usually 95% or 99%. Confidence intervals are usually expressed along with a margin of error, which expresses the uncertainty in the measurement and is usually half the length of the confidence interval. In most surveys, the margin of error is expressed for the confidence interval at a 95% confidence level. So, for the above example, if we say, “30% of the US population are Independents with a 3% margin of error,” we’re really saying, “We are 95% confident that between 27% and 33% of the US population are Independents,” where the confidence interval is between 27% and 33%, the confidence level is 95%, and the margin of error is 3%.

1.3.3 Conditional Probabilities and Sample Spaces
In probability, though, the similar sounding term, sample space, has a different meaning: a sample space is simply the set of all possible outcomes of an experiment. For our dice example, the Sample Space = \{2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12\}. We can calculate the probability of each outcome in the sample space, just as we did before for snake-eyes.

Suppose we roll our two dice and see that the first one is showing a 1 but can’t quite make out the value on the second die. A question that might immediately come to mind could be, “I wonder what the likelihood is of getting a second 1 if I’ve already rolled a 1?” This is an example of conditional probability.

A conditional probability is the probability of an event \(A\) occurring (rolling snake-eyes) if another event \(B\) has already occurred (already rolled a 1 on the first die). This kind of conditional probability can be expressed as

\[ P(A|B) = \frac{P(A \cap B)}{P(B)} . \]

You can, once again, ignore the funky notation for now if it’s unfamiliar to you. We’ll be using this equation later to derive Bayes’ Rule; this will lead us to discover Bayesian Analysis which is the underpinning of so many analyses, especially in statistics and machine learning. Being able to reason and think probabilistically, in general, is increasingly becoming an essential aspect of conducting any kind of systematic, scientific investigation.

1.4 Logical Thinking
We can even think about “reasoning with evidence” in this manner to be one way to characterize what we do in physics and computer science. Our present concern, though, is with reasoning or thinking about computational problems. Specifically, computational thinking requires thinking about a process for solving some problem that is computable. As we’ve seen in Section 0.5, a function is a process that takes some input and converts it to some output. The input can be considered to be the data, or evidence, and the output can be thought of as our answer, or prediction.

More broadly, computational thinking normally involves forming a model of some sort and reasoning with it in some way. So it’s no wonder, then, that the model is usually mathematical and the reasoning is some kind of formal mathematical reasoning, like formal logic. But the model could also be graphical or linguistic or logical, depending on the kind of problem we’re tackling and the resources we have available. Thus, the task of solving a problem is often reduced to finding a suitable representation of the model and a logical process to use that model in order to solve the problem.

1.4.1 Origins of Formal Logic
In general, logic is a way to reason about the world and discover answers. Its origins can be traced to the ancient Greeks, as we saw in Section 1.1. The Greek approach to logic helped establish guiding
principles for assessing the validity of arguments or statements.

In Section 1.1, we learned about Socrates and his approach of asking questions as a way to discovery. If two people had different opinions about a certain topic, Socrates would engage them in a conversation where he’d ask questions that forced them to think critically about that topic; this approach is often called the dialectic method of inquiry. So if you asked Socrates a question, he’d more likely than not respond by asking you more questions so you could discover the truth for yourself.

His pupil, Plato, also thought people could discover truths about the world and ideas by self-examination and inquiry; in fact, they both thought that universal knowledge was already inherently within people. But Plato thought our senses might deceive us sometimes and so the universal characteristics about things must exist separately from the thing itself, in the idealized world of Forms which can only be known through the mind. In fact, Plato was intensely concerned with inquiring into the nature of reality whereas Socrates was mainly interested in logic and ethics and not so much about the fundamental nature of the physical universe or metaphysics in that sense. As such, Plato sought to answer questions about this abstract nature of reality and its relation to the physical world we experience through our senses.

Aristotle, on the other hand, deviated from his teacher in this regard. Like Plato, he was also interested in the nature of reality but, like Democritus before him, Aristotle was a firm believer in the evidence he gained from his senses. He didn’t care too much for alternate planes of existence and whatnot; instead, he argued that the Universal ideas both Plato and Socrates pursued were derived from experience and were already present in the material objects we examine. In this regard, Aristotle was interested in finding what was true and valid about the physical world we experience via our senses.

### 1.4.2 Propositional Logic and Argumentation

Logic, in general, deals with assessing the validity of assertions. In order to determine the validity, or truth, of a particular assessment, you have to establish the idea that an assertion can be evaluated as true or false. Building upon the work of Plato and Socrates, Aristotle observed how reason was used in conversations and was able to codify an approach for reasoning about the validity of statements.\(^{11}\)

He helped establish a formal system for answering questions and laid the foundation for deductive logic based on True/False statements. Statements that are either True or False are called propositional statements and using logic to determine the validity of such statements is called propositional logic.\(^{12}\)

But before you can reason about the things in the world, Aristotle thought that you have to be able to define what those things are; i.e., what kind of a thing it is or what is its essence or Substance.\(^{13}\) Aristotle was thus specifically interested in propositional statements that would allow us to find the

---

10. Logic itself has a rich history and was simultaneously developed and discovered in ancient cultures such as in India with Medhatithi Gautama, China with Mozi, Egyptian Geometry, Babylonian Astronomy, etc.

11. The word logic is, in fact, derived from logos which Aristotle took to mean “reasoned discourse.”

12. Another way to think about it is that a statement that asserts a truth about the world is making a proposition about it which can be verified. There are, however, many kinds of logics, including syllogistic, predicate, modal, temporal, mathematical, probabilistic, intuitionistic, fuzzy, etc. Some of these are further related, like probabilistic and fuzzy logic, for example, as most statements are rarely true or false only but instead allow for multiple degrees of certainty. Such systems measure degree of belief or degree of membership in the given concept.

13. Aristotle actually laid down three basic axioms, statements that are assumed to be true without requiring a formal proof, as the bases for his logical system:
- No statement can be both true and false (Law of Non-contradiction)
- Every statement is either true or false (Law of the Excluded Middle)
- An object is made up of its own particular characteristics that are a part of what it is (Principle of Identity)
categories to which things belong. His approach is usually called Aristotelian Logic or Syllogistic Logic and is based, unsurprisingly, on syllogisms.\footnote{A syllogism is an example of a form of argument called *modus ponens*, “method of affirming”, which simply says: If \( X \) is true, then \( Y \) is true. \( Y \) is true. Therefore, \( X \) is true. An example is the Socrates syllogism: If Socrates is a Man, then Socrates is Mortal. Socrates is a Man. Therefore, Socrates is Mortal. An alternative is *modus tollens*, “method of denying”, which is at the heart of Karl Popper’s falsification, the keystone of scientific proof, and says: \( Y \) is not true. If \( X \) is true, then \( Y \) is true. Therefore, \( X \) is not true. Leaning on Socrates again: Socrates is not a Dolphin. If Socrates can swim, then Socrates is a Dolphin. Therefore, Socrates cannot swim. I should add the disclaimer I have no idea if Socrates could swim or not but I’m relatively confident he was not a Dolphin.}

More formally, a categorical syllogism is an argument composed of two premises and one conclusion. Most people dislike arguments, thinking of them as involving bickering or fighting. But formal argumentation is a method of working through ideas or hypotheses critically and systematically.

An argument usually consists of a series of statements (called the premises) that provide reasons to support a final statement (called the conclusion). A deductive argument guarantees the validity of the conclusion while an inductive argument makes the conclusion more likely or probable.

In a syllogism, each of these statements (the two premises and one conclusion) are categorical statements where the first two (the premises) are true and the truth of the premises implies the truth of the conclusion. As Wikipedia, the font of all knowledge, puts it, a syllogism “is a kind of logical argument that applies deductive reasoning to arrive at a conclusion based on two or more propositions that are asserted or assumed to be true.”

The most famous example of a syllogism is:
1. Socrates is a man.
2. All men are mortal.
3. Therefore, Socrates is mortal.

Each of these three statements is made up of a subject and a predicate. Just as in grade school grammar, a subject is the main noun phrase that controls the verb and a predicate is the verb phrase that describes or modifies some property of the subject. A predicate can be a single word, a group of words consisting of a main verb and helping verbs, or a complete verb phrase consisting of the main verb and all the words related to that verb. For example, in the first propositional statement above, “Socrates” is the subject and “is a man” is the predicate.\footnote{Since there is a distinction between a simple predicate and a complete predicate, the verb phrase can either describe a property of the subject (complete predicate) or be more narrowly defined as a relationship between the subject and object (simple predicate) as a linking verb. This terminology can get quite confusing in different fields. For example, in Aristotelian logic, the linking verb is often called a copula and the object the predicate. In the Resource Description Framework (RDF), the RDF triples are expressed as Subject-Predicate-Object as in “Jack is a Friend of Jill” where “Jack” is the Subject, “is a Friend of” is the predicate, a simple predicate in this case, which describes the relationship between the Subject and the Object (in this case, “Jill”).}

Propositional logic extends Aristotelian logic to go beyond only dealing with the categories to which things belong. It is a more mathematical model for reasoning about whether propositional statements like logical expressions in general are true or false. We can represent individual statements with propositional variables, traditionally called \( p \) and \( q \). We can then generalize propositional logic further to predicate logic, also called first-order logic, in which the predicates are functions of variables that return Boolean values, as we’ll see in the next chapter. A predicate, in this context, describes the relationship between objects represented by these variables.

These logics, propositional and predicate, can also be represented symbolically. In propositional logic, an entire proposition is represented by a variable, typically a single letter like \( p, q, r \), etc. Then, a logical operation can be used to combine propositions and create a compound proposition. These
operations are represented by operators like \( \land \) (AND), \( \lor \) (OR), \( \neg \) (NOT), \( \Rightarrow \) (Implication/If-then), and \( \iff \) (Bi-conditional/If and Only If).

Statements and operators can be combined and the various combinations can be represented with a **truth table**, as below:

- \( p = \) “It is snowing.”
- \( q = \) “The driveway needs shoveling.”
- Truth Table for \( p \land q \): “It is snowing AND The driveway needs shoveling”:

<table>
<thead>
<tr>
<th>( p )</th>
<th>( q )</th>
<th>( p \land q )</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
</tbody>
</table>

However, the formalism of propositional logic is somewhat limited and cannot even express the Socrates syllogism we’ve used as our running example! As such, predicate logic introduces formalism to express both the objects as well as the properties of those objects. It does so by introducing **variables** (like, \( x \), \( y \), \( z \), etc.) to represent objects instead of entire propositions and **predicates** (like \( P \), \( Q \), etc.) to express the properties of the objects.

These predicates are then generalizations of propositions and are functions that return either True or False values only; thus, they become propositions when their variables are replaced with their actual values. All the logical operators from propositional logic carry over; in addition, we can add quantifiers like \( \forall \) (Universal/For-All) and \( \exists \) (Existential/There-Exists) to make propositional statements like: \( \forall x P(x) \) which asserts that the predicate \( P(x) \) is True for all \( x \). We can, of course, still apply all the normal propositional logic operators to such quantified statements, as well, to make compound statements like: \( \exists x (P(x) \land Q(x)) \equiv True \), which says that there exists an \( x \) such that both \( P(x) \) and \( Q(x) \) are True.

### 1.4.3 Declarative vs Imperative Statements

An argument is a way to reason from given information (the premises) to new information (the conclusion). The premises and conclusions, in our conception of formal logic so far, must be statements that then express information, in particular this information is either true or false. Such statements are called **declarative** statements. An example of a declarative statement, one that either states a fact or provides some information, is, “Socrates is a man.”

This is different from statements that are either questions or commands and so don’t express information. These kinds of statements, which are used to do something or give a command of some sort, are called **imperative** statements. An example of an imperative statement is, “Drink the hemlock!” These declarative and imperative statements are analogous to the declarative and imperative knowledge we met in Section 0.2.

Although the logics we’ve seen so far only involve declarative statements, it is entirely possible to develop a logic that includes imperative statements and, in fact, is the basis of many programming languages as we’ll see soon when our adventure continues in the next section.
1.5 Computational Thinking and Computational Solutions

At the end of our last adventure, though, we learned that not all problems in the universe have a computational solution. The good news, though, is that if a function is computable, a Universal Turing Machine can be programmed to compute it. The Universal Turing Machine solves these problems by an algorithm, a rule for solving a mathematical problem in a finite number of precise, unambiguous steps. These algorithms can be expressed in any language that is appropriate for a particular computing agent. In dealing with digital computers, we can write these algorithms in a plethora of different programming languages, including Java and Python.

It turns out that if a programming language can be used to simulate a Universal Turing Machine, that language is said to be Turing Complete as it can then solve any computable problem in the universe as shown by the Church-Turing Thesis! All modern programming languages, like Java and Python, are Turing Complete. What this means is that any solution that can be programmed in one programming language can be programmed in any other programming language: they are all equivalent in their computational ability!

This implies that a computational solution, or program, written in Java can be translated or expressed in Python without loss of correctness. For this reason, most computer scientists are language agnostics and choose whichever programming language is most appropriate for a particular problem since all languages are fundamentally equivalent in regards to their computational power. All you need to do is come up with a computational solution, or algorithm, for the problem at hand!

Listen to your heart... there is something else you can do...

How do you solve problems in your life, whether they’re interpersonal or practical or physical? Most people start by just using their intuition, which they often characterize as following their heart or listening to their gut. As it turns out, the gut doesn’t do a lot of thinking and the heart does even less. Almost all of our thinking is in our brains and it often builds upon sub-conscious processing as well as our experiences. Intuitive solutions are perfectly acceptable as long as they work and make accurate predictions reliably and consistently. Most of us, though, have more of a hit-and-miss record with intuitive solutions. Over time, people realized that systematic solutions to problems often yield more effective and consistent results.

As the early Greeks systematized those approaches to thinking which resulted in more consistent solutions to problems, formal logic became an accepted method for solving complex problems. A problem could then be modelled as an argument, a series of statements, or premises, and a conclusion; the rules that determined whether the conclusion was a reasonable consequence of the premises constituted the argument’s logic. So logic is just a structured, systematic way of thinking or reasoning about an argument.

Natural philosophers like Galileo Galilei used empirical thinking, where the premises are based on data from the natural world, which led to a particularly effective iterative process for solving problems that is often called the scientific method. In fact, computational thinking can be thought of as a set of skills or ideas that are a generalization of that iterative methodology.

---

16 What is a program? A translation of the algorithm into a language that can be interpreted and executed by a digital computer. A digital computer thus becomes the computing agent which carries out, or executes, that algorithm. E.g., our algorithm for finding the average in Section 0.1.3 can be thought of as \( \sum \frac{x}{N} \) which, when we translate into the algorithms shown in Section 1.7.

17 Surprisingly, the gut does have a hundred million or so nerve cells in the Enteric Nervous System (ENS) and also
1.5.1 Computational Thinking Overview

Computational problems, in general, require a certain mode of approach or way of thinking. This approach is often called computational thinking and is similar, in many ways, to the scientific method where we’re concerned with making predictions.\(^{18}\) We’ll delineate the steps involved in Computational Thinking in much more detail in Section 2.9.

**Definition 1.5.1 — Computational Thinking.** In order to make predictions using computational thinking, we need to define three things related to the problem and its solution:

1. **Problem Specification:** We’ll start by analyzing the problem, stating it precisely, and establishing the criteria for solution. A computational thinking approach to solution often starts by breaking complex problems down into more familiar or manageable sub-problems, sometimes called problem decomposition, frequently using deductive or probabilistic reasoning. This can also involve the ideas of abstraction and pattern recognition, both of which we saw in our temperature sensor problem when we found a new representation for our data and then plotted it to find a pattern. More formally, we’d use these techniques in creating models and simulations, as we’ll learn later on.

2. **Algorithmic Expression:** We then need to find an algorithm, a precise sequence of steps, that solves the problem using appropriate data representations. This process uses inductive thinking and is needed for transferring a particular problem to a larger class of similar problems. This step is also sometimes called algorithmic thinking. We’ll be talking about imperative, like procedural or modular, and declarative, like functional, approaches to algorithmic solutions.

3. **Solution Implementation & Evaluation:** Finally, we create the actual solution and systematically evaluate it to determine its correctness and efficiency. This step also involves seeing if the solution can be generalized via automation or extension to other kinds of problems.

I should add a little caveat here: these rules for computational thinking are all well and good but they’re not really rules, per se; instead, think of them more like well-intentioned heuristics, or rules of thumb.

These heuristics for computational thinking are very similar to the heuristics usually given for the 5-step scientific method\(^ {19}\) taught in grade school: they’re nice guidelines but they’re not mandatory. They’re suggestions of ideas you’ll likely need or require for most efforts but it’s not some process to pigeonhole your thinking or approach to solution.

The best way to think about it might be in more general terms and harkens back to how Niklaus Wirth, the inventor of the computer language Pascal, put it in the title of his book, interacts with the nervous system via gut hormones and microbiota, bacteria in the digestive system. Similarly, the heart has a few thousand ganglia, or clusters of nerve cells, but these neurons are mainly used for regulating cardiac function. So, as you might imagine, the vast majority of our thinking experience is in the brain.

\(^{18}\)The term computational thinking was popularized by Jeanette Wing, who studied under John Guttag, in her landmark 2006 viewpoint paper [10].

\(^{19}\) The scientific method is usually written out as something like:

1. **Observe** some aspect of the universe
2. Use those observations to inform some hypothesis about it
3. Make some prediction using that hypothesis
4. Test the prediction via experimentation and modify the hypothesis accordingly
5. **Repeat** steps 3 and 4 until the hypothesis no longer needs modification
Figure 1.1: **Computational Thinking is a set of techniques for solving complex problems** that can be classified into three steps: *Problem Specification, Algorithmic Expression, and Solution Implementation & Evaluation*. The skills involved in each step of the Computational Thinking Approach are above.

Data Structures + Algorithms = Programs. The title of the book suggests that both data structures and algorithms are essential for writing programs and, in a sense, they define programs.

We can think of programs as being the computational solutions, the solutions to *computable functions*, that we express in some particular programming language. We also know that an algorithm is an *effective procedure*, a sequence of step-by-step instructions for solving a specific kind of problem. We haven’t met the term data structures yet but we can think of it as a particular data representation. Don’t worry if this isn’t completely clear yet as we’ll be getting on quite intimate terms with both that and algorithms later on. For now, you can think of this equation as:

Representations of Data + Representations of Effective Procedures = Computational Solutions
At its core, the central aspect of all fundamental physical science is *prediction*, usually through experimentation as Feynman said in the quote earlier; if you’re able to make repeated, precise, quantitative predictions, whichever model you use or mode of thinking you employ is working and should likely be re-employed. If it’s a formal method, great; if it’s something less formal, yet still structured and repeatable like the above equation, and leads to correct computational solutions, that’s also fine.

Any structured thinking process or approach that lets you get to this state would be considered *computational thinking*. You can even think of it as an alternative definition of *critical thinking* or *evidence-based reasoning* where your solutions result from the data and how you think about that data:

\[
\text{Data } + \text{ How to Think about that Data } = \text{ Computational Thinking}
\]

In this sense, being able to represent the data and then manipulate it is itself a computational solution to a computable problem! We’ll look at the detailed steps involved in computational problem solving in a little bit.

### Some formal definitions

Before we can formally examine the differences between the declarative and imperative programming languages, we’ll need to define some basic programming concepts. We’ll be discussing these ideas more deeply when we get into the guts of machines and programming later on.

- **Variable**: a variable is the name for a stored value that represents some fundamental characteristic of the system or function. This stored value is *mutable*, i.e., it can be changed by the function or computation or program. We’ve previously seen input and output variables for a function as well as the variables for Time and Temperature when we collected sensor data in the previous chapter.

- **State**: the state of a system or function can be thought of as the values for some set of variables that fully describe the system or function. Usually, the state at a position in time does not include anything about its history. The state of the system for our Temperature Sensor example might be determined by the location of each sensor and the temperature reading on each sensor at a certain time; the state would then be determined by the location, temperature, and time variables for each sensor.

- **Function**: Another way to think about functions is to characterize them as a structured method for transitioning from one state (the input set of variables) to another state (the output set of variables). This state transition is not a mutable stored value.

### 1.6 Two Fundamental Models of Programming

In our previous discussion, we described functions in two ways: one was as a mapping or relation between the domain and the range of that function and the other was a mathematical representation of the function in functional notation.

These two approaches can also be categorized, respectively, as *declarative* and *imperative* (sometimes called *procedural*), the same two categories of knowledge we saw earlier when we discussed *know-what* and *know-how* knowledge and when we discussed declarative vs imperative *statements* in formal logic in Section 1.4.3.
So another way to characterize the function is to directly utilize our procedural knowledge about that function. In the imperative or procedural approach to defining a function, we found a procedure for determining an output value from a set of input values. More specifically, we found that the values of computable functions can be specified by effective procedures, or algorithms.

As we saw above, these computational solutions, or algorithms, for computable functions can be expressed in some programming language, especially when we want to use a digital computer as the computing agent for that solution. Approaches to computation using programming languages can also be classified in this same way.

Thus, the two primary models of computation for programming languages are: declarative approaches to computation, which include functional programming, and imperative approaches to computation, which include procedural programming and object-oriented programming.

1.6.1 Declarative vs Imperative Programming Languages

A programming language can choose to provide either, or both, of these approaches to computation although most languages are pre-dominantly one or the other.

Declarative programming languages define the program logic but not the actual control flow that determines how that solution is found. They can be reporting languages like SQL, for example. This declarative approach can be further extended to functional programming, which avoids the use of state and changeable data and instead approaches computational problems as functional transformations of data collections.

Imperative programming languages consist of statements that directly change the computed state. It can be further classified into the categories of structured programming and procedural programming, which we’ll meet in more detail later. Most modern languages like Java and Python are imperative languages that support both procedural and object-oriented paradigms.

Referencing these formal definitions, we can think of declarative programming as a model of computation that is based on a system where relationships are specified by definitions or equations which specify what is to be computed, not how it is to be computed. In the declarative programming paradigm, variables can only ever have one value which cannot be changed during a program’s execution (while a program is running). This approach is called a non-destructive assignment.

A Quick Recap

The state of a system can be thought of as the set of variables that give a complete description of the mathematical or physical system at a specific time. A real life system is much messier and more complex so you have to decide which variables form a complete description of the mathematical model of that system. The variables that completely describe the state of a system are called state variables and the set of all possible values of the state variables is called the state space of that system.

---

20 On a more technical note, structured programming has subroutines and avoids the use of the simple tests used in goto statements and instead uses more abstract and complex control structures like for and while. Procedural programming in addition supports full modularization, including event-driven and object-oriented programming.

21 We can think of declarative languages as specifying what to do and imperative programming specifying how to do it. This is similar to the two types of knowledge we saw earlier, know-what and know-how in Section 0.1.3.
Chapter 1. Computational Thinking and Information Theory

Variables | Control Structures
---|---
LIST_OF_SENSOR_LOCATIONS | foreach
RUNNING_SUM
TEMPERATURE
NUMBER_OF_SENSORS
AVERAGE_VALUE

Table 1.1: List of Variables and Control Structures in Algorithm 1.1

The imperative model of computation, on the other hand, uses well-ordered sequences of commands to modify variables by successive destructive assignments. They use three standard control structures to determine how the program carries out its set of instructions: either in **sequence**, one after the other; by **branching** or selecting between different series of options; or by **looping** or iterating over a series of instructions a certain number of times. We’ll look at these three control structures (sequence, selection, and repetition) in much more detail later.\(^{22}\)

### 1.7 Pseudocode

For now, let’s see one representation of an algorithm that’s not designed for a computer but rather for humans. In addition to programming languages like Java and Python, we can translate an algorithm into **pseudocode**, an English-like language that’s not designed for any specific computer or programming language.

Let’s see what all these variables and functions look like in pseudocode by designing an algorithm to calculate the average temperature at a certain time on campus from the notes collected in Figure 0.1.

In order to calculate the average temperature, we have to add up all the temperatures on all the sensors and then divide that sum by the number of sensors.

The algorithm shown in Algorithm 1.1 is a method for calculating the average temperature at a certain time across all the temperatures. It uses some variables (like LIST_OF_SENSOR_LOCATIONS, RUNNING_SUM, TEMPERATURE, NUMBER_OF_SENSORS, and AVERAGE_VALUE) and control structures (like the **foreach** looping control structure) as shown in Table 1.1.

So the first thing we need is to get the LIST_OF_SENSOR_LOCATIONS that were recording temperatures at a certain time, say 8am. Then, in Line 1, we create a variable called RUNNING_SUM to hold the sum of all the temperature values. This is the variable that will hold the sum of all the temperatures. Each time we read a sensor value from the LIST_OF_SENSOR_LOCATIONS, we’ll add it to the RUNNING_SUM. Since we haven’t yet read any sensor values at Line 1, we’ll initialize RUNNING_SUM to 0 in Line 1.

Next, in Lines 2 – 5, we loop over the LIST_OF_SENSOR_LOCATIONS and, for each SENSOR in the LIST_OF_SENSOR_LOCATIONS, we read the Temperature value (in Line 3) and then add that TEMPERATURE to the RUNNING_SUM.

\(^{22}\)Declarative languages, like functional languages, don’t have the looping control structure as not being able to change variable values would lead to infinite loops. Since functional programming cannot have loops, the only way to implement repetition is via functional recursion. Functional programming also treats functions as first-class components, which essentially means that they can be utilized everywhere other components, like variables, can be used.
After checking all the sensors in the LIST_OF_SENSOR_LOCATIONS, we then get the NUMBER_OF_SENSORS that were on that LIST_OF_SENSOR_LOCATIONS on Line 6. Finally, we calculate the actual AVERAGE_VALUE on Line 7 and, on Line 8, we print out that average value.

**Algorithm 1.1: Algorithm for calculating average temperature**

```
Input: The LIST_OF_SENSOR_LOCATIONS from Post-It Notes
1 Set RUNNING_SUM = 0
2 foreach SENSOR in the LIST_OF_SENSOR_LOCATIONS do
3     Set TEMPERATURE = The Temperature value read from the SENSOR
4     Set RUNNING_SUM = RUNNING_SUM + TEMPERATURE
5 end
6 Set NUMBER_OF_SENSORS = Length of the LIST_OF_SENSOR_LOCATIONS
7 Set AVERAGE_VALUE = RUNNING_SUM / NUMBER_OF_SENSORS
8 Print the AVERAGE_VALUE
```

If all of Algorithm 1.1 made sense to you, you’ve got a handle on most of the basic ideas needed for any and all programming paradigms!

The state of the algorithm can be considered to be the list of temperature values.

**Problem 1.1** Would the average temperature value also be considered part of the state of the system? Why or why not?

### 1.7.1 Pseudocode Expanded: Now with Twice the Pseudo!

We can now expand the Algorithm to also incorporate the last two major constructs used in basic procedural programming: selection control structures and procedural units, also called procedures.

**Nota Bene:** This section is just for fun for now, where fun is, obviously, a very relative term. We won’t define any of these terms, like selection control structures or procedures, precisely until later so don’t worry if none of this makes sense right now. If it does make sense, all the better! The main purpose of this section is to give an intuitive feel and hopefully the expanded pseudocode algorithm will be as approachable as Algorithm 1.1.

Keeping with this informal theme, you can think of a selection control structure as something that lets you make a choice, often called an if-statement: e.g., “if I’m hungry, then I’ll eat some pizza” is an example of a selection statement.

Similarly, we’ll use a procedural unit, or procedure, by way of preview of a modular programming approach we’ll explore in detail later; for now, let’s parcel off some of the calculation steps into their own procedure, called HelperProcedure. HelperProcedure is defined on Lines 1 – 13 and then we can just call, or execute, that procedure separately on Line 14.

We’ll shorten some of the variable names to make them descriptive but somewhat less verbose (e.g., we’ll change RUNNING_SUM to SUM, etc.). We’ll also include a selection control structure to ensure that the denominator is not 0. This might happen, for example, if there were no sensor values at all in the LIST_OF_SENSOR_LOCATIONS as you’d end up with an average value calculation of:
Algorithm 1.2: Algorithm for calculating average temperature using Procedures

Input: The LIST_OF_SENSOR_LOCATIONS from Post-It Notes and initial AVERAGE

// First, Define the Helper Procedure which uses the INPUT
1 Procedure HelperProcedure():
2    Set SUM = 0
3    foreach SENSOR in the LIST_OF_SENSOR_LOCATIONS do
4        Set TEMPERATURE = The Temperature value read from the SENSOR
5        Set SUM = SUM + TEMPERATURE
6    end
7    Set NUM_SENSORS = Length of the LIST_OF_SENSOR_LOCATIONS
8    // Make sure that number of sensors is not 0 to avoid division by 0
9    if NUM_SENSORS equals 0 then
10       Set NUM_SENSORS = 1
11    end
12    Set AVERAGE = SUM / NUM_SENSORS
13 end

// This is the START OF THE PROGRAM where we call the HelperProcedure
14 HelperProcedure()
15 Print the AVERAGE

\[
AVERAGE = \frac{SUM}{NUM_SENSORS} = \frac{0}{0}
\]

This is a problem since dividing by 0 is undefined so we can use the selection statement to just force the denominator to be 1 and make that a valid, but perhaps inaccurate, average value (e.g., it’s unlikely the temperature is actually 0 so you’d have to deal with this kind of semantic error, as we’ll see in Section 3.5).

The algorithm shown in Algorithm 1.2 also creates the same variables and same control structures as Algorithm 1.1; however, Algorithm 1.2 also includes a procedure (HelperProcedure) that does all the grunt-work of the actual calculation.

This means that the guts of our program only consists of the last two lines, Lines 14 – 15!

We can already see some of the essential characteristics of algorithms and effective procedures in our two example algorithms. Each step is a finite, doable step and the algorithm, or program, consists of a finite number of these finite steps.

Most of the statements are executable, i.e., they do something. Some of them, the comments, are intended only as remarks for us and not the computing agent. The comments, in this case, are lines that start with //. The essential computation, though, is contained in the instructions on Lines 1 – 13 and Lines 14 – 15 of our program.
1.8 Functional and Imperative Models of Computation

We’ve seen that two models of computation for programming languages are functional programming and imperative programming.

The functional model establishes some relation between the domain (the input) and the range (the output) of a function. This approach assumes that the function or computation always reads the input, processes it somehow, and then produces an output. Exactly how it processes it, the procedures it employs, are essentially irrelevant in the functional model of computation. All you have to do is declare the function that specifies what to do.

Looking in detail at the procedures, or sequence of commands or imperatives, that manipulate the input data representation to produce the output data representation leads us directly to the imperative model of computation. The imperative model of computation relies upon commands or statements called imperatives that map the program state before the statement is executed to a program state after the statement is executed or carried out by the computing agent.

If the sequence of statements, the algorithm, don’t involve any non-deterministic steps (like rolling a random die), they will maintain a strict correspondence between the inputs and the outputs and always produces the same output data for a given set of input data. So if they both maintain the strict correspondence between inputs and outputs, what’s the difference between these two models?

The most salient difference is that programs in the functional programming paradigm only specify what their outputs are, whereas programs in the imperative programming paradigm specify exactly how to compute the outputs. Some texts will even define Computer Science as the study of describing algorithms, constructing them, and comparing rival algorithms for computing the same function.

\textbf{Example 1.1} A simple example of the difference between the declarative programming and imperative programming paradigms is the common problem of answering someone who asks, “Where do you live?” In the declarative, or functional, programming paradigm, your answer might be, “I live at 137 Quantum Way, Beverly Hills, CA 90210.” The imperative programming approach, on the other hand, might be something along the lines of, “Take a right on Heisenberg Way, go 1.5 miles, turn left onto Galileo Alley, go 0.5 miles but stay in the rightmost lane. Turn on your lights when you go into the tunnel and turn them off when you come out of it. Finally, turn right onto Quantum Way and pull into the third parking spot with the large sign saying 137 above it.”

\textbf{Problem 1.3} Can you express the above example as a pseudocode algorithm, similar to Algorithm 1.1 or Algorithm 1.2?
Chapter 1. Computational Thinking and Information Theory

The advantage of the imperative model of computation is that it is much less restrictive than the functional model of computation since it covers any kind of manipulation of any data representation and it doesn’t, in the general sense, require both specific inputs and outputs. All it requires is some data representation and a procedure or algorithm for using that representation.

The Big Lebowski

We can think of imperative programming as involving procedures that are a collection of instructions, or algorithm. These procedures don’t always return a value, though. A function in mathematics, on the other hand, is something that maps specific inputs to specific outputs. As such, a function in the functional programming paradigm can be thought of as a collection of instructions, or algorithm, that also returns something, usually an output value of some sort.

1.8.1 Computation in General

Computation can then be broadly defined as the manipulation of some data representation by following some specific algorithm; but the strange thing is that the same computation, or mapping in the functional sense, can be carried out by many different permutations of algorithms, data representations, and computing agents.

In fact, algorithms themselves are representations since we can represent them as instructions for a human or as a program written for a digital machine to carry out. We can translate the algorithm, or procedure, from one language to another. When the computing agent needs to carry out, or execute, those instructions, it simply reads and follows the stored representation of the algorithm or program in a process called interpretation.

A Computational Universe

In this sense, computers can be thought of as devices that can store and interpret representations of algorithms and data, regardless of whether they are implemented in transistors, neurons, or quarks!

1.9 Information Theory

We talked about a function as transforming some input into the output. Both the input and the output contain information which the function processes and transforms.

These inputs and outputs are also encoded in some specific representation. In fact, the choice of a specific data representation can often affect our choice of corresponding algorithm to solve some problem. As a result, computer scientists often study the properties of different kinds of representations of data, or data structures.

Since we know we can transform data from one representation to another, we can start to abstract this process by ignoring the particular representation and instead think about the algorithm as manipulating the information in the input. But what is the information content of some particular data representation?
1.9 Information Theory

Is there more or less information content in another data representation? Can we somehow quantify this idea of information?

1.9.1 The Birth of Information Theory

This is just what occupied Claude Shannon, a mathematician at Bell Labs, in the 1940s. At the time, Bell Labs had a very liberal working policy: scientists were allowed to spend a significant portion of their time working on any problem that interested them as long as they kept their doors open for other scientists to drop by. Since Bell Labs was owned by AT&T Corporation, Shannon decided to take some of his free time and look at the meat and potatoes of their business: communication.

In particular, he realized that no one at AT&T had actually defined what it was they were communicating and what was at the very core of their extensive business enterprise: the information. Other people had, of course, thought deeply about this idea of quantifying information both at Bell Labs (like Harry Nyquist and Ralph Hartley) and outside (like Leo Szilard, Edwin James, and Leon Brillouin). But Shannon wanted to nail this down precisely and figure out the limits of communication. So, building upon some of this previous work, Shannon started work on a memorandum to analyze the transmission of information. Finally, in 1948, Shannon managed to fully quantify this idea of information in the context of communication channels and published his seminal paper, “A Mathematical Theory of Communication.” [11]

1.9.2 First Rule of Communication Club: There Is No Meaning

The fundamental problem Shannon examined was how to send a message from a source to a receiver without any degradation of the message itself. In order to examine this transmission of messages, the first thing Shannon did was ignore the meaning of the message! This kind of abstraction, or ignoring of what are considered non-essential details for a specific approach to solution, is something we’ll do quite often when we solve computational problems, as we saw in Section 1.5.1. As Shannon put it, the “semantic aspects of communication are irrelevant to the engineering problem”. [11]

Instead, he tried to distill the problem down to its essential aspect which he finally conceptualized as selecting the actual message to be transmitted from the set of possible messages that could be transmitted. So how might this one message, selected from the set of all possible messages, suffer degradation when it’s actually transmitted?

Let’s start by looking at the same problem Shannon and Hartley had examined: suppose we had a very simple system that consisted of a transmitter, a wire, and a receiver. The person on the receiving end can receive three kinds of symbols:

1. HIGH signal (a spike up)
2. NO signal (no spike at all)
3. LOW signal (a spike down)

This is shown in the original image from Hartley’s paper [12], as shown in Figure 1.2. As Hartley points out, although these symbols might have some psychological meaning for the sender or receiver, we won’t delve into those semantics. We’re just concerned with the transmission of this sequence of symbols, which we call our message.

You can think of Figure 1.2(A) as showing the original signal as sent by the sender at the Transmitter. We can then think of Figure 1.2(B) as showing the signal after it has gone down the wire for quite a while. (C) is the same signal after it has gone down even more of the wire’s length and, finally, (D) is

---

23 You can also think of this as a telegraph system that sends Morse code. For more on Morse code, please see [https://en.wikipedia.org/wiki/Morse_code](https://en.wikipedia.org/wiki/Morse_code)
that same signal after it has traveled down the wire for perhaps a few miles. As you can see, the signal in (D) is unrecognizable from the signal in (A)!

What is the source of this degradation? As any continuous signal travels down any communication channel, there are small, unpredictable perturbations that creep into that signal. These small perturbations that arise due to the physical transmission of the signal down the wire are called **noise**.

If that signal encodes, or represents, a message, then the message itself gets corrupted! The degree of corruption, or degradation, of the message can be represented by a simple ratio, the **signal-to-noise** ratio. When the message is first sent from the receiver, as in (A), the message has very little noise so the signal-to-noise ratio is high. By the time the message gets to (D), however, the noise has increased greatly, due to the perturbations in the transmission of the signal down the wire, and the signal-to-noise ratio decreases significantly, thus garbling the final message received at the receiver end.

---

### A unit by any other name...

Whenever we make a quantitative measurement, we usually express it in terms of some defined quantity. This lets us compare the measurements to an agreed-upon standard. For example, suppose I measure the distance between my desk and my chair and declare it to be 1 foot long. What do I mean by a foot? Do I mean my own foot or the much larger foot of my cousin, who stands a full head taller than me?

These kinds of discrepancies led people to standardize the measurements. Although the original standard may well have been the foot of some prominent personage, the foot is nowadays defined to be exactly $0.3048$ meters. Thus, if I say the distance is 1 foot, someone on the other side of the world will know exactly what length I mean.

The unit in the term, **unit of measurement**, refers to the number 1 so that all measurements in terms of that unit of measurement will be multiples of that 1 unit. So $3\text{ feet}$ is $3 \times 1\text{ foot} = 3\text{ feet}$. In other words, all measurements are reported in terms of multiples of that unit. Once we quantify information, we need to express a standard unit of information, as well, so we an measure the amount of information we have. That is exactly what Shannon did when he quantified the idea of “information.”
Encoding the Message

Hartley [12] had also tried to quantify the amount of information in a message without regard to its semantic meaning. The approach Hartley took was to quantify it by first finding the number of possible symbols, \( S \), which, in our case above, are HIGH, NO, and LOW. For a specific message, made up of \( n \) symbols, Hartley characterized the quantity of information as:

\[
H = \log_b S^n = n \log_b S
\]  

(1.3)

If we set \( n = 1 \) in Equation 1.3, we end up with a single unit of information in this representation, \( \log_b S \)!

Log this way... add this way...

A logarithm is a way to allow multiplication to be performed via addition. A log can be thought of as the opposite, or inverse, of an exponential. The two important terms in a logarithm are the base, \( b \), and the exponent, \( x \), just as in exponentiation. For example, if I have the following exponent:

\[
y = b^x
\]

which raises the base, \( b \), to the exponent or power, \( x \), I can represent the inverse operation as the following logarithm:

\[
\log_b(y) = x
\]

Another neat thing about logs is that raising a number to a certain power lets you bring that power down as a multiple; this is probably easier to see in an equation than explain in words:

\[
\log_b(y^A) = A \cdot \log_b(y)
\]

This also implies that a multiplicaton by a negative factor, \(-A \log_b(y)\), is the same as raising the number to a negative, or inverse, power, \(\log_b(y^{-A})\).

Just as doing an operation, and then its inverse, gives us back the original, in the same way, doing a log followed by an exponentiation (or the opposite) gives us back the original:

\[
b^{\log_b(m)} = m \quad \text{and} \quad \log_b(b^m) = m
\]

So how does this simple inversion allow us to do multiplication as addition instead? If you had to multiply the following:

\[
\log_b(m \times n)
\]

you could instead just add the following:

\[
\log_b(m \times n) = \log_b(b^{\log_b(m)} \times b^{\log_b(n)}) = \log_b(b^{\log_b(m)+\log_b(n)}) = \log_b(m) + \log_b(n)
\]

log calculations for a large number of values were historically recorded in voluminous log tables but today you’re more likely to just type it into a calculator or a log calculator online.

Contact adopt@cognella.com, 800.200.3908, for more information © COGNELLA

DO NOT DUPLICATE, DISTRIBUTE, OR POST
If we then use $\log$ with the base $b = 10$, as in $\log_{10}$, the unit of information will be expressed in decimal digits; these units of information eventually came to be called the Hartley units and were our first measure of information. In fact, $H = \log_b S$, which is just Equation 1.3 with $n = 1$, is sometimes also called the **Hartley Function** or **Hartley’s Information Entropy**.

$$H = \log_b S$$ (1.4)

### 1.9.3 Shannon Information

Shannon decided to look at this problem more closely. He started by examining the transmitter and continued the process of *removing “semantic” meaning* from his examination of sending and receiving messages. So, rather than imagining a person sending a message, he thought about a mathematical process, a **stochastic process**, generating a string of symbols at the source. A stochastic process, in this case, is just some mathematical model of a system that produces a sequence of symbols based on some set of probabilities (probability rears its head once again!).

If the set of possible symbols is again as before (i.e., **HIGH**, **NO**, and **LOW**), then the number of possible messages, if all of the symbols are equally likely, would be $3^n$, where $n$ is the number of symbols that make up our particular message and the set of all possible symbols, our alphabet, is made up of just 3 possible symbols (i.e., **HIGH**, **NO**, and **LOW**). So if our message consisted of just two symbols, there would be $3^2 = 9$ possibilities (like {**HIGH** ↪ **HIGH**} or {**LOW** ↪ **HIGH**} or perhaps {**NO** ↪ **LOW**} for our 2-symbol message).

Shannon decided to take a somewhat different route here than Hartley. He started by imagining all the possible messages of a certain length he could make given a certain set of symbols (e.g., in our example with 3 possible symbols, we could make 9 2-symbol-long messages). Since he had removed all psychological meaning from these messages, a la Hartley, he figured all these 9 messages were **equally likely**.

He next asked the question, “Can we find a measure of how much ‘choice’ is involved in the selection of the event or of how uncertain we are of the outcome?” [11] In other words, how much “choice” is involved in picking one of these messages from the 9 possible messages to send down the pipeline? If you’ve been paying close attention all along, alarm bells must be going off for you now (I, of course, never was that good of a student so I never heard any bells while reading textbooks). This is exactly what all those **probabilities** we looked at earlier measure!

He decided to base his measure of the information that is “produced” when one message is chosen (from a finite set of possible messages) on the probability of that message occurring or being selected. The raw probability, however, wasn’t mathematically convenient so he outlined three reasons for doing what Hartley had done earlier and used a logarithm function. Since he was interested in finding out how “…uncertain we are of the outcome,” he also inverted the probability, $p$, to end up with **Shannon’s Information Function**:

$$\text{Information Function} = h(p) = \log_2 \left( \frac{1}{p} \right) = -\log_2(p)$$ (1.5)
This measure of information is calculated in base-2 rather than the base-10 of Hartley. Since Shannon was looking at the fundamentals of communication, base-2 was a convenient way to measure the amount of information if the device used was something like a flip-flop circuit, which only had two stable positions: ON or OFF, even simpler than our 3-position, or 3-state, system of HIGH, NO, and LOW above.

These kinds of devices with just two states, ON and OFF, can be represented using binary digits. A binary digit is a 1 or a 0 and can be represented in a base-2 positional system, as opposed to the base-10 of our normal decimal digit representations which use a base-10 positional system. So a binary positional system only uses 2 digits (a 0 or 1) but a decimal positional system uses 10 digits (0, 1, 2, 3, 4, 5, 6, 7, 8, and 9). We’ll talk more about these positional representations in Section 2.3.1, as well.

### Abstracting Information into Binary Systems

The idea of abstracting information in a system didn’t originate with Shannon, of course. In 1801, the French weaver Joseph Marie Jacquard patented a mechanical loom that used perforated cards to control the weaving mechanism. The perforated cards had holes and blank spaces which controlled the movement of counterweights, thus allowing the wires to move in the pattern specified by the holes and spaces on the card.

The holes and blank spaces allowed Jacquard to abstract the information contained in any image or picture by using a binary representation of two symbols, a hole or a blank space, just like the flip-flops studied by Shannon! In fact, Leibniz had also advocated the use of a binary system in 1703.

---

**A bit is not the same thing as a binary digit!**

Even though the word bit is derived from the words binary digit, a bit and a binary digit are not the same thing in information theory. A binary digit is a specific value, a 0 or 1, that is assigned to a binary variable. A bit, on the other hand, represents the amount of information, \( h(p) = \log_2\left(\frac{1}{p}\right) = -\log_2(p) \), carried by a symbol when the probability of seeing that symbol is \( p = 0.5 \) or 50% (this is the case when there are just two possible symbols, for example). If we plug this probability, \( p = 0.5 \), into our formula, we get the result that one bit of information is 1:

\[
h(p = 0.5) = \log_2\left(\frac{1}{0.5}\right) = \log_2(2) = 1 \text{ bit}
\]

The amount of information, measured in bits, can be any real value. E.g., if the probability of seeing a symbol is \( p = \frac{3}{4} = 0.75 \), then the amount of information would be \( h(p = 0.75) = \log_2\left(\frac{1}{0.75}\right) = 0.41 \text{ bits} \)
1.9.4 Information, Uncertainty, and... Surprise!

For now, we can think of one unit of information as the information carried by a symbol (or message) when the probability of seeing that symbol (of that event occurring) is $\frac{1}{2}$. In other words, one unit of information represents a choice between two equally likely possibilities. This unit of information is referred to as a bit, which John W. Tukey had recently coined as a portmanteau from binary digit. Thus, Shannon’s measure of information came to be measured in bits, which are also sometimes called Shannons (Sh) in his honour.

Nota Bene: A bit by definition is the amount of information needed to choose between two equally probable possibilities. Sometimes, a bit is also used as a short-hand to mean a binary digit, which is the value of a binary variable and can be a 0 or a 1. In some circumstances, the context will be the only way to determine which meaning is intended.

It’s important to note that Shannon actually defined information as a decrease in uncertainty at the receiver’s end and posed 5 postulates for the information function, $h(p)$. This representation of information as $h(p) = \log_2(\frac{1}{p}) = -\log_2(p)$ can thus also be interpreted as a measure of the amount of surprise associated with observing some symbol from our set of possible symbols (or, alternatively, the surprise of some event occurring). The smaller the probability, $p$, the larger the surprise, $h(p)$.

Shannon’s 5 postulates for information, $h(p)$

1. $h(p)$ is continuous for $0 \leq p \leq 1$
   (Continuity: just as for a normal probability)
2. $h(p) = \infty$ if $p = 0$
   (an event with 0% probability would result in an infinite amount of surprise)
3. $h(p) = 0$ if $p = 1$
   (an event with 100% probability has 0 surprise)
4. $h(p_i) > h(p_j)$ if $p_j > p_i$
   (if the probability of one event, $j$, is greater than another event, $i$, the surprise of that first event, $i$, is lower)
5. $h(p_i) + h(p_j) = h(p_i + p_j)$ if the two states $s_i$ and $s_j$ are independent
   (Additivity: the information from two independent symbols is linearly additive)

It can be shown [11] that the only function that satisfies these five properties is $h(p) = -C \log_b(p) = C \log_b(\frac{1}{p})$ where $b$ is some base, not necessarily base-2, and $C > 0$ is just a scaling factor or choice of unit of measure.

To get Shannon’s Information Function, we set $C = 1$ since this gives a nice, simple 1 unit of “surprise” as a measure for information: $h(p) = \log_b(\frac{1}{p})$

---

26 As we’ll see later on, Shannon defined the Rate of Information Transmission as $R = H_{\text{before}} - H_{\text{after}}$, where $H$ is his uncertainty function or entropy, as John von Neumann suggested in Section 1.10. Information is thus the reduction in uncertainty after subtracting what he called the equivocation or the “average ambiguity of the received signal,” $H_e(x)$, the average rate of conditional entropy and this difference gives a sense of what he called the “missing information.”

Interestingly, we can apply Shannon’s Uncertainty Function, $H$, to the probability distribution of locations and momenta of a system of many particles. We can then use MaxEnt to find the distribution that maximizes $H$ to get back the statistical physics result that, for a system of non-interacting particles, the distribution of locations is uniform and the distribution of the momenta is the Maxwell-Boltzmann distribution and is equivalent to Boltzmann’s entropy, as we’ll see later. Arieh Ben-Naim further showed that this equivalence is in the “sense that calculations of entropy changes between two equilibrium states of an ideal gas, based on this definition agree with the result based on Clausius entropy.” In fact, David Layzer even went so far as to call Information, which was defined as the difference between Maximum Possible Entropy and Actual Entropy, the Arrow of Time.
Put another way, the rarer the message, the greater the information. The only caveat I’d add is to not extrapolate our idea of information to what Hartley called psychological aspects of information. As Mark Twain famously said, “We should be careful to get out of an experience only the wisdom that is in it and stop there.”

1.9.5 Data → Information → Knowledge Revisited

We’ve previously talked about finding different representations of data and how we could translate from one representation to another with no loss of information. Now we can see why: we can encode any message into binary digits with no loss of information as defined above as long as the probabilities (or frequencies) of the symbols remains the same. The most impactful aspect of Shannon’s seminal work consists in the implications of his simple formulation.

Digitizing information, converting it to a binary digit representation as a sequence of 0’s and 1’s, allows us to transmit messages without degradation by amplifying it using readers and amplifiers, also known as regenerative repeaters, instead of simple amplifiers. Not only did Shannon find an approach to quantify information but he was also able to find the limits on the storage and transmission of that information. Shannon was thus able to explain the fundamental bounds on signal processing and communication operations such as data compression, including explaining why languages like English have to contain a certain amount of redundancy!

Earlier, we also saw that we can convert our observations of the physical world into data by finding a suitable representation for them. Giving the data a context of some sort transformed it into information. So how does Shannon’s Information relate to the framework we developed in the last chapter?

For example, if we go back to the 3-symbol or 3-state system of HIGH, NO, and LOW, the data could be the observations of the symbols as they come down the wire. After observing a lot of symbols over a long time, we can transform those observations into tables sorted by each of the three symbols. Then, the observed frequencies for each of the three symbols would give us a sense of the probability for each of the three symbols. This probability, \( p \), would be the context for the data and is directly related to the Shannon Information via \( h(p) \)

A new kind of context...

The ability to change our representations of data without loss of information also allows us to digitize the message and utilize the full Shannon framework, as well. Once we have that, the context is simply the probability of each of the symbols and, hence, the probability of the messages.

So we can, in some sense, think of information as data that’s been given some context. If that context is the probability of occurrence of the messages or the symbols that constitute the messages, we end up with Shannon’s Information measure. If that context is some logic-based reasoning system, we

---

27 Since the probability of two events occurring together is the product of their individual probabilities for independent events, this property also states, that the information from both events occurring together (\( e_1 \) and \( e_2 \)) is equal to the information from \( e_1 \) plus the information from \( e_2 \).

28 Simple amplifiers will just increase the amplitude of a signal whereas a regenerative repeater will amplify, reshape, retime, and then retransmit a digital signal.
can use it to argue for some conclusion based on the evidence; that conclusion is then the information we can use as the basis for creating actionable knowledge.

**Data → Information → Knowledge Defined!**

In somewhat understated terms, we can then say that data is an organized sequence of symbols representing ideas or observations where the symbols might have some value, which is some meaning in some interpretation. Information is data that has some context which helps resolve uncertainty or answers a question of some sort and so reflects our prediction about the world. Finally, knowledge is an implicit (practical, know-how skills) or explicit (theoretical, know-what facts) understanding of formalized, organized information that can help solve some problem.

### 1.10 Shannon’s Information Entropy

We can use this quantification of information, $h(p)$, to find the average information per symbol in a set of symbols with some a priori probabilities [11]. Shannon’s information function is symbolized by a lower-case $h$ while Shannon’s uncertainty function, or Shannon’s entropy, is symbolized by a capital $H$. If we take the sum of the information totals associated with each symbol, and weight each symbol’s information by the probability of observing that symbol, we end up with Equation 1.6, which is called the Shannon Information Entropy, for $N$ possible symbols.

$$H = C p_1 h(p_1) + C p_2 h(p_2) + C p_3 h(p_3) + \cdots + C p_{N-1} h(p_{N-1}) + C p_N h(p_N)$$

$$= C \sum_{i=1}^{N} p_i h(p_i)$$

$$= -C \sum_{i=1}^{N} p_i \log_2(p_i)$$

(1.6)

Here, we used the capital Greek sigma, $\Sigma$, as a shorthand to represent the Sum of numbers from $i = 1$ to $i = N$. If we again set the scaling factor, or choice of unit of measure, as $C = 1$, we end up with:

$$H = - \sum_{i}^{N} p_i \log_2(p_i)$$

(1.7)

In this equation, $p_i$ represents the probability of observing the $i$th possible symbol and $H$ is given in units of bits per symbol, as well. This means that the entropy is the average information per symbol. If this is the case, in order for that average information per symbol to be high, the distribution of probabilities will likely have a large number of unlikely events in it. In other words, there is a lot of uncertainty in the distribution and entropy itself can be thought of as a measure of the spreading out of the probabilities. In fact, for a given distribution of states, entropy is maximized when the states are equiprobable and all have the same likelihood.

Alternatively, if we’re looking at observing messages, rather than individual symbols, we can think of the probability, $p_i$, as the probability of the message, $m_i$, that is taken from the entire message space,
or set of messages, $M$. If the messages are all equally likely, the $p_i$ are all the same and Equation 1.7 reduces to **Hartley’s Information Entropy**:\(^{29}\)

$$H = \log_2 |M| \quad (1.8)$$

In this equation, the $|M|$ is the cardinality of the message space $M$. As we saw earlier, when base-2 is used for the logarithm, as in Equation 1.7, the unit of information entropy is called either the *bit* or the *Shannon*. If we were to use a different base for the logarithm, the resulting units have a different name. If we set the base to Euler’s number $e$, and thus change the log to $\ln$ (or $\log_e$), the unit is called a *nat* (with $1\text{nat} = 1.443\text{bits}$) whereas setting the base to 10, or decimals, results in a unit called *Hartley*, as we saw before.

If we change $C$, we can have different units entirely as $C$ is also a unit of measure factor. In fact, we’ll soon change it to $k_B$ which will allow us to connect our formulation to Thermodynamics. In some formulations, $k_B$ is also set to 1 to make things more convenient. \([13]\)

Shannon’s information entropy is defined for a specific context, a specific probability of the symbols, and is equal to the *average amount of Shannon’s Information provided by those symbols* in the above case, we assume that each symbol has the same probability, $p$.

Shannon’s information entropy can thus quantify the information content of a system; the larger the information entropy, the greater the information content calculated on the receiver’s end as the unconditional entropy minus the conditional entropy, as we’ll see later. In terms of the amount of surprise, we can think of $H$ as an average of the surprise associated with all the possible observations.

We can also think of $H$ as a **measure of the amount of uncertainty** associated with the value of a *random variable* when we only know the distribution of that random variable. We can then use information entropy to define other kinds of information. Just as information entropy is a measure of the amount of information in a single random variable, we can define **mutual information** as a measure of the amount of information that is in common to two different random variables, as we’ll see later in Section 10.6.

In fact, Shannon’s information entropy, $H$, can be used to help predict the efficiency limits on any binary encoding for a given set of symbols.\(^{30}\) It also shows, via Shannon’s source coding theorem, that the fundamental limit of the (lossless) encoding or compression of a message is such that a message cannot be compressed to have more than one bit of information, on average, for each bit of the message. In this way, Shannon’s information entropy can even provide a lower bound on the most efficient encoding for compressing text in languages like English.

### 1.10.1 Entropy in Physics

In 1854, physicist Rudolf Clausius was wrestling with a different kind of efficiency, the efficiency of heat engines. A heat engine uses the temperature difference between two heat reservoirs to do work. Heat is the transfer of energy between two bodies which, in this case, are two reservoirs at two different temperatures.\(^{31}\)

---

\(^{29}\) If there are $|M|$ messages and each is equally likely, we get probabilities, $p_i = \frac{1}{M}$. This gives $H = -\sum \frac{M}{M} \log_2 \left( \frac{1}{M} \right) = -M \log_2 \left( \frac{1}{M} \right) = \log_2 |M|$.

\(^{30}\) For example, we can define efficiency as entropy divided by average number of bits per symbol used in code.

\(^{31}\) Temperature is the average kinetic energy of the atoms of a physical system and is an intensive property of the system, which means it doesn’t depend on how much of the substance you have. Heat, on the other hand, is energy that is transferred from one substance to another and is an extensive property of matter that does depend on how many atoms you have in the system. We’ll also discuss work and energy further in Section 1.10.5.
Heat can be thought of as energy in transition, just like work, and involves the transfer of the internal energy, both kinetic and potential, of a system of particles. This heat energy is transferred via conduction, convection, and radiation and, in a heat engine, it’s transferred via conduction from the hotter reservoir to the colder reservoir. As it’s transferred, some of the heat energy can be used to do work in this full cycle of energy transfer from the hotter to colder reservoir. A heat pump works the opposite way: it uses work to transfer heat from the colder reservoir to the hotter reservoir.

The efficiency of a heat engine is defined to be the ratio of work done over input heat and tells us how much of the input energy ends up doing useful work. Maximum efficiency can be achieved when there are no losses to friction or other effects. This implies that the processes are reversible; if energy is lost to friction then the process is irreversible.

A Carnot engine is an idealized engine that uses this most efficient heat engine cycle, called the Carnot cycle; the Carnot engine is idealized because it has to be perfectly reversible and not have any losses to frictional effects, etc. All other engines are non-reversible and have a lower efficiency than this maximum.

Clausius characterized this thermodynamic irreversibility by defining a thermodynamic quantity that depends on the size of heat energy, $Q$, and temperature, $T$, involved in a heat transfer. He called this quantity entropy, $S$, after the Greek word, entropía, which means “in transformation,” and defined the change in entropy ($\Delta S$) to be:

$$\Delta S \geq \frac{\Delta Q}{T} \Rightarrow \frac{Heat}{Temperature}$$

The inequality, the Clausius Thermodynamic Entropy, becomes an equality only for a reversible process whereas, for an irreversible process, it is always the inequality. So, for an irreversible process, the entropy always increases and the change in entropy is positive. This implies that the total entropy of the universe always increases and is often called the Second Law of Thermodynamics.

Statistical Physics and Entropy

There is a strong connection between probability and thermodynamic entropy. Thermodynamic systems can be described either from a microscopic or macroscopic perspective. You can describe the state of a system from a microscopic perspective as its microstate, which is determined by the position and velocity of every particle in the system.

The macrostate, on the other hand, describes the state of a system using macroscopic properties like temperature, pressure, and volume of the set of particles in the system. All of these macroscopic properties are measurable but only give partial information about the system unlike the microstate, which gives complete information about the system of particles.

In 1877, the brilliant physicist Ludwig Eduard Boltzmann used atomic theory to define the entropy of a system of gases. He started by assuming that all microstates are equally likely where each microstate was a specific state for each of the particles of that system. Every time any particle changes its position or its velocity, you end up with a new microstate.
Castles in the air...
One way to think of this might be with a system made out of Lego’s. Let’s say you have a few thousand tiny little Lego’s and you use them to build a really ornate castle. The outside of your castle consists of about 150 Lego’s and all the rest go on the inside, making up the structure and volume of the castle. You can then think of the castle as seen from the outside as the macrostate: you can measure how tall it is and how much it goes around, for example. The exact layout of each and every Lego, including the ones that are hidden on the inside of the castle, is a description of one microstate. If you move even one Lego from inside the tower to inside the moat, you’ve changed to a brand new microstate. Just as with the physical case, many different microstates can correspond to the same macrostate. For example, if I move around the Lego’s that are inside the tower, every move determines a different microstate but the macrostate, how the tower looks from the outside, might be unchanged.

In fact, given the inordinately large number of microstates, there will usually be a huge number of microstates that correspond to a particular macrostate with specific values of temperature, pressure, volume, etc.\[^{32}\] So for a system with \( N \) different possible microstates consistent with some given macrostate, the individual probability for each equally likely microstate would be \( p = \frac{1}{N} \). The entropy of a statistical thermodynamic system, called Boltzmann’s Thermodynamic Entropy, can then be shown to be:

\[
S = k_B \ln N \tag{1.10}
\]

Another physicist, J. Willard Gibbs, came up with a formulation of the entropy in terms of the probability of each of the microstates as:

\[
S = -k_B \sum_i^N p_i \ln p_i \tag{1.11}
\]

The formulation in Equation 1.11, called Gibbs’ Thermodynamic Entropy, is analogous to Shannon’s Information Entropy in Equation 1.7. As we saw with Shannon’s Information Entropy, if all the microstates are equally likely, you end up with Equation 1.10, just as the equally likely messages

\[^{32}\] More than one microstate can correspond to a specific macrostate; the microstates are continuous and so you partition phase space into cells where each cell is considered to be a microstate, thus making them discrete and countable.
for Shannon’s Information Entropy in Equation 1.7 gave us Hartley’s Information Entropy in Equation 1.8:33

\[ S = -k_B \sum_i N p_i \ln p_i = \frac{\delta Q}{T} = \frac{\text{Heat}}{\text{Temperature}} \]  

(1.12)

Unpredictability and Life, The Universe, and Everything...

In a way, you can think of Shannon’s Information Entropy as a measure of unpredictability or uncertainty about a system. In fact, the most important consequence of Shannon’s formulation of information was to show that what we had been calling entropy in thermodynamics or statistical mechanics or communication channels could be generalized well beyond these fields. Instead, that general concept could apply to any context where you can define probabilities for that system.[13]

Although we call this quantitative measure “information”, remember that our “information” is stripped of all semantic meaning! It’s essentially a mathematical model of some aspect of a system which we can then use to make predictions about the system. If these predictions are borne out, then we usually conclude that the mathematical model is very likely to describe the underlying physical reality and so “describes” something about the universe.

But really, concepts like energy, entropy, and information are all abstractions. As Feynman might say, they’re bookkeeping tools: just a bunch of numbers we use to characterize some system, like we see in Section 10.6.2. Even force, mass, particles, waves, etc., at the fundamental level, are abstractions or mathematical models. These models have a collection of numbers associated with them, along with some rules for the model to follow. In other words, Numbers + Rules About Those Numbers = Physical Theories, which we use to predict various aspects of the world and how its components will behave.

1.10.2 Information Entropy and Physical Entropy

So where does all this get us? Well, so far, as shown by Jaynes, the classical entropy, Clausius’ Thermodynamic Entropy, is related to a probabilistic description of the microstates of a system in Gibbs’ Thermodynamic Entropy. This, in turn, has a formulation that looks just like the general formulation of Shannon’s Information Entropy in Equation 1.6, where that pesky scaling factor or choice of unit of measure, \( C \), is set to the Boltzmann factor, \( k_B \), which gives us units of specific heat.

33 You can easily get the Boltzmann entropy, \( S_B \), from the Gibbs entropy, \( S_G \), for a system with microstates of probability \( p_i \) each, which is: \( S_G = -k_B \sum_i N p_i \ln(p_i) \). In equilibrium, all microstates associated with the equilibrium macrostate are equally likely so, for \( N \) equally likely microstates, we get probabilities, \( p_i = \frac{1}{N} \). This gives \( S_G = -k_B \sum_i N \frac{1}{N} \ln\left(\frac{1}{N}\right) \Rightarrow -k_B N \frac{1}{N} \ln\left(\frac{1}{N}\right) \Rightarrow k_B \ln(N) = S_B \), which is the Boltzmann entropy for a system with \( N \) microstates. There is a slight caveat that \( S_B \) is for an isolated, micro-canonical (where all microstates are equiprobable) system whereas \( S_G \) is for a canonical system that can exchange energy with its environment.
This similarity in the two mathematical formulations is what led John von Neumann to recommend the term entropy for Shannon’s measure of the average amount of Shannon’s Information. In a personal message, Shannon, who had been struggling with finding a meaningful name for his new construction, was told by von Neumann not to call it **uncertainty**. He instead recommended:

> You should call it entropy, for two reasons:  
> In the first place your uncertainty function has been used in statistical mechanics under that name, so it already has a name. In the second place, and more important, nobody knows what entropy really is, so in a debate you will always have the advantage.

This famous anecdote was said to have occurred in 1940 at Princeton’s Institute for Advanced Study and you can read various amusing summaries of this anecdote online. But von Neumann might have had a deeper motivation for the nomenclature as he had already formulated his own version of Quantum Entropy in 1932!

Jaynes saw this connection as being more fundamental and went even further to suggest that Thermodynamic Entropy can be seen as an actual *application* of Shannon’s Information Theory [15]. He was one of the first to realize that entropy essentially quantifies the amount of information about the microstate that we lose when we monitor the system *macroscopically*.

Another way to think about this is with the Lego’s Castle analogy. In that analogy, we imagined that the castle’s external appearance determined its macrostate and the detailed internal layout of each of the Lego’s determined a single microstate. If you were to observe just the external macrostate, say a specific configuration of the castle’s towers and moats, you wouldn’t know exactly which internal Lego’s were placed where.

Based on just the currently known macrostate of the castle system, the amount of information that’s not available is related to knowing which particular microstate, out of the many different microstates that correspond to that same macrostate, is the actual current microstate of the castle system. For example, the tower could have 10, 50, or 200 Lego’s on the inside and the entropy would tell us how many are actually in there.

So Clausius’ Thermodynamic Entropy is tied to Gibbs’ Statistical Entropy via the missing microstate information. In fact, Shannon’s Information Entropy talks about exactly this kind of information in the general sense, without being mired in Thermodynamics via $k_B$ at all!

As it happens, in the Boltzmann formulation entropy, all microstates are considered to be equally likely; thus, the average amount of information is contained in the logarithm of the number of microstates, which is the same as applying Shannon’s Information Entropy to equiprobable microstates!

---

34For example, you can find several transcripts here: [http://www.eoht.info/page/Neumann-Shannon+anecdote](http://www.eoht.info/page/Neumann-Shannon+anecdote)
This means that thermodynamic entropy as derived by statistical mechanics is proportional, with the constant of proportionality being the Boltzmann constant $k_B$, to the amount of Shannon Information, $h(p)$, needed to define the aspects of the microstate that we can’t define based solely on the macroscopic parameters of classical thermodynamics.

### Different Entropy Formulations

#### INFORMATION ENTROPY

- Shannon’s Entropy: $H_S = -C \sum_i p_i \log_2 p_i$
- Hartley Entropy: $H_H = \log_b S$

#### THERMODYNAMIC ENTROPY

- Gibbs’ Entropy: $S_G = -k_B \sum_i p_i \ln p_i$
- Boltzmann Entropy: $S_B = k_B \ln N$

This leads to another significant interpretation of entropy as characterized by the second law of thermodynamics, which states that entropy cannot decrease in the universe. We can now think of that as the increasing impossibility of being able to define precisely on a macroscopic level the probabilities that determine the microstate at the microscopic level.

### 1.10.3 Demons Arise!

Determining what’s precisely happening at the microscopic level occupied some of our greatest thinkers, including James Clerk Maxwell. Maxwell specifically examined a thought experiment, or *gedanken experiment* in Einstein’s terminology, about an imaginary creature in a microscopic system that was designed to try to contradict the Second Law of Thermodynamics about increasing entropy.

The particular system he imagined was a box of gas molecules at a certain temperature, which is related to the average kinetic energy of the molecules. This means that some gas molecules will be moving faster than that average while others will be moving slower than that average. At this point, a wily demon sneaks in and places a little partition in the box, as shown in Figure 1.3.

The particles on either side of the partition will still have the same average kinetic energy or temperature. Now suppose the partition had a tiny door that was controlled by this demon: whenever the demon saw a particle approaching that was moving faster than a certain threshold, he’d open the partition and let it go to the right side of the box. Over time, all the fast moving particles would end up on the right side, which will now have a higher temperature than the left side, which contains all the particles that are moving slower than some threshold.

By forcing an ordering on the particles in the two partitions, the demon not only reduces entropy but also allows for heat to flow from a cold reservoir to a hot reservoir, thus violating all sorts of the

---

35Of course, Albert Einstein came after Maxwell and was, in fact, inspired by Maxwell’s work.
various aspects of the Second Law of Thermodynamics! In fact, you could even use the demon’s work to create a perpetual motion machine that would take advantage of the temperature differential it creates. So how can we resolve this apparent paradox?

In 1929, Hungarian physicist Leo Szilard demonstrated that Maxwell’s *gedanken experiment* did not actually violate the Second Law of Thermodynamics as the demon has to utilize energy in determining whether a particle is fast or slow. In particular, the demon had to have a way of making measurements and then storing and recalling the results. These measurements will increase the entropy by $k_B \ln 2$, which appears in the form of $k_B T \ln 2$ of heat; this increase in entropy will cancel out the entropy decrease!

Maxwell’s demon: the link between thermodynamics and information

Leon Brillouin suggested there was no difference between physical entropy and information entropy. The point of connection between physical entropy and information entropy is in the fact that, in order to obtain information about a system, it must be measured. This act of measurement, in turn, increases the physical entropy of the universe by an amount exactly equal to the amount of information that was measured. So he thought of information creating entropy but also of entropy creating information and said:

“Every physical system is incompletely defined. We only know the values of some macroscopic variables, and we are unable to specify the exact positions and velocities of all the molecules contained in a system. We only have scanty, partial information on the system, and most of the information on the detailed structure is missing. Entropy measures the lack of information; it gives us the total amount of missing information on the ultramicroscopic structure of the system.”

This connection is exemplified by Maxwell’s Demon and how it keeps track of particles: Szilard had said $k_B T \ln 2$ Joules of energy were needed for getting information about particles in a 1-particle system; Brillouin argued that changing 1 bit of information requires $k_B T \ln 2$ Joules of energy; Landauer showed that erasing 1 bit of information costs $k_B T \ln 2$ Joules of energy.

---

36 A very nice recounting of this by Philippe Jacquet is here: http://www.bibnum.education.fr/sites/default/
Chapter 1. Computational Thinking and Information Theory

The $T$ comes in because of the relationship between entropy and heat energy, as shown in Equation 1.12. This then says that you need energy in order to acquire information and this is very weird!

While Szilard showed that $k_B T \ln 2$ Joules of energy were needed for acquisition of information in a one-particle system, French physicist Leon Brillouin suggested that changing one bit of information requires $k_B T \ln 2$ energy or $k_B \ln 2$ entropy.

Others made this connection even more explicit. Rolf Landauer and Charles Bennett both looked at the connection between information entropy and thermodynamics. Landauer published a paper in 1971 in which he explained that certain computing operations, as carried out by digital computers, necessarily resulted in an increase in entropy because they were not reversible. In particular, he argued that erasing 1 bit of information requires $k_B T \ln 2$ Joules of energy. The idea that erasing one bit of information increases physical entropy by $k_B \ln 2$ and generates $k_B T \ln 2$ Joules of heat energy is also known as Landauer’s Principle or Landauer’s Limit.

1.10.4 Connecting Information Entropy and Thermodynamic Entropy

Suppose you have a system where a particle can be in one of two equally likely states, each with probability $p_i = 0.5$. Initially, the system’s Shannon Information Entropy ($H$) as seen in Equation 1.13 will be:

$$H_{initial} = \sum_i p_i \ln(p_i) = C \sum_i \frac{1}{p_i} \ln \left( \frac{1}{p_i} \right)$$

$$= C \cdot p_1 \ln \left( \frac{1}{p_1} \right) + C \cdot p_2 \ln \left( \frac{1}{p_2} \right)$$

$$= C \cdot 0.5 \cdot \ln \left( \frac{1}{0.5} \right) + C \cdot 0.5 \cdot \ln \left( \frac{1}{0.5} \right)$$

$$= \frac{1}{2} C \cdot \ln(2) + \frac{1}{2} C \cdot \ln(2)$$

$$\Rightarrow H_{initial} = C \cdot \ln(2)$$

We use ln instead of log for convenience since we can convert from one base to any other base by multiplying by an appropriate constant.

Following Shannon’s formulation [11], if we set $C = \log_2(e)$, we end up with Shannon’s Information Entropy of:

$$H_{initial} = \log_2(2) \Rightarrow 1 \text{ bit of Shannon Information}$$

This means the system can, in its initial state, store 1 bit of Shannon Information.

We also saw that Shannon’s Information Entropy is exactly the same as Gibbs’ Entropy with $C = k_B$, as shown in Equation 1.13. So if we set $C = k_B$ in Equation 1.14 above, we end up with Boltzmann’s Thermodynamic Entropy ($S_{initial}$) of:

$$S_{initial} = k_B \cdot \ln(2)$$

---

37 Shannon just set $C = 1$ but we also add in a change of base from base-$e$ to base-2 using a change of base: $\ln(2) = \log_e(2) = \log_2(2) = \ln(2) \cdot \log_2(e)$.
Suppose we check the system after some time and find that the particle is actually in the first state and so it is not in the other state. That means $p_1 = 1$ and $p_2 = 0$. The Shannon Information Entropy for the final state is:

\[
H_{\text{final}} = C \cdot p_1 \ln \left( \frac{1}{p_1} \right) + C \cdot p_2 \ln \left( \frac{1}{p_2} \right) \\
= C \cdot 1 \cdot \ln \left( \frac{1}{1} \right) + C \cdot 0 \cdot \ln \left( \frac{1}{0} \right) \\
= C \cdot 1 \cdot \ln(1) = C \cdot 1 \cdot 0
\]

\[\Rightarrow H_{\text{final}} = 0\]

The final state of this system can now store only 0 bits of Shannon Information. This means that the original 1 bit of Shannon Information has been deleted. The change in any variable is calculated as $\Delta \text{Variable} = \text{Variable}_{\text{final}} - \text{Variable}_{\text{initial}}$. So the change in entropies, both Shannon and Boltzmann, for deleting 1 bit of Shannon Information are:

\[
\Delta H = H_{\text{final}} - H_{\text{initial}} = 0 - \log_2(2) = -\log_2(2) \Rightarrow \text{Shannon Information Entropy}
\]

\[
\Delta S = S_{\text{final}} - S_{\text{initial}} = 0 - k_B \ln(2) = -k_B \ln(2) \Rightarrow \text{Boltzmann Physical Entropy}
\]

Using Equation 1.9, we see that the amount of heat energy required to delete 1 bit of Shannon Information is:

\[
\Delta S = \frac{\Delta Q}{T} \\
\Rightarrow \Delta Q = \Delta S \cdot T \Rightarrow \text{Amount of Heat released upon deletion of 1 bit of Shannon Information}^{38}
\]

Thus, any irreversible logical transformation of classical information will require at least $k_B \cdot T \ln(2)$ of heat per bit of Shannon Information; alternatively, the deletion of one bit of Shannon Information will result in heat production of $k_B \cdot T \ln(2)$ Joules, where $T$ is the temperature of the storage medium, and increase in physical entropy of $k_B \ln(2)$. This result is known as Landauer’s Principle or Landauer’s Limit of energy needed to delete one bit of information and firmly makes the connection between information entropy and physical entropy.\(^{39}\)

---

\(^{38}\)The negative $\Delta Q$ indicates this is exothermic, so it releases that amount of heat when 1 bit of Shannon Information is deleted. This was calculated from the perspective of the $Q_{\text{reservoir}}$ rather than $Q_{\text{system}}$. We also used the equal part of the inequality in Equation 1.9 as we can equivalently look at the $Q_{\text{reservoir}}$ and $Q_{\text{res}} = T \Delta S_{\text{res}} \geq -T \Delta S_{\text{res}}$ and $\Delta S = -k_B \ln(2)$. From that perspective, the $\Delta Q$ is exothermic and deletion of 1 bit of Shannon Information gives off that amount of heat. Also, $Q_{\text{res}}$, the heat produced in the reservoir, is calculated in the quasistatic limit, which means a long cycle duration.

\(^{39}\)In fact, Landauer went on to show in 1996 that information itself is physical since it has to be stored in physical systems and information itself, stored in units of bits, has to obey the laws of physics and the erasure of information requires some minimum heat dissipation.
Chapter 1. Computational Thinking and Information Theory

1.10.5 The Universe Itself Is Informational

The implications of these connections is fundamentally staggering. In Section 0.7, we discovered that what was knowable in the universe is, at least so far, also computable. In Section 1.10, we have seen that what is doable in the universe, what the universe does at a most fundamental level, is computable, as well!

We saw that being *computable* meant that something was able to be carried out by a Universal Turing Machine. A Turing machine consists of certain processes, or *effective procedures*, in mathematics that could be carried out by following a set of rules. We looked at this as being something equivalent to:

\[
\text{Computation} = \text{Data} + \text{Instructions to Transform that Data} \quad (1.18)
\]

The *data* are representations of fundamental *observations* and transforming them gives the data a *context*, which results in *information*. Information, we found, is a property of a *message* that is intended to be communicated between a system made of a sender and a receiver. It is an abstract idea that is encoded into some carrier and that reflects some *measure of uncertainty* about the system. We found the fundamental unit of information, the *bit*, can be thought of as the atom of communication.

These bits of information, it turns out, are related to physical entropy in thermodynamics and statistical mechanics. *Entropy*, in the physical sense, can be thought of as the *ability to exchange work and heat energy* and is something that increases in spontaneous transformations, as encapsulated in the Second Law of Thermodynamics. *Energy* is the ability to do work. *Work*, in turn, can be thought of as a Force times a Displacement of some sort; e.g., in mechanical work, it’s mechanical force times mechanical displacement.

This idea of entropy not only designates the so-called arrow of time but, as physicists like Stephen Hawking and Jacob Bekenstein discovered, the entropy associated with a black hole can be thought to be contained within an imaginary spherical shell around that black hole. As the black hole’s entropy increases, they found it did something strange: instead of increasing with the 3-d volume of the sphere, it increases with the black hole’s 2-d surface area!

By studying the entropy, or information content of black holes, we’ve learned something about the information content of the universe itself. There is a finite amount of information that you need to describe everything that is going on in the universe.

We know how much information can fit inside a black hole and we’ve learned how much information a black hole carries. It turns out that information cannot get lost. We also know we can convert everything in the universe into a black hole and it would be finite since we could compress all information in the universe into that black hole. This information would record every single thing about the universe: information about every single microstate of every single system in the universe.

As physicist Raphael Bousso found, this number is about \(10^{123}\) bits. This is the bound on information in the universe. In fact, following the holographic principle, the entire universe can be seen as two-dimensional information on the cosmological horizon!

Since entropy is so fundamental in the universe, it turns out that the universe itself is informational at its most elementary level. As we saw in the previous chapter, we manipulate this information in our computational solutions but we only deal with finite problems that are computable in the physical universe (in physics but not all of mathematics, in general).

So in a very real sense, we can then think of the universe itself as being computational! Instead of computation for Turing machines as we saw in Equation 1.18, we can think of computation in the
Figure 1.4: Data to Information to Knowledge: Some essential ideas in the transformation of data to information and knowledge.

The physical universe is made of particles which have some state. As renowned physicist Leonard Susskind noted, the laws of motion are just the rules for updating the states of particles when you express the state of the system in terms of bits, usually in proportion to their corresponding degrees of freedom. You can then express the complete dynamical information of a system solely in terms of binary bits.\footnote{We can represent the dynamical state of any system by some point in a phase space whose dimension is two times the number of degrees of freedom it has. A \textit{phase space} usually consists of the generalized momentum versus the generalized position of all the particles in the system. It contains all the dynamical information about the system and is usually expressed via the Hamiltonian function, which determines the trajectory of the point in phase space and thus encapsulates the complete dynamics of the system. The phase space is then divided into hypercubes where a representative phase space point can only occupy a single cell.}
The brilliant physicist John Archibald Wheeler, who was also Richard Feynman’s doctoral advisor, had a penchant for coming up with pithy quotes that keenly capture deep insights about the universe. Not only did he coin descriptive terms like “black hole” and “wormhole” but he’s famous for capturing the essence of general relativity succinctly in his saying, “Spacetime tells matter how to move; matter tells spacetime how to curve.”

Similarly, Wheeler summarized the hints of emerging fields like Digital Physics by saying, “it from bit”, which suggests that all material, physical existence actually has an information-theoretic underpinning and that information is the essential aspect of the universe. In other words, “the universe is made of information; matter and energy are only incidental.” He even went further to postulate, 41 “Every it–every particle, every field of force, even the spacetime continuum itself–derives its function, its meaning, its very existence entirely–even if in some contexts indirectly–from the apparatus-elicited answers to yes-or-no questions, binary choices, bits.”

Shannon, in a sense, told us that bits could be decoupled from their physical implementation when he developed Shannon Information and Shannon Information Entropy. When Szilard, Landauer, et al., looked at the physical limits of number crunching, they found that computing can be thermodynamically reversible and the physics had to be put back in. Interestingly, some of the biggest names in physics and computing, including such luminaries as Richard Feynman, John Wheeler, Freeman Dyson, Konrad Zuse, Rolf Landauer, Paul Benioff, etc., met at the Endicott House in 1981 for the first Physics of Computation Conference and worked on such ideas as the physical limits of computation.

In fact, some of the questions that came out of it prompted computer scientists to look at physics: “How carefully should computer scientists listen to nature if they want to use its resources for computing?” as well as prompting physicists to look at computation: “And can physicists gain any significant insights by looking at physical processes as a kind of ongoing computation?” Feynman’s keynote, Simulating Physics with Computers, even proposed a new device based on quantum mechanics, a quantum computer. But his perspective on computation, as expressed near the end of his keynote speech, might best summarize the need for computational thinking, as well:

“...[T]he discovery of computers and the thinking about computers has turned out to be extremely useful in many branches of human reasoning, ... And all I was doing was hoping that the computer-type of thinking would give us some new ideas.”

41The first quote is from Wheeler’s famous paper “Information, Physics, Quantum: The Search for Links.” Please see the following Scientific American article for more of the quotes: https://www.scientificamerican.com/article/pioneering-physicist-john-wheeler-dies/
2. Computational Problem Solving

When I am working on a problem I never think about beauty. I only think about how to solve the problem. But when I have finished, if the solution is not beautiful, I know it is wrong.
– R. Buckminster Fuller

2.1 What Is a Model?

Our adventure started off by trying to predict whether or not we would need a jacket when we rushed off to campus. If you recall, we even went so far as recruiting our friends to embed sensors all over campus so we could stay comfy and cozy if we happened to venture out into the cold.

We then recorded temperature values and GPS locations at each of those sensors. As you start to examine the patterns and trends in the data, you might wonder if there are other values you should record from these sensors.

For example, should you keep track of the colour of the sensors? Some sensors might be yellow while others might be black. Knowing a little bit about black-body radiation, you know that a black-body is a perfect emitter and absorber of light. That means the ambient temperature around the black sensors will be ever-so-slightly higher than the ambient temperature around the yellow sensors. Should we account for this?

---

1At the risk of being presumptuous, I might have suggested changing the wrong to incomplete. But you can see the original quote here: https://simple.wikiquote.org/wiki/Richard_Buckminster_Fuller

2This is why people in the desert can get away with wearing black clothes; the black clothes will absorb all the incident radiation on the outside part and the heat radiated from the body on the inside part. If there are sufficient layers between the outer part and the body, the inner black layers can absorb your body heat and radiate it away to the wind so you can stay nice and cool in a very hot environment. In addition, thick black clothing is also slightly better at blocking UV radiation.
I don’t know, it depends. If the problem you’re tackling demands that level of *precision*, you’ll have to account for it in your *model*. If the problem, on the other hand, is only concerned with nice, round temperature values, you can safely ignore any contributions to the ambient temperature by the darker coloured sensors.

### Accuracy vs Precision

Let’s say you’re checking to see if a group of possible values matches some specific value. **Accuracy** is a measure of how close those values are to the actual value. So if the group of values you measure is \{10.0, 11.0, 11.1\} and the actual, correct value is 10.003, these values are pretty close to the actual value and are relatively accurate but don’t contain the correct number of significant figures.

If, on the other hand, the group of values were \{78.101, 78.010, 78.111\}, the values contain the correct number of significant figures and are all very close to each other but none of them are really close to the actual value of 10.003. These values would be **precise** but not accurate. Please see Figure 2.1 for the various permutations.

![Figure 2.1: Accuracy vs Precision](image)

---

### 2.1.1 Models and Abstractions

This process of not modeling certain details, like the colour of sensors, is called **abstraction**. Abstraction, in one sense, can be thought of as ignoring details that you deem to be unessential.

Suppose your friend, who is new to your campus, asks how to get from the Science Building to the Dining Hall. You might draw them a map which shows the Science Building and the Dining Hall. What else should you include on the map? Should you include the trees and grass? Probably not... but how about the walkways and roads? Depending on whether your friend decides to walk or drive, that would certainly be pertinent.

So in the final map you draw, you will decide to ignore certain details about the real campus and only include those details you deem to be relevant for the problem at hand. This approach of ignoring the non-essential details is the essence of *abstraction* and the map that you make would be the **model** you create of the real-world campus.
“All models are wrong, but some are useful.”

That famous quote, attributed to the statistician George E.P. Box, encapsulates the idea of abstraction. This is a variation of the saying by the mathematician Norbert Wiener, “The best material model of a cat is another, or preferably the same, cat.”

Both of these quotes point to the idea that a model, whether mathematical or computational or graphical or physical, is an abstraction of some real-world entity or phenomenon. Since a full examination of a real-world entity or phenomenon is usually too complex for us to describe and understand to the highest levels of granularity, we usually settle for solving an easier version of the problem in a first approximation.

As such, we place constraints on our model and abstract out non-essential details resulting in a simpler, but easier to solve and understand, model. But it’s important to remember that all of these models are abstractions and not the actual underlying entity or phenomenon they represent. The model might be useful but it is imperative not to confuse the model for the actual thing.

As we create models and abstract out certain details, we place constraints on our model. This constrained model is usually much simpler than the real-world underlying entity or phenomenon, at least in the first approximation. As our initial model becomes increasingly validated, and our confidence in the underlying theory increases, we can start to relax some of those constraints and make the model more complex.

For example, in our temperature sensor model, we might start by only recording the temperature values and the GPS locations. Once we find this first-approximation model is nominally good at discerning a trend or pattern in the data, we can start to relax some of the constraints and make a more complex and possibly more precise model. So we could relax the constraint of ignoring the colour of the sensors, for instance. In addition, we might want to start recording the height of the sensors to see if the temperature variations with altitude make a difference or give us a better idea of the temperature distribution or (scalar) field.
The heart of the scientific method is still beating...

As we start to relax more and more of the constraints, the model gets more complex but the problem it solves starts to approach the real-world, underlying entity or phenomenon. Our incremental solutions start to incrementally solve this more complex problem by building on each smaller success. This incremental improvement in our knowledge and models is the heart of the scientific method!

The idea of incremental progress is captured well in a (hopefully funny but strikingly realistic) joke about a trapper who asks a physicist how best to trap a bear. The physicist, after a moment’s reflection, responds, “Consider a spherical bear, in simple harmonic motion...” This is (perhaps) funny because a bear obviously is not spherical and doesn’t exhibit simple harmonic motion. But, since we know how spherical objects behave in simple harmonic motion quite well, the physicist decides to model the problem similarly as a first approximation. One presumes the bear lived a long, healthy life while the physicist slowly refined the model to the necessary approximation for creating an effective bear trap.

2.2 Data Representations

A computational or mathematical model is often represented by a function of some sort. These models usually have associated inputs and outputs which are encoded in some specific representation. For example, if we’re trying to calculate the average temperature across all sensors, we might find it easier to do that calculation using a decimal representation than, say, a binary representation or, even more masochistically, a Roman numeral representation.

The choice of data representation is often very important as it will dictate the particular algorithm we use to solve the problem. If I represent the temperature values in Roman numerals, I might have to do quite a few machinations in order to do simple addition.

**Algorithm 2.1:** Algorithm for adding $29 + 14 = 43$ using Roman numerals:

$\Rightarrow XXIX + XIV = XLIII$

1. Convert any subtractive (e.g., IV) to its un-compacted version (i.e., IIII)
   $\Rightarrow XXIII + XIII$
2. Concatenate the two values together
   $\Rightarrow XXIII + XIII$
3. Sort the symbols in order from left-to-right with the largest-value symbols on the left
   $\Rightarrow XXXIX + VIII$
4. Starting with the right end, combine groups of the same symbols that can make a larger-value and move the single larger symbol to the left
   $\Rightarrow XXIII$
5. Add subtractives back in where possible
   $\Rightarrow XLIII$
2.2 Data Representations

For example, an algorithm for doing addition using a Roman numeral representation of the data is shown in Algorithm 2.1. As you can see, this is a much more complex undertaking than simply saying $29 + 14 = 43$ when we represent the data in the Hindu-Arabic positional numeric system instead of the Roman numeral system.

Figure 2.2: Values, Data, and Data Representation: Values, when given a certain context, become data. That context is the data representation, or structure, imposed upon the raw facts.

2.2.1 Data Structures

Just as we saw in Section 1.5.1, Niklaus Wirth said,

Data Structures + Algorithms = Programs

(2.1)

We reinterpreted this as:

Representations of Data + Representations of Algorithmic Procedures = Computational Solutions

(2.2)

Thus, the choice of data representation for a certain problem is intimately connected to the choice of algorithm for solving that problem. As a result, computer scientists spend a great deal of time studying the properties and consequences of different data representations, or data structures, as we’ll see in a little bit.
From Observations to Data...
As we discussed in Section 0.3, observations can be thought of as recording the manifestations of the universe and when these observations are given a structure, these representations can be thought of as data. Further giving the data a context transforms it into information.

2.3 Number Representations

Since Shannon’s Information is based on the binary representation, let’s explore that number representation in some detail. Let’s start by looking at the ordinary decimal system we use daily. The decimal number system we use is the Hindu-Arabic positional numeric system rather than the older Roman number system. A number is made up of one or more digits and each digit can be from 0–9.

2.3.1 Positional Number Representations

The decimal number system uses positional math where the position of a digit determines the value of the digit in the number, as seen in Figure 2.3. This is very different from the Roman numeral system where each digit has the same value regardless of its position, as we saw in Algorithm 2.1.

3 As mentioned in [16]:

The key to the interaction between abstract and physical entities in physics is via the representation relation. This is the method by which physical systems are given abstract descriptions: an atom is represented as a wave function, a billiard ball as a point in phase space, a black hole as a metric tensor and so on. That this relation is possible is a prerequisite for physics: without a way of describing objects abstractly, we cannot do science. We have given examples of mathematical representation, but this is not necessary: it can be any abstract description of an object, logical, mathematical or linguistic. Which type of representation has an impact on what sort of physics is possible: if we have a linguistic representation of object weight that is simply “heavy” or “light”, then we are able to do much less precise physics than if we use a numerical amount of newtons.

4In a sense, you can think of this as when, say, electrons manifest from the electron field. In that case, you can think of the state of the electron as the data and the processing or transformation of the data being carried out by the particles. In this conception, physics can be thought of as the computational processing and transformation of information.
In the base-10 decimal system, the right-most digit, the 1st digit from the right, of any number has a value that’s a multiple of \(10^0 = 1\); the 2nd digit from the right has a value that’s a multiple of \(10^1 = 10\); the 3rd digit from the right has a value that’s a multiple of \(10^2 = 100\); etc.

For example, in the number 321, the 1 is in the 1’s place, the 2 is in the 10’s place, and the 3 is in the 100’s place. When you add the values of all three digits, you end up with the number three hundred twenty-one: 321.

### 2.3.2 Computational Representations

When we looked at the theoretical Universal Turing Machines (UTM) as an idealized computing agent and mathematical model of computation in Section 0.7.1, we realized that a UTM needed some sort of alphabet, or set of symbols.\(^5\)

This input alphabet can be any set of symbols like \(\{a, b, c\}\) or \(\{↑, ↓\}\) or \(\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}\). As we saw in Sections 1.9.3 and 1.9.4, it turns out that a binary representation, \(\{0, 1\}\), is sufficient to encode any message at all, as pointed out in Section 1.9.5.\(^6\)

You can use this binary system to represent any number and, in fact, encode any data. So if we use a binary system with 1’s and 0’s, we could represent the number 13 using those 1’s and 0’s as the following string in that base-2 number system: 000101.

Here, just like in the decimal representation, the right-most digit, the 1st digit from the right, 1, has a value that’s a multiple of \(2^0 = 1\) and is 1; the 2nd digit from the right, 0, has a value that’s a multiple of \(2^1 = 2\) and is 0; the 3rd digit from the right, 1, has a value that’s a multiple of \(2^2 = 4\) and is 4; the 4th digit from the right, 1, has a value that’s a multiple of \(2^3 = 8\) and is 8. All the other values in this number are 0 so when we add up the values we calculated above, we get \(1 + 0 + 4 + 8 = 13\).

We could represent our binary system with \(\{0, 1\}\) but we could just as well use any set of symbols since 1’s and 0’s are also just symbols. So we could represent our binary, base-2 number system with the symbols \(\uparrow\) and \(\downarrow\) instead, as in \(\{↑, ↓\}\)!. We would then represent the number 13 by the following string of up- and down-arrows: \(↓↓↓↓↑↑↑\).

So depending on the computing agent, we can use any numeric representation that would make the task of computation more efficient. For example, base-10 numbers work great for human computers who happen to have 10 fingers.

#### Problem 2.1

Some people posit that we use a decimal (base-10) positional system because we have 10 fingers. Suppose you lived instead in the world of The Simpsons, where people only have 8 fingers, rather than 10, as shown in Figure 2.4. What base would they have likely evolved to use? Can you represent the base-10 decimal number 8,357 in a base-Simpsons representation?

### 2.3.3 Physical Representations

But the digital computers that underly most of the hardware we use today rely on encoding all data as binary digits, in base-2. This works well because, at their core, these digital computers use transistors...
Hardware, Software, and Firmware

**Hardware** normally refers to a digital computer which will be the physical computing agent used in our computational solution. The desktop on your desk (for us old fogeys, at least), the notebook on your lap, or the Smartphone/Snapchat/Instagram device that serves as your hub to connect to the world would all be considered hardware. Any of the physical, electronic devices that surround you would fall under this category. Almost all of these electronic devices use Integrated Circuits (IC) that contain billions and billions of transistors and gates to carry out the actual computation. These general ICs are used in components like the Random Access Memory (RAM) units and the microprocessors in the Central Processing Unit (CPU).

**Software**, on the other hand, is the ordered set of instructions that are realizable by the specific hardware computing agent. This is exactly what we called programs or source code or code in Section 0.7. Software is often stored on some digital medium like a flash drive and is loaded into the physical computer’s RAM so that it can be executed by the operating system that controls the CPU on that machine. These programs can be written by anyone, including the operator of the computer, and can be changed and re-loaded and re-executed as many times as desired. Data is also stored on the same digital medium as these software instructions in the von Neumann computer architecture which we’ll be discussing in just a bit.

**Firmware** is semi-permanent software that is not intended to change often as it is software that is embedded in hardware directly at manufacturing time, rather than loaded by a user, for example. The contents of the RAM memory are lost whenever a computer loses power, resulting in a loss of any software that has been loaded into the RAM. Firmware, on the other hand, maintains its state when hardware loses power since it is burned, or flashed, directly on the hardware in Read-Only Memory (ROM). A Field-Programmable Gate Array (FPGA) is similar in that it can be programmed after it has been manufactured and allows users to create their own digital circuits. You can create these circuit designs using some Hardware Description Language (HDL) like Verilog or VHDL.

and gates that deal with high and low voltages which we can then label as 1’s and 0’s to simulate the binary values in computational models.
2.4 Digital Representations

Some computer scientists study hardware, the physical systems that underly the digital machines that surround us. The fundamental processing element of the Integrated Circuits (IC) in digital electronic computers is the logic gate, sometimes simply called the gate. A gate is an electronic device that takes one or more binary inputs and produces a single binary output, like Figure 2.6b.

These logic gates are usually made using switches. A switch is exactly what you think of intuitively: it’s an on/off device that allows current to flow through the circuit when it’s closed and that stops all current flow when it’s open, as shown in Figure 2.5 where we can see the schematic of a mechanical switch. Multiple switches are combined to make devices like AND gates, OR gates, XOR gates, etc. Examples of two such gates are shown in Figure 2.6.

The logic gates in Figure 2.6 are shown with the switch closed or open. The open state is denoted by 0 and the closed state by 1. The AND gate shown in Figure 2.6a outputs a 1 only if both inputs are 1. The OR gate shown in Figure 2.6b outputs a 1 if at least one input is 1.

There have been many kinds of switches in the past: electromechanical relays, vacuum tubes, gas tubes, etc. Most modern logic gates are built using tiny electronic transistor switches. A transistor is a switch with no moving parts, a so-called “solid-state switch”, that is composed of different semiconductors in the Complementary Metal-Oxide-Semiconductor (CMOS) approach used to design most digital circuit boards. Modern ICs contain billions of transistors on a single chip and are able to do trillions of operations per second. These ICs are also sometimes called chips or microchips and

---

(a) AND gate: flow of electricity only if both switches are closed
(b) OR gate: flow of electricity if at least one switch is closed
multiple chips are connected together on a circuit board.

The number of transistors on a single chip, and hence the computing power of these microprocessors, have roughly doubled approximately every two years or so since 1965. This trend is sometimes called Moore’s Law but it should more properly be called Moore’s Observation as it’s not a physical law at all. In fact, we’re coming close to the limits of this observed trend as we reach the physical limits of computation.\(^9\)

---

**Boolean vs Binary: Round 1!**

Nota bene: Boolean numbers are **not** the same as binary numbers! A binary number can contain any number of binary digits whereas a Boolean number can only contain a single binary digit.

---

The combination of multiple gates allows a physical implementation of all the binary mathematics that’s possible for binary values. The 0 and 1 values of a binary number are represented by the physical position of the on/off switches. This allows us to physically implement all the algorithms that can be described by binary mathematics. In just a bit, we’ll meet Boolean Algebra, the branch of mathematics that deals with binary values. Thus, the logic gates in ICs are able to physically implement all aspects of Boolean Algebra!

### 2.5 Boolean Algebra

George Boole (1815 – 1864) was a deeply religious man who had a mystic experience that motivated him to think about how the mind processes thought. He published *An Investigation of the Laws of Thought* in which he dealt with the basis of Aristotelean Logic and Probabilities by applying symbolic algebra to formal logic. In the introduction, he alluded to these lofty goals,

*The design of the following treatise is to investigate the fundamental laws of those operations of the mind by which reasoning is performed; to give expression to them in the symbolical language of a Calculus, and upon this foundation to establish the science of Logic ... and, finally, to collect ... some probable intimations concerning the nature and constitution of the human mind.*\(^10\)

Since Aristotelean logic was rooted in syllogisms, which we met in Sections 1.2 and 1.4, Boole decided to tackle such logical constructs using an objective mathematical structure, similar to the framework Descartes had laid for Euclidean geometry by representing geometric ideas as algebraic formulae in which you could manipulate symbols using formal rules.\(^11\) To follow the example syllogism we gave earlier, Boole took the beginning of it, “All men are mortal”, and replaced the nouns with variables and the modifiers and verbs with arithmetic operators. These variables, which came to be called Boolean variables, could only have two possible values, two binary values: **TRUE** or **FALSE**.

---

\(^9\) Richard Feynman famously addressed this in his 1959 talk entitled, There’s Plenty of Room at the Bottom, which also inspired the nascent field of nanotechnology. New approaches in physics and biology are looking for ways past the quantum blocks of further miniaturization.

\(^10\) A very nice summary of these connections is here: https://a16z.com/2017/03/21/logic-philosophy-computer-science-dixon-atlantic/ The one thing I might add to that is the contributions of Leibniz to formal logic and Boolean representations.

\(^11\) This is also similar to what Shannon did for Information by stripping out all semantic content and giving it a mathematical underpinning.
2.5 Boolean Algebra

Boole was able to develop a set of variables, operators, and rules for transforming those values that could be applied to the principles of logic and reasoning. We can then express these Boolean operators, like AND, OR, and NOT, using truth tables that use such Boolean variables, variables whose state can only have one of two possible values.

In this way, he derived an entire mathematical framework of logic that relied on only two kinds of values: TRUE and FALSE. He applied these formal laws of algebra to the principles of logic to create the field of Boolean Algebra. Since we can use any two symbols for these two base values, as we saw earlier, we can use symbols like HIGH and LOW or 1’s and 0’s; if we use the two binary numbers, \{0, 1\}, as our two base values, we end up with the mathematics of binary values.

This is exactly what inspired people like Tukey, Hartley, and Shannon, who realized this same Boolean Algebra could be used to analyze the relays, the on/off mechanical electrical switches, used in telephone switching circuits. Boolean Algebra can be used to describe the output state of not just relay-based systems but also the transistor-based ICs used in modern digital computers!

2.5.1 Boolean Algebra and Digital Circuits

The connection between Boolean Algebra and these digital circuits is also rooted in history. When Descartes first defined the formal rules for manipulating symbols representing geometric ideas, it allowed people to reason about abstract ideas without being constrained by their own spatial intuitions and language. This change in language or representation allowed people like Isaac Newton and Gottfried Wilhelm Leibniz to independently discover the calculus in the 17th century.

Leibniz went even further and postulated the development of a new language, called Characteristica Universalis, that could represent all possible mathematical and scientific knowledge. He also imagined a machine, called the Calculus Ratiocinator, that could use that language to perform logical deductions and settle any philosophical dispute by calculation.

Alan Turing was inspired by ideas like Leibniz’s universal characteristics language and logic calculating machines when he started thinking about his mathematical models of computation, the so-called Universal Turing Machines we met in Section 0.7.1. In particular, he was looking at them in the context of the Entscheidungsproblem (“Decision Problem”) as formulated by David Hilbert in 1928.

In the Entscheidungsproblem, Hilbert posed a challenge that asked if an algorithm existed which could determine whether an arbitrary mathematical statement is true or false. This was part of his larger program to formalize all of mathematics and required:

1. Formulation: Is mathematics complete? All mathematical statements should be written in a precise formal language with well-defined rules
2. Completeness: Is mathematics consistent? There should be a proof that all true mathematical statements can be proved in the formalism (along with Consistency and Conservation)
3. Decidability: Is every statement in mathematics decidable? There should be a clearly formulated procedure that can definitively establish within a finite time (an algorithm!) the truth or falsehood of any mathematical statement (based on the given axioms)

That last item is the “Decision Problem” and it captured the attention of many famous mathematicians including Alan Turing, Alonzo Church, and Kurt Gödel. In fact, as we saw in Section 0.7.1, Gödel’s First Incompleteness Theorem proved that, for any consistent system with a computable set of axioms that is capable of expressing arithmetic, it’s possible to construct a proposition that cannot be proved from the given axioms (and neither can its negation). This means that any consistent logical system that encompasses arithmetic necessarily also contains propositions that are true but can’t be
proven to be so!

The bad news didn’t stop there for Hilbert’s program; as seen in Section 0.7.1, both Turing and Church showed the answer to the “Decision Problem” was “No,” although Church did it using the lambda calculus. Turing proved this result in his 1936 paper entitled, “On Computable Numbers, With an Application to the Entscheidungsproblem,” in which he created a mathematical model of an all-purpose computing machine and showed that no such machine could determine whether or not a given proposed conclusion follows from given premises using a formal logical system.\(^\text{12}\)

While Turing showed the connection between logic and computing, Claude Shannon was able to show the connection between electronics, logic, and computing, as well! Shannon’s 1938 paper, “A Symbolic Analysis of Switching and Relay Circuits,” showed how Boole’s system of formal logic, a Boolean algebra consisting of True/False values only, could describe the operation of two-valued electrical switching circuits. Remember that Boolean logic, or Boolean algebra, is a form of propositional logic and relied upon just three fundamental operators (AND, OR, and NOT) to perform logic functions.

Shannon first showed that this Boolean algebra, and binary arithmetic, could be used to specify the design of logic circuits using electromechanical relays. He then further showed that arrangements of these relays, in turn, could be used to also solve Boolean algebra problems! This correspondence between the “symbolic study of logic” (Boolean algebra) and the relays and switches in the circuit is shown in Figure 2.7. This meant that any circuit can be represented by a set of equations based in Boolean algebra. This was the first time the logical operation, or layer, of a computing machine was separated from its physical operation, or layer.

As we saw in Section 2.4, the fundamental element of integrated circuits is the logic gate, which is based on an electronic switch (rather than the electromechanical switches and relays of Shannon’s time). In fact, today’s computer circuits are built upon such electronic gates that operate according to Boolean algebra to determine the value of the output signal. That output value is usually designated as a 1 or 0. These gates can also come together to store information in storage units called flip-flops.

\(^{12}\)On an interesting side-note, the so-called Universal Turing Machines showed that the machine (the hardware), the instructions (the program), and the input (the data), were all combined in a single entity and further implied that any computing logic that could be expressed in hardware could also be expressed in software. But putting both the data and programs in the same architecture later came to be known as the von Neumann architecture, as we saw in Section 2.3.3.
2.5 Boolean Algebra

Although there are a few basic gates that correspond to the Boolean algebra operators (AND, OR, and NOT), compound gates like the NAND (NOT AND) gate are actually sufficient to build any computer circuit, including ones for arithmetic, memory, and executing instructions.

An example of both a NAND gate and an adder circuit that does arithmetic addition is shown in Figure 2.8. Modern computers usually have an Arithmetic Logic Unit (ALU), a key component of the Central Processing Unit (CPU), that consists of many such adder circuits strung together in order to carry out arbitrarily complex arithmetical operations. In fact, modern computer circuits can contain millions, some even billions, of these simple transistor gates.

2.5.2 Digital Logic Circuits

These simple transistor gates form complex digital logic circuits which come in two varieties: combinational logic circuits and sequential logic circuits. Combinational logic circuits are simple Boolean circuits whose output signals are a function of only the present input signals.

Sequential logic circuits, on the other hand, utilize flip-flops as their memory elements and use them to affect their final output, as well. Thus, sequential logic circuits depend on both the current input as well as the information from the history of past inputs that are stored in the flip-flops. In a way, we can think of the information stored in these memory elements as the state of the digital circuit.

Most complex digital logic circuits contain both combinational logic circuits as well as sequential logic circuits and use them to perform Boolean algebra on both the inputs as well as the stored data.

2.5.3 Theory of Computation

The relationship between Boolean algebra and digital logic circuits is extended in a model of computation known as Finite State Machines or Finite State Automata (FSA). These FSAs are abstract, mathematical models that are used for designing both digital logic circuits and algorithms. An FSA come in two flavours: a Deterministic Finite State Automaton (DFA) and a Non-Deterministic Finite State Automaton (NFA).

The states that are modeled in FSAs depend upon the system that’s being examined. In essence, a state can be any well-defined condition that characterizes the system at some point in time. When dealing with abstract mathematical automata, we can delineate a finite set of states that are valid conditions for an arbitrary, abstract system; each of these states can be represented either as a separate function or as the set of values for one or more variables associated with the system.

Thus, a DFA is defined by a finite set of states, which are used to determine both the initial state, as well as the final output, or accepted, set of states of the system represented by the DFA. The DFA
Chapter 2. Computational Problem Solving

is controlled by an input set of symbols, called the alphabet, and also has a transition function that determines the next state based on both the current state and the input symbol. Although the DFA can only transition to a single, unique state each time it reads an input symbol, the NFA can transition to multiple states at once and does not require an input symbol to make a state transition. All NFAs can be converted back to a DFA, albeit with more states (sometimes, exponentially more states).

DFAs are also used in formal language theory where they help determine a valid language given the rules of the language, or grammar. In this context, we can think of a language as being made of some set of strings. The strings, in turn, are composed of some sequence of symbols that come from an alphabet, a set of symbols defined for that language. These strings are combined via some grammar for that language which consists of both a lexicon and a set of rules for creating valid sentences. The rules of the grammar describe the allowed ways to construct strings and valid strings are called words in that language. The lexicon, or vocabulary, of the language is made up of a set of pairs in which each pair has a word and a category; the category, in this case, is like the part of speech (e.g., noun phrase, verb phrase, adjective, etc.).

There are many different kinds of grammars and there is a corresponding kind of FSA for each of them. For example, a context-free grammar, one in which all the production rules only have a single, non-terminal symbol on the left hand side of each production, is recognized by a PushDown Automata (PDA). The set of all strings accepted by some DFA defines a regular language.

In fact, a Turing Machine can be considered to be a generalization of the DFA such that it includes a set of tape symbols, including the empty symbol, and it can also change the symbol on the tape as part of its transitions, in effect gaining a memory. A PDA is basically a FSA with a memory but it is less powerful than the general-purpose Turing Machine as it cannot simulate an infinite tape. In fact, because a Turing Machine helps define what is computable, an FSA and its various derivative automata can therefore also model computation in general.

In particular, since computation itself is a process, which can be modeled as a function as we saw in Sections 0.5 and 1.4, we can characterize any model of computation as the process by which some set of inputs are transformed or mapped into some set of outputs using a particular algorithm. The Theory of Computation deals with such models of computation and how they can be used to compute some set of outputs efficiently. Thus, the theory of Computation deals with the theory of Computability (what is computable in the universe and what is uncomputable?), the theory of Computational Complexity (if we can compute something, can we do so efficiently?), and the theory of Automata and Language (what is the model of computation and how can it be used to solve problems using a given algorithm?).

2.6 What Is Information Processing?

Regardless of the particular model of computation or the data representation, the input and output contain a certain amount of “information”, where I’ve put it in quotes to indicate a more generic meaning than Shannon’s Information. We can then think of the function or procedure more generically manipulating the information in the input rather than the components of the specific representation used for that input.

When we do this, we can think of computation as the pure transformation of information where the function or procedure processes the information from one form into another. In fact, this is the

---

13 Since we can encode any discrete message into binary digits with no loss of information, as shown in Section 1.9.5, we can transform data without loss of information from one representation to another as long as the context, the probability of the symbols, stays the same.
2.7 What Is Computer Information Systems (CIS)?

This difference in emphasis naturally leads to establishing different sub-fields within computation. Two of the main divisions could be said to be between Computer Science (CS) and Computer Information Systems (CIS). We could then, in a sense, claim that Computer Science is mainly concerned with the processes that facilitate the transformations whereas Computer Information Systems is concerned more with the flow and management of data or information in the system.

CS vs CIS

More generally, we might say that Computer Science deals with data representation and transformation in computational solutions while Computer Information Systems deal with data curation, data flow, and information management in a computational system.

What exactly would constitute an information system? It would be any system that:

- delineates the conditions needed for data collection and curation
- helps enable the transformation of Data → Information → Knowledge
- has the ability to effectively manage both data and information

These services could be provided by manual devices like a pen and paper but, nowadays, they’re more likely to be computational systems that include both the hardware (the actual computing machines) and the software (like a DataBase Management System or DBMS), as well as the constraints on those systems as embodied in rules and procedures for a specific system or organization.

In fact, most of these information systems are usually embedded within an organization and so there are many business rules and processes that are encapsulated in them. When we say business rules, in this context, we almost always mean the rules for any large organization, not just a commercial enterprise. So these business rules apply for academic, governmental, and industrial organizations, as well.
Organizations are interested in making good decisions in order to grow their enterprise appropriately. The purpose of an information system is to facilitate good decision-making by ensuring the information, and its underlying data, are managed properly. CIS, in fact, often deals with business aspects of organizations, as well.

Many kinds of data might represent many different kinds of structured observations. As we start to collect the data, we might be able to group some of this data together. A collection of grouped data is likely related in some way and has internal relationships which might prove useful in considering how it could be transformed into information. The latent relationships in such a collection of data can be codified when the data is organized, perhaps in a relational DBMS.

CIS deals with data management and analysis, as well as the use and maintenance of the computational systems that allow for that processing. Maintaining and using the actual information systems and their components is usually encompassed in the field of Information Technology (IT). IT is thus the application of computational systems to business processes.

CS, on the other hand, can be thought of as concentrating more on the computational processes and their implementation rather than the maintenance and use of the components of these computational systems. CS looks at the processes or, following the physical metaphor, how the universe does things. Part of that is the flow of data so there is quite a bit of overlap between CS and CIS. Although there isn’t a clear division between these two fields, we can, in general, think of CIS as being more about the flow of data and CS as being more about the processes that transform and process the data.

**What about Computational Thinking?**

Computational Thinking, on the other hand, is mainly about data and how to think critically about that data and transform it. Thus, Computational Thinking is highly pertinent to all aspects of computation, both CS and CIS.

### 2.7.1 What Is Software Engineering?

No matter whether you’re dealing with Computer Science or Computer Information Systems, the first step in finding a computational solution is to clearly define the problem. In fact, finding the problem requirements, and converting them into problem specifications, is a clearly defined process in software engineering that is often carried out by a systems analyst.

Engineering ideas were incorporated into software development by people like computer scientist Margaret Hamilton in the 1960s. As digital computers and programs became more ubiquitous in business settings, the need to make programs more efficient came to prominence. Since engineering principles proved to be successful in other disciplines, computer scientists decided to apply engineering principles to the development of software which resulted in the sub-field of **Software Engineering**.

Economics often motivates innovation and it’s no different in computer science. We’ll meet many fundamental aspects of Software Engineering as we dive into the various aspects of the Software Development Life Cycle (SDLC) and the Unified Modeling Language (UML) later on.

### 2.8 Programming Languages

To recap, all of physics consists of computable functions. Computable functions are those whose output can be calculated using an effective procedure as carried out by an appropriate computing
agent. We then saw that if a function is computable, its final value can be computed by a theoretical Universal Turing Machine, the computing agent. These theoretical machines can be implemented in actual physical devices like the modern digital computer.

Digital computers can be used as our computing agent to process or run or execute the effective procedures, which are then called the software instructions or source code for the digital computer. Since functions use data, the information in that data can be encoded using a binary representation. The digital computers are built using transistors, which can carry out Boolean Algebra operations as they’re based on the on/off binary values of their switches, as well.

Since the computer program is an ordered set of instructions that should be carried out by the digital computer, our computing agent, those instructions have to be translated from the human programming language to binary, the language of the computer’s on/off transistor switches. This process of translation is often called **compiling** or **interpreting** and the final binary instructions are called machine code, as opposed to the source code which is written in the human programming language.\(^{14}\)

All software and data is stored as binary in a computer. In fact, the input data is also encoded as binary for use by a computer. Computation thus involves the manipulation of different representations of data by following some specified instructions or procedures. That same computation can usually be carried out by different equivalent procedures, data representations, and computing agents.

What might not be immediately apparent is that the procedures or instructions encoded in the software programs are represented as text in the appropriate programming language. But this textual content itself can be encoded in any other representation, as well! These representations of the software programs can then be stored together with the representations of the input data in the computer’s memory.

---

**The von Neumann Architecture**

The von Neumann Architecture, unsurprisingly attributed to the mathematician John von Neumann, described just this design architecture in 1945 when he proposed that a computer should have some space for memory that can store both the data and the instructions that process that data in the same storage space.

Today, any stored-program computer, one which stores the computer program instructions in electronic memory, is said to follow the von Neumann architecture if it cannot fetch both the instruction and the data from the same memory at the same time as they both are stored in the same place and share the same *bus*, or communication pathway.

There are other architectures, like the Harvard architecture, but the von Neumann architecture is by far the most popular in digital computers (analog computers are another matter altogether, of course\(^{15}\)).

---

\(^{14}\)We will talk about this process in more detail in the next chapter when we discuss their sub-components like tokenizers and parsers, translation to binary machine language for the logic gates, and then having that data sent back up again.
2.8.1 Programming Language Generations

As we’ve seen, computer programs are just ordered sets of instructions expressed in some programming language. Each instruction in the computer program is translated from source code into machine code, the binary language of the CPU. Programmers write the source code that is converted by either a compiler or an interpreter into binary instructions that the CPU can then execute. All data is also stored in a computer using binary, as specified in the von Neumann architecture.

Machine code, also called machine language or simply binary, is the binary representation of the instructions and is called a First Generation Language. Since the 1’s and 0’s of a binary representation are essentially incomprehensible to a human, scientists developed a higher-level, more abstract and human-friendly, language. That language, a Second Generation Language, is called assembly, although some might argue its odd mnemonic codes aren’t much friendlier than binary.

So, as computers and their associated software programs became more ubiquitous and useful in different enterprises, software engineers realized they needed to make the development of computer programs more accessible to people other than those wearing white lab coats with long, complex titles. This led to the development of Third Generation Languages, also called high-level languages, like Java, C++, Python, etc.

For example, 0010 0001 0000 1000 0000 0000 0000 0001 is an instruction to add 1 in a low-level binary language for MIPS ISA while addi $8, $8, 1 is the same add instruction in an assembly language. But a high level language like Python expresses that instruction much more intuitively as \( x = x + 1 \).

Scientists and engineers have continued developing higher-level, more abstract programming languages in Fourth and Fifth Generation Languages. Fourth Generation Languages are non-procedural languages like query and reporting languages (e.g., SQL) or application generators and decision support systems languages. These languages work on collections of information rather than individual bits and so languages like Python are sometimes also included here.

Finally, some Fifth Generation Languages use natural language while other Fifth Generation Languages remove humans from the loop altogether by using constraints given to the program rather than using a human-generated algorithm. This generation of languages would include declarative and logic-based languages like Prolog, as well. Computer scientists are, for the most part, language agnostics and pick whichever programming language is most appropriate for the problem at hand.

2.9 Computational Thinking Defined

But before we implement our solution in a particular programming language, we have to define an algorithmic solution for the problem we’re examining. Let’s look at how to actually find such a computational solution!

We’ll use these steps in evaluating every problem that we meet from here on out. The individual steps will be customized as different problems will require different detailed approaches.

In this approach, we can also think of the Principles as the Strategy, the high level concepts needed to find a computational solution; the Ideas can then be seen as the particular Tactics, the patterns or methods that are known to work in many different settings; and, finally, the Techniques as the Tools that can be used in specific situations. All of these are needed to come up with the eventual computational solution to the problem.

As a side note, this labelling of programming languages as belonging to different generations was all done retroactively. When people first used binary or assembly, they didn’t do it knowing there were easier, more abstract, representations available!
2.9 Computational Thinking Defined

Computational Thinking Steps

Computational Thinking is an iterative process composed of three stages:

1. **Problem Specification**: analyze the problem and state it precisely, using abstraction, decomposition, and pattern recognition as well as establishing the criteria for solution
2. **Algorithmic Expression**: find a computational solution using appropriate data representations and algorithm design
3. **Solution Implementation & Evaluation**: implement the solution and conduct systematic testing

Details of the Computational Thinking Approach

Let’s list the details of the various computational thinking **principles** and the accompanying computer science **ideas** and software engineering **techniques** that can come into play for each of these three steps. Please note, this is just a listing and we’ll be talking about all these principles, ideas, and techniques in more detail in the next few chapters.

1. **Problem Specification**
   - Computational Thinking **Principles**: Problem Analysis and Abstraction
   - Computer Science **Ideas**: Model Development using decomposition and pattern recognition
   - Software Engineering **Techniques**: Problem Requirements, Problem Specifications, UML diagrams, etc.

2. **Algorithmic Expression**
   - Computational Thinking **Principles**: Computational Problem Solving using Data Representation and Algorithmic Development
   - Computer Science **Ideas**: Data representation via some symbolic system and Algorithmic development to systematically process information using modularity, flow control (including sequential, selection, and iteration), recursion, encapsulation, and parallel computing
   - Software Engineering **Techniques**: Flowcharts, Pseudocode, Data Flow Diagrams, State Diagrams, Class-responsibility-collaboration (CRC) cards for Class Diagrams, Use Cases for Sequence Diagrams, etc.

3. **Solution Implementation & Evaluation**
   - Computational Thinking **Principles**: Systematic Testing and Generalization
   - Computer Science **Ideas**: Algorithm implementation with analysis of efficiency and performance constraints, debugging, testing for error detection, evaluation metrics to measure correctness of solution, and extending the computational solution to other kinds of problems
   - Software Engineering **Techniques**: Implementation in a Programming Language, Code Reviews, Refactoring, Test Suites using a tool like JUnit for Unit and System Testing, Quality Assurance (QA), etc.
Chapter 2. Computational Problem Solving

2.9.1 Computational Thinking Skills

The first step of the computational solution, Problem Specification, relies upon some essential computational thinking skills. Although computational thinking isn’t a formal methodology for reasoning, it does encompass some basic skills that are useful in all fields and disciplines. They constitute a way of reasoning or thinking logically and methodically about solving any problem in any area! These essential skills are also the buzzwords you can put on your résumé or CV so let’s delve into an intuitive understanding of the more important ones, especially decomposition, pattern recognition, and abstraction, as well as its cousin, generalization.

Decomposition is simply the idea that you’ll likely break a complex problem down into more manageable pieces. If the problem is some complex task, you might break it down into a sequence of simpler sub-tasks. If the problem deals with a complex system, you might break the system down into a bunch of smaller sub-components. For example, if you’re faced with writing a large, complex paper, you might choose to tackle it by decomposing the paper into smaller sub-sections and tackling each of those separately.

Pattern recognition is the idea of spotting similarities or trends or regularities of some sort in a problem or some dataset. These patterns that we might identify help us make predictions or find solutions outright. For example, if you’re driving on the freeway and you notice cars bunching together in the left lane down the road, you might decide to change into the right lane. Or if you see a consistent trend upward in a stock for a number of months, you might decide to buy some shares in that stock.

Abstraction is the idea, as alluded to earlier, of ignoring what you deem to be unessential details. It allows us to thus prioritize information about the system under examination. We can use this idea of abstraction to do things like make models, such as the map to represent the campus mentioned before. Another example of abstraction might be creating a summary of a book or movie. We can also generalize to form a “big picture” that ignores some of the inessential details.

Generalization like this allows us to identify characteristics that are common across seemingly disparate models, thus allowing us to adapt a solution from one domain to a supposedly unrelated domain. Generalization can help us to organize ideas or components, as we do when we classify some animals as vertebrates and others as invertebrates. In addition, being able to identify the general principles that underly the patterns we’ve identified allows us to generalize patterns and trends into rules. These rules, in turn, can directly inform the final algorithm we’ll use in the second step of constructing the computational solution.

2.9.2 Algorithmic Expression: Computational Problem Solving

The second step of the computational solution, Algorithmic Expression, is the heart of computational problem solving. So far we’ve learned that computer science is the study of computational processes and information processes. Information is the result of processing data by putting it in a particular context to reveal its meaning. Data are the raw facts or observations of nature and computation is the manipulation of data by some procedure carried out by some computing agent.

This conversion of Data to Information and then Knowledge can be done via computational problem solving. After defining the problem precisely, it involves these three steps:

- **Data**: structure raw facts for evidence-based reasoning
- **Representation**: create a problem abstraction that captures the relevant aspects of the system
- **Algorithm**: delineate a systematic procedure that solves the problem in a finite amount of time

17 Disclaimer: correlation does not equal causation; even if you spot a pattern, you might want to confirm or validate that prediction with other analyses before actually putting your money where your pattern is.
Algorithmic Expression, Step 2 in the *Computational Thinking Approach* above, can also be called **Computational Problem Solving**.

Computational Problem Solving involves finding an appropriate *representation* of, or context for, the data, and using that representation in an *algorithmic*, step-by-step procedure that solves the problem once the problem is clearly defined.

The contextualization of data can be considered a first approximation of information and the solution transforms the data to information and then actionable knowledge. This can be seen in Figure 2.9, which shows the *information processing* workflow.

One way to think about information is data in some *context*. If that context is the probability of occurrence, we end up with Shannon’s Information measure, as discussed in Section 1.9.5. If we put data in the context of some logic-based reasoning structure, we can reach some conclusion based on the evidence; this conclusion becomes our usable information that can form the basis of actionable knowledge. We can also codify this information in some knowledge-based system that is curated using knowledge management techniques.

One way to show a particular problem can be solved is to actually design a solution. This is done by developing an algorithm, a step-by-step procedure for achieving the desired result, using algorithmic thinking. If you can find an algorithm to solve the problem, then it is computable and you’re set! But if you cannot find an algorithm to solve it, that doesn’t mean the problem is not solvable, as we’ll see when we talk about P vs NP problems later on.

Of course, there are some mathematical problems that are not computable. **Formal Analysis** is the process of examining algorithms and problems mathematically, and, as we saw in Section 0.7.1, Gödel’s incompleteness theorems says there are some problems in the universe that are not solvable mathematically. In fact, some seemingly simple problems are *uncomputable*, not solvable by any algorithm!

Other problems are solvable but are *intractable*: they would take too much time or require too much memory space in order to solve them in any practical way. Such intractable problems, problems that are too complex to be solved analytically, can be examined *empirically* instead by implementing a system and studying its behaviour.

The majority of the computational problem solving process is spent in the design and analysis of the data representations and final algorithm, which is Step 2 in Section 2.9 above. The actual programming of the algorithmic solution in an appropriate programming language, called **Solution Implementation & Evaluation**, is only the final part, Step 3, of the Computational Thinking approach and what we’ll explore in the next chapter!

---

**Figure 2.9: Information Processing Workflow**
2.9.3 Strategies for Computational Problem Solving

The best scientists in the world are babies: they observe the world, formulate hypotheses, and constantly test their hypotheses via experimentation. For example, they might start off by seeing a very tempting three-pronged hole in the wall and formulate the hypothesis that sticking your wet finger in that hole will be a lot of fun. After being duly shocked, successful babies modify their hypothesis and no longer stick their fingers in those inviting holes in walls. If you are reading this, congratulations, you were a successful baby, as well.

Over time, we intuit the lessons we learned but often forget the kind of scientific thinking that led us to those initial conclusions. One of the goals in delineating the computational thinking approach is to help re-discover and formalize how we think about solutions so we can generalize it to unseen problems in a systematic manner.

We can then apply that same structured thinking approach to new problems with which we might be wrestling. Along the way, we’ll also come up with some strategies we can employ to aid and structure that innovative thinking. One approach to classifying the various problem solving strategies would be dividing them up into the following three categories:

- **Brute Force** Approaches that employ Ad Hoc Thinking:
  When we first face a new problem, we can delineate the initial state (the givens) and the final state (the goal) as precisely as possible. Once we have that, we can start to examine the various paths through the problem space that would solve the problem. Often, if the problem is simple enough, we can just take a brute-force approach where we do an exhaustive search of all the possibilities via trial-and-error.

  This kind of hacking is often easy to implement and will likely find a solution. But if the problem is sufficiently complex, it can quickly become intractable. We can even try a forward or reverse approach: i.e., we can either look for solutions that go from the givens to the goal or reverse the process and go from the goal to the givens. The goal to the givens strategy is most useful either when there are a large number of initial states or when there are a small number of possibilities for the final state.

  For example, if we’re trying to solve a maze, we can determine the initial state (the givens) as the starting point of the maze. We can determine the final state (the goal) as the ending point of the maze. Then, we can go from givens to goal by starting at the entrance to the maze and hacking our way through the maze. We could also reverse the process and start at the goal and work our way backwards. This can work great as long as the maze is simple enough but, as soon as it gets sufficiently complex, you’ll invariably be lost until the Minotaur ends your adventure. For these more complex problems, we can use more structured approaches that either take a deductive or inductive approach.

- **Deductive** Approaches that employ Analytical Thinking:
  Deductive approaches to solution involve laying out the evidence or resources for some problem domain in a logical structure and then examining that structure to determine to which solution it points. These kinds of approaches are often called analytical approaches and they can be classified as either top-down decomposition or bottom-up construction. Detectives and lawyers sometimes employ such deductive approaches; e.g., Sherlock Holmes will often take a bottom-up construction when he’s trying to figure out who was the killer whereas Perry Mason will take a top-down approach to trap the killer once he’s figured out whodunnit (and how).

  Top down analytical approaches start by defining the problem and the goal and then decomposing the problem into smaller and smaller sub-problems. The assumption is that eventually these...
sub-problems will be simple enough to solve easily and the re-combine into a final solution for
the original problem. In the maze problem, this would be analogous to determining the goal and
working your way backward to intermediate goals.

Bottom up approaches, on the other hand, start with the resources or skills that are available to
solve a problem and construct a “super-resource” by combining and reusing simpler resources
or skills. This requires that we have a very good idea of what resources or skills are available
before we embark upon finding a solution. This approach can also be seen in the maze problem
where this would be analogous to delineating the skills that are needed to maneuver in a maze;
things like going left until you hit a dead-end or being able to go left or right. You can then
combine these different skills or resources in the correct combination to make your way to the
final goal. We will also see this strategy in the taxi cab problem, which we’ll meet when we
discuss different distance metrics in later chapters.

- **Inductive Approaches that employ Analogical Thinking:**

  Finally, we can also take an inductive approach via reasoning by analogy. This kind of analogical
  thinking is exemplified by the British economist and philosopher John Stuart Mill’s Methods,
a set of five patterns of inductive inference for evaluating causal relationships, published in
*A System of Logic* in 1843. For our purposes, the methods that are most relevant are the Method
of Agreement and the Method of Difference.

  These kinds of methods are used by lawyers when they argue a case by comparing it to other
cases from the past which were similar or dissimilar to the current circumstances. In fact,
such inductive approaches were also used by Aristotle in his elucidation of categories; today,
software engineers use these methods in the modern computational equivalent of **object-oriented**
programming (OOP) where you try to find classes of objects that you can reuse or extend. A
class, in the computational sense, specifies the properties and behaviours of some set of objects
that are categorized, or classified, together. We’ll examine OOP in much more detail in a bit.

2.10 Problem Space and System State

The computational approach to problem solving depends upon a well-defined problem and a sequence
of precise, unambiguous steps (the algorithm) for solving the problem. A well-defined problem specifies
the **givens** (which can be the input or the **initial conditions**), the **goals** (also called the outputs), and the
**resources** (includes the means or methods required to accomplish the tasks or steps of the algorithm).

A systems analyst can clarify and, hopefully, quantify the goals, givens, and resources and also
determine the representation of each. As we’ve seen, the choice of representation can significantly
impact the choice of solution, or algorithm. Different representations can have different expressive
powers, reveal different features of the problem, or suggest different routes to solution. In computational
applications, a mathematical representation or notation sometimes makes the most sense, expressing
the problem (and solution) precisely and concisely.

Similarly, the algorithm has to consist of precise, unambiguous steps. An algorithm can sometimes
be expressed in a **problem space**, especially when it deals with a mathematical problem. A problem
space, in this context, is the set of all the states of a system. A specific **state** of the system is a record of
the values of all the elements of the problem under examination.

Let’s look at the task of making a peanut butter and jelly sandwich once again: the **elements** of the
problem in this case might be the bread, peanut butter, jelly, utensils, and whether you want to keep the
crust or not. A particular **state** of the PB&J system before you make the sandwich might then consist of:
Chapter 2. Computational Problem Solving

<table>
<thead>
<tr>
<th>State_Number</th>
<th>BREAD</th>
<th>PEANUT_BUTTER</th>
<th>JELLY</th>
<th>KNIFE</th>
<th>CRUST</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>20</td>
<td>700g</td>
<td>600g</td>
<td>clean</td>
<td>no</td>
</tr>
<tr>
<td>S2</td>
<td>18</td>
<td>690g</td>
<td>595g</td>
<td>dirty</td>
<td>no</td>
</tr>
</tbody>
</table>

Table 2.1: State Table for PB&J System

- the number of bread slices: 20
- the amount of peanut butter: 700 grams
- the amount of jelly: 600 grams
- a usable butter knife: clean
- answer to keep the crust? no

After you make the sandwich, the state of the elements might become:
- the number of bread slices: 18
- the amount of peanut butter: 690 grams
- the amount of jelly: 595 grams
- a usable butter knife: dirty
- answer to keep the crust? no

We can delineate the initial and final states of our PB&J system as shown in Table 2.1. We can thus represent the initial state of the PB&J system as:

\[
S1 = [20, 700, 600, \text{clean, no}] 
\]

and the end state as:

\[
S2 = [18, 690, 595, \text{dirty, no}] 
\]

where each state consists of the following set, or vector, of elements:

\[\text{[BREAD, PEANUT\_BUTTER, JELLY, KNIFE, CRUST]}\]

The problem space would then consist of just these two states, S1 and S2:

\[\{S1, S2\}\]

In general, though, a problem space consists of many possible states and a solution can be represented as some path from the initial state (which represents the givens) to the end state (which represents the goal state). We can often represent the complete state space of a problem as a state space graph. We can then traverse this graph in a depth-first or breadth-first manner in order to find the solution to our problem.

We can apply this kind of reasoning not just to making peanut butter and jelly sandwiches but to any general problem, from playing card games to computing the distribution of galaxies. Let’s look at the slightly simpler problem of card games next!
2.10 Problem Space and System State

2.10.1 Turing Machine Example

As we saw in Section 0.7.1, a Turing Machine is a generic symbol manipulator that has a tape head and a (possibly infinite) tape with squares for symbols.

The tape head can only do a few things:
1. **Read** the symbol on the current square
2. **Write** a symbol to the current square
3. **Move** from one square to an adjacent square (either left or right)
4. **Change** its internal state from one state to another
5. **Halt** when its program, the sequence of instructions, is done

Steps 3 and 4 above are a form of memory for the Turing Machine as they allow it to keep track of what it has just read (Step 3) and what it should do in response to that (Step 4). As we saw earlier, the tape the Turing Machine uses is also a form of memory. The (possibly infinite length) tape consists of a bunch of cells where each cell can either be blank or have some symbol on it. The particular symbol it might have depends on the problem the Turing Machine is trying to solve.

Let’s imagine a scenario where we’re playing a card game that converts all diamond cards to spade cards. So we might have a segment of the input tape with some diamond cards and, once our Turing Machine is done, that segment of tape should have only spade cards, as seen below:

```
[●, ●, ●, ●, □] → [♦, ♦, ♦, ♦, □]
```

So each cell on the Turing Machine tape for this game can only have one of three symbols, where two of them represent two suits from a standard card deck (i.e., only diamonds and spades, we’ll ignore the other two suits to simplify the problem somewhat) and the third symbol is a blank (to indicate there is no symbol in that particular cell of the tape).

The possible symbols that can be in a single cell of the tape, in this case, are:
- None (□)
- Diamond (●)
- Spade (♦)

We can then represent the different states of a Turing Machine with the following set, or vector, just like in the PB&J system we saw earlier:

```
[CURRENT_STATE, CURRENT_SYMBOL, MOVE, WRITE, NEXT_STATE]
```

Let’s assume the Turing Machine we want to construct should go through and convert all diamonds to spades. So it should scan through the tape and, when it finds a diamond, it should change it to a spade. When it reaches the end of the current diamond/spade sequence, it should go on to the next diamond/spade sequence.

**CURRENT_STATE** then refers to one of the possible states of the system which, in this case, are:
- **Check_Suit**: in this state, the tape head checks the current symbol and, if it’s a ●, it over-writes it with a ♦, and then goes on to the next state. However, if the current symbol is a ♦, then it leaves the symbol alone and instead moves to the right and goes to the Check_Stop state.
• *Check Stop*: in this state, the tape head checks the current symbol and, if it’s a ♦, it moves to the right. However, if the cell was empty (i.e., the symbol is None, □), it switches to the HALT state.

• *HALT*: in this state, the tape head checks the current symbol and, if it’s blank (i.e., the symbol is None, □), it moves to the right of the tape and enters the Check_Suit state.

These different states, and how you can transition from one state to another, can be represented in a State Transition Table as shown in Table 2.2.

So if we take the example of a 5-card poker hand that has all diamonds, our input tape might look like so:

[♦, ♦, ♦, ♦, □]

Our Turing Machine can then convert all ♦’s to ♠’s using the following program, or sequence of state transitions, in our Turing Machine:

{S1,S2,S3,S1,S2,S3,S1,S2,S3,S1,S2,S4,S5}

The above program would change the input tape to result in the following:

[♠, ♠, ♠, ♠, □]

Finally, we can see that our definition of computable functions uses this same machinery: i.e., we can then say that some function, \( f(x) \), is computable if we can construct a Turing Machine, with its input tape initialized to the string \( x \), such that, when it finishes its program, it writes the string \( f(x) \) as its output. Since Turing Machines are general symbol manipulators, we can use any set of symbols to represent our values; for example, if \( x \) and \( f(x) \) are integer values, we can represent those integer values as binary strings or decimal strings or really any other consistent representation.

Thus, any function that is computable has a corresponding Turing Machine representation; inversely, there are functions that are uncomputable or unsolvable as no Turing Machine exists to solve them. Since a Turing Machine is synonymous with the term *algorithm*, we can then say that there are some functions that are not solvable as there is no algorithm for solving those problems!

<table>
<thead>
<tr>
<th>Number</th>
<th>CURRENT_STATE</th>
<th>CURRENT_SYMBOL</th>
<th>WRITE</th>
<th>MOVE</th>
<th>NEXT_STATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>Check_Suit</td>
<td>Diamond</td>
<td>Spade</td>
<td>No</td>
<td>Check_Suit</td>
</tr>
<tr>
<td>S2</td>
<td>Check_Suit</td>
<td>Spade</td>
<td>No</td>
<td>Right</td>
<td>Check_Suit</td>
</tr>
<tr>
<td>S3</td>
<td>Check_Stop</td>
<td>Spade</td>
<td>No</td>
<td>Right</td>
<td>Check_Suit</td>
</tr>
<tr>
<td>S4</td>
<td>Check_Stop</td>
<td>None</td>
<td>No</td>
<td>No</td>
<td>HALT</td>
</tr>
<tr>
<td>S5</td>
<td>HALT</td>
<td>None</td>
<td>No</td>
<td>Right</td>
<td>Check_Suit</td>
</tr>
</tbody>
</table>

Table 2.2: State Transition Table for a simplified Turing Machine to replace all Diamonds with Spades. Each row represents a possible step, or state transition, from the CURRENT_STATE to the NEXT_STATE.
2.11 Computational Thinking in Action

Problem 2.2 Can you trace out the program above to see how it changes each cell? Once you do so, how many operations, or state transitions, are carried out on each input cell, including the final blank cell (□)?

Problem 2.3 Can you come up with a Turing Machine state transition table and its corresponding program that takes the same input tape and converts it to an alternating sequence of diamonds and spades? I.e., your tape should go from the original input to the new output as seen below:

\[
[\Diamond, \Diamond, \Diamond, \Diamond, \Diamond] \rightarrow [\Diamond, \spadesuit, \Diamond, \spadesuit, \Diamond]
\]

2.11 Computational Thinking in Action

Okay, enough theory, let’s look at this in the context of a real problem by revisiting our temperature sensor problem from Section 0.1. Let’s see the three steps of Computational Thinking in action.

Step 1: Problem Specification
We started off by first defining the problem: determining if it was cold enough to warrant taking a jacket to school. We then distributed the sensors and started collecting the raw facts, the GPS and temperature values. This involved ignoring things about the sensors like their colour, their height, etc.; this is the process of abstraction where we don’t model details that we deem to be irrelevant.

The problem abstraction is one of the first steps in defining the problem clearly and stating both the requirements and the specifications. We’ll discuss the difference between the two when we delve into software engineering ideas in more detail. But for now, the problem we finally stated was to ask if we need to wear a jacket when we go to our class this morning.

Step 2: Algorithmic Expression
As a first approximation, we organized the raw facts into a table; by giving the gathered facts this structure, we transformed them into data. That data was then transformed into information by giving it a context; the particular context we gave it was in a logic-based reasoning system that utilized a threshold to predict whether it was hot or not.

We then further processed that information into an actionable decision by using this rule-based knowledge to decide whether we would need a jacket or not. The decision rule, which used a threshold to decide if a jacket was necessary, is the procedure, or algorithm, we designed to solve this problem. This computational process and overview is outlined in Figure 2.10.

Step 3: Solution Implementation & Evaluation
Finally, we need to instruct the computing agent to calculate the result using the algorithm we designed. We would normally do this by implementing the solution instructions in some language that was appropriate for the computing agent. In this case, we were the computing agent and so we gave the data a different representation by plotting the values in a graph which we used to extrapolate a temperature value for 8am.

You could, at this point, ask whether this is the most efficient or feasible way to go about solving the problem. Is your extrapolation justified? Is there an easier way to go about the process rather than the laborious task of gathering temperature values from all over campus using an army of your friends? Could your friends each implement this solution easily using their own friends? How about all the students at all the other universities? Questions like these will help you evaluate your solution and
assess how well your algorithm might generalize to this class of problems and whether it would scale to all the other students and universities, as well.

**Problem 2.4** You sank my... marble!?

Here’s a variation of the more familiar battleship game: suppose I place a single marble in one of 8 boxes. The goal is to find in which box the marble is located! Play this game with a friend twice where you’ll employ a simple strategy in the first iteration and a different strategy in the second go at it. The first strategy is to guess random boxes. The second strategy is to ask which half the marble is in.

What happens after 5 times with each strategy?

What would happen if you increased the number of boxes?

Bonus: How about if you increased the number of marbles? Now what about if you increased both?

**Problem 2.5** Suppose a computer scientist has discovered a rule of nature when observing nearby stars. The rule deals with a sequence of three numbers, each of which deals with the luminosity of the stars. The first such discovered sequence was:

\[
\{3, 9, 81\}
\]

Can you guess the rule by proposing three additional sequences, where each sequence also has three numbers that follow your rule? To confirm your rule, please check the footnote and see if you were right!\(^\text{18}\)

\(^{18}\)The rule was actually just three increasing numbers although you might be forgiven for thinking it was squaring each subsequent number or some other rule. This is just to show that not only can multiple rules apply for a particular sequence of numbers but you also have to consider how you might verify or validate the underlying rules. Can you propose a method or algorithm to help validate when the rule you have conjectured is the correct one or not?
• CS deals with data representation and transformation in computational solutions. It is mainly concerned with the computational processes that facilitate the actual transformation of Data to Information to Knowledge. Information Systems help enable and manage the transformation of Data to Information to Knowledge.

• CIS deals with data curation, data flow, and information management in such systems. It is concerned more with the flow and management of data or information in the system than the computational processes themselves. Computational systems include both the hardware (the actual computing machines) and the software (like a DataBase Management System, or DBMS), as well as the constraints on those systems as embodied in rules and procedures for a specific system or organization.

• IT is the application of computational systems to business processes. This often involves managing, maintaining, and using the actual information systems and their components, as well.

---

19 Other confusing terms are Information Science, which deals with defining information and its properties, including examinations of information flow and processing, as well as optimizing information storage, accessibility, and usability. Information Theory, which usually refers to a derivative of Shannon’s Information Theory. And Informatics, sometimes also confusingly called Information Science, which deals with things like text analytics and knowledge management and representation.
Basics: Algorithmic Expression

3 Computational Thinking and Structured Programming ...................... 91
3.1 Review of Computation
3.2 Computational Thinking Basics
3.3 Minimal Instruction Set
3.4 Getting Started with Python
3.5 Syntax, Semantic, or Logic Errors
3.6 State of a Computational System
3.7 Natural vs Formal Languages
3.8 Translating Your Programs
3.9 Playing with Python
3.10 An Example Using Computational Thinking

4 Data Types and Variables .............. 111
4.1 Different Types of Data
4.2 Data Type = Values + Operations
4.3 Variables and Expressions
4.4 Input/Output
4.5 An Example Using Computational Thinking

5 Control Structures .................... 123
5.1 Algorithms and Control Structures
5.2 Sequence
5.3 Selection
5.4 Repetition
5.5 An Example Using Computational Thinking

6 Data Structures ...................... 135
6.1 Abstract Data Types
6.2 A Non-Technical Abstract Type
6.3 Advantages of ADTs
6.4 Data Structures
6.5 Strings
6.6 Lists and Tuples
6.7 An Example Using Computational Thinking

7 Procedural Programming .............. 147
7.1 Functions Redux
7.2 Functions in Python
7.3 Sub-Routines with Parameters and Values
7.4 Namespaces and Variable Scope
7.5 Exception Handling
7.6 File I/O
7.7 An Example Using Computational Thinking
3. Computational Thinking and Structured Programming

And now I see with eye serene
The very pulse of the machine.
– William Wordsworth, “She Was a Phantom of Delight”

3.1 Review of Computation

If you skipped Part I, or zoned out while skimming it, this section is for you. When last we met, we learned about Alan Turing’s Universal Machines. This was our first step in trying to understand what it means for something to be computable. We found that a specific task can be said to be computable if we can enumerate a sequence of instructions, or algorithm, that can be carried out by some computing agent in order to complete the task.

Turing proposed a theoretical class of machines to serve as computing agents in order to develop a formal, mathematical notion of computation. Originally called a-machines or automatic machines, we now refer to them as Universal Turing Machines or, more simply, Turing Machines. In 1936, Turing showed that his hypothetical mathematical machines could solve every mathematical problem as long as there was an algorithm for it. Turing Machines were imagined to be very simple devices which only had a tape head, with some internal controller and memory, and an infinitely long tape on which it could read or write symbols under the control of some program, or ordered set of instructions.

3.1.1 Modern Digital Computers

All modern digital computers are Turing-complete and can do everything a Turing Machine can do. Modern computers are multi-purpose machines that use programs to accomplish certain tasks. In this

1Quoted in “VMS Internals and Data Structures”, V4.4, when referring to software interrupts.
sense, all modern computers can simply be thought of as machines that are able to store and manipulate information under the control of some changeable program, just like the Turing Machines.

So what is a changeable program? Programs are algorithms, the detailed, step-by-step ordered set of instructions that tell a computer exactly what to do. If we change the program, the computer will then change what it does. And if you need to perform two different tasks (e.g., calculating a sum and writing a thesis), you don’t need to get two different specialized machines (i.e., a calculator and a typewriter) as you had to in the past. Instead, you can use a general-purpose computer and just change the program to the one that’s appropriate for each task. The programs that are run change but the machine stays the same!

Thus, there are three main aspects of modern digital electronic computers:

- Computers are devices for manipulating information derived from the input data.
- Programs are the instructions that a computer carries out or executes. These instructions are also called the algorithm, which is the computational solution to some computable problem.
- Computers operate under the control of a changeable program. They follow the von Neumann architecture and store both the data and the programs in the same memory space.

3.1.2 Evolution of Computers

These modern digital electronic computers were preceded by many different kinds of mechanical and electromechanical computers. The earliest computers, in fact, weren’t machines at all; they were people who did computations. Even as late as the 19th century, you could still find ads in newspapers looking to hire human computers and, in fact, the term persisted until the middle of the 20th century, where it was even applied to the female mathematicians that worked for NASA.

Humans also started to invent devices to help compute the result of complex and arduous calculations. Although many ingenious calculating devices were invented throughout human history, from the Mesopotamian Abacus to the Incan Quipus to German mathematician Gottfried Wilhelm Leibniz’s Stepped Reckoner, none of these devices used memory or were capable of operating without human intervention at every step. All this changed when the French merchant Joseph-Marie Jacquard invented the programmable loom in 1804. A loom is a device used to make fabric by weaving yarn or thread. A loom would hold the longitudinal warp threads in place so the weaver could interweave the filling threads. Most people used a handloom of some sort until the power loom was invented in the 18th century.

Jacquard invented a device that could be fitted to these power looms to help with the complex and tedious task of manually producing tapestries and patterned fabric. Jacquard’s programmable loom could be operated without intervention by a human operator at every step and used metal punched cards for memory. These interchangeable metal cards with holes punched in them were used to represent different thread patterns thus allowing for the automatic and reproducible production of intricate woven fabrics. The loom was programmed using a chain of these punched cards where each punched card encoded one row of the overall design. These patterns of holes and non-holes are the same kind of binary representation that can be utilized in Boolean logic!

2Using the term computer to mean someone who computes dates back to the early 1600’s: https://en.wikipedia.org/wiki/Human_computer

3 In addition to the South American Quipu or Khipu, some of which might even have used vines, there were similar knotted string accounting systems used in ancient China and Hawaii. The Chinese of course also had the abacus, as did the Mesamerican Aztecs who used maize kernels in a version called Nepohualtzitzin. Even the Romans used counting boards and the Greeks further developed complex machines like the Antikythera mechanism, which might be considered a mechanical analog computer as it used physical gears to carry out astronomical calculations.
3.1 Review of Computation

A Punchy Historical Detour...

This same idea of using patterns of holes and non-holes was also used to encode musical compositions. As mentioned in Charles Fowler’s *The Museum of Music: A History of Mechanical Instruments*, using a binary representation to encode musical information dates back to the 9th century Persian scholars, the Banu Musa. The Banu Musa invented a water organ that used interchangeable cylinders with raised pins on their surface, where the raised pins would represent the on/off states.

Over the years, many musical instruments and automata of various kinds, including musical boxes, clocks, and dolls that were powered by either water or sand or weights or springs or electricity, have used such encoding mechanisms with wooden barrels or perforated metal disks to encode binary information. By the 1670s, organs used wooden barrels with pins to play musical pieces automatically, eventually leading to mechanical organs and player pianos which used sheets perforated with holes to represent music.

In 1725, the French textile worker Basile Bouchon, inspired by such musical machines that were controlled by pegged cylinders, created a device to control a loom using perforated paper. His assistant, Jean-Baptiste Falcon, improved upon this initial invention but Jacques Vaucanson, in 1745, extended it to design the first fully automatic loom. His designs were further extended and perfected by Jacquard in his highly successful loom.

In 1832, the punch cards used by Jacquard also inspired the Russian inventor Semyon Nikolaevich Korsakov to use punched cards for storing arbitrary information as well as allowing for mechanized searches through large information stores, mainly of homeopathic medicines in his examples. You can learn about this fascinating bit of information technology history here: http://history-computer.com/ModernComputer/thinkers/Korsakov.html.

Twenty years after Jacquard, the English mathematician Charles Babbage used the punched-card idea to design his *Difference Engine* in 1822. The Difference Engine was a steam-powered mechanical calculator for automating the computation of polynomial functions, which are known to be good approximations of many useful functions, by using the “method of finite differences” to calculate the polynomial values without doing multiplication. In 1833, Babbage expanded upon the Difference Engine and designed the *Analytical Engine*, a programmable computer that also used metal punched cards for input and printed output on paper.

Although the Difference Engine was a special-purpose calculator, the Analytical Engine was a general purpose computer. This general-purpose machine could perform any mathematical operation in theory. The Difference Engine had a built-in sequence of steps which couldn’t be modified; the Analytical Engine, on the other hand, enabled a person, eventually called a programmer, to modify the sequence of instructions the machine could execute via punched metal cards. Just like in Jacquard’s loom, different patterns of holes in the cards would lead to different mechanical behaviours. By mechanizing the logical control, Babbage’s machine could solve any formal problem by carrying out any mathematical operation!

Like modern computers, the Analytical Engine contained an arithmetic processing control unit for fetching and executing instructions (called the mill), a readable/writeable memory for holding data and programs (called the store), and input/output devices (the punched metal cards). It also used stored programs to modify its behaviour by modifying two boxes of the punched metal cards where one box
was for the data, or variables, while the other box held the sequence of instructions, or program, for the machine. Babbage even imagined maintaining stores, or libraries, of the programs and data which could be reused easily.

In addition, the Analytical Engine even had a conditional branching statement, thus making it Turing complete. This allowed the program, or sequence of instructions, to branch to different options depending upon some condition. Not only did this branching allow the possibility of choosing between alternate actions, it also allowed the programmer to repeat some set of actions, as well.

Although Babbage died before the Analytical Engine could be completed, its innovative design was popularized by the writings of the English mathematician and writer Lady Augusta Ada King-Noel, Countess of Lovelace and daughter of the great English poet, Lord Byron. Her Notes also contained a step-by-step sequence of instructions (an algorithm!) that could be carried out by the machine in order to compute Bernoulli numbers.

The algorithm written in her Notes is considered to be the first computer program and Lady Ada Lovelace is recognized as the world’s first programmer, which led to a language, Ada, being named in her honour. In fact, despite never having seen the machine actually built, she not only envisioned this kind of machine being able to solve any abstract mathematical problem that was solvable but even being applied to the composition of music, images, and non-mathematical applications by having the instructions and data represent images and sounds instead of numbers!

The next computer developed by the human race used electromechanical relays, which are just simple mechanical switches used to control the flow of electricity in circuits, and was invented by the German engineer Konrad Zuse in the 1930s. Although it was also a general-purpose programmable computer, similar to the Analytical Engine, it was originally supposed to use vacuum tubes instead of electromechanical relays. Vacuum tubes can amplify, switch, and modify signals by controlling the flow of electrons without any moving parts.

However, since Zuse didn’t have sufficient financial resources living in Germany in World War II, he ended up using electromechanical relays instead. Unfortunately, the Nazis kept his work secret and his models (e.g., the Z1, Z2, Z3, etc.) were destroyed during the bombing of Berlin. Other electromechanical relay based computers were built in the US by people like John Atanasoff (eventually leading to the ABC, which was not programmable and so not Turing-complete), George Stibitz (the Model V, which was programmable), and Howard Aiken (the Harvard Mark I, which was also programmable). The Harvard Mark II led to the first computer bug being discovered by Admiral Grace Hopper, as we’ll see later on.

The next generation of computers did, however, use vacuum tubes and the first electronic, general-purpose digital computer was the room-sized computer called the ENIAC (the Electronic Numerical Integrator and Computer) by the physicist John Mauchley and electrical engineer J. Presper Eckert. The brilliant mathematician Alan Turing helped design the COLOSSUS at roughly the same time in England. The mathematician John von Neumann was also involved with the ENIAC and formalized the idea of storing programs in the same memory along with data to help simplify the tedious task of programming via switches and cables.

Of course, both Babbage and Turing had suggested these ideas earlier but von Neumann’s insight helped avoid having to unplug and replug hundreds or thousands of wires into boards every time a programmer wanted to perform a different computation. Von Neumann also recommended representing memory in binary and even suggested using the conditional control transfer machine instruction to invoke subprograms by storing the machine’s current state in memory.

This architecture, which has come to be called the von Neumann architecture, was used in the ENIAC’s successor, EDVAC. All modern digital computers, including our current transistor-based
computers, use this von Neumann hardware architecture which consists of a Central Processing Unit (CPU), a memory which holds both programs and data, and an input/output system!

### 3.1.3 What Is a Computer Program?

In modern computers, programs are the **software** that control the **hardware**, the physical machine. The central processing unit (CPU) is the brain of a computer and carries out all the basic operations on the input **data**. At its core, the CPU is made of switches and gates using transistors and does very simple operations like arithmetic operations that test to see if two numbers are equal using the Arithmetic Logic Unit (ALU). The power of a computer derives from its ability to do these simple operations very fast and very accurately.

![Diagram of CPU](image)

**Figure 3.1:** The Fetch-Execute Cycle in the CPU consists of Fetching the next Instruction; Decoding that Instruction and Getting the corresponding Data; and finally Executing that Instruction

The CPU can directly access information that is stored in its main memory, as seen in Figure 3.1. The main memory is usually called Random Access Memory (RAM) and is very fast but volatile. Memory itself is divided into cells where each cell can store a small amount of data. Each cell in memory also has a unique address that can be accessed using a bunch of wires called the **address bus**; the address bus is uni-directional and can only carry data from the CPU to the main memory. As we saw in Section 2.8, there is another bus called the **data bus** which carries the data from the CPU to the main memory and vice versa and so is bi-directional. Each wire in the bus can send a high-voltage to represent a 1 and a low-voltage to represent a 0 so that it can transmit binary digits. There is one additional wire, called the **command wire**, that goes from the CPU to the main memory and sets the mode of the memory to either read or write, depending on the operation dictated by the CPU.

Since the RAM itself is made of electrical components, if power to the machine ever goes away, the contents of this main memory are lost. Secondary memory provides greater persistence and allows for more permanent storage on different media like magnetic media (e.g., hard drives, tape drives, etc.), optical media (e.g., CDs, DVDs, etc.), and non-volatile transistor-based semiconductors (e.g., USB Flash drives).

---

4In RAM, data can be accessed at random rather than sequentially, as in hard drives. RAM also needs power to keep the data persistent; this is different from Read Only Memory (ROM) which is a non-volatile storage medium.
flash drives). Data can be moved from RAM to secondary memory (and vice versa) via a local, internal bus, as well.

Programs are the instructions that are carried out by the CPU, our computing agent. Since a CPU consists of simple on/off switches, all program instructions are eventually translated to binary, the actual on/off values to be processed by these circuits of switches. The programs are carried out, also called run or executed, by the CPU using the Fetch-Execute Cycle. This involves repeating the following steps until the computer is powered off:

- Retrieve, or fetch, the next instruction from memory
- Decode the instruction to see what it represents and Get the corresponding data
- Carry out, or execute, the appropriate action

The instructions that are executed by the CPU constitute the program that’s written by the programmer to solve some problem.

### 3.2 Computational Thinking Basics

The process of creating these programs, the software, is called programming. Programming is a fundamental tool in the computer scientist’s toolkit. It’s one of the most powerful tools for solving computational problems. Writing these programs, these computational solutions, requires us to think computationally.

*Computational thinking* can be thought of as involving three steps that are used iteratively to help solve the problem at hand:

1. **Problem Specification**: analyze the problem using abstraction, decomposition, and pattern recognition; state the problem precisely; and establish the criteria for its solution
2. **Algorithmic Expression**: solve the problem using algorithmic thinking, which involves designing an algorithm with an appropriate data representation
3. **Solution Implementation & Evaluation**: implement the algorithmic solution and systematically test it for correctness and efficiency

We’ll use this computational thinking approach to help design and implement our computational solutions as programs for computers.

### 3.3 Minimal Instruction Set

In the end, these programs are executed, or carried out, by the CPU at the heart of modern digital computers. Turing’s hypothetical machines also showed us that all modern computers have the same power as long as they have the appropriate programming. That means, given sufficient resources of memory and time, every computer can do all the things any other computer can do.

All that power is encapsulated by the simple functionality of a hypothetical Turing Machine\(^5\), which can only do three things:

1. Write or Delete a symbol
2. Move the tape head Left or Right
3. Go to a new state or stay in the same state

\(^5\)Nota Bene: we don’t actually use Turing Machines as they are only hypothetical machines and do not exist in reality. But a real physical machine like a modern digital computer can do all the things a theoretical Turing Machine can do.
3.3 Minimal Instruction Set

That’s it! That’s all that a machine needs to be able to do to solve any problem that is solvable by any computer. That means any machine with this kind of minimal set of operations can calculate anything that is computable.\(^6\)

Turing Machines are hypothetical machines so when you have an actual, physical computer, that minimal set of operations increases slightly, depending on the kind of computer with which you’re dealing. The set of all operations that the CPU of a physical machine can carry out is called its instruction set. The CPU is, in turn, controlled by a program, which is simply the ordered set of instructions needed to solve some problem; all the program’s instructions are eventually broken down into combinations of the operations that constitute the CPU’s instruction set which are then carried out, or executed, by the CPU.

Just as we have a minimal set of operations for a Turing Machine, it turns out we can define a minimal set of instructions, or minimal instruction set, for a real, physical machine, as well. That means all computer programs can be written using various combinations of just this minimal set of instructions! The importance of this statement deserves some additional emphasis: all possible programs that could ever be created can be written in terms of just this minimal set of instructions. For a specific approach to a certain class of computers, that minimal set of instructions consists of only five instructions. Thus, all computable functions ever can be computed with just these five instructions!

So what are these five instructions and what kind of a computer do you need to utilize them? Suppose you have a CPU with a separate, internal memory space, called a register, that can store a single value. Let’s call this memory space, or register, the accumulator. You could re-write any computer program to instead be written using just the following five instructions.\(^7\) This means that all programs ever written can be reduced to a combination of just these five instructions for this kind of a computer!

1. **LOAD A**: load data contents at RAM address \(A\) into the accumulator
2. **STORE A**: store the accumulator’s data contents to RAM address \(A\)
3. **INC**: increment the accumulator’s value
4. **DEC**: decrement the accumulator’s value
5. **BRZ X**: branch program execution to address \(X\) in RAM if the accumulator value is zero

These instructions are in an Assembly language, which is not very intuitive and is thus not used extensively nowadays. Instead, today we tend to use more user-friendly, high-level languages like Python! Most high-level programming languages provide a much richer set of possible instructions that are more abstract and easier for humans to understand. But we can think of the task of programming as the process of expressing a computational solution as a sequence of instructions in a high-level language like Python. Each of those instructions can, in turn, be broken down into smaller and smaller sub-tasks until the individual sub-tasks are simple enough to be expressed as a sequence of instructions in an instruction set that is appropriate for the particular CPU. Let’s jump right in and see how we can write programs in Python and run, or execute, them on an actual CPU.

\(^6\)As we’ll see later when we delve into P and NP problems, not everything knowable is computable.

\(^7\)These would be single-cycle instructions in a Reduced Instruction Set Computer (RISC) architecture as opposed a Complex Instruction Set Computer (CISC) architecture or the so-called One Instruction Set Computer (OISC). Another way to think of these is that CISC processors favour richer functionality in the hardware whereas RISC processors minimize hardware complexity and push more work onto the software. There are also variations on this that can be based on anything from two to eight instructions.
3.4 Getting Started with Python

Let’s start off by installing Python and getting the lay of the land in terms of the mechanics of writing a program in Python. Once we get our hands dirty, so to speak, we’ll step back and review some of the fundamental ideas of programming like statements, expressions, etc. and even implement the algorithm for calculating the average temperature value we met way back in Section 1.7.

3.4.1 What Is Python and Where Do I Get It?

Python is one of the most popular programming languages today. Not only is it used ubiquitously in applications like data science and machine learning, but it is also used extensively in various pedagogical settings due to its ease of use and low learning curve.

Editors, Interpreters, Debuggers, Oh My!

So what exactly does an Integrated Development Environment (IDE) give you? As you can see in Figure 3.2, which shows an example image of the NetBeans IDE in action, most IDEs give you:

1. **An Editor**: An IDE usually includes a window that behaves like a regular editor, which is just a program that lets you write and edit text (in this case, the text is your source code, your instructions in some programming language like Python)

2. **A Translator**: This translates your instructions, or code (e.g., in Java or Python), into **machine language** so the CPU can understand and execute them. It usually comes in two flavours: an interpreter or a compiler. An **interpreter** is kind of like an over-eager doggy, always ready to play catch or follow your next instruction, and, as soon as you give it a command, it immediately goes out and does it without waiting to see what the next one might be! Its cousin, a **compiler**, is more like a calm, trained helper dog who patiently waits for all your instructions before carrying them out in a batch.

3. **A Debugger**: When you write code, you’re guaranteed to get tons of errors, which are also called bugs. We’ll be talking about the various kinds of errors later but, rest assured, the IDE will help you catch some of these errors and remove any bugs from your program using the debugger.

You can install Python on your computer relatively easily but the exact process will vary depending upon whether you’re using a Windows, Linux, or MacOS computer. Both Linux and the MacOS should come with Python pre-installed, although its Integrated Development Environment (IDE) likely isn’t installed by default. You can download the default IDE from the Python Software Foundation’s website at [https://www.python.org/downloads/](https://www.python.org/downloads/). In this book, we’ll only use Python 3.x as opposed to Python 2.7, which is the older version that is still quite ubiquitous and somewhat less secure.

With either version, the IDE for Python gives you three things: an editor, an interpreter, and a debugger. The default IDE included with the Python download is called IDLE. The inventor of Python, Guido van Rossum, was a fan of the comedy troupe, Monty Python, and named the IDE after one of its players, Eric Idle.\(^8\)

---

\(^8\)If you don’t want to install anything on your own machine, you can use one of a plethora of online Python interpreters listed here: [http://www.sethi.org/tutorials/references_java.shtml#online-compilers](http://www.sethi.org/tutorials/references_java.shtml#online-compilers)

\(^9\)In fact, large parts of early Python development were done under the influence of mass quantities of beer and stroopwafels: [https://mail.python.org/pipermail/python-dev/2004-September/049041.html](https://mail.python.org/pipermail/python-dev/2004-September/049041.html)
3.4 Getting Started with Python

3.4.2 “Hello World” in Python

If you install the default IDLE IDE, you’ll find it looks more like what’s shown in Figure 3.3, which shows the Python shell in Figure 3.3a and the Python text editor in Figure 3.3c, than the more feature-rich NetBeans IDE we saw earlier in Figure 3.2. When you first launch IDLE, you’ll see the Python shell, which is essentially an interpreter, like the little doggy that’s eagerly waiting for your very next command. You can enter any commands for the Python interpreter at the prompt, the three chevrons: `>>>`

Let’s type the code, or instructions, shown in Listing 3.1 directly at the prompt in the Python interactive shell. Nota bene: when you do type in the code, please make sure you don’t type in the prompt (the three chevrons, `>>>`) and only type in the code, itself: `print(“Hello World!”)`

```python
>>> print(“Hello World!”)
Listing 3.1: “Hello World!” program for the Python shell
```

Once you do that, you can press enter and you should see the result, Hello World!, printed out below your command as shown in Figure 3.3b. The interactive shell will execute every instruction as soon as you type it in.

If you’d prefer to have your program run in a batch so that the entire program, all your instructions for the complete computational solution, is run in one go rather than typing-and-running a single instruction at a time, you can instead write your program in script mode. In this approach, you can

Figure 3.2: The NetBeans Integrated Development Environment (IDE), consisting of an Editor and Output Console. It usually also has a Debugger (not pictured above) and can work with multiple languages, like Python, Java, etc.
type your entire program, also called your script, into IDLE’s text editor. In order to do so, you can choose File > New File to open a new Text Editor as shown in Figure 3.3c. You can then type in your code there as shown in Listing 3.2. As you no doubt suspect, most programs we write won’t consist of a single line of code, or instruction, like these initial sample programs we’re using to learn our way around the IDE.

```
print("Hello World!")
```

Listing 3.2: “Hello World!” program for the Python Text Editor

You can now save your file anywhere, just be sure to add the .py extension, especially if you’re on a Windows system. Once you’ve got all that squared away, you can run it by choosing Run > Run Module.

Assuming you didn’t make any mistakes, you should see the same output, Hello World!, in the Python interactive shell window. If you mis-spelled something or missed one of the steps above, you likely ended up with a dreaded error!

### 3.4.3 Error, Error... Does Not Compute!

But, following the sage advice of Douglas Adams, “DON’T PANIC!” Errors, or as they’re better known in computer parlance, bugs, are a frequent annoyance that can actually prove quite helpful.

Since the ordered set of instructions, or algorithm, which specify your computational solution are designed to be carried out by the digital computer that serves as our computing agent, we have to ensure the instructions are precise enough to translate into an accurate sequence of gate and circuit activations. Since the CPU is just a bunch of simple circuits, any mistakes can wreak havoc with our computational
solution. In our context, this means that we might get no answer or, even worse, a wrong answer when we run, or execute, our program. So one of the main tasks that occupies most software engineers is tracking down and fixing these bugs, a process we fondly call **debugging**.

### A Buggy History...

In 1947, the mathematician and computer scientist Grace Hopper was working on Harvard’s Mark II electromechanical computer when she noticed an error in its execution. Upon examination, she found that a moth had gotten stuck in one of the relays, the electrical switches, inside the machine. Being a good scientist, she dutifully recorded the first bug in a computer program in her research notebook:

Although her notebook, this anecdote, and the remains of that unfortunate moth are now officially in the Smithsonian Museum of American History, the term *bug* has an older history in engineering, dating back to at least the 1880s (in fact, even the actual moth was found by other engineers although it was recorded by Admiral Hopper herself). The term **debugging**, however, seems to have come into prominence starting in this time period.\(^\text{10}\)

---

### 3.5 Syntax, Semantic, or Logic Errors

Before we can fix these potential errors, we have to figure out what kind of errors we might have. There are three broad kinds of errors we encounter in software development:

1. **Syntax Errors**: these are errors in following the *rules* of the language, like mis-spelling words, forgetting punctuation, etc. In English, a convoluted example would be, “THE dog bit, the mahn.” These are usually caught, often rather annoyingly, by the translator (either the compiler or the interpreter) every time you try to run your code. These are also the easiest to correct and often the IDE will even offer to fix them for you.

2. **Semantic Errors**: these are issues with constructing the statements of a language in a **prohibited** or improper manner. In English, it might be a typical conversation with Yoda, “Bit me the dog did.” In software engineering, they come in two flavours: *dynamic* (e.g., disallowed operations like dividing by 0 or trying to access invalid data like an out of bounds array index) or *static* (e.g., trying to use a variable without assigning it a value).

\(^{10}\)Please see this article: [https://www.wired.com/2013/12/googles-doodle-honors-grace-hopper-and-entomology/](https://www.wired.com/2013/12/googles-doodle-honors-grace-hopper-and-entomology/)
3. **Logical Errors**: these occur when the statements are fine but the result is incorrect or unexpected (e.g., printing the sum when you intended to print the average). The only way to find such errors is by testing. In English, this is what might happen in a journalism class when you might have intended to say, “A dog bit a man” but end up creating the headline, “A man bit a dog.” In computing, companies like Microsoft often call these a Beta Release of their software.\(^{11}\)

These errors can occur either at compile-time (syntax and static semantic) or at run-time (dynamic semantic and logical). Runtime errors are also sometimes referred to as exceptions and we’ll see later how we can use exception handling to take care of such exceptional situations.

The important takeaway in this is to not get frustrated when you see errors. Most errors are syntactical errors so that means the translator is helping you by catching any imprecision in your instructions for the CPU. Not only is that a good thing unto itself but everyone’s code usually contains some kind of error at first; in fact, the more complex the solution, the greater the likelihood of errors creeping in regardless of who is doing the programming. The toughest errors, as you might well imagine, are the logical errors since the only way to find them is to create an appropriate test. But as the computer scientist Edsger W. Dijkstra famously said:

> Testing shows the presence, not the absence, of bugs.

These bugs can, in fact, be incredibly persistent; some take literally decades to discover as the appropriate test wasn’t created or discovered in that amount of time. For example, Bill Joy, co-founder of Sun Microsystems, wrote a program called head that, under a certain set of conditions, would crash but that particular set of conditions went undiscovered for 15 years!\(^ {12} \)

### 3.5.1 Debugging

Once you’ve identified the bugs, the next step is to get rid of them by debugging them. This isn’t an easy undertaking as the sources of the most significant errors aren’t usually apparent. The syntactical errors, as we mentioned earlier, are the easiest to find and eliminate since the translator (either the compiler or the interpreter) will usually give you a heads-up on those. If appropriate for your language of choice, the static semantic errors will also be flagged by a compiler. The tough ones are the dynamic semantic and logic errors, both of which are only detectable at run-time.

A lot of languages, like Java, will give you clues about dynamic semantic errors and force you to handle these exceptional situations. But the only way to discover the logical errors is by becoming a bit of a detective yourself. Testing your program will give you clues as to which parts might have logical errors. Then, you can investigate what might have led to the error in your program. You can use the debugger that’s included in most IDEs, including in IDLE, or you can manually insert statements that output the current state of the program in an effort to get a handle on what went awry.

---

\(^{11}\) In the interest of avoiding any potential lawsuits, please note that this is a PJ (poor joke). On a more serious note, though, static program analysis can also be used to catch some subsets of logic errors.

\(^{12}\) Please see here for the gory details: [https://www.csoonline.com/article/2927441/application-security/11-software-bugs-that-took-way-too-long-to-meet-their-maker.html#slide12](https://www.csoonline.com/article/2927441/application-security/11-software-bugs-that-took-way-too-long-to-meet-their-maker.html#slide12)
One small step...

One of the best software engineering tips might be to make small modifications and check often. That is, as you write your program, make **small changes** each time and run it after making each change so you don’t introduce any new errors that you have to track down after typing in 378 lines of code. This technique also serves you well when debugging or trying to fix your code: introduce **small corrections** and check to see if they make a positive difference before persisting further in that direction. The essential idea is to make sure the **state** of your program is valid at every step and you have a functional, working program at each step.

### 3.6 State of a Computational System

In Section 2.5, we saw that we could model any computational system as a process which takes some input and computes some desired output. The **state** of a computational system, as we saw in Section 2.5.2, can be specified for a digital circuit as the information stored in its memory elements or for a general mathematical system as the values for one or more of its variables. We also met the idea of a state and its associated variables in Section 1.6.

In our case, this process is some program we might write in a programming language and run on a modern digital computer. As we started to think about programming languages, we saw, in Section 2.5.3, that a formal language has a vocabulary, or lexicon, and a set of grammatical rules for constructing valid sentences, or statements. In the same way, a programming language has a specific vocabulary and a specific grammar for instructing the CPU, the computing agent, how to perform a particular task.

The vocabulary of a programming language consists of some **reserved words**, also called **keywords**, that it understands and a particular syntax for creating and organizing the instructions using those keywords. The program we write is just the specific implementation of our abstract algorithm, the step by step instructions, for solving some computational problem written in that particular programming language.

We can think of the algorithm as a sort of recipe and, just like in a cooking recipe, an algorithm can have variables, corresponding to the ingredients in a recipe, as well as statements, corresponding to the directions in a recipe. The directions in a recipe usually utilize or manipulate the ingredients in the process of transforming the input to the desired output. The **state** of the recipe at any point in time can be represented by the values of the ingredients (e.g. the cake is half-baked).

In the same way, your program can create variables, symbols for representing some data that’s stored in the main memory of the computer, which it can utilize or manipulate as it carries out the instruction statements in an algorithm. The values represented by the program’s variables at a specific point in time thus constitute the **state** of the program at that time, as well.
3.7 Natural vs Formal Languages

Although the high-level programming languages we use to write our programs can sometimes look very similar to human languages, they are very different. Human languages that evolve without the imposition of any specific design, like English, are called natural languages. These differ from formal languages which are artificial, invented languages that are designed by people for a specific purpose. For example, chemists might use a specific language to represent chemical shapes and structures. We met some of the rules for formal languages in Section 2.5.3 where we learned that computation can be expressed using programming languages that are designed to follow these same mathematical rules.

Programming languages can broadly fall into two main categories, imperative and declarative, similar to what we saw in Section 1.6. Imperative programming languages use an explicit sequence of statements to do something or give a command of some sort.

As we saw in Section 1.4.3, statements can be either declarative or imperative. Declarative statements indicate what we want and what is true. Imperative statements, on the other hand, are either questions or commands. Since we’re only dealing with imperative programming languages, as opposed to functional languages which are declarative, our statements issue commands that are carried out, or executed, by the interpreter in Python. An example of a command we used earlier is the print() command in Listing 3.2.

Imperative programming languages, like C, C++, Java, etc., can be further classified as:

1. Structured: structured programming uses control flow structures over goto statements to avoid problems like creating “spaghetti code”.
2. Procedural: procedural programming is based on the modular programming idea of dividing your program into individual elements called sub-routines, usually procedures or functions, named blocks of code with optional inputs and outputs, as seen in Section 7.1.
3. Object-Oriented: object-oriented (OO) programming encapsulates the behaviour and state data in an object with the actual computation being carried out by sending messages between objects. There is a subtle difference between modular programs and OO programs as modular programs can be purely procedural and rely upon either functions or classes but OO programs use a class-based representation as their primary model, as seen in Section 8.2.

Declarative programming languages, like SQL, Prolog, etc., on the other hand, specify the desired result but don’t delineate an explicit procedure for how to compute it, more in line with a calculus than an algebra, for example. We can think of these approaches as consisting of programming commands or mathematical assertions.

Functional programming languages, like Haskell, Ruby, etc., are usually considered a subset of declarative programming languages and are characterized primarily by not having any mutable data. They follow the no side-effects approach where the only result of a functional program should be the value that’s computed but nothing else should change during the computation, including the variables and the state. One other quick point that might not make sense till we get to the functions chapter in a little bit but most functional programming paradigms also treat functions as “first-class elements,” which just means that functions can be treated like variables and can be the values of data structures, or used as arguments, or used within control structures directly.

Although there are significant differences between formal and natural languages they both share some commonalities, as well. For example, they each are composed of tokens, which are the basic

---

13 That might be more in the sense of modern, or abstract, algebra which is concerned with structure whereas calculus is not.
elements of the language. For natural languages, tokens might be the normal words we use when we talk to each other. For languages used in chemistry, they might consist of the chemical elements. For most programming languages, they are usually things like numbers, words, operators, delimiters, etc. As we saw with formal languages, they each also have a certain structure, an allowed syntax, and some sense of semantics, as well.

But formal languages, and programming languages, are distinguished from natural languages in that they cannot deal well with ambiguity. For example, in English, you might understand the phrase, “raise the roof,” as an idiom in a social context or a directive in the context of a construction job. Sentences, or statements, in programming languages have to be unambiguous and precise.

Unlike natural languages, this means that each statement in a formal language has to have a unique meaning regardless of the context. This also helps reduce the redundancy in formal languages as compared to natural languages.

So you can’t just tell a computer, “Give me a buck!” Computers take things literally and don’t understand idioms or metaphors. A computer, upon hearing that sentence, just might try to get you a male deer. Instead, you have to be literal in all instructions sent to the CPU as all it understands is the 1’s and 0’s of the digital logic circuits that compose its guts.

Talking about our Generations...

As we saw in Section 2.8.1, there are different generations of programming languages. A First-Generation Language is the machine language that is unique for each different kind of CPU and is the only kind of language the computer really understands. Second-Generation Languages are the assembly languages that replace binary digit codes with mnemonic keywords. Third-Generation Languages are the high-level computer languages with syntax and keywords that are almost English-like and designed to be used by people.

These include the languages we’ll be studying like Python and Java. We’ll use these high-level programming languages to write our programs, the instructions the computer will follow to solve our computational problem. Because these languages are still not completely natural, software engineers often refer to the program as their code and the process of writing an algorithm in a programming language is usually called coding. In the end, our high-level programs have to be translated to low-level languages like assembly or machine language in order for the CPU to understand them.

3.8 Translating Your Programs

When people speak different languages, there are a couple ways to communicate ideas, other than frantically waving your arms. One approach might be to hire an interpreter, someone that will listen to every sentence you say and immediately translate it to the other person’s language. Another approach might be for you to write down all that you want to say and then have someone translate the entire note and give it to the other person. An interpreter immediately translates what’s said but it slows down the

14Thankfully, we’re still a few years away from computers actually being able to do this when they misunderstand you. We can, however, use programming to do Natural Language Processing (NLP) which can decode human idioms into something the computer can understand and interpret so it might just give you that $1 bill, or whatever denomination is appropriate for your locale.
communication of ideas as the speaker has to pause while each sentence is translated; a translation of the entire note takes a long time initially but then is much faster to read in one go by the other person.

In the same way, all programs written in a high-level, English-like programming language eventually have to be converted into a low-level machine language that the CPU can understand and execute. In fact, each different brand of CPU has its own unique machine language which makes the task of translating from a high-level language to a low-level language somewhat tricky.

In general, there are three main paradigms we can use to translate a program from a high-level language like Python or Java to a low-level language so that we can execute it. We can do so by using a:

- **Compiler**: this is when the entire program written in a high-level programming language is translated, using a tokenizer and (usually) a parser, into low-level code, first into assembly and then into machine language (also called object code) for the particular processor you’re using. Those object files are bundled into an executable by a linker which can attach them to any needed libraries. When you want to finally launch the newly-compiled program, a loader kicks into gear to load your instructions into the CPU’s main memory so it can then start executing those instructions.

- **Interpreter**: a traditional interpreter takes a single instruction in a high-level programming language and immediately performs the requested actions. This can sometimes involve doing the same steps as in a compiler where some high-level instructions might need to be converted to machine language, as well.

- **Virtual Machine Interpreter**: In this scheme, both a compiler and an interpreter are used. A compiler, in this case, doesn’t translate to machine language for the physical machine but rather to machine language for a virtual machine that only exists in software (i.e., it’s some program that simulates a physical machine). The object code for this virtual machine is usually called bytecode and the bytecode, in turn, is interpreted by a Virtual Machine and the appropriate actions (or opcodes) are executed to produce output, as shown in Figure 3.4.

The advantage of this approach is that it allows a program converted to bytecode to be executed...
on any physical machine that can run the virtual machine for that language. For a traditionally compiled program, every time you want to run it on a different hardware platform (e.g., MacOS vs Windows vs Linux), you would have to re-compile the program for the new platform. Interpreted programs, on the other hand, can run on any platform that has the appropriate interpreter since they’re not compiled ahead of time.

As you might imagine, an interpreted program is often slower than a compiled program since it has to be interpreted each time it is run. As such, some interpreted languages also allow for Just-In-Time (JIT) compilation where the bytecode is converted into executable object code for the physical machine.

The Python shell is a way to interact with the Python interpreter. You can access the Python interpreter in either shell mode or script/batch mode. The Python shell might more accurately be called “interactive shell” rather than just shell.

3.9 Playing with Python

All right, let’s dig in and start playing with Python a bit before we explore some of the more advanced ideas in the next few chapters. Assuming you have IDLE up and running, as in Figure 3.3, we can start seeing what Python can do. We’ve already learned how to say hello to Python in Section 3.4 using the print() command so let’s see what else we can do with it.

We could, if we wanted, use Python as an overpowered calculator:

```python
gt> 3 + 2
5
gt> 3 - 2
1
gt> 3 * 2
6
gt> 3 / 2
1.5
```

We can have it do simple arithmetic as above. If we need more complicated math functionality, we can import the math library. One of Python’s biggest strengths is the various libraries, also called modules, that are available with Python. These modules contain commands or functions that other people have already written for us and all we need to do to use them is import that library or module:

```python
gt> import math
gt> math.sqrt(16)
4.0
gt> math.sin(60)
-0.3048106211022167
```

Here, we imported the math module and, once we’d imported it into the Python shell, we were able to use it to invoke its sqrt() command and its sin() command. As you can tell with these examples, and the print() command, Python also follows the functional notation we met in Section 0.5 where the name of the function (like print or sin) is followed by the opening and closing parentheses,“(“).

You might have noticed that the actual call to the sine function in Python wasn’t sin() but rather math.sin(). We can think of the sin() function as living in the math module so when we want to use, or invoke, the sin() function, we have to use its full name or full location as math.sin(). The
period or full stop that separates the name of the module (math) from the call to the function (sin()) has a special name, the **dot operator**. The dot operator can be used to access any function or variable in a module or object and is very powerful, as we’ll see when we get to the chapter on classes. Because the print() function is part of the base Python language and doesn’t live inside another module, we could call or invoke it directly without needing to qualify its location.

There are tons of other commands available in both the math module and the base Python language. Sometimes, though, you might not remember where a particular function lives. When that happens, Python lets you know right away:

```python
>>> math.abs(-500)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: module 'math' has no attribute 'abs'
>>> abs(-500)
500
```

We thought the math library had the abs() absolute value command but, when we tried it, Python let us know that the math module did not have an attribute named abs. Since the abs() command is part of the base language, we were able to call it directly on the next line to find the absolute value of -500.

Similarly, if we had messed up typing in the print() command we’ve used before, Python will let us know that something went awry:

```python
>>> print("Hello World!")
Hello World!
>>> print("Hello World!

File "<stdin>", line 1
    print("Hello World!

^ SyntaxError: EOL while scanning string literal
```

We printed out the string “Hello World!” in the first two lines above but, when we tried to do it again, we forgot to enter the closing quote and closing parenthesis. Python immediately let us know there was an issue with our syntax with the warning, “SyntaxError: EOL while scanning string literal”.

### 3.9.1 Python and Mathematics

Python syntax very closely reflects standard mathematical syntax. As such, you can almost type mathematical statements just as they are directly into the Python interactive shell. For example, in mathematics, if you wanted to create a variable x and assign it a value of 5, you might say, x=5. This, and other mathematical operations, carry over identically in Python:

```python
>>> x=5
>>> print(x)
5
>>> y=x*5
>>> print(y)
25
>>> hypotenuse_squared = x*x + y*y
>>> print(hypotenuse_squared)
650
```
In fact, as you can see by the last statement in the listing above, we aren’t constrained to short, non-descriptive names for variables as we are by convention in mathematics. So instead of calling our hypotenuse something uninformative like \( z \), we can instead choose to call the variable by the much more descriptive name, \( \text{hypotenuse}_\text{squared} \).

### 3.10 An Example Using Computational Thinking

We can easily extend this to solve the problem we started at the beginning of the book in Section 0.1.1, namely, computing the average of a bunch of temperatures. Let’s follow the three Computational Thinking steps we learned in Section 3.2 and develop an initial computational solution for this problem:

1. **Problem Specification**: Compute the average value of three temperature values

2. **Algorithmic Expression**: Store 3 temperature values in separate variables and use them to compute average Temperature using our mathematical model as the algorithm: 

\[
T = \frac{\sum_{i=1}^{3} T_i}{3}
\]

3. **Solution Implementation & Evaluation**: Implement this algorithm in Python and test with the following input values and results:

<table>
<thead>
<tr>
<th>Temperature #</th>
<th>Temperature Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>68</td>
</tr>
<tr>
<td>2</td>
<td>71</td>
</tr>
<tr>
<td>3</td>
<td>73</td>
</tr>
</tbody>
</table>

Average of above 3 temperatures = 70.666667

Now we’re ready to implement our solution in Python by creating the following program as seen in Listing 3.3.

```python
>>> temperature1 = 68
>>> temperature2 = 71
>>> temperature3 = 73
>>> number_of_temperatures = 3
>>> average = (temperature1 + temperature2 + temperature3)/number_of_temperatures
>>> print("Average temperature: ", average)
Average temperature: 70.66666666666667
```

Listing 3.3: First attempt at computing the average temperature

In Listing 3.3, we needed to know exactly what the temperature values were and how many we had. As you might imagine, Python gives you a lot of power and flexibility so that you don’t have to provide all this explicit information ahead of time. We’re going to discover all the functionality Python offers to let us wield that power in the next few chapters.
Figure 3.5: Different Kinds of Errors: Syntax, Semantic, and Logical errors occur at either compile-time (syntax and static semantic) or run-time (dynamic semantic and logical). Semantic and logical errors are also sometimes referred to as exceptions and we can use exception handling to take care of such exceptional situations. Most languages use exceptions for both compile-time and run-time errors. In languages like Java, exceptions that occur at compile-time are called checked exceptions whereas run-time exceptions are called unchecked exceptions. In languages like Python, C++, and C#, the compiler does not force the programmer to deal with exceptions so all exceptions are unchecked.
Bibliography


Abstract Data Type, 142, 195
Abstract Data Type (ADT), 135
  Characteristics, 136
  Operations, 136
Abstraction, 60, 191, 203, 204
  Data Abstraction, 191
  Procedural Abstraction, 191
Accuracy, 59, 232, 269
ACID, 218
Alan Kay, 169
Algorithm, 12, 13, 91
Algorithmic Thinking, 29, see Thinking
American Standard Code for Information
  Interchange (ASCII), 112
Application Programming Interface (API), 191
  Public Interface, 191
Artificial Intelligence, 227
  General AI, 227
  Hard AI, 227
  Narrow AI, 227
  Soft AI, 227
  Strong AI, 227
  Weak AI, 227
Assignment Operator, 114
Bayes Theorem, 262
  Evidence, 262
  Likelihood, 262
  Prior, 262
Bayesian Inference, 260
Bayesian Statistics, 260
Big Data, 247
  Semi-structured Data, 247
  Structured Data, 247
  Unstructured Data, 247
Binary Files, 163
Bookkeeping Tool, 48, 275
Boolean Algebra, 68, 69
Breadth-First, 81
Bugs, 100
Business Analytics, 255
Business Intelligence, 255, 256
Business Rules, 210
Byte, 112
Changeable Program, 91
Charles Babbage, 92
Church-Turing Thesis, 16
Comment, 114
Compiler, 105
Computable, 91
Computation, 12, 91
  Computable, 8
  Computable Functions, 12, 14
  Effective Procedure, 12
  Turing Complete, 29
  Un-Computable, 18
Computational Thinking, 21, see Thinking, 76
  Abstraction, 78
  Approach, 76
  Basics, 96
Computational Problem Solving, 78
  Ad Hoc Thinking, 80
  Bottom-Up, 80
  Brute Force, 80
  Deductive Thinking, 80
  Exhaustive Search, 80
  Inductive Thinking, 80
  Intractable, 80
  Strategies, 80
  Top-Down, 80
  Decomposition, 78
  Generalization, 78
  Pattern Recognition, 78
Computer Information Systems, 73
Computing Agent, 13
Computing Machines, 92
  Babbage Analytical Engine, 92
  Babbage Difference Engine, 92
  COLOSSUS, 92
  EDVAC, 92
  ENIAC, 92
  Greek Antikythera, 92
  Harvard Mark I, 92
  Incan Quipus, 92
  Mesopotamian Abacus, 92
  Programmable Loom, 92
  Z1, 92
Confusion Table, 269
Control Structures, 123
  Böhm Jacopini, 123
  Control Flow Graphs (CFGs), 123
  Iteration, 123
  Repetition, 123
  Selection, 123
Sequence, 123
Structured Program Theorem, 123
  Data, 3, 45
  Structured Data, 203
  Data Analysis, 218, 256
  Data Lake, 218
  Data Mart, 218
  Data Mining, 218
  Data Model, 142, 195, 206, 214, 218
  Data Representations, 62
  Data Science, 256
  Data Structure, 137, 138, 142, 195
  Data Structures, 62
  Data Type, 112, 137, 142, 195
  Data Warehouse, 218
  Database, 206
    Artificial Key, 220
    Attribute, 215
    Composite Key, 220
    Composite Primary Key, 220
    Data Type, 215
    Entity, 215
    Feature, 215
    Fields, 208
    Foreign Key, 214, 215
    Indexed, 215
    Instance, 208, 215
    Natural Key, 220
    Primary Key, 214, 215
    Relations, 215
    Schema, 208, 215
    Sub-Schema, 208
    Table, 208, 215
    Views, 208
    Virtual Table, 208
  DataBase Life Cycle (DBLC), 210
  DataBase Management System (DBMS), 208
  Database Model, 214, 218
  DBMS Catalog, 218
  Debugger, 98
  Decision Tree, 234, 241, 280
    Random Forest, 239
    DecisionTree
    Random Forest, 235
  Declarative programming, 104
<table>
<thead>
<tr>
<th>Term</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Depth-First</td>
<td>81</td>
</tr>
<tr>
<td>Descriptive Analytics</td>
<td>255</td>
</tr>
<tr>
<td>Diagnostic Analytics</td>
<td>255</td>
</tr>
<tr>
<td>Digital Circuits</td>
<td>69</td>
</tr>
<tr>
<td>Digital Computers</td>
<td>91</td>
</tr>
<tr>
<td>Digital Logic Circuits</td>
<td>71</td>
</tr>
<tr>
<td>Discrete Event Simulation</td>
<td>169</td>
</tr>
<tr>
<td>Editor</td>
<td>98</td>
</tr>
<tr>
<td>Embedded SQL</td>
<td>220</td>
</tr>
<tr>
<td>Call Level Interface (CLI)</td>
<td>220</td>
</tr>
<tr>
<td>Statement Level Interface (SLI)</td>
<td>220</td>
</tr>
<tr>
<td>Encapsulation</td>
<td>191</td>
</tr>
<tr>
<td>Energy</td>
<td>56</td>
</tr>
<tr>
<td>Ensemble Methods</td>
<td>239</td>
</tr>
<tr>
<td>Bagging</td>
<td>239</td>
</tr>
<tr>
<td>Boosting</td>
<td>239</td>
</tr>
<tr>
<td>BootStrapping</td>
<td>239</td>
</tr>
<tr>
<td>Stacking</td>
<td>239</td>
</tr>
<tr>
<td>Entities</td>
<td>204</td>
</tr>
<tr>
<td>Entity Relationship Diagrams (ERDs)</td>
<td>210</td>
</tr>
<tr>
<td>Cardinality</td>
<td>210</td>
</tr>
<tr>
<td>Constraints</td>
<td>210</td>
</tr>
<tr>
<td>Entropy</td>
<td>206, 271</td>
</tr>
<tr>
<td>Error</td>
<td>282</td>
</tr>
<tr>
<td>Bias Error</td>
<td>282</td>
</tr>
<tr>
<td>Bias-Variance Tradeoff</td>
<td>282</td>
</tr>
<tr>
<td>Irreducible Error</td>
<td>282</td>
</tr>
<tr>
<td>Noise</td>
<td>282</td>
</tr>
<tr>
<td>Overfitting</td>
<td>282</td>
</tr>
<tr>
<td>Signal</td>
<td>282</td>
</tr>
<tr>
<td>Underfitting</td>
<td>282</td>
</tr>
<tr>
<td>Variance Error</td>
<td>282</td>
</tr>
<tr>
<td>Errors</td>
<td></td>
</tr>
<tr>
<td>Compile-Time</td>
<td>101</td>
</tr>
<tr>
<td>Exception</td>
<td>101</td>
</tr>
<tr>
<td>Exception Handling</td>
<td>101</td>
</tr>
<tr>
<td>Logical</td>
<td>101</td>
</tr>
<tr>
<td>Run-Time</td>
<td>101</td>
</tr>
<tr>
<td>Semantic</td>
<td>101</td>
</tr>
<tr>
<td>Syntax</td>
<td>101</td>
</tr>
<tr>
<td>Exception Handling</td>
<td>158, 198</td>
</tr>
<tr>
<td>Catch</td>
<td>158</td>
</tr>
<tr>
<td>Checked Exceptions</td>
<td>158, 198</td>
</tr>
<tr>
<td>Compile-Time</td>
<td>158</td>
</tr>
<tr>
<td>Defensive Programming</td>
<td>158</td>
</tr>
<tr>
<td>Error</td>
<td>198</td>
</tr>
<tr>
<td>Raise</td>
<td>158</td>
</tr>
<tr>
<td>Run-Time</td>
<td>158</td>
</tr>
<tr>
<td>RuntimeException</td>
<td>198</td>
</tr>
<tr>
<td>Throwable</td>
<td>198</td>
</tr>
<tr>
<td>Unchecked Exceptions</td>
<td>158, 198</td>
</tr>
<tr>
<td>Exploratory Data Analysis (EDA)</td>
<td>245</td>
</tr>
<tr>
<td>Exploratory Data Analysis (EDA)</td>
<td>255</td>
</tr>
<tr>
<td>Expressions</td>
<td>112</td>
</tr>
<tr>
<td>Extract, Transform, Load (ETL)</td>
<td>218</td>
</tr>
<tr>
<td>F1-Score</td>
<td>269</td>
</tr>
<tr>
<td>Firmware</td>
<td>65</td>
</tr>
<tr>
<td>Flat Files</td>
<td>214</td>
</tr>
<tr>
<td>Flip-Flops</td>
<td>69</td>
</tr>
<tr>
<td>Floating Point</td>
<td>114</td>
</tr>
<tr>
<td>FooBar</td>
<td>154, 180</td>
</tr>
<tr>
<td>Formal Languages</td>
<td>104</td>
</tr>
<tr>
<td>Function</td>
<td>30, 33</td>
</tr>
<tr>
<td>Functional programming</td>
<td>104</td>
</tr>
<tr>
<td>Functions</td>
<td>8, 11, 112</td>
</tr>
<tr>
<td>Domain</td>
<td>8</td>
</tr>
<tr>
<td>Functional Notation</td>
<td>11</td>
</tr>
<tr>
<td>Range</td>
<td>8</td>
</tr>
<tr>
<td>Halting Problem</td>
<td>18</td>
</tr>
<tr>
<td>Hardware</td>
<td>65, 95</td>
</tr>
<tr>
<td>Accumulator</td>
<td>96</td>
</tr>
<tr>
<td>Address Bus</td>
<td>95</td>
</tr>
<tr>
<td>ALU</td>
<td>95</td>
</tr>
<tr>
<td>Command Wire</td>
<td>95</td>
</tr>
<tr>
<td>CPU</td>
<td>95</td>
</tr>
<tr>
<td>Data Bus</td>
<td>95</td>
</tr>
<tr>
<td>Fetch-Execute Cycle</td>
<td>95</td>
</tr>
<tr>
<td>RAM</td>
<td>95</td>
</tr>
<tr>
<td>Register</td>
<td>96</td>
</tr>
<tr>
<td>Harvard Architecture</td>
<td>74</td>
</tr>
<tr>
<td>Hello World</td>
<td>99</td>
</tr>
<tr>
<td>Hypothesis Testing</td>
<td>260</td>
</tr>
<tr>
<td>Identifier</td>
<td>114</td>
</tr>
<tr>
<td>Imperative Language</td>
<td>104</td>
</tr>
<tr>
<td>Imperative Programming</td>
<td></td>
</tr>
<tr>
<td>Modular Programming</td>
<td>173</td>
</tr>
<tr>
<td>Object Oriented programming</td>
<td>173</td>
</tr>
<tr>
<td>Procedural programming</td>
<td>173</td>
</tr>
<tr>
<td>Structured programming</td>
<td>173</td>
</tr>
</tbody>
</table>
Imperative programming, 173
Information, 4, 45
Information Gain, 206
Information Hiding, 191
Information Processing, 72
Information Technology, 73
Information Theory, 38
  Bit, 44
  Claude Shannon, 39
  Conditional Entropy, 46
  Hartley Information Entropy, 46
  Hartley’s Information Theory, 41
  Information, 44
  Ralph Hartley, 39
  Shannon Information, 42
  Shannon Information Entropy, 46
  Stochastic Process, 42
  Unit of Information, 41
Inheritance, 191
Instruction Set, 96
  Complex Instruction Set Computer (CISC), 96
  One Instruction Set Computer (OISC), 96
  Reduced Instruction Set Computer (RISC), 96
Interpreter, 105
Jacquard Loom, 92
Knowledge, 5, 45
Konrad Zuse, 92
Lady Ada Lovelace, 92
Lists, 141
Literals, 112
Logic
  Argumentation, 26
  Aristotelian Categories, 26, 175
  Formal Logic, 25, 26
  Predicate, 26
  Predicate Logic, 26
  Propositional Logic, 26
  Syllogisms, 26
  Truth Table, 26
Logic Gate, 67
Logic Gates, 69
  AND, 69
  NAND, 69
  NOT, 69
  OR, 69
Machine Learning, 229, 243
  Accuracy Paradox, 253
  Bias-Variance Tradeoff, 253
  Classification, 232, 241
  Cross-Validation, 253
  Curse of Dimensionality, 251
  Data Cleaning, 232
  Data Mining, 247
  Data Munging, 232
  Deep Learning, 247
  Dependent Variable, 232
  Dimensionality, 232, 241
  Feature Engineering, 232, 253
  Feature Selection, 232, 253
  Final Model, 241
  Final Prediction, 241
  Generalizing, 251
  Hyperparameters, 241
  Independent Variable, 232
  Instances, 232
  k-Means, 241, 243
  Label, 232
  Learn, 232
  Learned Model, 235
  Model, 232
  Observations, 232
  Overfitting, 251
  Parameters, 253
  Pre-Processing, 232
  Predictions, 232
  Predictor Variables, 232
  Samples, 232
  Statistics, 247
  Supervised Learning, 232, 241
  Target Variable, 232
  Testing, 235
  Testing Dataset, 241
  Training, 235
  Training Dataset, 241
  Unseen Dataset, 241
<table>
<thead>
<tr>
<th>Term</th>
<th>Page(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unsupervised Learning</td>
<td>241, 243</td>
</tr>
<tr>
<td>Validation Dataset</td>
<td>241</td>
</tr>
<tr>
<td>Markov chain Monte Carlo (MCMC)</td>
<td>266</td>
</tr>
<tr>
<td>Master Data</td>
<td>206</td>
</tr>
<tr>
<td>Master Data Management</td>
<td>206</td>
</tr>
<tr>
<td>Method Overloading</td>
<td>191</td>
</tr>
<tr>
<td>Method Overriding</td>
<td>191</td>
</tr>
<tr>
<td>Minimal Instruction Set</td>
<td>96</td>
</tr>
<tr>
<td>Model, 60, 203, 204</td>
<td></td>
</tr>
<tr>
<td>Conceptual Model, 204</td>
<td></td>
</tr>
<tr>
<td>Data Model, 204</td>
<td></td>
</tr>
<tr>
<td>Logical Model, 204</td>
<td></td>
</tr>
<tr>
<td>Physical Model, 204</td>
<td></td>
</tr>
<tr>
<td>Modules, 173</td>
<td></td>
</tr>
<tr>
<td>Monte Carlo, 169</td>
<td></td>
</tr>
<tr>
<td>Mutable, 114</td>
<td></td>
</tr>
<tr>
<td>Natural Languages, 104</td>
<td></td>
</tr>
<tr>
<td>NHST</td>
<td></td>
</tr>
<tr>
<td>Alternative Hypothesis, 260</td>
<td></td>
</tr>
<tr>
<td>Conditional Probability, 260</td>
<td></td>
</tr>
<tr>
<td>Null Hypothesis, 260</td>
<td></td>
</tr>
<tr>
<td>p-Value, 260</td>
<td></td>
</tr>
<tr>
<td>Significance Level, 260</td>
<td></td>
</tr>
<tr>
<td>Normalization, 220</td>
<td></td>
</tr>
<tr>
<td>Association Table, 220</td>
<td></td>
</tr>
<tr>
<td>Bridge Table, 220</td>
<td></td>
</tr>
<tr>
<td>Denormalization, 220</td>
<td></td>
</tr>
<tr>
<td>Functional Dependence, 220</td>
<td></td>
</tr>
<tr>
<td>Intersection Table, 220</td>
<td></td>
</tr>
<tr>
<td>Multi-Valued Attributes, 220</td>
<td></td>
</tr>
<tr>
<td>Partial Dependency, 220</td>
<td></td>
</tr>
<tr>
<td>Repeating Groups, 220</td>
<td></td>
</tr>
<tr>
<td>Transitive Dependency, 220</td>
<td></td>
</tr>
<tr>
<td>Null Hypothesis Significance Testing, 260</td>
<td></td>
</tr>
<tr>
<td>Null Hypothesis Significance Testing (NHST)</td>
<td>260</td>
</tr>
<tr>
<td>Number Representations, 64</td>
<td></td>
</tr>
<tr>
<td>Object-Oriented programming, 104</td>
<td></td>
</tr>
<tr>
<td>Object-Oriented Programming</td>
<td></td>
</tr>
<tr>
<td>Dot Operator, 141</td>
<td></td>
</tr>
<tr>
<td>Methods, 141</td>
<td></td>
</tr>
<tr>
<td>Objects, 141</td>
<td></td>
</tr>
<tr>
<td>Observations, 3</td>
<td></td>
</tr>
<tr>
<td>OLAP, 218</td>
<td></td>
</tr>
<tr>
<td>OLTP, 218</td>
<td></td>
</tr>
<tr>
<td>OOP</td>
<td></td>
</tr>
<tr>
<td>Aggregation, 193</td>
<td></td>
</tr>
<tr>
<td>Association, 193</td>
<td></td>
</tr>
<tr>
<td>Composition, 193</td>
<td></td>
</tr>
<tr>
<td>Generalization, 193</td>
<td></td>
</tr>
<tr>
<td>Operands, 114</td>
<td></td>
</tr>
<tr>
<td>Operator, 114</td>
<td></td>
</tr>
<tr>
<td>Assignment Operator, 114</td>
<td></td>
</tr>
<tr>
<td>Dot Operator, 107</td>
<td></td>
</tr>
<tr>
<td>Optimization, 229</td>
<td></td>
</tr>
<tr>
<td>Gradient Descent, 229</td>
<td></td>
</tr>
<tr>
<td>Loss Function, 229</td>
<td></td>
</tr>
<tr>
<td>Objective Function, 229</td>
<td></td>
</tr>
<tr>
<td>Simulated Annealing, 229</td>
<td></td>
</tr>
<tr>
<td>Stochastic Gradient Descent, 229</td>
<td></td>
</tr>
<tr>
<td>Parameters</td>
<td></td>
</tr>
<tr>
<td>Actual Parameters, 147</td>
<td></td>
</tr>
<tr>
<td>Arguments, 147</td>
<td></td>
</tr>
<tr>
<td>Formal Parameters, 147</td>
<td></td>
</tr>
<tr>
<td>Persistence, 161</td>
<td></td>
</tr>
<tr>
<td>Physical Data Independence, 210</td>
<td></td>
</tr>
<tr>
<td>Physics</td>
<td></td>
</tr>
<tr>
<td>Boltzmann Thermodynamic Entropy, 48</td>
<td></td>
</tr>
<tr>
<td>Clausius Thermodynamic Entropy, 47</td>
<td></td>
</tr>
<tr>
<td>Entropy, 47</td>
<td></td>
</tr>
<tr>
<td>Gibbs Thermodynamic Entropy, 48</td>
<td></td>
</tr>
<tr>
<td>Heat Engine, 47</td>
<td></td>
</tr>
<tr>
<td>Macrostate, 48</td>
<td></td>
</tr>
<tr>
<td>Maxwell’s Demon, 52</td>
<td></td>
</tr>
<tr>
<td>Microstate, 48</td>
<td></td>
</tr>
<tr>
<td>Polymorphism, 191</td>
<td></td>
</tr>
<tr>
<td>Precision, 59</td>
<td></td>
</tr>
<tr>
<td>Prescriptive Analytics, 255</td>
<td></td>
</tr>
<tr>
<td>Probability, 23</td>
<td></td>
</tr>
<tr>
<td>Conditional Probability, 25</td>
<td></td>
</tr>
<tr>
<td>Confidence Interval, 24</td>
<td></td>
</tr>
<tr>
<td>Kolmogorov’s Axioms, 23</td>
<td></td>
</tr>
<tr>
<td>Population, 24</td>
<td></td>
</tr>
<tr>
<td>Random Sample, 24</td>
<td></td>
</tr>
<tr>
<td>Sample Space, 25</td>
<td></td>
</tr>
<tr>
<td>Problem Requirements, 74</td>
<td></td>
</tr>
<tr>
<td>Problem Space, 81</td>
<td></td>
</tr>
<tr>
<td>Problem Specifications, 74</td>
<td></td>
</tr>
</tbody>
</table>
Procedural programming, 104
Procedure, 12, 14
Procedures, see Sub-routines
Program, 91
Programming Languages, 21
  Binary, 76
  Declarative Programming Languages, 33
  Dynamically-Typed, 114
  Fifth Generation Language, 76
  First Generation Language, 76
  Fourth Generation Language, 76
  Functional Programming, 37
  High-Level Languages, 76
  Imperative Programming, 37
  Imperative Programming Languages, 33
  Machine Code, 76
  Machine Language, 76
  Second Generation Language, 76
  Strongly-Typed, 114
  Third Generation Language, 76
Project Triangle, 210
Pseudocode, 34, 35, 173
Python
  Dot Operator, 107
  Libraries, 107
  Modules, 107
Quantum Computer, 56
Query, 208
Query Language, 208
Random Variable, 46
Random Variables, 271
  Cumulative Distribution Function, 271
  Event Space, 271
  Outcome, 271
  Probability Density Function, 271
  Probability Distribution, 271
  Probability Distribution Function, 271
  Probability Mass Function, 271
  Sample Space, 271
Relation, 218
Relational Database Model, 213, 214, 218
Schema, 218
Script, 99
Self-Documenting Code, 180
Set, 8
  n-tuple, 8
  Ordered Set, 8
  Sequence, 8
  Series, 8
  Vector, 8
Shannon Information, 271
Shannon Information Entropy, 271
Side effects, 173
SIMULA, 169
Smalltalk, 169
Software, 65, 95
Software Engineering, 74
Source Code, 14, 104
SQL
  Ad-Hoc Query, 213
  State, 30, 33, 81, 103, 175
  State Space, 33
  State Variable, 33
  State Space, 81
  State Space Graph, 81
  Statements, 112
Streams, 161
  Standard Error (STDERR), 118, 161
  Standard Input (STDIN), 118, 161
  Standard Output (STDOUT), 118, 161
Strings, 139
Structured programming, 104
Structured Query Language (SQL), 210
Sub-routine, 147, 173
  Call, 150
  Flow of Execution, 155
  Functions, 147, 173
  Invoke, 150
  Procedures, 147, 173
Switch, 67
Symbol Table, 114
System Development Life Cycle (SDLC), 210
Systems Analyst, 74
Theory
  Context-Free Grammar, 71
  Finite State Automata, 71
  Formal Language Theory, 71
  Grammar, 71
  Regular Language, 71
INDEX

Thinking
  Computational Thinking, 30
  Abstraction, 30
  Algorithmic Expression, 30
  Algorithmic Thinking, 30
  Models, 30
  Pattern Recognition, 30
  Problem Decomposition, 30
  Problem Specification, 30
  Solution Evaluation, 30
  Solution Implementation, 30
  Declarative Statements, 28
  Deductive Reasoning, 22
  Imperative Statements, 28
  Inductive Reasoning, 22
  Probabilistic Inference, 22
Translation, 74
  Compilation, 74
  Interpretation, 74
Translator, 98, 105
Tuples, 141
  Comparable, 142
  Hashable, 142
Turing Machine, 83
  Turing Machines, see Universal Turing Machine, 16, 81, 227
  Type, 111, 135, 142, 195
  Typecasting, 114
  Unified Modeling Language (UML), 193
  Universal Turing Machine, 14
  Use Cases, 206
  Values, 3
  Variable, 30, 33
    State Variable, 33
  Variables, 11, 112
    Dependent Variable, 11
    Global, 155
    Independent Variable, 11
    Lifetime, 155
    Local, 155
    Namespace, 155
    Scope, 155
    Virtual Machine, 105
    Virtual Machine Interpreter, 105
    von Neumann Architecture, 74
    Work, 56
Ricky J. Sethi is an Associate Professor of Computer Science at Fitchburg State University. Ricky is also Director of Research for the Madsci Network and Team Lead for SNHU Online at Southern New Hampshire University.

Prior to that, he was a Research Scientist at UMass Amherst/U-Mass Medical School and at UCLA/USC Information Sciences Institute, where he was chosen as an NSF Computing Innovation Fellow (CIFellow) by the CCC and the CRA. Even earlier, he was a Post-Doctoral Scholar at UCR, where he was the Lead Integration Scientist for the WASA project and participated in ONR’s Empire Challenge 10.

Ricky has authored or co-authored over 30 peer-reviewed papers, book chapters, and reports and made numerous presentations on his research in machine learning, computer vision, social computing, and data science.

Ricky’s work has been generously funded by the National Science Foundation (NSF), the National Endowment for the Humanities (NEH), the Institute for Advanced Study (IAS), and Amazon. He has also served as a panelist on several NSF programs, an Editorial Board Member for the International Journal of Computer Vision & Signal Processing, and an organizer and program committee member for various conferences.

You can find out more at his website http://research.sethi.org/ricky/