

Math for Programmers

**3D GRAPHICS, MACHINE LEARNING AND
SIMULATIONS WITH PYTHON**

PAUL ORLAND



MANNING
SHELTER ISLAND

For online information and ordering of this and other Manning books, please visit www.manning.com. The publisher offers discounts on this book when ordered in quantity. For more information, please contact


Special Sales Department
Manning Publications Co.
20 Baldwin Road
PO Box 761
Shelter Island, NY 11964
Email: orders@manning.com

©2020 by Manning Publications Co. All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by means electronic, mechanical, photocopying, or otherwise, without prior written permission of the publisher.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in the book, and Manning Publications was aware of a trademark claim, the designations have been printed in initial caps or all caps.

© Recognizing the importance of preserving what has been written, it is Manning's policy to have the books we publish printed on acid-free paper, and we exert our best efforts to that end. Recognizing also our responsibility to conserve the resources of our planet, Manning books are printed on paper that is at least 15 percent recycled and processed without the use of elemental chlorine.

 Manning Publications Co.
20 Baldwin Road
PO Box 761
Shelter Island, NY 11964

Development editor: Jenny Stout
Technical development editor: Kris Athi
Review editor: Aleks Dragosavljević
Production editor: Lori Weidert
Copy editor: Frances Buran
Proofreader: Jason Everett
Technical proofreader: Mike Shepard
Typesetter and cover designer: Marija Tudor

ISBN 9781617295355
Printed in the United States of America

*To my first math teacher and
my first programming teacher—Dad.*

brief contents

1	■	Learning math with code	1
PART 1		VECTORS AND GRAPHICS	19
2	■	Drawing with 2D vectors	21
3	■	Ascending to the 3D world	75
4	■	Transforming vectors and graphics	121
5	■	Computing transformations with matrices	158
6	■	Generalizing to higher dimensions	205
7	■	Solving systems of linear equations	257
PART 2		CALCULUS AND PHYSICAL SIMULATION	301
8	■	Understanding rates of change	303
9	■	Simulating moving objects	337
10	■	Working with symbolic expressions	354
11	■	Simulating force fields	392
12	■	Optimizing a physical system	422
13	■	Analyzing sound waves with a Fourier series	463
PART 3		MACHINE LEARNING APPLICATIONS	497
14	■	Fitting functions to data	499
15	■	Classifying data with logistic regression	526
16	■	Training neural networks	559

contents

<i>preface</i>	xvii
<i>acknowledgments</i>	xxi
<i>about this book</i>	xxiii
<i>about the author</i>	xxviii
<i>about the cover illustration</i>	xxix

1 *Learning math with code* 1

1.1	Solving lucrative problems with math and software	2
	<i>Predicting financial market movements</i>	2
	<i>Finding a good deal</i>	5
	<i>Building 3D graphics and animations</i>	7
	<i>Modeling the physical world</i>	9
1.2	How not to learn math	11
	<i>Jane wants to learn some math</i>	12
	<i>Slogging through math textbooks</i>	13
1.3	Using your well-trained left brain	13
	<i>Using a formal language</i>	14
	<i>Build your own calculator</i>	15
	<i>Building abstractions with functions</i>	16

PART 1 VECTORS AND GRAPHICS 19

2 *Drawing with 2D vectors* 21

2.1	Picturing 2D vectors	22
	<i>Representing 2D vectors</i>	24
	<i>2D drawing in Python</i>	26
	<i>Exercises</i>	29

2.2 Plane vector arithmetic 32

Vector components and lengths 35 ▪ *Multiplying vectors by numbers* 37 ▪ *Subtraction, displacement, and distance* 39
Exercises 42

2.3 Angles and trigonometry in the plane 51

From angles to components 52 ▪ *Radians and trigonometry in Python* 56 ▪ *From components back to angles* 57
Exercises 60

2.4 Transforming collections of vectors 67

Combining vector transformations 69 ▪ *Exercises* 70

2.5 Drawing with Matplotlib 72

3 *Ascending to the 3D world* 75

3.1 Picturing vectors in 3D space 77

Representing 3D vectors with coordinates 79 ▪ *3D drawing with Python* 80 ▪ *Exercises* 82

3.2 Vector arithmetic in 3D 83

Adding 3D vectors 83 ▪ *Scalar multiplication in 3D* 85
Subtracting 3D vectors 85 ▪ *Computing lengths and distances* 86
Computing angles and directions 87 ▪ *Exercises* 89

3.3 The dot product: Measuring vector alignment 92

Picturing the dot product 93 ▪ *Computing the dot product* 95
Dot products by example 97 ▪ *Measuring angles with the dot product* 97 ▪ *Exercises* 100

3.4 The cross product: Measuring oriented area 103

Orienting ourselves in 3D 103 ▪ *Finding the direction of the cross product* 106 ▪ *Finding the length of the cross product* 108
Computing the cross product of 3D vectors 109 ▪ *Exercises* 110

3.5 Rendering a 3D object in 2D 114

Defining a 3D object with vectors 114 ▪ *Projecting to 2D* 116
Orienting faces and shading 116 ▪ *Exercises* 119

4 *Transforming vectors and graphics* 121

4.1 Transforming 3D objects 123

Drawing a transformed object 124 ▪ *Composing vector transformations* 126 ▪ *Rotating an object about an axis* 129
Inventing your own geometric transformations 131
Exercises 134

4.2 Linear transformations 138

Preserving vector arithmetic 138 ▪ *Picturing linear transformations* 140 ▪ *Why linear transformations?* 142
Computing linear transformations 146 ▪ *Exercises* 149

5 *Computing transformations with matrices* 158

5.1 Representing linear transformations with matrices 159

Writing vectors and linear transformations as matrices 159
Multiplying a matrix with a vector 161 ▪ *Composing linear transformations by matrix multiplication* 163 ▪ *Implementing matrix multiplication* 166 ▪ *3D animation with matrix transformations* 166 ▪ *Exercises* 169

5.2 Interpreting matrices of different shapes 175

Column vectors as matrices 176 ▪ *What pairs of matrices can be multiplied?* 178 ▪ *Viewing square and non-square matrices as vector functions* 180 ▪ *Projection as a linear map from 3D to 2D* 181 ▪ *Composing linear maps* 184
Exercises 186

5.3 Translating vectors with matrices 191

Making plane translations linear 191 ▪ *Finding a 3D matrix for a 2D translation* 194 ▪ *Combining translation with other linear transformations* 195 ▪ *Translating 3D objects in a 4D world* 196 ▪ *Exercises* 199

6 *Generalizing to higher dimensions* 205

6.1 Generalizing our definition of vectors 206

Creating a class for 2D coordinate vectors 207 ▪ *Improving the Vec2 class* 208 ▪ *Repeating the process with 3D vectors* 209
Building a vector base class 210 ▪ *Defining vector spaces* 212
Unit testing vector space classes 214 ▪ *Exercises* 216

6.2 Exploring different vector spaces 219

Enumerating all coordinate vector spaces 219 ▪ *Identifying vector spaces in the wild* 221 ▪ *Treating functions as vectors* 223
Treating matrices as vectors 226 ▪ *Manipulating images with vector operations* 227 ▪ *Exercises* 230

6.3 Looking for smaller vector spaces 237

Identifying subspaces 238 ▪ *Starting with a single vector* 240
Spanning a bigger space 240 ▪ *Defining the word dimension* 243 ▪ *Finding subspaces of the vector space of functions* 244 ▪ *Subspaces of images* 245 ▪ *Exercises* 248

7 Solving systems of linear equations 257

- 7.1 Designing an arcade game 258
 - Modeling the game* 259 ▪ *Rendering the game* 260
 - Shooting the laser* 261 ▪ *Exercises* 262
- 7.2 Finding intersection points of lines 263
 - Choosing the right formula for a line* 263 ▪ *Finding the standard form equation for a line* 265 ▪ *Linear equations in matrix notation* 267 ▪ *Solving linear equations with NumPy* 268
 - Deciding whether the laser hits an asteroid* 270 ▪ *Identifying unsolvable systems* 271 ▪ *Exercises* 273
- 7.3 Generalizing linear equations to higher dimensions 278
 - Representing planes in 3D* 278 ▪ *Solving linear equations in 3D* 280 ▪ *Studying hyperplanes algebraically* 282 ▪ *Counting dimensions, equations, and solutions* 283 ▪ *Exercises* 285
- 7.4 Changing basis by solving linear equations 294
 - Solving a 3D example* 296 ▪ *Exercises* 297

PART 2 CALCULUS AND PHYSICAL SIMULATION 301

8 Understanding rates of change 303

- 8.1 Calculating average flow rate from volume 305
 - Implementing an average_flow_rate function* 305 ▪ *Picturing the average flow rate with a secant line* 306 ▪ *Negative rates of change* 308 ▪ *Exercises* 309
- 8.2 Plotting the average flow rate over time 310
 - Finding the average flow rate in different time intervals* 310
 - Plotting the interval flow rates* 311 ▪ *Exercises* 313
- 8.3 Approximating instantaneous flow rates 315
 - Finding the slope of small secant lines* 315 ▪ *Building the instantaneous flow rate function* 318 ▪ *Currying and plotting the instantaneous flow rate function* 320 ▪ *Exercises* 322
- 8.4 Approximating the change in volume 323
 - Finding the change in volume for a short time interval* 323
 - Breaking up time into smaller intervals* 324 ▪ *Picturing the volume change on the flow rate graph* 325 ▪ *Exercises* 328
- 8.5 Plotting the volume over time 328
 - Finding the volume over time* 328 ▪ *Picturing Riemann sums for the volume function* 329 ▪ *Improving the approximation* 332
 - Definite and indefinite integrals* 334

9 *Simulating moving objects* 337

- 9.1 Simulating a constant velocity motion 338
 - Adding velocities to the asteroids* 339 ▪ *Updating the game engine to move the asteroids* 339 ▪ *Keeping the asteroids on the screen* 340 ▪ *Exercises* 342
- 9.2 Simulating acceleration 342
 - Accelerating the spaceship* 343
- 9.3 Digging deeper into Euler's method 344
 - Carrying out Euler's method by hand* 344 ▪ *Implementing the algorithm in Python* 346
- 9.4 Running Euler's method with smaller time steps 348
 - Exercises* 349

10 *Working with symbolic expressions* 354

- 10.1 Finding an exact derivative with a computer algebra system 355
 - Doing symbolic algebra in Python* 356
- 10.2 Modeling algebraic expressions 358
 - Breaking an expression into pieces* 358 ▪ *Building an expression tree* 359 ▪ *Translating the expression tree to Python* 360
 - Exercises* 362
- 10.3 Putting a symbolic expression to work 365
 - Finding all the variables in an expression* 365 ▪ *Evaluating an expression* 366 ▪ *Expanding an expression* 369 ▪ *Exercises* 372
- 10.4 Finding the derivative of a function 374
 - Derivatives of powers* 374 ▪ *Derivatives of transformed functions* 375 ▪ *Derivatives of some special functions* 377
 - Derivatives of products and compositions* 378 ▪ *Exercises* 379
- 10.5 Taking derivatives automatically 381
 - Implementing a derivative method for expressions* 382
 - Implementing the product rule and chain rule* 383
 - Implementing the power rule* 384 ▪ *Exercises* 386
- 10.6 Integrating functions symbolically 387
 - Integrals as antiderivatives* 387 ▪ *Introducing the SymPy library* 388 ▪ *Exercises* 389

11 *Simulating force fields* 392

- 11.1 Modeling gravity with a vector field 393
 - Modeling gravity with a potential energy function* 394

- 11.2 Modeling gravitational fields 396
 - Defining a vector field* 396 ▪ *Defining a simple force field* 398
- 11.3 Adding gravity to the asteroid game 399
 - Making game objects feel gravity* 400 ▪ *Exercises* 403
- 11.4 Introducing potential energy 404
 - Defining a potential energy scalar field* 405 ▪ *Plotting a scalar field as a heatmap* 407 ▪ *Plotting a scalar field as a contour map* 407
- 11.5 Connecting energy and forces with the gradient 408
 - Measuring steepness with cross sections* 409 ▪ *Calculating partial derivatives* 411 ▪ *Finding the steepness of a graph with the gradient* 413 ▪ *Calculating force fields from potential energy with the gradient* 415 ▪ *Exercises* 418

12 *Optimizing a physical system* 422

- 12.1 Testing a projectile simulation 425
 - Building a simulation with Euler's method* 426 ▪ *Measuring properties of the trajectory* 427 ▪ *Exploring different launch angles* 428 ▪ *Exercises* 429
- 12.2 Calculating the optimal range 432
 - Finding the projectile range as a function of the launch angle* 432
 - Solving for the maximum range* 435 ▪ *Identifying maxima and minima* 437 ▪ *Exercises* 439
- 12.3 Enhancing our simulation 440
 - Adding another dimension* 441 ▪ *Modeling terrain around the cannon* 442 ▪ *Solving for the range of the projectile in 3D* 443
 - Exercises* 447
- 12.4 Optimizing range using gradient ascent 449
 - Plotting range versus launch parameters* 449 ▪ *The gradient of the range function* 450 ▪ *Finding the uphill direction with the gradient* 451 ▪ *Implementing gradient ascent* 453
 - Exercises* 457

13 *Analyzing sound waves with a Fourier series* 463

- 13.1 Combining sound waves and decomposing them 465
- 13.2 Playing sound waves in Python 466
 - Producing our first sound* 467 ▪ *Playing a musical note* 469
 - Exercises* 471
- 13.3 Turning a sinusoidal wave into a sound 471

Making audio from sinusoidal functions 471 ▪ *Changing the frequency of a sinusoid* 473 ▪ *Sampling and playing the sound wave* 475 ▪ *Exercises* 477

13.4 Combining sound waves to make new ones 478

Adding sampled sound waves to build a chord 478 ▪ *Picturing the sum of two sound waves* 479 ▪ *Building a linear combination of sinusoids* 481 ▪ *Building a familiar function with sinusoids* 483 ▪ *Exercises* 486

13.5 Decomposing a sound wave into its Fourier series 486

Finding vector components with an inner product 487 ▪ *Defining an inner product for periodic functions* 488 ▪ *Writing a function to find Fourier coefficients* 490 ▪ *Finding the Fourier coefficients for the square wave* 491 ▪ *Fourier coefficients for other waveforms* 492 ▪ *Exercises* 494

PART 3 MACHINE LEARNING APPLICATIONS 497

14 *Fitting functions to data* 499

14.1 Measuring the quality of fit for a function 502

Measuring distance from a function 503 ▪ *Summing the squares of the errors* 505 ▪ *Calculating cost for car price functions* 507 ▪ *Exercises* 510

14.2 Exploring spaces of functions 511

Picturing cost for lines through the origin 512 ▪ *The space of all linear functions* 514 ▪ *Exercises* 515

14.3 Finding the line of best fit using gradient descent 515

Rescaling the data 516 ▪ *Finding and plotting the line of best fit* 516 ▪ *Exercises* 518

14.4 Fitting a nonlinear function 519

Understanding the behavior of exponential functions 519 ▪ *Finding the exponential function of best fit* 521 ▪ *Exercises* 523

15 *Classifying data with logistic regression* 526

15.1 Testing a classification function on real data 528

Loading the car data 529 ▪ *Testing the classification function* 529 ▪ *Exercises* 530

15.2 Picturing a decision boundary 532

Picturing the space of cars 532 ▪ *Drawing a better decision boundary* 533 ▪ *Implementing the classification function* 534 ▪ *Exercises* 535

- 15.3 Framing classification as a regression problem 536
 - Scaling the raw car data* 536 ■ *Measuring the “BMWness” of a car* 538 ■ *Introducing the sigmoid function* 540 ■ *Composing the sigmoid function with other functions* 541 ■ *Exercises* 543
- 15.4 Exploring possible logistic functions 544
 - Parameterizing logistic functions* 545 ■ *Measuring the quality of fit for a logistic function* 546 ■ *Testing different logistic functions* 548 ■ *Exercises* 549
- 15.5 Finding the best logistic function 551
 - Gradient descent in three dimensions* 551 ■ *Using gradient descent to find the best fit* 552 ■ *Testing and understanding the best logistic classifier* 554 ■ *Exercises* 555

16 Training neural networks 559

- 16.1 Classifying data with neural networks 561
- 16.2 Classifying images of handwritten digits 562
 - Building the 64-dimensional image vectors* 563 ■ *Building a random digit classifier* 565 ■ *Measuring performance of the digit classifier* 566 ■ *Exercises* 567
- 16.3 Designing a neural network 568
 - Organizing neurons and connections* 568 ■ *Data flow through a neural network* 569 ■ *Calculating activations* 572
 - Calculating activations in matrix notation* 574 ■ *Exercises* 576
- 16.4 Building a neural network in Python 577
 - Implementing an MLP class in Python* 578 ■ *Evaluating the MLP* 580 ■ *Testing the classification performance of an MLP* 581 ■ *Exercises* 582
- 16.5 Training a neural network using gradient descent 582
 - Framing training as a minimization problem* 582 ■ *Calculating gradients with backpropagation* 584 ■ *Automatic training with scikit-learn* 585 ■ *Exercises* 586
- 16.6 Calculating gradients with backpropagation 588
 - Finding the cost in terms of the last layer weights* 589
 - Calculating the partial derivatives for the last layer weights using the chain rule* 590 ■ *Exercises* 591

appendix A Getting set up with Python 595

appendix B Python tips and tricks 607

appendix C Loading and rendering 3D Models with OpenGL and PyGame 635

index 645

preface

I started working on this book in 2017, when I was CTO of Tachyus, a company I founded that builds predictive analytics software for oil and gas companies. By that time, we had finished building our core product: a fluid-flow simulator powered by physics and machine learning, along with an optimization engine. These tools let our customers look into the future of their oil reservoirs and helped them to discover hundreds of millions of dollars of optimization opportunities.

My task as CTO was to productize and scale-out this software as some of the biggest companies in the world began to use it. The challenge was that this was not only a complex software project, but the code was very mathematical. Around that time, we started hiring for a position called “scientific software engineer,” with the idea that we needed skilled professional software engineers who also had solid backgrounds in math, physics, and machine learning. In the process of searching for and hiring scientific software engineers, I realized that this combination was both rare and in high demand. Our software engineers realized this as well and were eager to hone their math skills to contribute to our specialized back-end components of our stack. With eager math learners on our team already, as well as in our hiring pipeline, I started to think about the best way to train a strong software engineer to become a formidable math user.

I realized there were no books with the right math content, presented at the right level. While there are probably hundreds of books and thousands of free online articles on topics like linear algebra and calculus, I’m not aware of any I could hand to a typical professional software engineer, and expect them to come back in a few months having mastered the material. I don’t say this to disparage software engineers, I just mean that reading and understanding math books is a difficult skill to learn on its own. To do so, you often need to figure out what specific topics you need to learn

(which is hard if you don't know anything about the material yet!), read them, and then choose some high quality exercises to practice applying those topics. If you were less discerning, you could read every word of a textbook and solve *all* of its exercises, but it could take months of full-time study to do that!

With *Math for Programmers*, I hope to offer an alternative. I believe it's possible to read this book cover-to-cover in a reasonable amount of time, including completing all the exercises, and then to walk away having mastered some key math concepts.

How this book was designed

In the fall of 2017, I got in touch with Manning and learned that they were interested in publishing this book. That started a long process of converting my vision for this book into a concrete plan, which was much more difficult than I imagined, being a first-time author. Manning asked some hard questions of my original table of contents, like

- Will anyone be interested in this topic?
- Will this be too abstract?
- Can you really teach a semester of calculus in one chapter?

All of these questions forced me to think a lot more carefully about what was achievable. I'll share some of the ways we answered these questions because they'll help you understand exactly how this book works.

First, I decided to focus this book around one core skill—expressing mathematical ideas in code. I think this is a great way to learn math, even if you aren't a programmer by trade. When I was in high school, I learned to program on my TI-84 graphing calculator. I had the grand idea that I could write programs to do my math and science homework for me, giving me the right answer *and* outputting the steps along the way. As you might expect, this was more difficult than just doing my homework in the first place, but it gave me some useful perspective. For any kind of problem I wanted to program, I had to clearly understand the inputs and outputs, and what happened in each of the steps of the solution. By the end, I was sure I knew the material, and I had a working program to prove it.

That's the experience I'll try to share with you in this book. Each chapter is organized around a tangible example program, and to get it working, you need to put all the mathematical pieces together correctly. Once you're done, you'll have confidence that you've understood the concept and can apply it again in the future. I've included plenty of exercises to help you check your understanding on the math and code I've included, as well as mini-projects which invite you to experiment with new variations on the material.

Another question I discussed with Manning was what programming language I should use for the examples. Originally, I wanted to write the book in a functional programming language because math is a functional language itself. After all, the concept of a "function" originated in math, long before computers even existed. In various parts of math, you have functions that return other functions like integrals and

derivatives in calculus. However, asking readers to learn an unfamiliar language like LISP, Haskell, or F# *while* learning new math concepts would make the book more difficult and less accessible. Instead, we settled on Python, a popular, easy-to-learn language with great mathematical libraries. Python also happens to be a favorite for “real world” users of math in academia and in industry.

The last major question that I had to answer with Manning was what specific math topics I would include and which ones wouldn’t make the cut. This was a difficult decision, but at least we agreed on the title *Math for Programmers*, the broadness of which gave us some flexibility for what to include. My main criterion became the following: this was going to be “Math for Programmers,” not “Math for Computer Scientists.” With that in mind, I could leave out topics like discrete math, combinatorics, graphs, logic, Big O notation, and so on, that are covered in computer science classes and mostly used to *study* programs.

Even with that decision made, there was still plenty of math to choose from. Ultimately, I chose to focus on linear algebra and calculus. I have some strong pedagogical views on these subjects, and there are plenty of good example applications in both that can be visual and interactive. You can write a big textbook on either linear algebra *or* calculus alone, so I had to get even more specific. To do that, I decided the book would build up to some applications in the trendy field of machine learning. With those decisions made, the contents of the book became clearer.

Mathematical ideas we cover

This book covers a lot of mathematical topics, but there are a few major themes. Here are a few that you can keep an eye out for as you start reading:

- *Multi-dimensional spaces*—Intuitively, you probably have a sense what the words two-dimensional (2D) and three-dimensional (3D) mean. We live in a 3D world, while a 2D world is flat like a piece of paper or a computer screen. A location in 2D can be described by two numbers (often called x and y -coordinates), while you need three numbers to identify a location in 3D. We can’t picture a 17-dimensional space, but we can describe its points by lists of 17 numbers. Lists of numbers like these are called *vectors*, and vector math helps illuminate the notion of “dimension.”
- *Spaces of functions*—Sometimes a list of numbers can specify a function. With two numbers like $a = 5$ and $b = 13$, you can create a (linear) function of the form $f(x) = ax + b$, and in this case, the function would be $f(x) = 5x + 13$. For every point in 2D space, labeled by coordinates (a, b) , there’s a linear function that goes with it. So we can think of the set of all linear functions as a 2D space.
- *Derivatives and gradients*—These are calculus operations that measure the rates of change of functions. The *derivative* tells you how rapidly a function $f(x)$ is increasing or decreasing as you increase the input value x . A function in 3D might look like $f(x, y)$ and can increase or decrease as you change the values of either x or y . Thinking of (x, y) pairs as points in a 2D space, you could ask what

direction you could go in this 2D space to make f increase most rapidly. The gradient answers this question.

- *Optimizing a function*—For a function of the form $f(x)$ or $f(x, y)$, you could ask an even broader version of the previous question: what inputs to the function yield the biggest output? For $f(x)$, the answer would be some value x , and for $f(x, y)$, it would be a point in 2D. In the 2D case, the gradient can help us. If the gradient tells us $f(x, y)$ is increasing in some direction, we can find a maximum value of $f(x, y)$ if we explore in that direction. A similar strategy applies if you want to find a minimum value of a function.
- *Predicting data with functions*—Say you want to predict a number, like the price of a stock at a given time. You could create a function $p(t)$ that takes a time t and outputs a price p . The measure of predictive quality of your function is how close it comes to actual data. In that sense, finding a predictive function means minimizing the error between your function and real data. To do that, you need to explore a space of functions and find a minimum value. This is called *regression*.

I think this is a useful collection of mathematical concepts for anyone to have in their toolbelt. Even if you're not interested in machine learning, these concepts—and others in this book—have plenty of other applications.

The subjects I'm saddest to leave out of the book are probability and statistics. Probability and the concept of quantifying uncertainty in general is important in machine learning as well. This is a big book already, so there just wasn't time or room to squeeze a meaningful introduction for these topics. Stay tuned for a sequel to this book. There's a lot more fun and useful math out there, beyond what I've been able to cover in these pages, and I hope to be able to share it with you in the future.

acknowledgments

From start to finish, this book has taken about three years to create. I have gotten a lot of help in that time, and so I have quite a few people to thank and acknowledge.

First and foremost, I want to thank Manning for making this book happen. I'm grateful they bet on me to write a big, challenging book as a first-time author and had a lot of patience with me as the book fell behind schedule a few times. In particular, I want to thank Marjan Bace and Michael Stephens for pushing the project forward and for helping define what exactly it would be. My original development editor, Richard Wattenbarger, was also critical to keeping the book alive as we iterated on the content. I think he reviewed six total drafts of chapters 1 and 2 before we settled on how the book would be structured.

I wrote most of the book in 2019 under the expert guidance of my second editor, Jennifer Stout, who both got the project over the finish line and taught me a lot about technical writing. My technical editor, Kris Athi, and technical reviewer, Mike Shepard, also made it to the end with us, and thanks to them reading every word and line of code, we've caught and fixed countless errors. Outside of Manning, I got a lot of editing help from Michaela Leung, who also reviewed the whole book for grammatical and technical accuracy. I'd also like to thank the marketing team at Manning. With the MEAP program, we've been able to validate that this is a book people are interested in. It's been a great motivator to know a book will be at least a modest commercial success while working on the intensive final steps to get it published.

My current and former coworkers at Tachyus have taught me a lot about programming, and many of those lessons have made their way into this book. I credit Jack Fox for first getting me to think about the connections between functional programming and math, which comes up in chapters 4 and 5. Will Smith taught me about video game design, and we have had many good discussions about vector geometry for 3D

rendering. Most notably, Stelios Kyriacou taught me most of what I know about optimization algorithms and helped me get some of the code in this book to work. He also introduced me to the philosophy that “everything is an optimization problem,” a theme that you should pick up on in the latter half of the book.

To all the reviewers: Adhir Ramjiawan, Anto Aravinth, Christopher Haupt, Clive Harber, Dan Sheikh, David Ong, David Trimm, Emanuele Piccinelli, Federico Bertolucci, Frances Buontempo, German Gonzalez-Morris, James Nyika, Jens Christian B. Madsen, Johannes Van Nimwegen, Johnny Hopkins, Joshua Horwitz, Juan Rufes, Kenneth Fricklas, Laurence Giglio, Nathan Mische, Philip Best, Reka Horvath, Robert Walsh, Sébastien Portebois, Stefano Paluella, and Vincent Zhu, your suggestions helped make this a better book.

I’m by no means a machine learning expert, so I consulted a number of resources to make sure I introduced it correctly and effectively. I was most influenced by Andrew Ng’s “Machine Learning” course on Coursera and the “Deep Learning” series by 3Blue1Brown on YouTube. These are great resources, and if you’ve seen them, you’ll notice that part 3 of this book is influenced by the way they introduce the subject. I also need to thank Dan Rathbone, whose handy website CarGraph.com was the source of the data for many of my examples.

I also want to thank my wife Margaret, an astronomer, for introducing me to Jupyter notebooks. Switching the code for this book to Jupyter has made it much easier to follow. My parents have also been very supportive as I’ve written this book; on a few occasions, I’ve scrambled to get a chapter finished during a holiday visit with them. They also personally guaranteed that I would sell at least one copy (thanks, Mom!).

Finally, this book is dedicated to my Dad, who first showed me how to do math in code when he taught me how to program in APL when I was in fifth grade. If there’s a second edition of this book, I might enlist his help to rewrite all of the Python in a single line of APL code!

about this book

Math for Programmers teaches you how to solve mathematical problems with code using the Python programming language. Math skills are more and more important for professional software developers, especially as companies are staffing up teams for data science and machine learning. Math also plays an integral role in other modern applications like game development, computer graphics and animation, image and signal processing, pricing engines, and stock market analysis.

The book starts by introducing 2D and 3D vector geometry, vector spaces, linear transformations, and matrices; these are the bread and butter of the subject of linear algebra. In part 2, it introduces calculus with a focus on a few particularly useful subjects for programmers: derivatives, gradients, Euler's method, and symbolic evaluation. Finally, in part 3, all the pieces come together to show you how some important machine learning algorithms work. By the last chapter of the book, you'll have learned enough math to code-up your own neural network from scratch.

This isn't a textbook! It's designed to be a friendly introduction to material that can often seem intimidating, esoteric, or boring. Each chapter features a complete, real-world application of a mathematical concept, complemented by exercises to help you check your understanding as well as mini-projects to help you continue your exploration.

Who should read this book?

This book is for anyone with a solid programming background who wants to refresh their math skills or to learn more about applications of math in software. It doesn't require any previous exposure to calculus or linear algebra, just high-school level algebra and geometry (even if that feels long ago!). This book is designed to be read at

your keyboard. You'll get the most out of it if you follow along with the examples and try all the exercises.

How this book is organized

Chapter 1 invites you into the world of math. It covers some of the important applications of mathematics in computer programming, introduces some of the topics that appear in the book, and explains how programming can be a valuable tool to a math learner. After that, this book is divided into three parts:

- Part 1 focuses on vectors and linear algebra.
 - Chapter 2 covers vector math in 2D with an emphasis on using coordinates to define 2D graphics. It also contains a review of some basic trigonometry.
 - Chapter 3 extends the material of the previous chapter to 3D, where points are labeled by three coordinates instead of two. It introduces the dot product and cross product, which are helpful to measure angles and render 3D models.
 - Chapter 4 introduces linear transformations, functions that take vectors as inputs and return vectors as outputs and that have specific geometric effects like rotation or reflection.
 - Chapter 5 introduces matrices, which are arrays of numbers that can encode a linear vector transformation.
 - Chapter 6 extends the ideas from 2D and 3D so you can work with collections of vectors of *any* dimension. These are called vector spaces. As a main example, it covers how to process images using vector math.
 - Chapter 7 focuses on the most important computational problem in linear algebra: solving systems of linear equations. It applies this to a collision-detection system in a simple video game.
- Part 2 introduces calculus and applications to physics.
 - Chapter 8 introduces the concept of the rate of change of a function. It covers derivatives, which calculate a function's rate of change, and integrals, which recover a function from its rate of change.
 - Chapter 9 covers an important technique for approximate integration called Euler's method. It expands the game from chapter 7 to include moving and accelerating objects.
 - Chapter 10 shows how to manipulate algebraic expressions in code, including automatically finding the formula for the derivative of a function. It introduces symbolic programming, a different approach to doing math in code than used elsewhere in the book.
 - Chapter 11 extends the calculus topics to two-dimensions, defining the gradient operation and showing how it can be used to define a force field.
 - Chapter 12 shows how to use derivatives to find the maximum or minimum values of functions.

- Chapter 13 shows how to think of sound waves as functions, and how to decompose them into sums of other simpler functions, called Fourier series. It covers how to write Python code to play musical notes and chords.
- Part 3 combines the ideas from the first two parts to introduce some important ideas in machine learning.
 - Chapter 14 covers how to fit a line to 2D data, a process referred to as linear regression. The example we explore is finding a function to best predict the price of a used car based on its mileage.
 - Chapter 15 addresses a different machine learning problem: figuring out what model a car is based on some data about it. Figuring out what kind of object is represented by a data point is called classification.
 - Chapter 16 shows how to design and implement a neural network, a special kind of mathematical function, and use it to classify images. This chapter combines ideas from almost every preceding chapter.

Each chapter should be accessible if you’ve read and understand the previous ones. The cost of keeping all of the concepts in order is that the applications may seem eclectic. Hopefully the variety of examples make it an entertaining read, and show you the broad range of applications of the math we cover.

About the code

This book presents ideas in (hopefully) logical order. The ideas you learn in chapter 2 apply to chapter 3, then ideas in chapters 2 and 3 appear in chapter 4, and so on. Computer code is not always written “in order” like this. That is, the simplest ideas in a finished computer program are not always in the first lines of the first file of the source code. This difference makes it challenging to present source code for a book in an intelligible way.

My solution to this is to include a “walkthrough” code file in the form of a Jupyter notebook for each chapter. A Jupyter notebook is something like a recorded Python interactive session, with visuals like graphs and images built in. In a Jupyter notebook, you enter some code, run it, and then perhaps overwrite it later in your session as you develop your ideas. The notebook for each chapter has code for each section and sub-section, run in the same order as it appears in the book. Most importantly, this means you can run the code for the book as you read. You don’t need to get to the end of a chapter before your code is complete enough to work. Appendix A shows you how to set up Python and Jupyter, and appendix B includes some handy Python features if you’re new to the language.

This book contains many examples of source code both in numbered listings and in line with normal text. In both cases, source code is formatted in a fixed-width font like this to separate it from ordinary text.

Additionally, comments in the source code have often been removed from the listings when the code is described in the text. Code annotations accompany many of the

listings, highlighting important concepts. If errata or bugs are fixed in the source code online, I'll include notes there to reconcile any differences from the code printed in the text.

In a few cases, the code for an example consists of a standalone Python script, rather than cells of the walkthrough Jupyter notebook for the chapter. You can either run it on its own as, for instance, `python script.py` or run it from within Jupyter notebook cell as `!python script.py`. I've included references to standalone scripts in some Jupyter notebooks, so you can follow along section-by-section and find the relevant source files.

One convention I've used throughout the book is to represent evaluation of individual Python commands with the `>>>` prompt symbol you'd see in a Python interactive session. I suggest you use Jupyter instead of Python interactive, but in any case, lines with `>>>` represent inputs and lines without represent outputs. Here's an example of a code block representing an interactive evaluation of a piece of Python code, `"2 + 2"`:

```
>>> 2 + 2
4
```

By contrast, this next code block doesn't have any `>>>` symbols, so it's ordinary Python code rather than a sequence of inputs and outputs:

```
def square(x):
    return x * x
```

This book has hundreds of exercises, which are intended to be straightforward applications of material already covered, as well as mini-projects, which either are more involved, require more creativity, or introduce new concepts. Most exercises and mini-projects in this book invite you to solve some math problem with working Python code. I've included solutions to almost all of them, excluding some of the more open-ended mini-projects. You can find the solution code in the corresponding chapter's walkthrough Jupyter notebook.

The code for the examples in this book is available for download from the Manning website at <https://www.manning.com/books/math-for-programmers> and from GitHub at <https://github.com/orlandpm/math-for-programmers>.

liveBook discussion forum

Purchase of *Math for Programmers* includes free access to a private web forum run by Manning Publications where you can make comments about the book, ask technical questions, and receive help from the author and from other users. To access the forum, go to <https://livebook.manning.com/#!/book/math-for-programmers/discussion>. You can also learn more about Manning's forums and the rules of conduct at <https://livebook.manning.com/#!/discussion>.

Manning's commitment to our readers is to provide a venue where a meaningful dialogue between individual readers and between readers and the author can take

place. It is not a commitment to any specific amount of participation on the part of the author, whose contribution to the forum remains voluntary (and unpaid). We suggest you try asking the author some challenging questions lest his interest stray! The forum and the archives of previous discussions will be accessible from the publisher's website as long as the book is in print.

about the author

PAUL ORLAND is an entrepreneur, programmer, and math enthusiast. After a stint as a software engineer at Microsoft, he co-founded Tachyus, a start-up company building predictive analytics to optimize energy production in the oil and gas industry. As founding CTO of Tachyus, Paul led the productization of machine learning and physics-based modeling software, and later as CEO, he expanded the company to serve customers on five continents. Paul has a B.S. in math from Yale and an M.S. in physics from the University of Washington. His spirit animal is the lobster.

about the cover illustration

The figure on the cover of *Math for Programmers* is captioned “Femme Laponne,” or a woman from Lapp, now Sapmi, which includes parts of northern Norway, Sweden, Finland, and Russia. The illustration is taken from a collection of dress costumes from various countries by Jacques Grasset de Saint-Sauveur (1757–1810), titled *Costumes de Différents Pays*, published in France in 1797. Each illustration is finely drawn and colored by hand. The rich variety of Grasset de Saint-Sauveur’s collection reminds us vividly of how culturally apart the world’s towns and regions were just 200 years ago. Isolated from each other, people spoke different dialects and languages. In the streets or in the countryside, it was easy to identify where they lived and what their trade or station in life was just by their dress.

The way we dress has changed since then and the diversity by region, so rich at the time, has faded away. It is now hard to tell apart the inhabitants of different continents, let alone different towns, regions, or countries. Perhaps we have traded cultural diversity for a more varied personal life—certainly for a more varied and fast-paced technological life.

At a time when it is hard to tell one computer book from another, Manning celebrates the inventiveness and initiative of the computer business with book covers based on the rich diversity of regional life of two centuries ago, brought back to life by Grasset de Saint-Sauveur’s pictures.

Learning math with code



This chapter covers

- Solving lucrative problems with math and software
- Avoiding common pitfalls in learning math
- Building on intuition from programming to understand math
- Using Python as a powerful and extensible calculator

Math is like baseball, or poetry, or fine wine. Some people are so fascinated by math that they devote their whole lives to it, while others feel like they just don't get it. You've probably already been forced into one camp or another by twelve years of compulsory math education in school.

What if we learned about fine wine in school like we learned math? I don't think I'd like wine at all if I got lectured on grape varieties and fermentation techniques for an hour a day, five days a week. Maybe in such a world, I'd need to consume three or four glasses for homework as assigned by the teacher. Sometimes this would be a delicious educational experience, but sometimes I might not feel like getting loaded on a school night. My experience in math class went something like that, and it turned

me off of the subject for a while. Like wine, mathematics is an acquired taste, and a daily grind of lectures and assignments is no way to refine one's palate.

It's easy to think you're either cut out for math or you aren't. If you already believe in yourself, and you're excited to start learning, that's great! Otherwise, this chapter is designed for those less optimistic. Feeling intimidated by math is so common, it has a name: *math anxiety*. I hope to dispel any anxiety you might have and show you that math can be a stimulating experience rather than a frightening one. All you need are the right tools and the right mindset.

The main tool for learning in this book is the Python programming language. I'm guessing that when you learned math in high school, you saw it written on the blackboard and not in computer code. That's a shame, because a high-level programming language is far more powerful than a blackboard and far more versatile than whatever overpriced calculator you may have used. An advantage of meeting math in code is that the ideas have to be precise enough for a computer to understand, and there's never any hand-waving about what new symbols mean.

As with learning any new subject, the best way to set yourself up for success is to *want* to learn. There are plenty of good reasons for this. You could be intrigued by the beauty of mathematical concepts or enjoy the "brain-teaser" feel of math problems. Maybe there's an app or game that you dream of building, and you need to write some mathematical code to make it work. For now, I'll focus on a more pragmatic kind of motivation—solving mathematical problems with software can make you a lot of money.

1.1 Solving lucrative problems with math and software

A classic criticism you hear in high school math class is, "When am I ever going to use this stuff in real life?" Our teachers told us that math would help us succeed professionally and make money. I think they were right about this, even though their examples were off. For instance, I don't calculate my compounding bank interest by hand (and neither does my bank). Maybe if I became a construction site surveyor as my trigonometry teacher suggested, I'd be using sines and cosines every day to earn my paycheck.

It turns out the "real world" applications from high school textbooks aren't that useful. Still, there are real applications of math out there, and some of them are mind-bogglingly lucrative. Many are solved by translating the right mathematical idea into usable software. I'll share some of my favorite examples.

1.1.1 Predicting financial market movements

We've all heard legends of stock traders making millions of dollars by buying and selling the right stocks at the right time. Based on the movies I've seen, I always picture a trader as a middle-aged man in a suit yelling at his broker over a cell phone while driving around in a sports car. Maybe this stereotype was spot-on at one point, but the situation is different today.

Holed up in back offices of skyscrapers all over Manhattan are thousands of people called *quants*. Quants, otherwise known as quantitative analysts, design mathematical

algorithms to automatically trade stocks and earn a profit. They don't wear suits and they don't spend time yelling on their cell phones, but I'm sure many of them own very nice sports cars.

So how does a quant write a program that automatically makes money? The best answers to that question are closely-guarded trade secrets, but you can be sure they involve a lot of math. We can look at a brief example to get a sense of how an automated trading strategy might work.

Stocks are types of financial assets that represent ownership stakes in companies. When the market perceives a company is doing well, its stock price goes up—buying the stock becomes more costly and selling it becomes more rewarding. Stock prices change erratically and in real time. Figure 1.1 shows how a graph of a stock price over a day of trading might look.

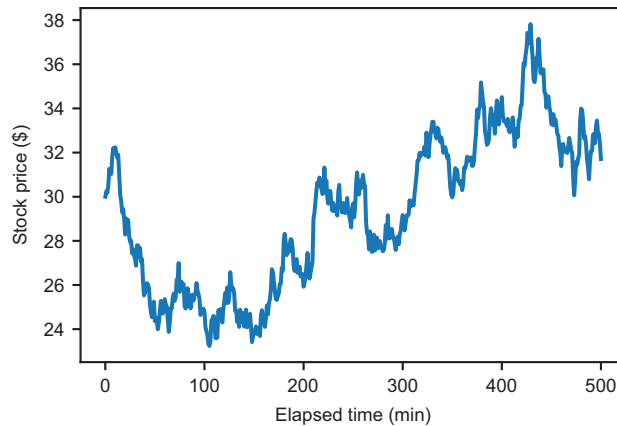


Figure 1.1 Typical graph of a stock price over time

If you bought a thousand shares of this stock for \$24 around minute 100 and sold them for \$38 at minute 400, you would make \$14,000 for the day. Not bad! The challenge is that you'd have to know in advance that the stock was going up, and that minutes 100 and 400 were the best times to buy and sell, respectively. It may not be possible to predict the exact lowest or highest price points, but maybe you can find relatively good times to buy and sell throughout the day. Let's look at a way to do this mathematically.

We could measure whether the stock is going up or down by finding a line of “best fit” that approximately follows the direction the price is moving. This process is called *linear regression*, and we cover it in part 3 of this book. Based on the variability of data, we can calculate two more lines above and below the “best fit” line that show the region in which the price is wobbling up and down. Overlaid on the price graph, figure 1.2 shows that the lines follow the trend nicely.

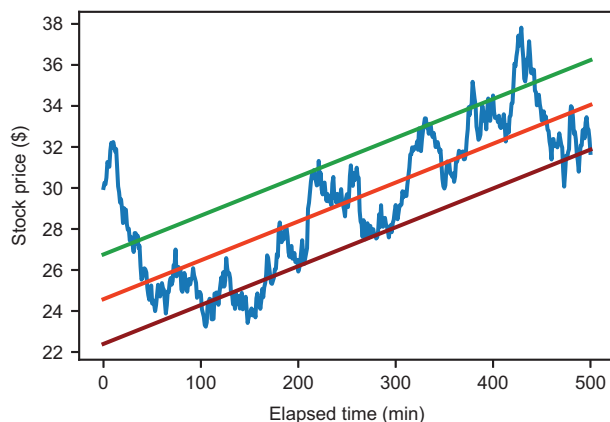


Figure 1.2 Using linear regression to identify a trend in changing stock prices

With a mathematical understanding of the price movement, we can then write code to automatically buy when the price is going through a low fluctuation relative to its trend and to sell when the price goes back up. Specifically, our program could connect to the stock exchange over the network and buy 100 shares when the price crosses the bottom line and sell 100 shares when the price crosses the top line. Figure 1.3 illustrates one such profitable trade: buying at around \$27.80 and selling at around \$32.60 makes you \$480 in an hour.

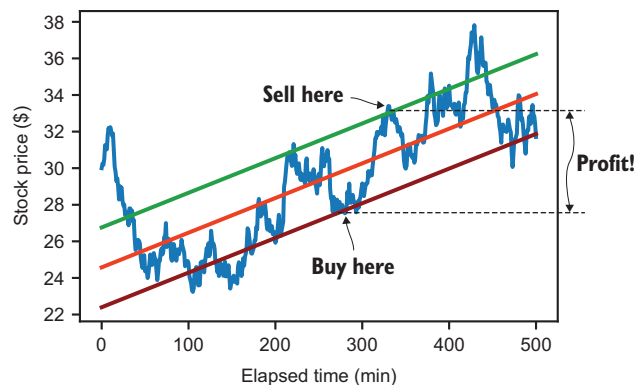


Figure 1.3 Buying and selling according to our rules-based software to make a profit

I don't claim I've shown you a complete or viable strategy here, but the point is that with the right mathematical model, you can make a profit automatically. At this moment, some unknown number of programs are building and updating models measuring the predicted trend of stocks and other financial instruments. If you write such a program, you can enjoy some leisure time while it makes money for you!

1.1.2 Finding a good deal

Maybe you don't have deep enough pockets to consider risky stock trading. Math can still help you make and save money in other transactions like buying a used car, for example. New cars are easy-to-understand commodities. If two dealers are selling the same car, you obviously want to buy from the dealer that has the lowest cost. But used cars have more numbers associated with them: an asking price, as well as mileage and model year. You can even use the duration that a particular used car has been on the market to assess its quality: the longer the duration, the more suspicious you might be.

In mathematics, objects you can describe with ordered lists of numbers are called *vectors*, and there is a whole field (called *linear algebra*) dedicated to studying them. For example, a used car might correspond to a *four-dimensional* vector, meaning a four-tuple of numbers:

(2015, 41429, 22.27, 16980)

These numbers represent the model year, mileage, days on the market, and asking price, respectively. A friend of mine runs a site called CarGraph.com that aggregates data on used cars for sale. At the time of writing, it shows 101 Toyota Priuses for sale, and it gives some or all of these four pieces of data for each one. The site also lives up to its name and visually presents the data in a graph (figure 1.4). It's hard to visualize four-dimensional objects, but if you choose two of the dimensions like price and mileage, you can graph them as points on a scatter plot.

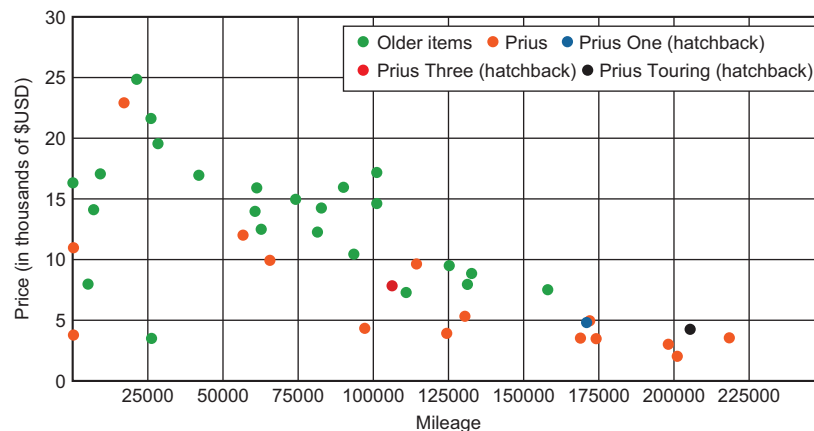


Figure 1.4 A graph of price vs. mileage for used Priuses from CarGraph.com

We might be interested in drawing a trend line here too. Every point on this graph represents someone's opinion of a fair price, so the trend line would aggregate these opinions together into a more reliable price at any mileage. In figure 1.5, I decided to

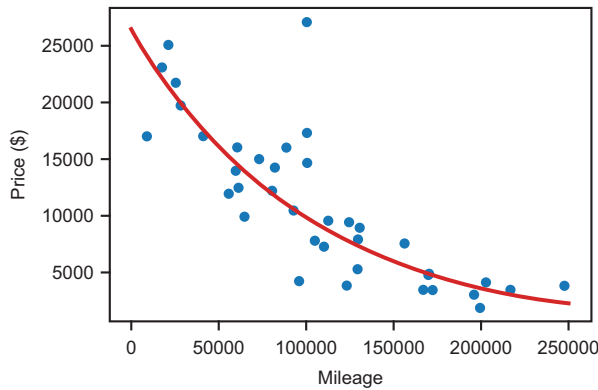


Figure 1.5 Fitting an exponential decline curve to price vs. mileage data for used Toyota Priuses

fit to an *exponential* decline curve rather than a line, and I omitted some of the nearly new cars selling for below retail price.

To make the numbers more manageable, I converted the mileage values to tens of thousands of miles, so a mileage of 5 represents 50,000 miles. Calling p the price and m the mileage, the equation for the curve of best fit is as follows:

$$p = \$26,500 \cdot (0.905)^m$$

Equation 1.1

Equation 1.1 shows that the best fit price is \$26,500 times 0.905 raised to the power of the mileage. Plugging the values into the equation, I find that if my budget is \$10,000, then I should buy a Prius with about 97,000 miles on it (figure 1.6). If I believe the curve indicates a *fair* price, then cars below the line should typically be good deals.

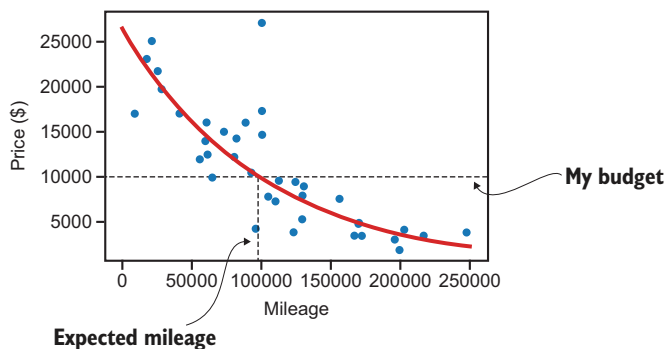


Figure 1.6 Finding the mileage I should expect on a used Prius for my \$10,000 budget

But we can learn more from equation 1.1 than just how to find a good deal. It tells a story about how cars depreciate. The first number in the equation is \$26,500, which is the exponential function's understanding of the price at zero mileage. This is an

impressively close match to the retail price of a new Prius. If we use a line of best fit, it implies a Prius loses a fixed amount of value with each mile driven. This exponential function says, instead, that it loses a fixed *percentage* of its value with each mile driven. After driving 10,000 miles, a Prius is only worth 0.905 or 90.5% of its original price according to this equation. After 50,000 miles, we multiply its price by a factor of $(0.905)^5 = 0.607$. That tells us that it's worth about 61% of what it was originally.

To make the graph in figure 1.6, I implemented a `price(mileage)` function in Python, which takes a mileage as an input (measured in 10,000s of miles) and returns the best-fit price as an output. Calculating `price(0) - price(5)` and `price(5) - price(10)` tells me that the first and second 50,000 miles driven cost about \$10,000 and \$6,300, respectively.

If we use a line of best fit instead of an exponential curve, it implies that the car depreciated at a fixed rate of \$0.10 per mile. This suggests that every 50,000 miles of driving leads to the same depreciation of \$5,000. Conventional wisdom says that the first miles you drive a new car are the most expensive, so the exponential function (equation 1.1) agrees with this, while a linear model does not.

Remember, this is only a *two-dimensional* analysis. We only built a mathematical model to relate two of the four numerical dimensions describing each car. In part 1, you learn more about vectors of various dimensions and how to manipulate higher-dimensional data. In part 2, we cover different kinds of functions like linear functions and exponential functions, and we compare them by analyzing their rates of change. Finally, in part 3, we look at how to build mathematical models that incorporate *all* the dimensions of a data set to give us a more accurate picture.

1.1.3 Building 3D graphics and animations

Many of the most famous and financially successful software projects deal with multi-dimensional data, specifically *three-dimensional* or *3D* data. Here I'm thinking of 3D animated movies and 3D video games that gross in the billions of dollars. For example, Pixar's 3D animation software has helped them rake in over \$13 billion at box offices. Activision's *Call of Duty* franchise of 3D action games has earned over \$16 billion, and Rockstar's *Grand Theft Auto* Valone has brought in \$6 billion.

Every one of these acclaimed projects is based on an understanding of how to do computations with 3D vectors, or triples of numbers of the form $v = (x, y, z)$. A triple of numbers is sufficient to locate a point in 3D space relative to a reference point called the *origin*. Figure 1.7 shows how each of the three numbers tells you how far to go in one of three perpendicular directions.

Any 3D object from a clownfish in *Finding Nemo* to an aircraft carrier in *Call of Duty* can

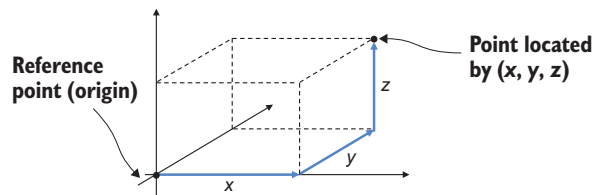


Figure 1.7 Labeling a point in 3D with a vector of three numbers, x , y , and z

be defined for a computer as a collection of 3D vectors. In code, each of these objects looks like a list of triples of float values. With three triples of floats, we have three points in space that can define a triangle (figure 1.8). For instance,

```
triangle = [(2.3, 1.1, 0.9), (4.5, 3.3, 2.0), (1.0, 3.5, 3.9)]
```

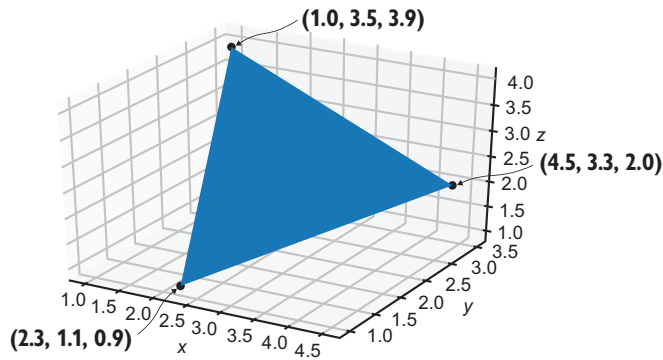


Figure 1.8 Building a 3D triangle using a triple of float values for each of its corners

Combining many triangles, you can define the surface of a 3D object. Using more, smaller triangles, you can even make the result look smooth. Figure 1.9 shows six renderings of a 3D sphere using an increasing number of smaller and smaller triangles.

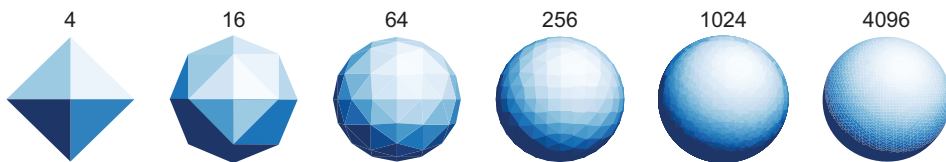


Figure 1.9 Three-dimensional (3D) spheres built out of the specified number of triangles.

In chapters 3 and 4, you learn how to use 3D vector math to turn 3D models into shaded 2D images like the ones in figure 1.9. You also need to make your 3D models smooth to make them realistic in a game or movie, and you need them to move and change in realistic ways. This means that your objects should obey the laws of physics, which are also expressed in terms of 3D vectors.

Suppose you're a programmer for *Grand Theft Auto V* and want to enable a basic use case like shooting a bazooka at a helicopter. A projectile coming out of a bazooka starts at the protagonist's location and then its position changes over time. You can use numeric subscripts to label the various positions it has over its flight, starting with $\mathbf{v}_0 = (x_0, y_0, z_0)$. As time elapses, the projectile arrives at new positions labeled by vectors $\mathbf{v}_1 = (x_1, y_1, z_1)$, $\mathbf{v}_2 = (x_2, y_2, z_2)$, and so on. The rates of change for the x , y , and z values are decided by the direction and speed of the bazooka. Moreover, the rates can change over time—the projectile increases its z position at a decreasing rate because of the continuous downward pull of gravity (figure 1.10).

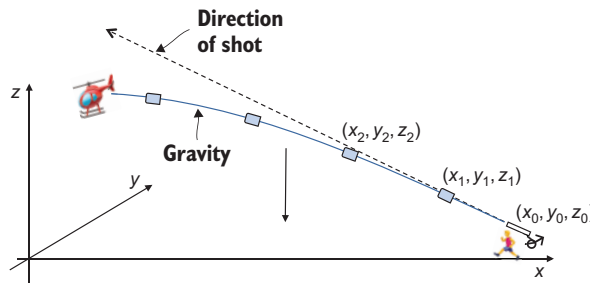


Figure 1.10 The position vector of the projectile changes over time due to its initial speed and the pull of gravity.

As any experienced action gamer will tell you, you need to aim slightly above the helicopter to hit it! To simulate physics, you have to know how forces affect objects and cause continuous change over time. The math of continuous change is called *calculus*, and the laws of physics are usually expressed in terms of objects from calculus called *differential equations*. You learn how to animate 3D objects in chapters 4 and 5, and then how to simulate physics using ideas from calculus in part 2.

1.1.4 Modeling the physical world

My claim that mathematical software produces real financial value isn't just speculation; I've seen the value in my own career. In 2013, I founded a company called Tachyus that builds software to optimize oil and gas production. Our software uses mathematical models to understand the flow of oil and gas underground to help producers extract it more efficiently and profitably. Using the insight it generates, our customers have achieved millions of dollars a year in cost savings and production increases.

To explain how our software works, you need to know a few pieces of oil terminology. Holes called *wells* are drilled into the ground until they reach the target layer of porous (sponge-like) rock containing oil. This layer of oil-rich rock underground is called a *reservoir*. Oil is pumped to the surface and is then sold to refiners who convert it into the products we use every day. A schematic of an oilfield (not to scale!) is shown in figure 1.11.

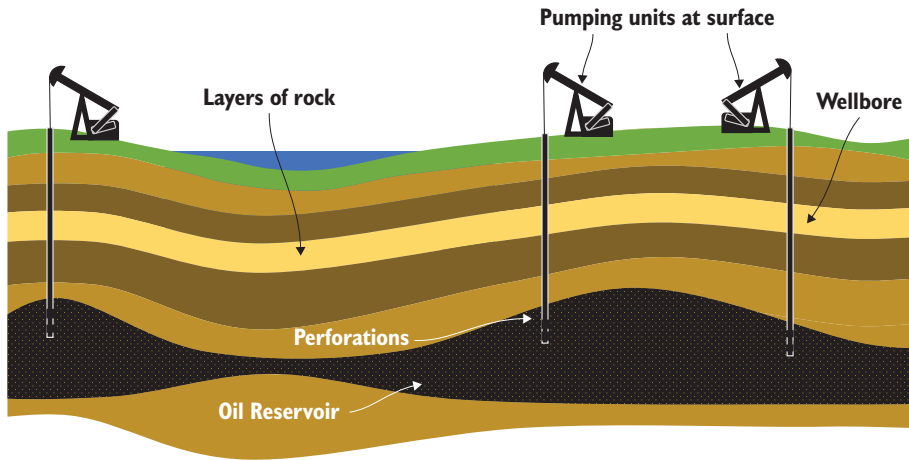


Figure 1.11 A schematic diagram of an oilfield

Over the past few years, the price of oil has varied significantly, but for our purposes, let's say it's worth \$50 a barrel, where a barrel is a unit of volume equal to 42 gallons or about 159 liters. If by drilling wells and pumping effectively, a company is able to extract 1,000 barrels of oil per day (the volume of a few backyard swimming pools), it will have annual revenues in the tens of millions of dollars. Even a few percentage points of increased efficiency can mean a sizable amount of money.

The underlying question is what's going on underground: where is the oil now and how is it moving? This is a complicated question, but it can also be answered by solving differential equations. The changing quantities here are not positions of a projectile, but rather locations, pressures, and flow rates of fluids underground. Fluid flow rate is a special kind of function that returns a vector, called a *vector field*. This means that fluid can flow at any rate in any three-dimensional direction, and that direction and rate can vary across different locations within the reservoir.

With our best guess for some of these parameters, we can use a differential equation called *Darcy's law* to predict flow rate of liquid through a porous rock medium like sandstone. Figure 1.12 shows Darcy's law, but don't worry if some symbols are unfamiliar! The function named \mathbf{q} representing flow rate is bold to indicate it returns a vector value.

The most important part of this equation is the symbol that looks

$$\mathbf{q}(x, y, z) = -\frac{\kappa}{\mu} \nabla p(x, y, z)$$

Permeability of porous medium
 Flow rate of fluid
 Pressure gradient
 Viscosity (thickness) of fluid

Figure 1.12 Darcy's law annotated for a physics equation, governing how fluid flows within a porous rock.

like an upside-down triangle, which represents the *gradient operator* in vector calculus. The gradient of the pressure function $p(x, y, z)$ at a given spatial point (x, y, z) is the 3D vector $\mathbf{q}(x, y, z)$, indicating the direction of increasing pressure and the rate of increase in pressure at that point. The negative sign tells us that the 3D vector of flow rate is in the *opposite* direction. This equation states, in mathematical terms, that fluid flows from areas of high pressure to areas of low pressure.

Negative gradients are common in the laws of physics. One way to think of this is that nature is always seeking to move toward lower potential energy states. The potential energy of a ball on a hill depends on the altitude h of the hill at any lateral point x . If the height of a hill is given by a function $h(x)$, the gradient points uphill while the ball rolls in the exact opposite direction (figure 1.13).

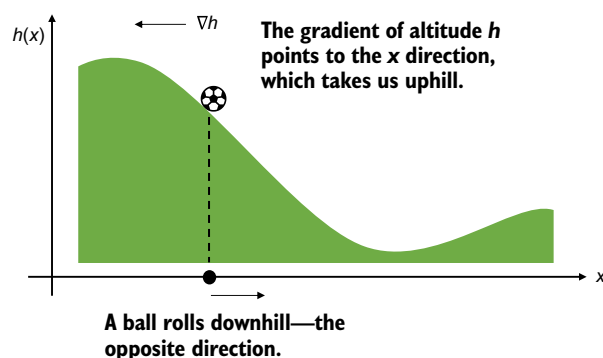


Figure 1.13 The positive gradient points uphill, while the negative gradient points downhill.

In chapter 11, you learn how to calculate gradients. There, I show you how to apply gradients to simulate physics and also to solve other mathematical problems. The gradient happens to be one of the most important mathematical concepts in machine learning as well.

I hope these examples have been more compelling and realistic than the real-world applications you heard in high school math class. Maybe, at this point, you're convinced these math concepts are worth learning, but you're worried that they might be too difficult. It's true that learning math can be hard, especially on your own. To make it as smooth as possible, let's talk about some of the pitfalls you can face as a math student and how I'll help you avoid them in this book.

1.2 How not to learn math

There are plenty of math books out there, but not all of them are equally useful. I have quite a few programmer friends who tried to learn mathematical concepts like the ones in the previous section, either motivated by intellectual curiosity or by career ambitions. When they use traditional math textbooks as their main resource, they often get stuck and give up. Here's what a typical *unsuccessful* math-learning story looks like.

1.2.1 *Jane wants to learn some math*

My (fictional) friend Jane is a full-stack web developer working at a medium-sized tech company in San Francisco. In college, Jane didn't study computer science or any mathematical subjects in depth, and she started her career as a product manager. Over the last ten years, she picked up coding in Python and JavaScript and was able to transition into software engineering. Now, at her new job, she is one of the most capable programmers on the team, able to build the databases, web services, and user interfaces required to deliver important new features to customers. Clearly, she's pretty smart!

Jane realizes that learning data science could help her design and implement better features at work, using data to improve the experience for her customers. Most days on the train to work, Jane reads blogs and articles about new technologies, and recently, she's been amazed by a few about a topic called "deep learning." One article talks about Google's AlphaGo, powered by deep learning, which beat the top-ranked human players in the world in a board game. Another article showed stunning impressionist paintings generated from ordinary images, again using a deep learning system.

After reading these articles, Jane overheard that her friend-of-a-friend Marcus got a deep learning research job at a big tech company. Marcus supposedly gets paid over \$400,000 a year in salary and stock. Thinking about the next step in her career, what more could Jane want than to work on a fascinating and lucrative problem?

Jane did some research and found an authoritative (and free!) resource online: the book *Deep Learning* by Goodfellow, et al., (MIT Press, 2016). The introduction read much like the technical blog posts she was used to and got her even more excited about learning the topic. But as she kept reading, the content of the book got harder. The first chapter covered the required math concepts and introduced a lot of terminology and notation that Jane had never seen. She skimmed it and tried to get on to the meat of the book, but it continued to get more difficult.

Jane decided she needed to pause her study of AI and deep learning until she learned some math. Fortunately, the math chapter of *Deep Learning* listed a reference on linear algebra for students who had never seen the topic before. She tracked down this textbook, *Linear Algebra* by Georgi Shilov (Dover, 1977), and discovered that it was 400 pages long and equally as dense as *Deep Learning*.

After spending an afternoon reading abstruse theorems about concepts like number fields, determinants, and cofactors, she called it quits. She had no idea how these concepts were going to help her write a program to win a board game or to generate artwork, and she no longer cared to spend dozens of hours with this dry material to find out.

Jane and I met to catch up over a cup of coffee. She told me about her struggles reading real AI literature because she didn't know linear algebra. Recently, I'm hearing a lot of the same form of lamentation:

I'm trying to read about [new technology] but it seems like I need to learn [math topic] first.

Her approach was admirable: she tracked down the best resource for the subject she wanted to learn and sought out resources for prerequisites she was missing. But in taking that approach to its logical conclusion, she found herself in a nauseating “depth-first” search of technical literature.

1.2.2 Slogging through math textbooks

College-level math books like the linear algebra book Jane picked up tend to be very formulaic. Every section follows the same format: it defines some new terminology, states some facts (called *theorems*) using that terminology, and then proves that those theorems are true.

This sounds like a good, logical order: you introduce the concept you’re talking about, state some conclusions that can be drawn, and then justify them. Then why is it so hard to read advanced mathematical textbooks?

The problem is that this is not how math is actually created. When you’re coming up with new mathematical ideas, there can be a long period of experimentation before you even find the right definitions. I think most professional mathematicians would describe their steps like this:

- 1 Invent a *game*. For example, start playing with some mathematical objects by trying to list all of them, find patterns among them, or find one with a particular property.
- 2 Form some *conjectures*. Speculate about some general facts you can state about your game and, at least, convince yourself these must be true.
- 3 Develop some *precise language* to describe your game and your conjectures. After all, your conjectures won’t mean anything until you can communicate them.
- 4 Finally, with some determination and luck, find a *proof* for your conjecture, showing why it *needs* to be true.

The main lesson to learn from this process is that you should start by thinking about big ideas, and the formalism can wait. Once you have a rough idea of how the math works, the vocabulary and notation become an asset for you rather than a distraction. Math textbooks usually work in the opposite order, so I recommend using textbooks as references rather than as introductions to new subjects.

Instead of reading traditional textbooks, the best way to learn math is to explore ideas and draw your own conclusions. However, you don’t have enough hours in the day to reinvent everything yourself. What is the right balance to strike? I’ll give you my humble opinion, which guides how I’ve written this non-traditional book about math.

1.3 Using your well-trained left brain

This book is designed for people who are either experienced programmers or for those who are excited to learn programming as they work through it. It’s great to write about math for an audience of programmers, because if you can write code, you’ve already trained your analytical left brain. I think the best way to learn math is with the

help of a high-level programming language, and I predict that in the not-so-distant future, this will be the norm in math classrooms.

There are several specific ways programmers like you are well equipped to learn math. I list those here, not only to flatter you, but also to remind you what skills you already have that you can lean on in your mathematical studies.

1.3.1 *Using a formal language*

One of the first hard lessons you learn in programming is that you can't write your code like you write simple English. If your spelling or grammar is slightly off when writing a note to a friend, they can probably still understand what you're trying to say. But any syntactic error or misspelled identifier in code causes your program to fail. In some languages, even forgetting a semicolon at the end of an otherwise correct statement prevents the program from running. As another example, consider the two statements:

```
x = 5
5 = x
```

I could read either of these to mean that the symbol x has the value 5. But that's not *exactly* what either of these means in Python, and in fact, only the first one is correct. The Python statement `x = 5` is an instruction to set the variable x to have the value 5. On the other hand, you can't set the number 5 to have the value x . This may seem pedantic, but you need to know it to write a correct program.

Another example that trips up novice programmers (and experienced ones as well!) is reference equality. If you define a new Python class and create two identical instances of it, they are not equal!

```
>>> class A(): pass
...
>>> A() == A()
False
```

You might expect two identical expressions to be equal, but that's evidently not a rule in Python. Because these are different instances of the `A` class, they are not considered equal.

Be on the lookout for new mathematical objects that look like ones you know but don't behave the same way. For instance, if the letters A and B represent numbers, then $A \cdot B = B \cdot A$. But, as you'll learn in chapter 5, this is not necessarily the case if A and B are *not* numbers. If, instead, A and B are matrices, then the products $A \cdot B$ and $B \cdot A$ are different. In fact, it's possible that only one of the products is even doable or that neither product is correct.

When you're writing code, it's not enough to write statements with correct syntax. The ideas that your statements represent need to make sense to be valid. If you apply the same care when you're writing mathematical statements, you'll catch your mistakes faster. Even better, if you write your mathematical statements in code, you'll have the computer to help check your work.

1.3.2 Build your own calculator

Calculators are prevalent in math classes because it's useful to check your work. You need to know how to multiply 6 by 7 without using your calculator, but it's good to confirm that your answer of 42 is correct by consulting your calculator. The calculator also helps you save time once you've mastered mathematical concepts. If you're doing trigonometry, and you need to know the answer to $3.14159 / 6$, the calculator is there to handle it so you can instead think about what the answer means. The more a calculator can do out-of-the-box, the more useful it should theoretically be.

But sometimes our calculators are too complicated for our own good. When I started high school, I was required to get a graphing calculator and I got a TI-84. It had about 40 buttons, each with 2 to 3 different modes. I only knew how to use maybe 20 of them, so it was a cumbersome tool to learn how to use. The story was the same when I got my first ever calculator in first grade. There were only 15 buttons or so, but I didn't know what some of them did. If I had to invent a first calculator for students, I would make it look something like the one in figure 1.14.

This calculator only has two buttons. One of them resets the value to 1, and the other advances to the next number. Something like this would be the right “no-frills” tool for children learning to count. (My example may seem silly, but you can actually buy calculators like this! They are usually mechanical and sold as tally counters.)

Soon after you master counting, you want to practice writing numbers and adding them. The perfect calculator at that stage of learning might have a few more buttons (figure 1.15).

There's no need for buttons like $-$, $*$, or \div to get in your way at this phase. As you solve subtraction problems like $5 - 2$, you can still check your answer of 3 with this calculator by confirming the sum $3 + 2 = 5$. Likewise, you can solve multiplication problems by adding numbers repeatedly. You could upgrade to a calculator that does all of the operations of arithmetic when you're done exploring with this one.

I think an ideal calculator would be extensible, meaning that you could add more functionality to it as needed. For instance, you could add a button to your calculator for every new mathematical operation you learn. Once you got to algebra, maybe you could enable it to understand symbols like x or y in addition to numbers. When you



Figure 1.14 A calculator for students learning to count



Figure 1.15 A calculator capable of writing whole numbers and adding them

learned calculus, you could further enable it to understand and manipulate mathematical functions.

Extensible calculators that can hold many types of data seem far-fetched, but that's exactly what you get when you use a high-level programming language. Python comes with arithmetic operations, a `math` module, and numerous third-party mathematical libraries you can pull in to make your programming environment more powerful whenever you want. Because Python is *Turing complete*, you can (in principle) compute anything that can be computed. You only need a powerful enough computer, a clever enough implementation, or both.

In this book, we implement each new mathematical concept in reusable Python code. Working through the implementation yourself can be a great way of cementing your understanding of a new concept, and by the end, you've added a new tool to your toolbox. After trying it yourself, you can always swap in a polished, mainstream library if you like. Either way, the new tools you build or import lay the groundwork to explore even bigger ideas.

1.3.3 *Building abstractions with functions*

In programming, the process I just described is called *abstraction*. For example, when you get tired of repeated counting, you create the abstraction of addition. When you get tired of doing repeated addition, you create the abstraction of multiplication, and so on.

Of all the ways that you can make abstractions in programming, the most important one to carry over to math is the *function*. A function in Python is a way of repeating some task that can take one or more inputs or that can produce an output. For example,

```
def greet(name):
    print("Hello %s!" % name)
```

allows me to issue multiple greetings with short, expressive code like this:

```
>>> for name in ["John", "Paul", "George", "Ringo"]:
...     greet(name)
...
Hello John!
Hello Paul!
Hello George!
Hello Ringo!
```

This function can be useful, but it's not like a mathematical function. Mathematical functions always take input values, and they always return output values with no side effects.

In programming, we call the functions that behave like mathematical functions *pure functions*. For example, the square function $f(x) = x^2$ takes a number and returns the product of the number with itself. When you evaluate $f(3)$, the result is 9. That doesn't mean that the number 3 has now changed and becomes 9. Rather, it means 9 is the corresponding output for the input 3 for the function f . You can picture this

squaring function as a machine that takes numbers in an input slot and produces results (numbers) in its output slot (figure 1.16).

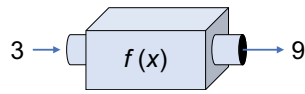


Figure 1.16 A function as a machine with an input slot and an output slot

This is a simple and useful mental model, and I'll return to it throughout the book. One of the things I like most about it is that you can picture a function as an object in and of itself. In math, as in Python, functions are data that you can manipulate independently and even pass to other functions.

Math can be intimidating because it is abstract. Remember, as in any well-written software, the abstraction is introduced for a reason: it helps you organize and communicate bigger and more powerful ideas. When you grasp these ideas and translate them into code, you'll open up some exciting possibilities.

If you didn't already, I hope you now believe there are many exciting applications of math in software development. As a programmer, you already have the right mindset and tools to learn some new mathematical ideas. The ideas in this book provided me with professional and personal enrichment, and I hope they will for you as well. Let's get started!

Summary

- There are interesting and lucrative applications of math in many software engineering domains.
- Math can help you quantify a trend for data that changes over time, for instance, to predict the movement of a stock price.
- Different types of functions convey different kinds of qualitative behavior. For instance, an exponential depreciation function means that a car loses a percentage of its resale value with each mile driven rather than a fixed amount.
- Tuples of numbers (called *vectors*) represent multidimensional data. Specifically, 3D vectors are triples of numbers and can represent points in space. You can build complex 3D graphics by assembling triangles specified by vectors.
- *Calculus* is the mathematical study of continuous change, and many of the laws of physics are written in terms of calculus equations that are called *differential equations*.
- It's hard to learn math from traditional textbooks! You learn math by exploration, not as a straightforward march through definitions and theorems.
- As a programmer, you've already trained yourself to think and communicate precisely; this skill will help you learn math as well.

Part 1

Vectors and graphics

In the first part of this book, we dig into the branch of math called *linear algebra*. At a very high level, linear algebra is the branch of math dealing with computations on multi-dimensional data. The concept of “dimension” is a geometric one; you probably intuitively know what I mean when I say “a square is 2-dimensional” while “a cube is 3-dimensional.” Among other things, linear algebra lets us turn geometric ideas about dimension into things we can compute concretely.

The most basic concept in linear algebra is that of a *vector*, which you can think of as a data point in some multi-dimensional space. For instance, you’ve probably heard of the 2-dimensional (2D) coordinate plane in high school geometry and algebra. As we’ll cover in chapter 2, vectors in 2D correspond to points in the plane, which can be labeled by ordered pairs of numbers of the form (x, y) . In chapter 3, we’ll consider 3-dimensional (3D) space, whose vectors (points) can be labeled by triples of numbers in the form (x, y, z) . In both cases, we see we can use collections of vectors to define geometric shapes, which can, in turn, be converted into interesting graphics.

Another key concept in linear algebra is that of a *linear transformation*, which we introduce in chapter 4. A linear transformation is a kind of function that takes a vector as input and returns a vector as output, while preserving the geometry (in a special sense) of the vectors involved. For instance, if a collection of vectors (points) lie on a straight line in 2D, after applying a linear transformation, they will still lie on a straight line. In chapter 5, we introduce *matrices*, which are rectangular arrays of numbers that can represent linear transformations. Our culminating application of linear transformations is to apply them sequentially over time to graphics in a Python program, resulting in some animated graphics in 3D.

While we can only picture vectors and linear transformations in 2D and 3D, it's possible to define vectors with any number of dimensions. In n dimensions, a vector can be identified as an ordered n -tuple of numbers of the form (x_1, x_2, \dots, x_n) . In chapter 6, we reverse-engineer the concepts of 2D and 3D space to define the general concept of a *vector space* and to define the concept of *dimension* more concretely. In particular, we'll see that digital images made of pixels can be thought of as vectors in a high-dimensional vector space and that we can do image manipulation with linear transformations.

Finally, in chapter 7, we look at the most ubiquitous computational tool in linear algebra: solving *systems of linear equations*. As you may remember from high school algebra, the solution to two linear equations in two variables like x and y tell us where two lines meet in the plane. In general, linear equations tell us where lines, planes, or higher-dimensional generalizations intersect in a vector space. Being able to automatically solve this problem in Python, we'll use it to build a first version of a video game engine.

Drawing with 2D vectors



This chapter covers

- Creating and manipulating 2D drawings as collections of vectors
- Thinking of 2D vectors as arrows, locations, and ordered pairs of coordinates
- Using vector arithmetic to transform shapes in the plane
- Using trigonometry to measure distances and angles in the plane

You probably already have some intuition for what it means to be two-dimensional or three-dimensional. A *two-dimensional* (2D) object is flat like an image on a piece of paper or a computer screen. It has only the dimensions of height and width. A *three-dimensional* (3D) object in our physical world, however, has not only height and width but also depth.

Models of 2D and 3D entities are important in programming. Anything that shows up on the screen of your phone, tablet, or PC is a 2D object, occupying some

width and height of pixels. Any simulation, game, or animation that represents the physical world is stored as 3D data and eventually projected to the two dimensions of the screen. In virtual and augmented reality applications, the 3D models must be paired with real, measured 3D data about the user's position and perspective.

Even though our everyday experience takes place in three dimensions, it's useful to think of some data as higher dimensional. In physics, it's common to consider time as the fourth dimension. While an object exists at a location in 3D space, an event occurs at a 3D location and at a specified moment. In data science problems, it's common for data sets to have far more dimensions. For instance, a user tracked on a website can have hundreds of measurable attributes, which describe usage patterns. Grappling with these problems in graphics, physics, and data analysis requires a framework for dealing with data in higher dimensions. This framework is *vector mathematics*.

Vectors are objects that live in multi-dimensional spaces. These have their own notions of arithmetic (adding, multiplying, and so on). We start by studying 2D vectors, which are easy to visualize and compute with. We use a lot of 2D vectors in this book, and we also use them as a mental model when reasoning about higher-dimensional problems.

2.1 Picturing 2D vectors

The 2D world is flat like a piece of paper or a computer screen. In the language of math, a flat, 2D space is referred to as a *plane*. An object living in a 2D plane has the two dimensions of height and width but no third dimension of depth. Likewise, you can describe locations in 2D by two pieces of information: their vertical and horizontal positions. To describe the location of points in the plane, you need a reference point. We call that special reference point the *origin*. Figure 2.1 shows this relationship.

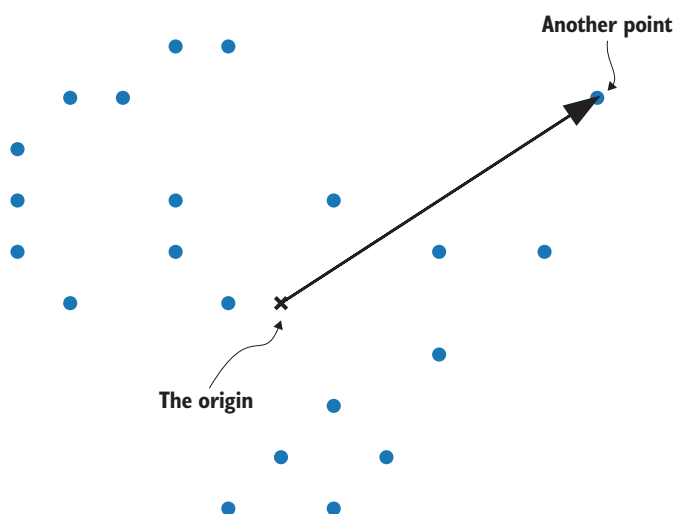


Figure 2.1 Locating one of several points in the plane, relative to the origin

There are many points to choose from, but we have to fix one of them as our origin. To distinguish it, we mark the origin with an x instead of with a dot as in figure 2.1. From the origin, we can draw an arrow (like the solid one in figure 2.1) to show the relative location of another point.

A *two-dimensional vector* is a point in the plane relative to the origin. Equivalently, you can think of a vector as a straight arrow in the plane; any arrow can be placed to start at the origin, and it indicates a particular point (figure 2.2).

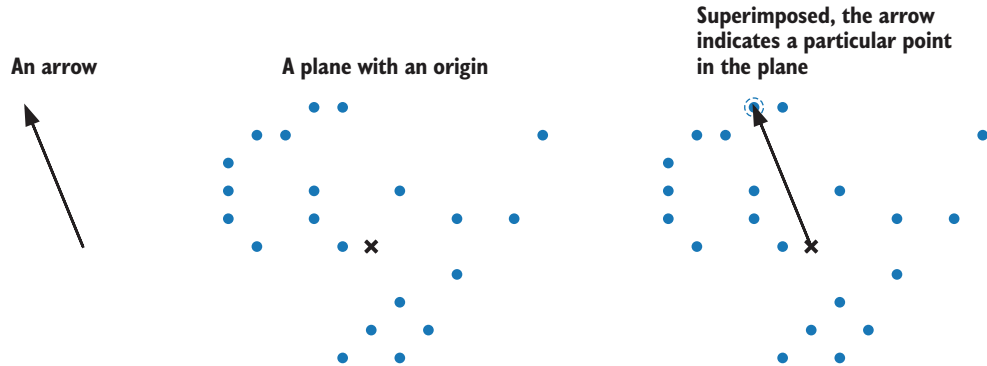


Figure 2.2 Superimposing an arrow on the plane indicates a point relative to the origin.

We'll use both arrows and points to represent vectors in this chapter and beyond. Points are useful to work with because we can build more interesting drawings out of them. If I connect the points in figure 2.2 as in figure 2.3, I get a drawing of a dinosaur:

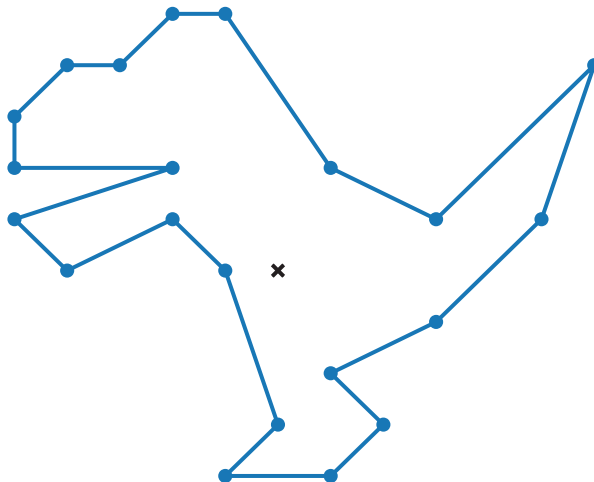
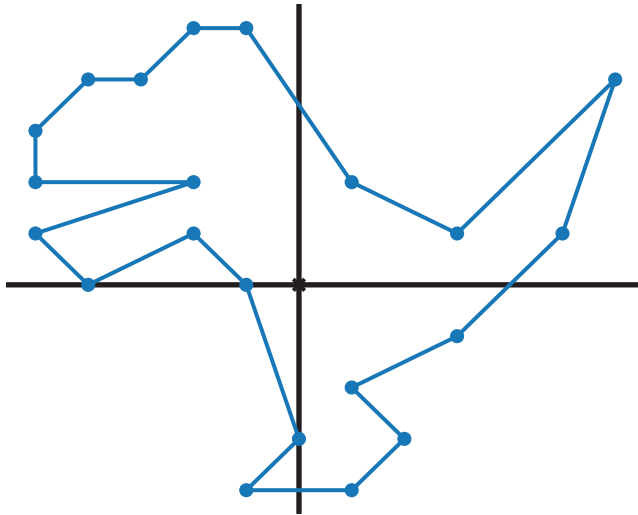


Figure 2.3 Connecting points in the plane to draw a shape

Any time a 2D or 3D drawing is displayed by a computer, from my modest dinosaur to a feature-length Pixar movie, it is defined by points—or vectors—connected to show the desired shape. To create the drawing you want, you need to pick vectors in the right places, requiring careful measurement. Let's take a look at how to measure vectors in the plane.

2.1.1 Representing 2D vectors

With a ruler, we can measure one dimension such as the length of an object. To measure in two dimensions, we need two rulers. These rulers are called *axes* (the singular is *axis*), and we lay them out in the plane perpendicular to one another, intersecting at the origin. Drawn with axes, figure 2.4 shows that our dinosaur has the notions of up and down as well as left and right. The horizontal axis is called the *x-axis* and the vertical one is called the *y-axis*.



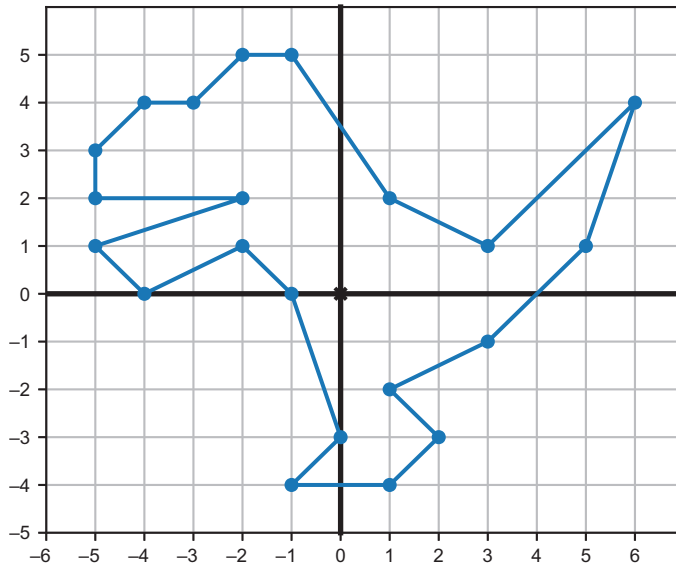


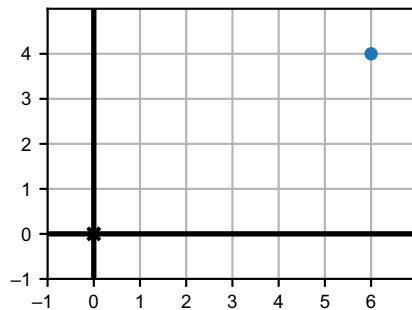
Figure 2.5 Grid lines let us measure the location of points relative to the axes.

The numbers 6 and 4 are called the *x- and y-coordinates* of the point, and this is enough to tell us exactly what point we're talking about. We typically write coordinates as an *ordered pair* (or *tuple*) with the *x*-coordinate first and the *y*-coordinate second, for example, $(6, 4)$. Figure 2.6 shows how we can now describe the same vector in three ways.

1. An ordered pair of numbers (*x*- and *y*-coordinates)

$(6, 4)$

2. A point in the plane relative to the origin



3. An arrow of a specific length in a specific direction

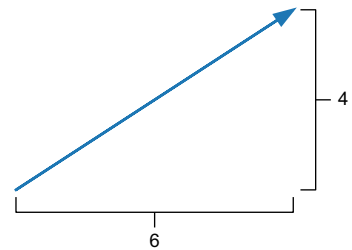


Figure 2.6 Three mental models describing the same vector.

From another pair of coordinates like $(-3, 4.5)$, we can find the point in the plane or the arrow that represents them. To get to the point in the plane with these coordinates, start at the origin and then travel three grid lines to the left (because the x -coordinate is -3) and then four and a half grid lines up (where the y -coordinate is 4.5). The point won't lie at the intersection of two grid lines, but that's fine; any pair of real numbers gives us some point on the plane. The corresponding arrow will be the straight-line path from the origin to that location, which points up and to the left (northwest, if you prefer). Try drawing this example for yourself as practice!

2.1.2 2D drawing in Python

When you produce an image on a screen, you're working in a 2D space. The pixels on the screen are the available points in that plane. These are labeled by whole number coordinates rather than real number coordinates, and you can't illuminate the space between pixels. That said, most graphics libraries let you work with floating-point coordinates and automatically handle translating graphics to pixels on the screen.

We have plenty of language choices and libraries to specify graphics and to get them on the screen: OpenGL, CSS, SVG, and so on. Python has libraries like Pillow and Turtle that are well equipped for creating drawings with vector data. In this chapter, I use a small set of custom-built functions to create drawings, built on top of another Python library called Matplotlib. This lets us focus on using Python to build images with vector data. Once you understand this process, you'll be able to pick up any of the other libraries easily.

The most important function I've included, called `draw`, takes inputs representing geometric objects and keyword arguments specifying how you want your drawing to look. The Python classes listed in table 2.1 represent each kind of drawable geometric object.

Table 2.1 Some Python classes representing geometric figures, usable with the `draw` function.

Class	Constructor example	Description
Polygon	<code>Polygon(*vectors)</code>	Draws a polygon whose vertices (corners) are represented by a list of vectors
Points	<code>Points(*vectors)</code>	Represents a list of points (dots) to draw, one at each of the input vectors
Arrow	<code>Arrow(tip)</code> <code>Arrow(tip, tail)</code>	Draws an arrow from the origin to the <code>tip</code> vector or from the <code>tail</code> vector to the <code>head</code> vector if a tail is specified
Segment	<code>Segment(start, end)</code>	Draws a line segment from the <code>start</code> to the vector <code>end</code>

You can find these functions implemented in the file `vector_drawing.py` in the source code. At the end of the chapter, I'll say a bit more about how these are implemented.

NOTE For this chapter (and each subsequent chapter), there is a Jupyter notebook in the source code folder showing how to run (in order) all of the code in the chapter, including importing the functions from the `vector_drawing` module. If you haven't already, you can consult appendix A to get set up with Python and Jupyter.

With these drawing functions in hand, we can draw the points outlining the dinosaur (figure 2.5):

```
from vector_drawing import *
    dino_vectors = [(6,4), (3,1), (1,2), (-1,5), (-2,5), (-3,4), (-4,4),
        # insert 16 remaining vectors here
    ]

draw(
    Points(*dino_vectors)
)
```

I didn't write out the complete list of `dino_vectors`, but with the suitable collection of vectors, the code gives you the points shown in figure 2.7 (matching figure 2.5 as well).

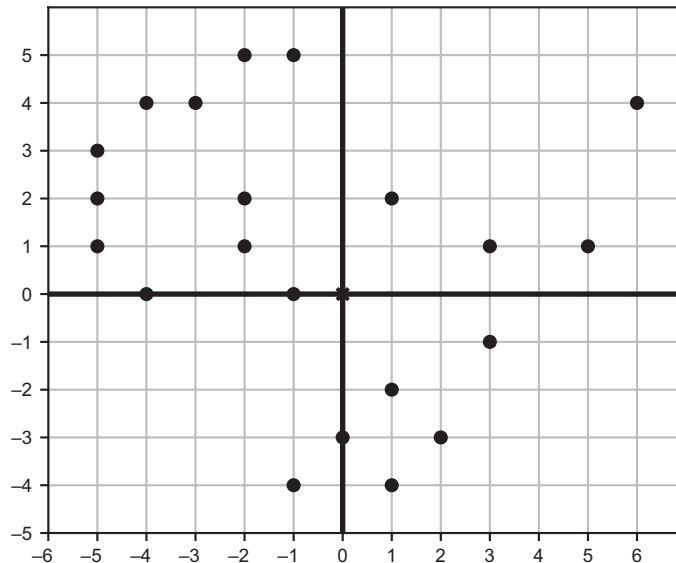


Figure 2.7 Plotting the dinosaur's points with the `draw` function in Python

As a next step in our drawing process, we can connect some dots. A first segment might connect the point (6, 4) with the point (3, 1) on the dinosaur's tail. We can draw the points along with this new segment using this function call, and figure 2.8 shows the results:

```
draw(
    Points(*dino_vectors),
    Segment((6, 4), (3, 1))
)
```

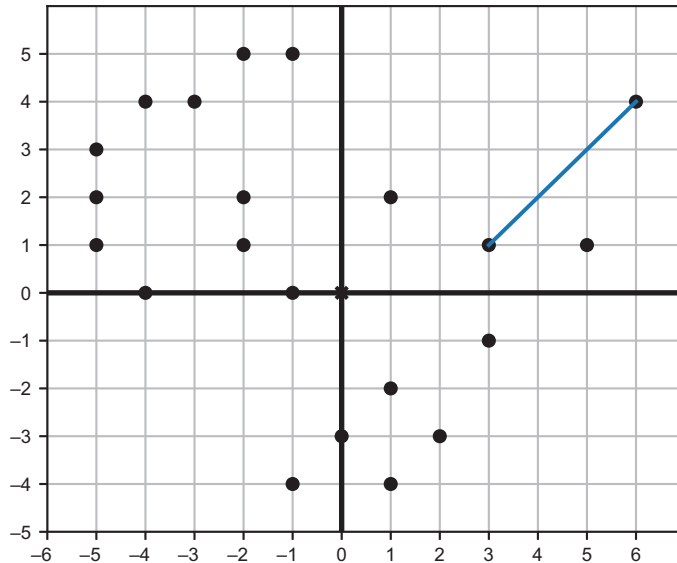


Figure 2.8 The dinosaur's points with a line segment connecting the first two points (6, 4) and (3, 1)

The line segment is actually the collection consisting of the points (6, 4) and (3, 1) as well as all of the points lying on the straight line between them. The `draw` function automatically colors all of the pixels at those points blue. The `Segment` class is a useful abstraction because we don't have to build every segment from the points that make up our geometric object (in this case, the dinosaur). Drawing 20 more segments, we get the complete outline of the dinosaur (figure 2.9).

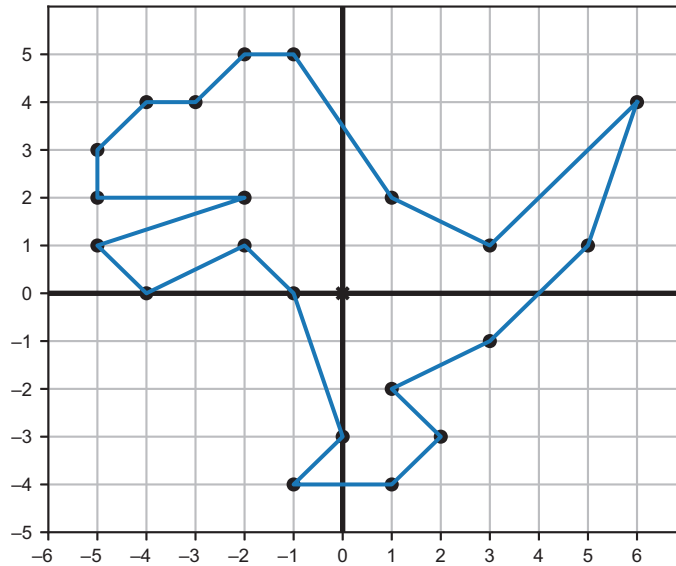


Figure 2.9 A total of 21 function calls give us 21 line segments, completing the outline of the dinosaur.

In principle, we can now outline any kind of 2D shape we want, provided we have all of the vectors to specify it. Coming up with all of the coordinates by hand can be tedious, so we'll start to look at ways to do computations with vectors to find their coordinates automatically.

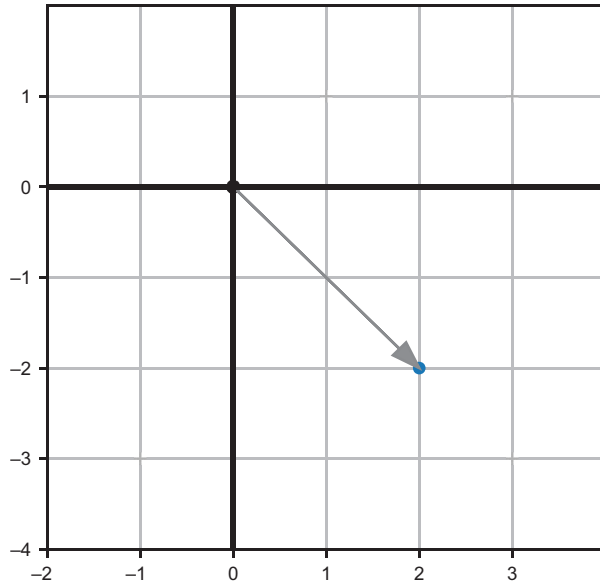
2.1.3 Exercises

Exercise 2.1 What are the x - and y -coordinates of the point at the tip of the dinosaur's toe?

Solution $(-1, -4)$

Exercise 2.2 Draw the point in the plane and the arrow corresponding to the point $(2, -2)$.

Solution Represented as a point on the plane and an arrow, $(2, -2)$ looks like this:



The point and arrow representing $(2, -2)$

Exercise 2.3 By looking at the locations of the dinosaur's points, infer the remaining vectors not included in the `dino_vectors` list. For instance, I already included $(6, 4)$, which is the tip of the dinosaur's tail, but I didn't include the point $(-5, 3)$, which is a point on the dinosaur's nose. When you're done, `dino_vectors` should be a list of 21 vectors represented as coordinate pairs.

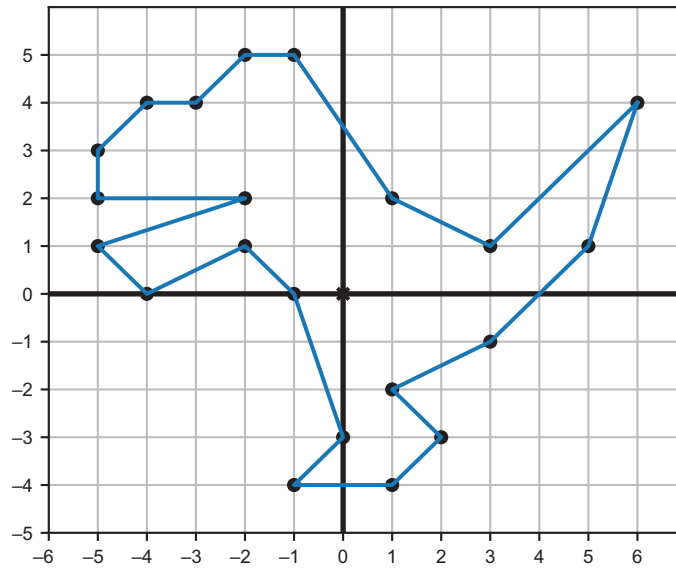
Solution The complete set of vectors outlining the dinosaur is as follows:

```
dino_vectors = [(6,4), (3,1), (1,2), (-1,5), (-2,5), (-3,4), (-4,4),
                (-5,3), (-5,2), (-2,2), (-5,1), (-4,0), (-2,1), (-1,0), (0,-3),
                (-1,-4), (1,-4), (2,-3), (1,-2), (3,-1), (5,1)
                ]
```

Exercise 2.4 Draw the dinosaur with the dots connected by constructing a Polygon object with the dino_vectors as its vertices.

Solution

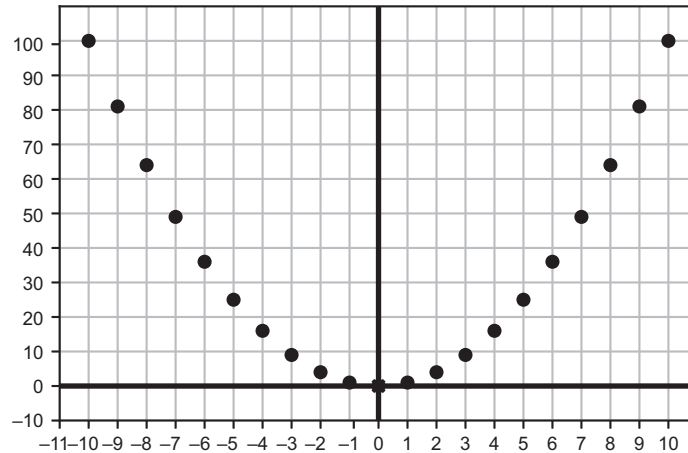
```
draw(
    Points(*dino_vectors),
    Polygon(*dino_vectors)
)
```



The dinosaur drawn as a polygon.

Exercise 2.5 Draw the vectors (x, x^2) for x in the range from $x = -10$ to $x = 11$ as points (dots) using the `draw` function. What is the result?

Solution The pairs draw the graph for the function $y = x^2$, plotted for the integers from 10 to 10:



Points on the graph for $y = x^2$

To make this graph, I used two keyword arguments for the `draw` function. The `grid` keyword argument of `(1, 10)` draws vertical grid lines every one unit and horizontal grid lines every ten units. The `nice_aspect_ratio` keyword argument set to `False` tells the graph it doesn't have to keep the x -axis and the y -axis scales the same:

```
draw(
    Points(*[(x,x**2) for x in range(-10,11)]),
    grid=(1,10),
    nice_aspect_ratio=False
)
```

2.2 Plane vector arithmetic

Like numbers, vectors have their own kind of arithmetic; we can combine vectors with operations to make new vectors. The difference with vectors is that we can visualize the results. Operations from vector arithmetic all accomplish useful geometric transformations, not just algebraic ones. We'll start with the most basic operation: *vector addition*.

Vector addition is simple to calculate: given two input vectors, you add their x -coordinates to get the resulting x -coordinate and then you add their y -coordinates to get the resulting y -coordinate. Creating a new vector with these summed coordinates

gives you the *vector sum* of the original vectors. For instance, $(4, 3) + (-1, 1) = (3, 4)$ because $4 + (-1) = 3$ and $3 + 1 = 4$. Vector addition is a one-liner to implement in Python:

```
def add(v1, v2):
    return (v1[0] + v2[0], v1[1] + v2[1])
```

Because we can interpret vectors as arrows or as points in the plane, we can visualize the result of the addition in both ways (figure 2.10). As a point in the plane, you can reach $(-1, 1)$ by starting at the origin, which is $(0, 0)$, and move one unit to the left and one unit up. You reach the vector sum of $(4, 3) + (-1, 1)$ by starting instead at $(4, 3)$ and moving one unit to the left and one unit up. This is the same as saying you traverse one arrow and then traverse the second.

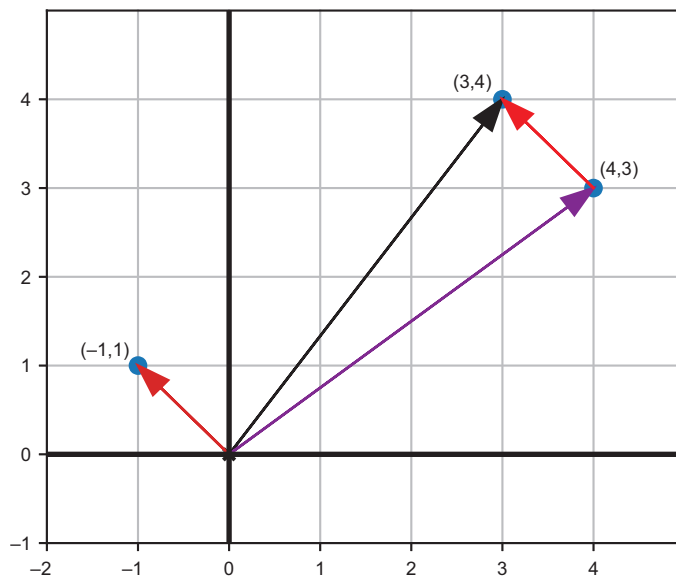


Figure 2.10 Picturing the vector sum of $(4, 3)$ and $(-1, 1)$

The rule for vector addition of arrows is sometimes called *tip-to-tail* addition. That's because if you move the tail of the second arrow to the tip of the first (without changing its length or direction!), then the sum is the arrow from the start of the first to the end of the second (figure 2.11).

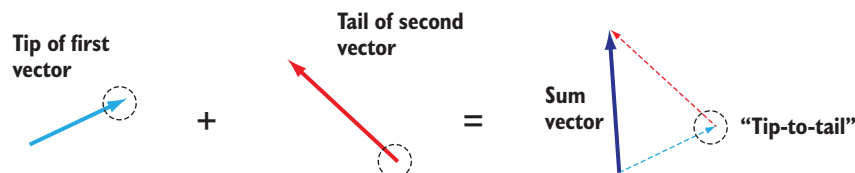


Figure 2.11 Tip-to-tail addition of vectors

When we talk about arrows, we really mean “a specific distance in a specific direction.” If you walk one distance in one direction and another distance in another direction, the vector sum tells you the overall distance and direction you traveled (figure 2.12).

Adding a vector has the effect of moving or *translating* an existing point or collection of points. If we add the vector $(-1.5, -2.5)$ to every vector of `dino_vectors`, we get a new list of vectors, each of which is 1.5 units left and 2.5 units down from one of the original vectors. Here’s the code for that:

```
dino_vectors2 = [add((-1.5, -2.5), v) for v in dino_vectors]
```

The result is the same dinosaur shape shifted down and to the left by the vector $(-1.5, -2.5)$. To see this (figure 2.13), we can draw both dinosaurs as polygons:

```
draw(
    Points(*dino_vectors, color=blue),
    Polygon(*dino_vectors, color=blue),
    Points(*dino_vectors2, color=red),
    Polygon(*dino_vectors2, color=red)
)
```

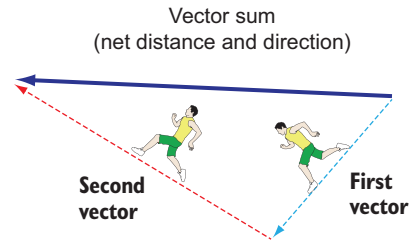


Figure 2.12 The vector sum as an overall distance and direction traveled in the plane.

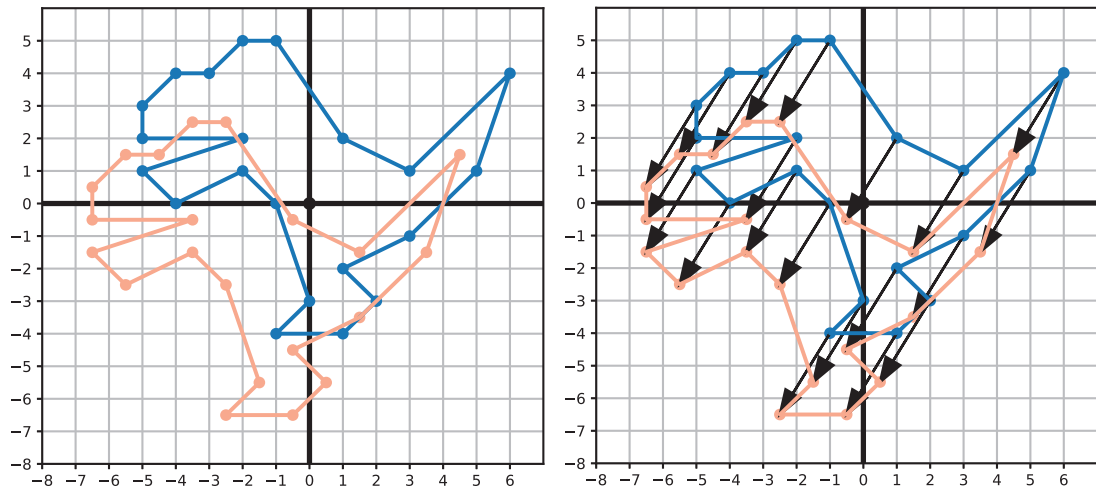


Figure 2.13 The original dinosaur (blue) and the translated copy (red). Each point on the translated dinosaur is moved by $(-1.5, -2.5)$ down and to the left from its location on the original dinosaur.

The arrows in the copy on the right show that each point moved down and to the left by the same vector: $(-1.5, -2.5)$. A translation like this is useful if, for instance, we want to make the dinosaur a moving character in a 2D computer game. Depending on the button pressed by the user, the dinosaur could translate in the corresponding direction on the screen. We'll implement a real game like this with moving vector graphics in chapters 7 and 9.

2.2.1 Vector components and lengths

Sometimes it's useful to take a vector we already have and decompose it as a sum of smaller vectors. For example, if I were asking for walking directions in New York City, it would be much more useful to hear “go four blocks east and three blocks north” rather than “go 800 meters northeast.” Similarly, it can be useful to think of vectors as a sum of a vector pointing in the x direction and a vector pointing in the y direction.

As an example, figure 2.14 shows the vector $(4, 3)$ rewritten as the sum $(4, 0) + (0, 3)$. Thinking of the vector $(4, 3)$ as a navigation path in the plane, the sum $(4, 0) + (0, 3)$ gets us to the same point along a different path.

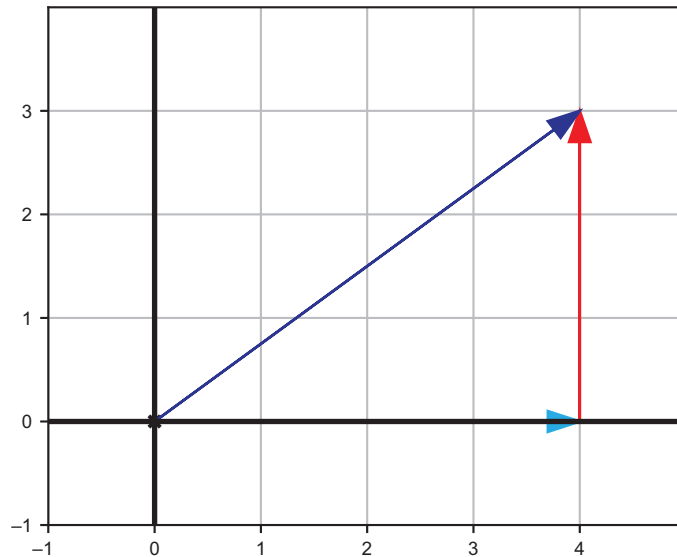


Figure 2.14 Breaking the vector $(4, 3)$ into the sum $(4, 0) + (0, 3)$

The two vectors $(4, 0)$ and $(0, 3)$ are called the x and y components, respectively. If you couldn't walk diagonally in this plane (as if it were New York City), you would need to walk four units to the right and then three units up to get to the same destination, a total of seven units.

The *length* of a vector is the length of the arrow that represents it, or equivalently, the distance from the origin to the point that represents it. In New York City, this could be the distance between two intersections “as the crow flies.” The length of a vector in the x or y direction can be measured immediately as a number of ticks passed on the corresponding axis: $(4, 0)$ or $(0, 4)$ are both vectors of the same length, 4, albeit in different directions. In general, though, vectors can lie diagonally, and we need to do a calculation to get their lengths.

You may recall the relevant formula: the *Pythagorean theorem*. For a right triangle (a triangle having two sides meeting at a 90° angle), the Pythagorean theorem says that the square of the length of the longest side is the sum of squares of the lengths of the other two sides. The longest side is called the *hypotenuse*, and its length is denoted by c in the memorable formula $a^2 + b^2 = c^2$, where a and b are the lengths of the other two sides. With $a = 4$ and $b = 3$, we can find c as the square root of $4^2 + 3^2$ (figure 2.15).

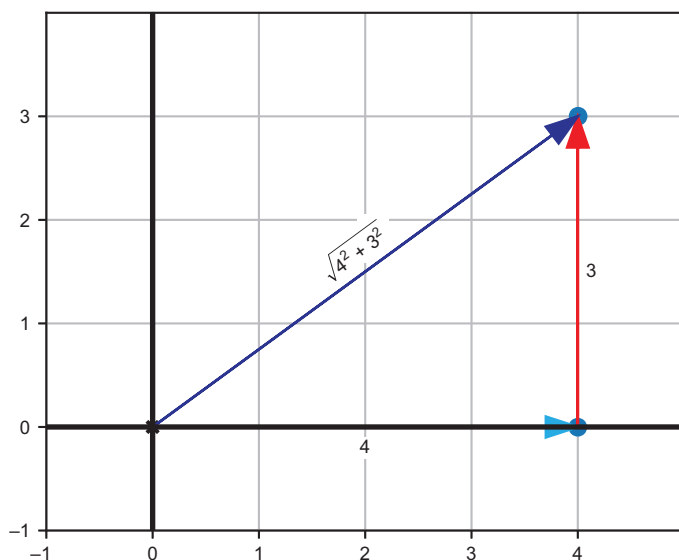


Figure 2.15 Using the Pythagorean theorem to find the length of a vector from the lengths of its x - and y -components

Breaking a vector into components is handy because it always gives us a right triangle. If we know the lengths of the components, we can compute the length of the hypotenuse, which is the length of the vector. Our vector $(4, 3)$ is equal to $(4, 0) + (0, 3)$, a sum of two perpendicular vectors whose sides are 4 and 3, respectively. The length of the vector $(4, 3)$ is the square root of $4^2 + 3^2$, which is the square root of 25, or 5. In a city with perfectly square blocks, traveling 4 blocks east and 3 blocks north would take us the equivalent of 5 blocks northeast.

This is a special case where the distance turns out to be an integer, but typically, lengths that come out of the Pythagorean theorem are not whole numbers. The

length of $(-3, 7)$ is given in terms of the lengths of its components 3 and 7 by the following computation:

$$\sqrt{3^2 + 7^2} = \sqrt{9 + 49} = \sqrt{58} = 7.61577\dots$$

We can translate this formula into a `length` function in Python, which takes a 2D vector and returns its floating-point length:

```
from math import sqrt
def length(v):
    return sqrt(v[0]**2 + v[1]**2)
```

2.2.2 Multiplying vectors by numbers

Repeated addition of vectors is unambiguous; you can keep stacking arrows tip-to-tail as long as you want. If a vector named \mathbf{v} has coordinates $(2, 1)$, then the fivefold sum $\mathbf{v} + \mathbf{v} + \mathbf{v} + \mathbf{v} + \mathbf{v}$ would look like that shown in figure 2.16.

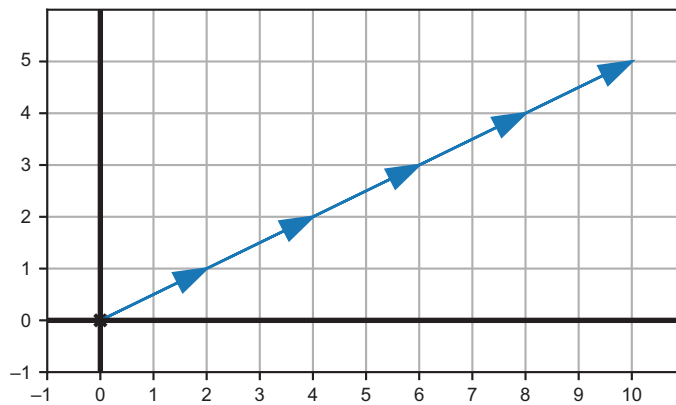


Figure 2.16 Repeated addition of the vector $\mathbf{v} = (2, 1)$ with itself

If \mathbf{v} were a number, we wouldn't bother writing $\mathbf{v} + \mathbf{v} + \mathbf{v} + \mathbf{v} + \mathbf{v}$. Instead, we'd write the simpler product $5 \cdot \mathbf{v}$. There's no reason we can't do the same for vectors. The result of adding \mathbf{v} to itself 5 times is a vector in the same direction but with 5 times the length. We can run with this definition, which lets us multiply a vector by any whole or fractional number.

The operation of multiplying a vector by a number is called *scalar multiplication*. When working with vectors, ordinary numbers are often called *scalars*. It's also an appropriate term because the effect of this operation is *scaling* the target vector by the given factor. It doesn't matter if the scalar is a whole number; we can easily draw a vector that is 2.5 times the length of another (figure 2.17).

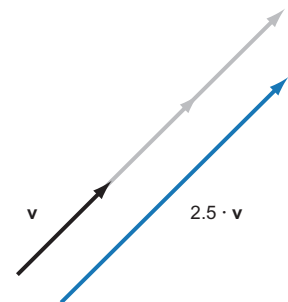


Figure 2.17 Scalar multiplication of a vector \mathbf{v} by 2.5

The result on the vector components is that each component is scaled by the same factor. You can picture scalar multiplication as changing the size of the right triangle defined by a vector and its components, but not affecting its aspect ratio. Figure 2.18 superimposes a vector \mathbf{v} and its scalar multiple $1.5 \cdot \mathbf{v}$, where the scalar multiple is 1.5 times as long. Its components are also 1.5 times the length of the original components of \mathbf{v} .

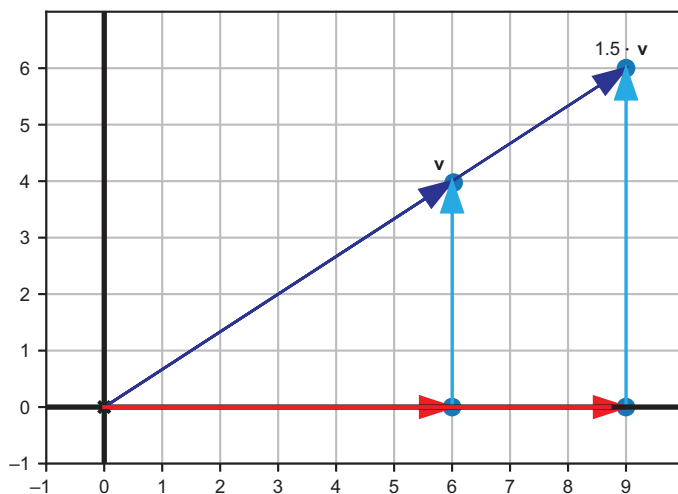


Figure 2.18 Scalar multiplication of a vector scales both components by the same factor.

In coordinates, the scalar multiple of 1.5 times the vector $\mathbf{v} = (6, 4)$ gives us a new vector $(9, 6)$, where each component is 1.5 times its original value. Computationally, we execute any scalar multiplication on a vector by multiplying each coordinate of the vector by the scalar. As a second example, scaling a vector $\mathbf{w} = (1.2, -3.1)$ by a factor 6.5 can be accomplished like this:

$$6.5 \cdot \mathbf{w} = 6.5 \cdot (1.2, -3.1) = (6.5 \cdot 1.2, 6.5 \cdot -3.1) = (7.8, -20.15)$$

We tested this method for a fractional number as the scalar, but we should also test a negative number. If our original vector is $(6, 4)$, what is $-\frac{1}{2}$ times that vector? Multiplying the coordinates, we expect the answer to be $(-3, -2)$. Figure 2.19 shows that this vector is half the length of the original and points in the opposite direction.

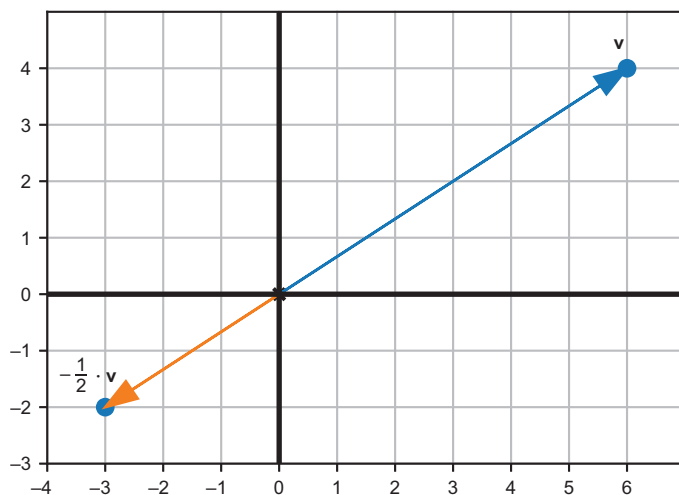


Figure 2.19 Scalar multiplication of a vector by a negative number, $-\frac{1}{2}$

2.2.3 Subtraction, displacement, and distance

Scalar multiplication agrees with our intuition for multiplying numbers. A whole number multiple of a number is the same as a repeated sum, and the same holds for vectors. We can make a similar argument for negative vectors and vector subtraction.

Given a vector \mathbf{v} , the *opposite* vector, $-\mathbf{v}$, is the same as the scalar multiple $-1 \cdot \mathbf{v}$. If \mathbf{v} is $(-4, 3)$, its opposite, $-\mathbf{v}$, is $(4, -3)$ as shown in figure 2.20. We get this by multiplying each coordinate by -1 , or in other words, changing the sign of each.

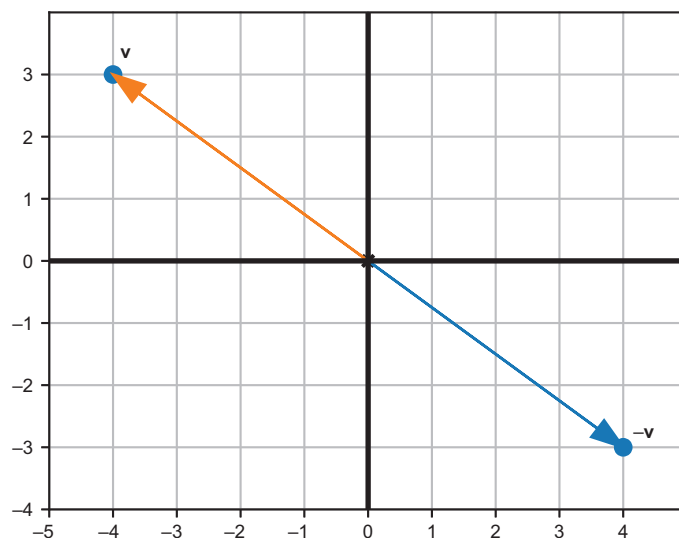


Figure 2.20 The vector $\mathbf{v} = (-4, 3)$ and its opposite $-\mathbf{v} = (4, -3)$.

On the number line, there are only two directions from zero: positive and negative. In the plane, there are many directions (infinitely many, in fact), so we can't say that one of \mathbf{v} and $-\mathbf{v}$ is positive while the other is negative. What we can say is that for any vector \mathbf{v} , the opposite vector $-\mathbf{v}$ will have the same length, but it will point in the opposite direction.

Having a notion of negating a vector, we can define *vector subtraction*. For numbers, $x - y$ is the same as $x + (-y)$. We set the same convention for vectors. To subtract a vector \mathbf{w} from a vector \mathbf{v} , you add the vector $-\mathbf{w}$ to \mathbf{v} . Thinking of vectors \mathbf{v} and \mathbf{w} as points, $\mathbf{v} - \mathbf{w}$ is the position of \mathbf{v} relative to \mathbf{w} . Thinking instead of \mathbf{v} and \mathbf{w} as arrows beginning at the origin, figure 2.21 shows that $\mathbf{v} - \mathbf{w}$ is the arrow from the tip of \mathbf{w} to the tip of \mathbf{v} .

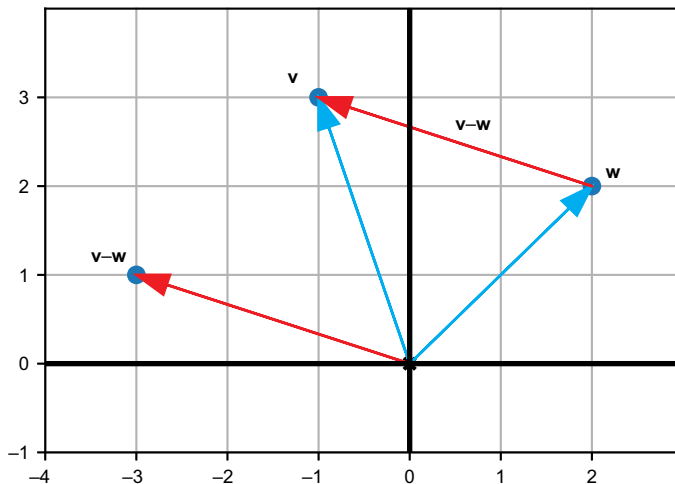


Figure 2.21 The result of subtracting $\mathbf{v} - \mathbf{w}$ is an arrow from the tip of \mathbf{w} to the tip of \mathbf{v} .

The coordinates of $\mathbf{v} - \mathbf{w}$ are the differences of the coordinates \mathbf{v} and \mathbf{w} . In figure 2.21, $\mathbf{v} = (-1, 3)$ and $\mathbf{w} = (2, 2)$. The difference for $\mathbf{v} - \mathbf{w}$ has the coordinates $(-1 - 2, 3 - 2) = (-3, 1)$.

Let's look at the difference of the vectors $\mathbf{v} = (-1, 3)$ and $\mathbf{w} = (2, 2)$ again. You can use the draw function I gave you to plot the points \mathbf{v} and \mathbf{w} and to draw a segment between them. The code looks like this:

```
draw(
    Points((2,2), (-1,3)),
    Segment((2,2), (-1,3), color=red)
)
```

The difference for the vectors $\mathbf{v} - \mathbf{w} = (-3, 1)$ tells us that if we start at point w , we need to go three units left and one unit up to get to point v . This vector is sometimes called the *displacement* from w to v . The straight, red line segment from w to v in figure 2.22, drawn by this Python code, shows the *distance* between the two points.

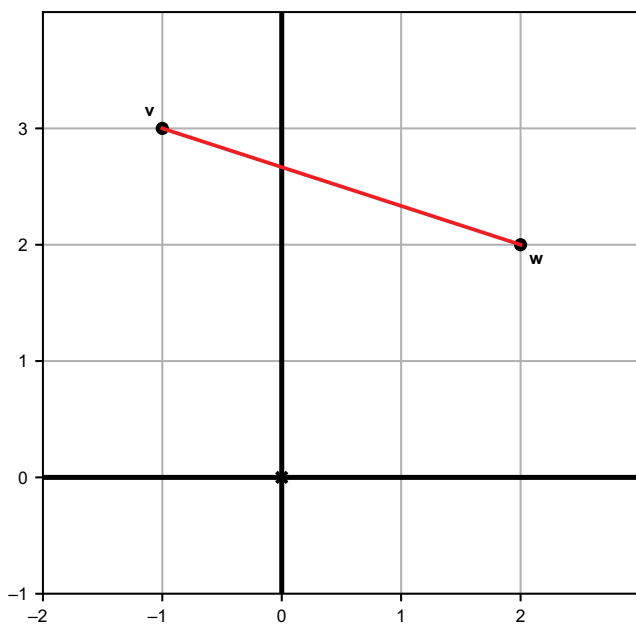


Figure 2.22 The distance between two points in the plane

The length of the line segment is computed with the Pythagorean theorem as follows:

$$\sqrt{(-3)^2 + 1^2} = \sqrt{9 + 1} = \sqrt{10} = 3.162 \dots$$

While the displacement is a vector, the distance is a scalar (a single number). The distance on its own is not enough to specify how to get from w to v ; there are plenty of points that have the same distance from w . Figure 2.23 shows a few others with whole number coordinates.

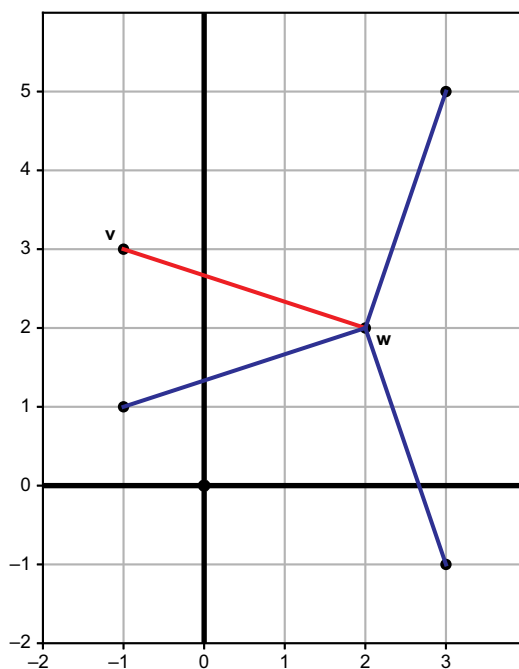


Figure 2.23 Several points equidistant from $w = (2, 2)$

2.2.4 Exercises

Exercise 2.6 If the vector $\mathbf{u} = (-2, 0)$, the vector $\mathbf{v} = (1.5, 1.5)$, and the vector $\mathbf{w} = (4, 1)$, what are the results of $\mathbf{u} + \mathbf{v}$, $\mathbf{v} + \mathbf{w}$, and $\mathbf{u} + \mathbf{w}$? What is the result of $\mathbf{u} + \mathbf{v} + \mathbf{w}$?

Solution With the vector $\mathbf{u} = (-2, 0)$, the vector $\mathbf{v} = (1.5, 1.5)$, and the vector $\mathbf{w} = (4, 1)$, the results are as follows:

$$\mathbf{u} + \mathbf{v} = (-0.5, 1.5)$$

$$\mathbf{v} + \mathbf{w} = (5.5, 2.5)$$

$$\mathbf{u} + \mathbf{w} = (2, 1)$$

$$\mathbf{u} + \mathbf{v} + \mathbf{w} = (3.5, 2.5)$$

Exercise 2.7—Mini Project You can add any number of vectors together by summing *all* of their *x*-coordinates and *all* of their *y*-coordinates. For instance, the fourfold sum $(1, 2) + (2, 4) + (3, 6) + (4, 8)$ has *x* component $1 + 2 + 3 + 4 = 10$ and *y* component $2 + 4 + 6 + 8 = 20$, making the result $(10, 20)$. Implement a revised `add` function that takes any number of vectors as arguments.

Solution

```
def add(*vectors):
    return (sum([v[0] for v in vectors]), sum([v[1] for v in vectors]))
```

Exercise 2.8 Write a function `translate(translation, vectors)` that takes a translation vector and a list of input vectors, and returns a list of the input vectors all translated by the translation vector. For instance, `translate((1, 1), [(0, 0), (0, 1), (-3, -3)])` should return `[(1, 1), (1, 2), (-2, -2)]`.

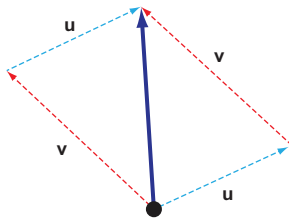
Solution

```
def translate(translation, vectors):
    return [add(translation, v) for v in vectors]
```

Exercise 2.9—Mini Project Any sum of vectors $\mathbf{v} + \mathbf{w}$ gives the same result as $\mathbf{w} + \mathbf{v}$. Explain why this is true using the definition of the vector sum on coordinates. Also, draw a picture to show why it is true geometrically.

Solution If you add two vectors $\mathbf{u} = (a, b)$ and $\mathbf{v} = (c, d)$, the coordinates a, b, c , and d are all real numbers. The result of the vector addition is $\mathbf{u} + \mathbf{v} = (a + c, b + d)$. The

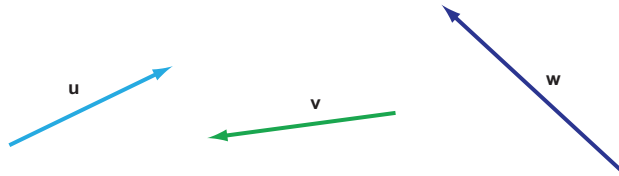
result of $\mathbf{v} + \mathbf{u}$ is $(c + a, d + b)$, which is the same pair of coordinates because order doesn't matter when adding real numbers. Tip-to-tail addition in either order yields the same sum vector. Visually, we can see this by adding an example pair of vectors tip-to-tail:



Tip-to-tail addition in either order yields the same sum vector.

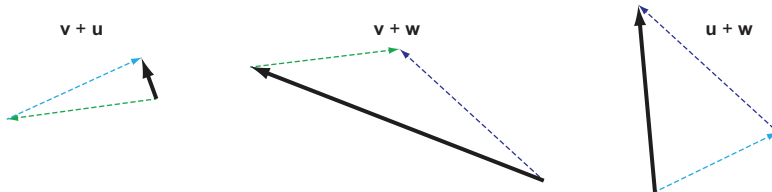
It doesn't matter whether you add $\mathbf{u} + \mathbf{v}$ or $\mathbf{v} + \mathbf{u}$ (dashed lines), you get the same result vector (solid line). In geometric terms, \mathbf{u} and \mathbf{v} define a parallelogram, and the vector sum is the length of the diagonal.

Exercise 2.10 Among the following three arrow vectors (labeled \mathbf{u} , \mathbf{v} , and \mathbf{w}), which pair has the sum that gives the *longest* arrow? Which pair sums to give the *shortest* arrow?



Which pair sums to the longest or shortest arrow?

Solution We can measure each of the vector sums by placing the vectors tip-to-tail:



Tip-to-tail addition of the vectors in question

Inspecting the results, we can see that $\mathbf{v} + \mathbf{u}$ is the shortest vector (\mathbf{u} and \mathbf{v} are in nearly opposite directions and come close to canceling each other out). The longest vector is $\mathbf{v} + \mathbf{w}$.

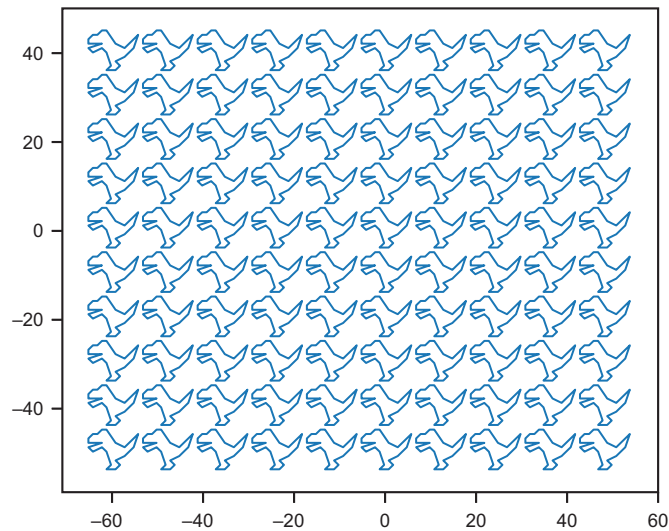
Exercise 2.11—Mini Project Write a Python function using vector addition to show 100 simultaneous and non-overlapping copies of the dinosaur. This shows the power of computer graphics; imagine how tedious it would be to specify all 2,100 coordinate pairs by hand!

Solution With some trial and error, you can translate the dinosaurs in the vertical and horizontal direction so that they don't overlap, and set the boundaries appropriately. I decided to leave out the grid lines, axes, origin, and points to make the drawing clearer. My code looks like this:

```
def hundred_dinos():
    translations = [(12*x, 10*y)
                    for x in range(-5, 5)
                    for y in range(-5, 5)]
    dinos = [Polygon(*translate(t, dino_vectors), color=blue)
              for t in translations]
    draw(*dinos, grid=None, axes=None, origin=None)

hundred_dinos()
```

The result is as follows:



100 dinosaurs. Run for your life!

Exercise 2.12 Which is longer, the x or y component of $(3, -2) + (1, 1) + (-2, -2)$?

Solution The result of the vector sum $(3, -2) + (1, 1) + (-2, -2)$ is $(2, -3)$. The x component is $(2, 0)$ and the y component is $(0, -3)$. The x component has a length of 2 units (to the right), while the y component has a length of 3 units (downward because it is negative). This makes the y component longer.

Exercise 2.13 What are the components and lengths of the vectors $(-6, -6)$ and $(5, -12)$?

Solution The components of $(-6, -6)$ are $(-6, 0)$ and $(0, -6)$, both having length 6. The length of $(-6, -6)$ is the square root of $6^2 + 6^2$, which is approximately 8.485.

The components of $(5, -12)$ are $(5, 0)$ and $(0, -12)$, having lengths of 5 and 12, respectively. The length of $(5, -12)$ is given by the square root of $5^2 + 12^2 = 25 + 144 = 169$. The result of the square root is exactly 13.

Exercise 2.14 Suppose I have a vector \mathbf{v} that has a length of 6 and an x component $(1, 0)$. What are the possible coordinates of \mathbf{v} ?

Solution The x component of $(1, 0)$ has length 1 and the total length is 6, so the length b of the y component must satisfy the equation $1^2 + b^2 = 6^2$, or $1 + b^2 = 36$. Then $b^2 = 35$ and the length of the y component is approximately 5.916. This doesn't tell us the direction of the y component, however. The vector \mathbf{v} could either be $(1, 5.916)$ or $(1, -5.916)$.

Exercise 2.15 What vector in the `dino_vectors` list has the longest length? Use the `length` function we wrote to compute the answer quickly.

Solution

```
>>> max(dino_vectors, key=length)
(6, 4)
```

Exercise 2.16 Suppose a vector \mathbf{w} has the coordinates $(\sqrt{2}, \sqrt{3})$. What are the approximate coordinates of the scalar multiple $\pi \cdot \mathbf{w}$? Draw an approximation of the original vector and the new vector.

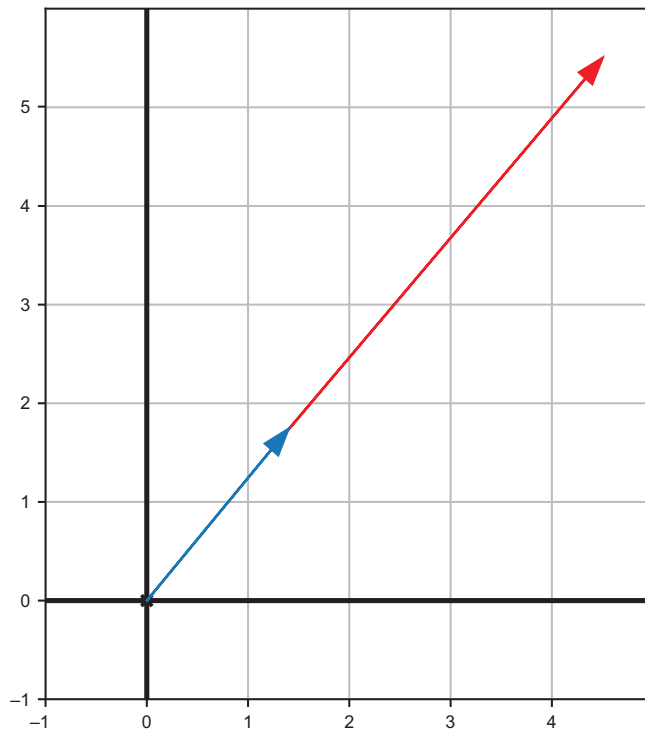
Solution The value of $(\sqrt{2}, \sqrt{3})$ is approximately

(1.4142135623730951, 1.7320508075688772)

Scaling each coordinate by a factor of π (pi), we get

(4.442882938158366, 5.441398092702653)

The scaled vector is longer than the original as shown here:



The original vector
(shorter) and its
scaled version (longer)

Exercise 2.17 Write a Python function `scale(s,v)` that multiplies the input vector `v` by the input scalar `s`.

Solution

```
def scale(scalar,v):  
    return (scalar * v[0], scalar * v[1])
```

Exercise 2.18—Mini Project Convince yourself algebraically that scaling the coordinates by a factor also scales the length of the vector by the same factor. Suppose a vector of length c has the coordinates (a, b) . Show that for any non-negative real number s , the length of $(s \cdot a, s \cdot b)$ is $s \cdot c$. (This can't work for a negative value of s because a vector can't have a negative length.)

Solution We use the notation $|(a, b)|$ to denote the length of a vector (a, b) . So, the premise of the exercise tells us:

$$c = \sqrt{a^2 + b^2} = |(a, b)|$$

From that, we can compute the length of (sa, sb) :

$$\begin{aligned} |(sa, sb)| &= \sqrt{(sa)^2 + (sb)^2} \\ &= \sqrt{s^2 a^2 + s^2 b^2} \\ &= \sqrt{s^2 \cdot (a^2 + b^2)} \\ &= |s| \cdot \sqrt{a^2 + b^2} \\ &= |s| \cdot c \end{aligned}$$

As long as s isn't negative, it's the same as its absolute value: $s = |s|$. Then the length of the scaled vector is sc as we hoped to show.

Exercise 2.19—Mini Project Suppose $\mathbf{u} = (-1, 1)$ and $\mathbf{v} = (1, 1)$, and suppose r and s are real numbers. Specifically, let's assume $-3 < r < 3$ and $-1 < s < 1$. Where are the possible points on the plane where the vector $r \cdot \mathbf{u} + s \cdot \mathbf{v}$ could end up?

Note that the order of operations is the same for vectors as it is for numbers. We assume scalar multiplication is carried out first and then vector addition (unless parentheses specify otherwise).

Solution If $\mathbf{r} = 0$, the possibilities lie on the line segment from $(-1, -1)$ to $(1, 1)$. If \mathbf{r} is not zero, the possibilities can leave that line segment in the direction of $(-1, 1)$ or $(1, 1)$ by up to three units. The region of possible results is the parallelogram with vertices at $(-2, 0)$, $(2, 0)$, $(-1, 1)$, and $(1, 1)$. We can test many random, allowable values of \mathbf{r} and \mathbf{s} to validate this:

```
from random import uniform
u = (-1,1)
v = (1,1)
def random_r():
    return uniform(-3,3)
def random_s():
    return uniform(-1,1)
```

(continued)

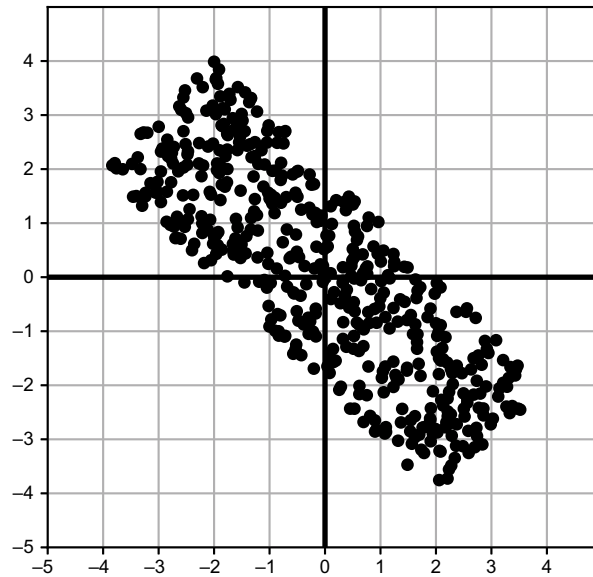
```

possibilities = [add(scale(random_r(), u), scale(random_s(), v))
                  for i in range(0,500)]

draw(
    Points(*possibilities)
)

```

If you run this code, you get a picture like the following, showing the possible points where $r \cdot \mathbf{u} + s \cdot \mathbf{v}$ could end up given the constraints:



Location of possible points for $r \cdot \mathbf{u} + s \cdot \mathbf{v}$ given the constraints.

Exercise 2.20 Show algebraically why a vector and its opposite have the same length.

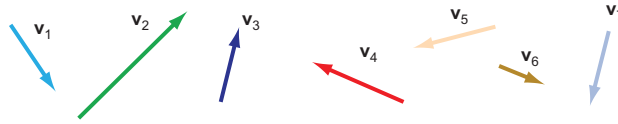
Hint Plug the coordinates and their opposites into the Pythagorean theorem.

Solution The opposite vector of (a, b) has coordinates $(-a, -b)$, but this doesn't affect the length:

$$\sqrt{(-a)^2 + (-b)^2} = \sqrt{(-a) \cdot (-a) + (-b) \cdot (-b)} = \sqrt{a^2 + b^2}$$

The vector $(-a, -b)$ has the same length as (a, b) .

Exercise 2.21 Of the following seven vectors, represented as arrows, which two are a pair of opposite vectors?



Solution Vectors v_3 and v_7 are the pair of opposite vectors.

Exercise 2.22 Suppose \mathbf{u} is any 2D vector. What are the coordinates of $\mathbf{u} + -\mathbf{u}$?

Solution A 2D vector \mathbf{u} has some coordinates (a, b) . Its opposite has coordinates $(-a, -b)$, so:

$$\mathbf{u} + (-\mathbf{u}) = (a, b) + (-a, -b) = (a - a, b - b) = (0, 0)$$

The answer is $(0, 0)$. Geometrically, this means that if you follow a vector and then its opposite, you end up back at the origin, $(0, 0)$.

Exercise 2.23 For vectors $\mathbf{u} = (-2, 0)$, $\mathbf{v} = (1.5, 1.5)$, and $\mathbf{w} = (4, 1)$, what are the results of the vector subtractions $\mathbf{v} - \mathbf{w}$, $\mathbf{u} - \mathbf{v}$, and $\mathbf{w} - \mathbf{v}$?

Solution With $\mathbf{u} = (-2, 0)$, $\mathbf{v} = (1.5, 1.5)$, and $\mathbf{w} = (4, 1)$, we have

$$\mathbf{v} - \mathbf{w} = (-2.5, 0.5)$$

$$\mathbf{u} - \mathbf{v} = (-3.5, -1.5)$$

$$\mathbf{w} - \mathbf{v} = (2.5, -0.5)$$

Exercise 2.24 Write a Python function `subtract(v1,v2)` that returns the result of $\mathbf{v}_1 - \mathbf{v}_2$, taking two 2D vectors as inputs and returning a 2D vector as an output.

Solution

```
def subtract(v1,v2):
    return (v1[0] - v2[0], v1[1] - v2[1])
```

Exercise 2.25 Write a Python function `distance(v1, v2)` that returns the *distance* between two input vectors. (Note that the `subtract` function from the previous exercise already gives the *displacement*.)

Write another Python function `perimeter(vectors)` that takes a list of vectors as an argument and returns the sum of distances from each vector to the next, including the distance from the last vector to the first. What is the perimeter of the dinosaur defined by `dino_vectors`?

Solution The distance is just the length of the difference of the two input vectors:

```
def distance(v1, v2):
    return length(subtract(v1, v2))
```

For the perimeter, we sum the distances of every pair of subsequent vectors in the list, as well as the pair of the first and the last vectors:

```
def perimeter(vectors):
    distances = [distance(vectors[i], vectors[(i+1)%len(vectors)])
                 for i in range(0, len(vectors))]
    return sum(distances)
```

We can use a square with side length of one as a sanity check:

```
>>> perimeter([(1,0), (1,1), (0,1), (0,0)])
4.0
```

Then we can calculate the perimeter of the dinosaur:

```
>>> perimeter(dino_vectors)
44.77115093694563
```

Exercise 2.26—Mini Project Let \mathbf{u} be the vector $(1, -1)$. Suppose there is another vector \mathbf{v} with positive integer coordinates (n, m) such that $n > m$ and has a distance of 13 from \mathbf{u} . What is the displacement from \mathbf{u} to \mathbf{v} ?

Hint You can use Python to search for the vector \mathbf{v} .

Solution We only need to search possible integer pairs (n, m) where n is within 13 units of 1 and m is within 13 units of -1 :

```
for n in range(-12, 15):
    for m in range(-14, 13):
        if distance((n, m), (1, -1)) == 13 and n > m > 0:
            print((n, m))
```

There is one result: $(13, 4)$. It is 12 units to the right and 5 units up from $(1, -1)$, so the displacement is $(12, 5)$.

The length of a vector is not enough to describe it, nor is the distance between two vectors enough information to get from one to the other. In both cases, the missing ingredient is *direction*. If you know how long a vector is and you know what direction it is pointing, you can identify it and find its coordinates. To a large extent, this is what *trigonometry* is about, and we'll review that subject in the next section.

2.3 Angles and trigonometry in the plane

So far, we've used two "rulers" (called the x -axis and the y -axis) to measure vectors in the plane. An arrow from the origin covers some measurable displacement in the horizontal and vertical directions, and these values uniquely specify the vector. Instead of using two rulers, we could just as well use a ruler and a protractor. Starting with the vector $(4, 3)$, we can measure or calculate its length to be 5 units, and then use our protractor to identify the direction as shown in figure 2.24.

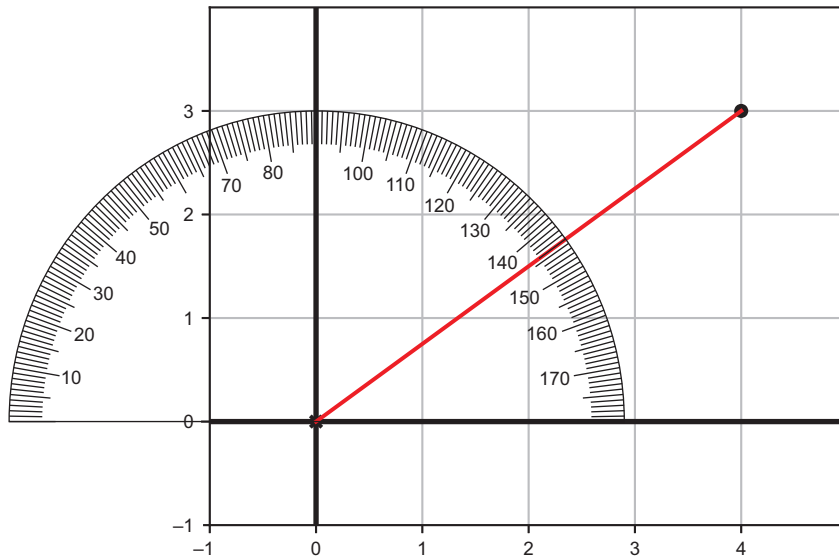


Figure 2.24 Using a protractor to measure the angle at which a vector points

This vector has a length of 5 units, and it points in a direction approximately 37° counterclockwise from the positive half of the x -axis. This gives us a new pair of numbers $(5, 37^\circ)$ that, like our original coordinates, uniquely specify the vector. These numbers are called *polar coordinates* and are just as good at describing points in the plane as the ones we've worked with so far, called *Cartesian coordinates*.

Sometimes, like when we're adding vectors, it's easier to use Cartesian coordinates. Other times, polar coordinates are more useful; for instance, when we want to look at vectors rotated by some angle. In code, we don't have literal rulers or protractors available, so we use trigonometric functions to convert back and forth instead.

2.3.1 From angles to components

Let's look at the reverse problem: imagine we already have an angle and a distance, say, 116.57° and 3. These define a pair of polar coordinates $(3, 116.57^\circ)$. How can we find the Cartesian coordinates for this vector geometrically?

First, we can position our protractor at the origin to find the right direction. We measure 116.57° counterclockwise from the positive x -axis and draw a line in that direction (figure 2.25). Our vector $(3, 116.57^\circ)$ lies somewhere on this line.

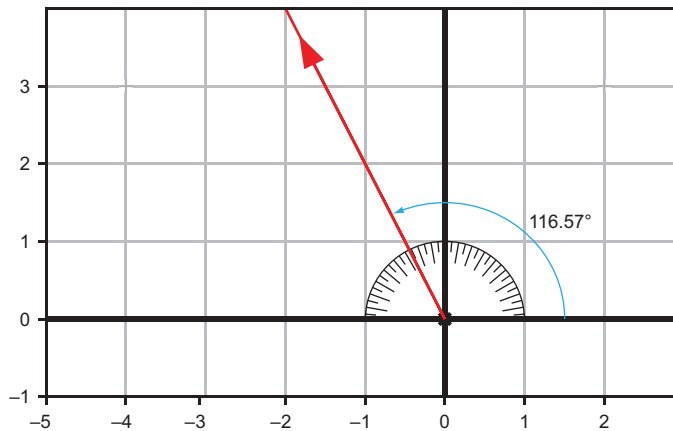


Figure 2.25 Measuring 116.57° from the positive x -axis using a protractor

The next step is to take a ruler and measure a point that is three units from the origin in this direction. Once we've found it, as in figure 2.26, we can measure the components and get our approximate coordinates $(-1.34, 2.68)$.

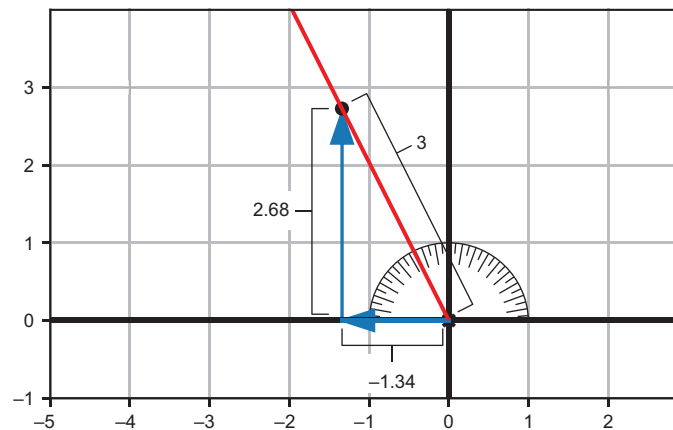


Figure 2.26 Using a ruler to measure the coordinates of the point that is three units from the origin

It may look like the angle 116.57° was a random choice, but it has a useful property. Starting from the origin and moving in that direction, you go up two units every time

you go one unit to the left. Vectors that approximately lie along that line include $(-1, 2)$, $(-3, 6)$ and, of course, $(-1.34, 2.68)$; the y -coordinates are -2 times their x -coordinates (figure 2.27).

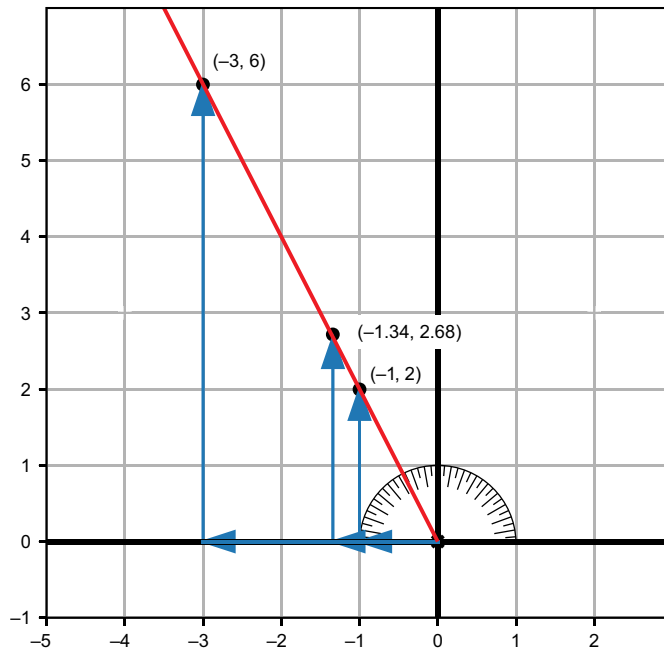


Figure 2.27 Traveling in the direction of 116.57° , you travel two units up for every unit you travel to the left.

The strange angle 116.57° happens to give us a nice round ratio of -2 . We won't always be lucky enough to get a whole number ratio, but every angle does give us a *constant* ratio. The angle 45° gives us one vertical unit for every one horizontal unit or a ratio of 1. Figure 2.28 shows another angle, 200° . This gives us a constant ratio of -0.36 vertical units for every -1 horizontal unit covered or a ratio of 0.36 .

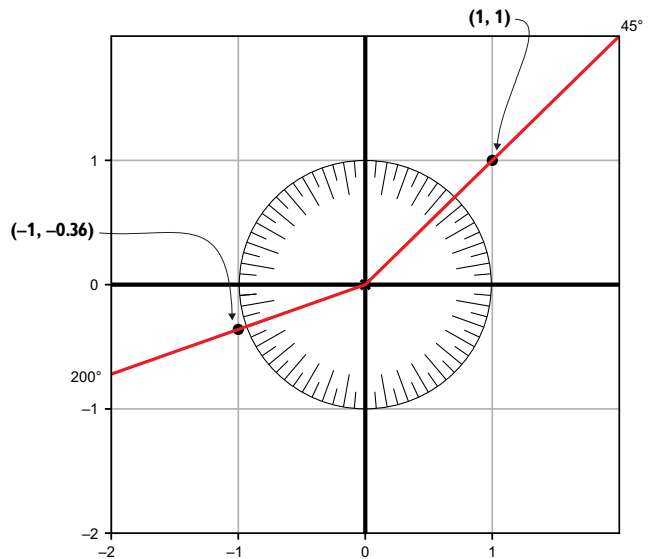


Figure 2.28 How much vertical distance is covered per unit of horizontal distance at different angles?

Given an angle, the coordinates of vectors along that angle will have a constant ratio. This ratio is called the *tangent* of the angle, and the tangent function is written as *tan*. You’ve seen a few of its approximate values so far:

$$\begin{aligned}\tan(37^\circ) &\approx \frac{3}{4} \\ \tan(116.57^\circ) &\approx -2 \\ \tan(45^\circ) &= 1 \\ \tan(200^\circ) &\approx 0.36\end{aligned}$$

Here, to denote *approximate* equality, I use the symbol \approx as opposed to the symbol $=$. The tangent function is a *trigonometric* function because it helps us measure triangles. (The “trigon” in “trigonometry” means triangle and “metric” means measurement.) Note that I haven’t told you *how* to calculate the tangent yet, only what a few of its values are. Python has a built-in tangent function that I’ll show you how to use shortly. You almost never have to worry about calculating (or measuring) the tangent of an angle yourself.

The tangent function is clearly related to our original problem of finding Cartesian coordinates for a vector given an angle and a distance. But it doesn’t actually provide the coordinates, only their ratio. For that, two other trigonometric functions are helpful: *sine* and *cosine*. If we measure some distance at some angle (figure 2.29), the tangent of the angle gives us the vertical distance covered divided by the horizontal distance.

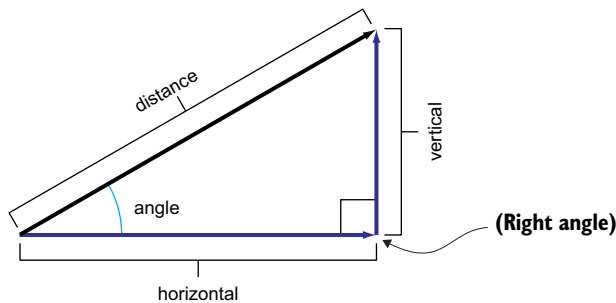


Figure 2.29 Schematic of distances and angles for a given vector

By comparison, the sine and cosine give us the vertical and horizontal distance covered relative to the overall distance. These are written *sin* and *cos* for short, and this equation shows the definitions for both:

$$\sin(\text{angle}) = \frac{\text{vertical}}{\text{distance}} \quad \cos(\text{angle}) = \frac{\text{horizontal}}{\text{distance}}$$

Let’s look at the angle 37° for a concrete example (figure 2.30). We saw that the point (4, 3) lies at a distance of 5 units from the origin at this angle.

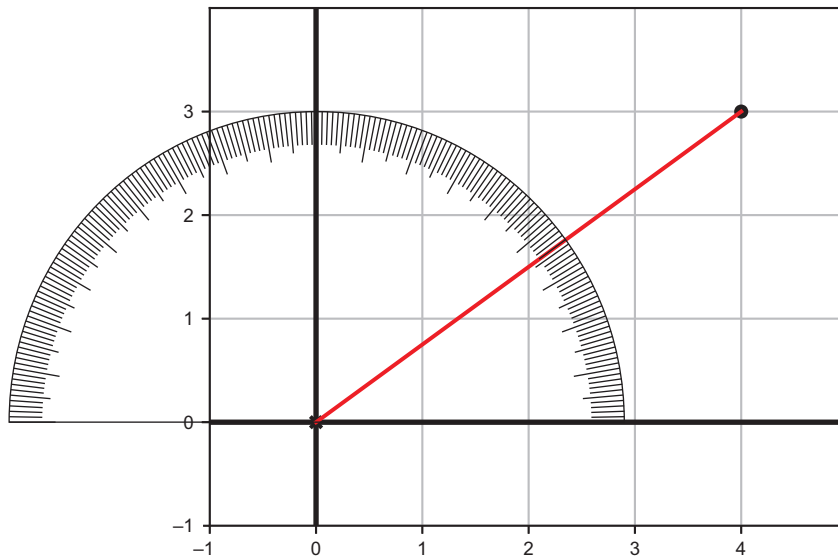


Figure 2.30 Measuring the angle to the point (4, 3) with a protractor

For every 5 units you travel at 37° , you cover approximately 3 vertical units. Therefore, we can write:

$$\sin(37^\circ) \approx 3/5$$

Similarly, for every 5 units you travel at 37° , you cover approximately 4 horizontal units, so we can write:

$$\cos(37^\circ) \approx 4/5$$

This is a general strategy for converting a vector in polar coordinates to corresponding Cartesian coordinates. If you know the sine and cosine of an angle θ (the Greek letter theta, commonly used for angles) and a distance r traveled in that direction, the Cartesian coordinates are given by $(r \cdot \cos(\theta), r \cdot \sin(\theta))$ and shown in figure 2.31.

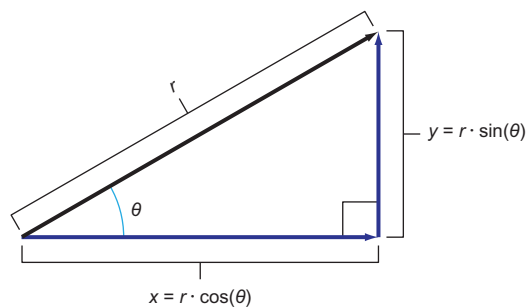


Figure 2.31 Picturing the conversion from polar coordinates to Cartesian coordinates for a right triangle

2.3.2 Radians and trigonometry in Python

Let's turn what we've reviewed about trigonometry into Python code. Specifically, let's build a function that takes a pair of polar coordinates (a length and an angle) and outputs a pair of Cartesian coordinates (lengths of x and y components).

The main hurdle is that Python's built-in trigonometric functions use different units than the ones we've used. We expect $\tan(45^\circ) = 1$, for instance, but Python gives us a much different result:

```
>>> from math import tan
>>> tan(45)
1.6197751905438615
```

Python doesn't use degrees, and neither do most mathematicians. Instead, they use units called *radians* to measure angles. The conversion factor is

$$1 \text{ radian} \approx 57.296^\circ$$

This may seem like an arbitrary conversion factor. Some more suggestive relationships between degrees and radians are given in terms of the special number π (pi), whose value is approximately 3.14159. Here are a few examples:

$$\pi \text{ radians} = 180^\circ$$

$$2\pi \text{ radians} = 360^\circ$$

In radians, half a trip around a circle is an angle of π and a whole revolution is 2π . These respectively agree with the half and whole circumference of a circle of radius 1 (figure 2.32).

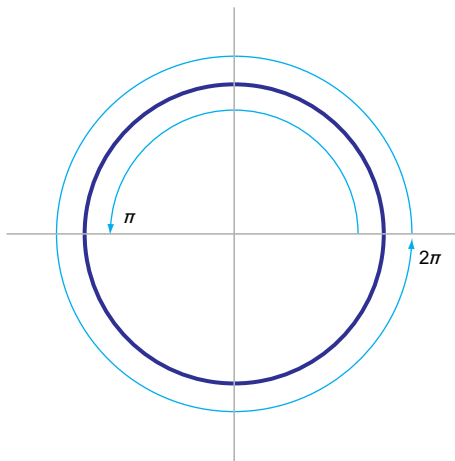


Figure 2.32 A half revolution is π radians, while a whole revolution is 2π radians.

You can think of radians as another kind of ratio: for a given angle, its measurement in radians tells you how many radiuses you've gone around the circle. Because of this special property, angle measurements without units are assumed to be radians. Noting that $45^\circ = \pi/4$ (radians), we can get the correct result for the tangent of this angle:

```
>>> from math import tan, pi
>>> tan(pi/4)
0.9999999999999999
```

We can now make use of Python's trigonometric functions to write a `to_cartesian` function, taking a pair of polar coordinates and returning corresponding Cartesian coordinates:

```
from math import sin, cos
def to_cartesian(polar_vector):
    length, angle = polar_vector[0], polar_vector[1]
    return (length*cos(angle), length*sin(angle))
```

Using this, we can verify that 5 units at an angle of 37° gets us close to the point (4, 3):

```
>>> from math import pi
>>> angle = 37*pi/180
>>> to_cartesian((5,angle))
(3.993177550236464, 3.0090751157602416)
```

Now that we can convert from polar coordinates to Cartesian coordinates, let's see how to convert in the other direction.

2.3.3 From components back to angles

Given a pair of Cartesian coordinates like $(-2, 3)$, we know how to find the length with the Pythagorean theorem. In this case, it is $\sqrt{13}$, which is the first of the two polar coordinates we are looking for. The second is the angle, which we can call θ (theta), indicating the direction of this vector (figure 2.33).

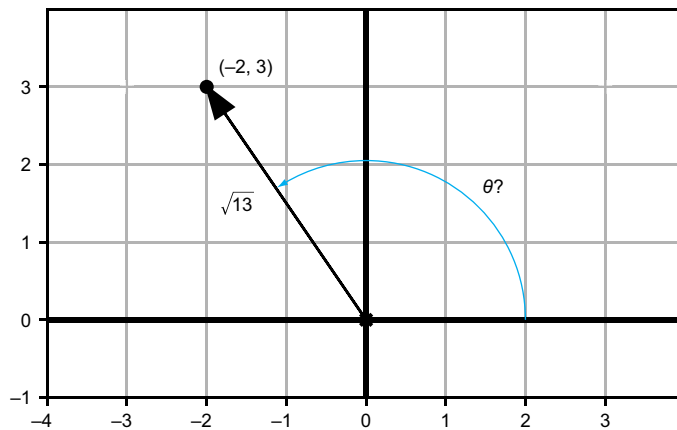


Figure 2.33 In what angle does the vector $(-2, 3)$ point?

We can say some facts about the angle θ that we're looking for. Its tangent, $\tan(\theta)$, is $3/2$, while $\sin(\theta) = 3/\sqrt{13}$ and $\cos(\theta) = -2/\sqrt{13}$. All that's left is finding a value of θ that satisfies these. If you like, you can pause and try to approximate this angle yourself by guessing and checking.

Ideally, we'd like a more efficient method than this. It would be great if there were a function that took the value of $\sin(\theta)$, for instance, and gave you back θ . This turns out to be easier said than done, but Python's `math.asin` function makes a good attempt. This is an implementation of the *inverse trigonometric function* called the *arcsine*, and it returns a satisfactory value of θ :

```
>>> from math import asin
>>> sin(1)
0.8414709848078965
>>> asin(0.8414709848078965)
1.0
```

So far, so good. But what about the sine of our angle $3/\sqrt{13}$?

```
>>> from math import sqrt
>>> asin(3/sqrt(13))
0.9827937232473292
```

This angle is roughly 56.3° , and as figure 2.34 shows, that's the wrong direction!

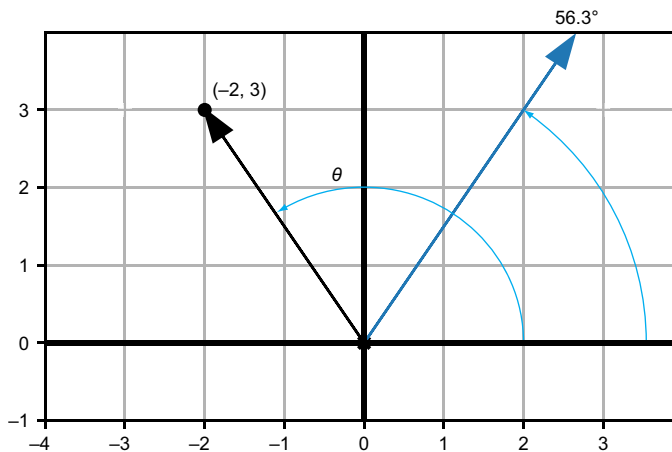


Figure 2.34 Python's `math.asin` function appears to give us the wrong angle.

It's not wrong that `math.asin` gives us this answer; another point (2, 3) *does* lie in this direction. It is at length $\sqrt{13}$ from the origin, so the sine of this angle is also $3/\sqrt{13}$. This is why `math.asin` is not a full solution for us. There are multiple angles that can have the same sine.

The inverse trigonometric function, called *arccosine* and implemented in Python as `math.acos`, happens to give us the right value:

```
>>> from math import acos
>>> acos(-2/sqrt(13))
2.1587989303424644
```

This many radians is about the same as 123.7° , which we can confirm to be correct using a protractor. But this is only by happenstance; there are other angles that could have given us the same cosine. For instance, $(-2, -3)$ also has distance $\sqrt{13}$ from the origin, so it lies at an angle with the same cosine as θ : $-2/\sqrt{13}$. To find the value of θ that we actually want, we'll have to make sure the sine *and* cosine agree with our expectation. The angle returned by Python, which is approximately 2.159, satisfies this:

```
>>> cos(2.1587989303424644)
-0.5547001962252293
>>> -2/sqrt(13)
-0.5547001962252291
>>> sin(2.1587989303424644)
0.8320502943378435
>>> 3/sqrt(13)
0.8320502943378437
```

None of the arcsine, arccosine, or arctangent functions are sufficient to find the angle to a point in the plane. It *is* possible to find the correct angle by a tricky geometric argument you probably learned in high school trigonometry class. I'll leave that as an exercise and cut to the chase—Python can do the work for you! The `math.atan2` function takes the Cartesian coordinates of a point in the plane (in reverse order!) and gives you back the angle at which it lies. For example,

```
>>> from math import atan2
>>> atan2(3,-2)
2.158798930342464
```

I apologize for burying the lede, but I did so because it's worth knowing the potential pitfalls of using inverse trigonometric functions. In summary, trigonometric functions are tricky to do in reverse; multiple different inputs can produce the same output, so an output can't be traced back to a unique input. This lets us complete the function we set out to write: a converter from Cartesian to polar coordinates:

```
def to_polar(vector):
    x, y = vector[0], vector[1]
    angle = atan2(y,x)
    return (length(vector), angle)
```

We can verify some simple examples: `to_polar((1,0))` should be one unit in the positive x direction or an angle of zero degrees. Indeed, the function gives us an angle of zero and a length of one:

```
>>> to_polar((1,0))
(1.0, 0.0)
```

(The fact that the input and the output are the same is coincidental; they have different geometric meanings.) Likewise, we get the expected answer for $(-2, 3)$:

```
>>> to_polar((-2,3))
(3.605551275463989, 2.158798930342464)
```

2.3.4 Exercises

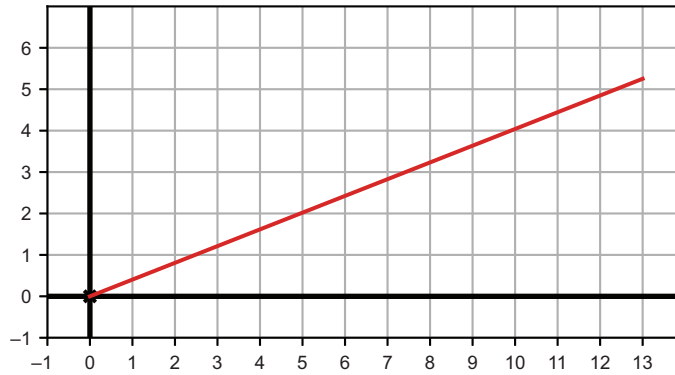
Exercise 2.27 Confirm that the vector given by Cartesian coordinates $(-1.34, 2.68)$ has a length of approximately 3 as expected.

Solution

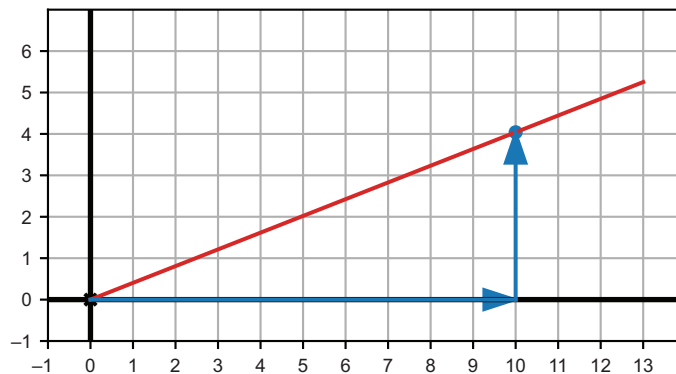
```
>>> length((-1.34, 2.68))  
2.9963310898497184
```

Close enough!

Exercise 2.28 The figure shows a line that makes a 22° angle in the counter-clockwise direction from the positive x -axis. Based on the following picture, what is the approximate value of $\tan(22^\circ)$?



Solution The line passes close to the point $(10, 4)$, so $4 / 10 = 0.4$ is a reasonable approximation of $\tan(22^\circ)$ as shown here:



Exercise 2.29 Turning the question around, suppose we know the length and direction of a vector and want to find its components. What are the x and y components of a vector with length 15 pointing at a 37° angle?

Solution

The sine of 37° is roughly $\frac{3}{5}$, which tells us that every 5 units of distance covered at this angle takes us 3 units upward. So, 15 units of distance give us a vertical component of $\frac{3}{5} \cdot 15$, or 9.

The cosine of 37° is roughly $\frac{4}{5}$, which tells us that each 5 units of distance in this direction take us 4 units to the right, so the horizontal component is $\frac{4}{5} \cdot 15$ or 12. In summary, the polar coordinates $(15, 37^\circ)$ correspond approximately to the Cartesian coordinates $(12, 9)$.

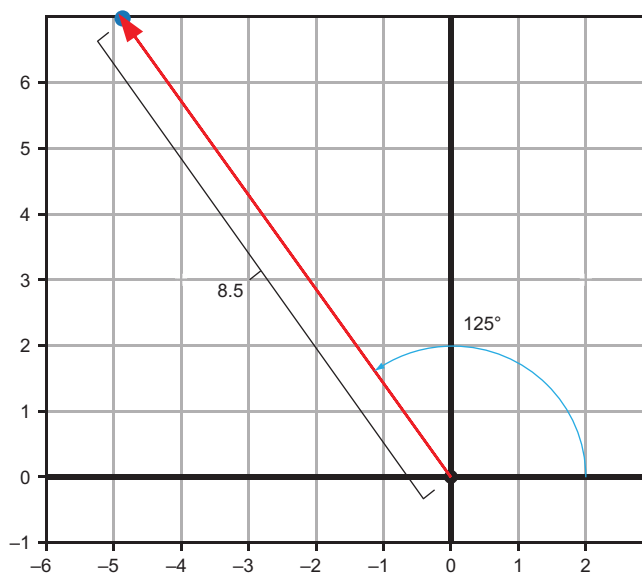
Exercise 2.30 Suppose I travel 8.5 units from the origin at an angle of 125° , measured counterclockwise from the positive x -axis. Given that $\sin(125^\circ) = 0.819$ and $\cos(125^\circ) = -0.574$, what are my final coordinates? Draw a picture to show the angle and path traveled.

Solution

$$x = r \cdot \cos(\theta) = 8.5 \cdot -0.574 = -4.879$$

$$y = r \cdot \sin(\theta) = 8.5 \cdot 0.819 = 6.962$$

The following figure shows the final position, $(-4.879, 6.962)$:



Exercise 2.31 What are the sine and cosine of 0° ? Of 90° ? Of 180° ? In other words, how many vertical and horizontal units are covered per unit distance in any of these directions?

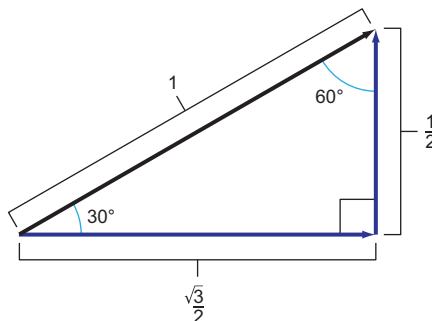
Solution At 0° , no vertical distance is covered, so $\sin(0^\circ) = 0$; rather, every unit of distance traveled is a unit of horizontal distance, so $\cos(0^\circ) = 1$.

For 90° (a quarter turn counterclockwise), every unit traveled is a positive vertical unit, so $\sin(90^\circ) = 1$, while $\cos(90^\circ) = 0$.

Finally, at 180° , every unit of distance traveled is a negative unit in the x direction, so $\cos(180^\circ) = -1$ and $\sin(180^\circ) = 0$.

Exercise 2.32 The following diagram gives some exact measurements for a right triangle:

First, confirm that these lengths are valid for a right triangle because they satisfy the Pythagorean theorem. Then, calculate the values of $\sin(30^\circ)$, $\cos(30^\circ)$, and $\tan(30^\circ)$ to three decimal places using the measurements in the diagram.



Solution These side lengths indeed satisfy the Pythagorean theorem:

$$\sqrt{\left(\frac{1}{2}\right)^2 + \left(\frac{\sqrt{3}}{2}\right)^2} = \sqrt{\frac{1}{4} + \frac{3}{4}} = \sqrt{\frac{4}{4}} = 1$$

Plugging the side lengths into the Pythagorean theorem

The trigonometric function values are given by the appropriate ratios of side lengths:

$$\sin(30^\circ) = \frac{\left(\frac{1}{2}\right)}{1} = 0.500$$

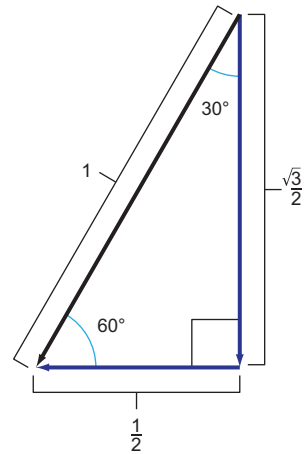
$$\cos(30^\circ) = \frac{\left(\frac{\sqrt{3}}{2}\right)}{1} \approx 0.866$$

$$\tan(30^\circ) = \frac{\left(\frac{1}{2}\right)}{\left(\frac{\sqrt{3}}{2}\right)} \approx 0.577$$

Calculating the sine, cosine, and tangent by their definitions

Exercise 2.33 Looking at the triangle from the previous exercise from a different perspective, use it to calculate the values of $\sin(60^\circ)$, $\cos(60^\circ)$, and $\tan(60^\circ)$ to three decimal places.

Solution Rotating and reflecting the triangle from the previous exercise has no effect on its side lengths or angles.



A rotated copy of the triangle from the previous exercise

The ratios of the side lengths give the trigonometric function values for 60° :

$$\sin(60^\circ) = \frac{\left(\frac{\sqrt{3}}{2}\right)}{1} \approx 0.866$$

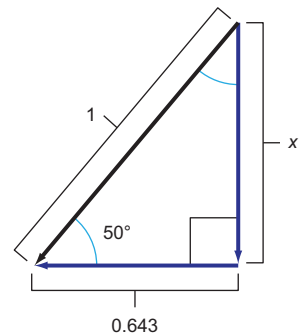
$$\cos(60^\circ) = \frac{\left(\frac{1}{2}\right)}{1} = 0.500$$

$$\tan(60^\circ) = \frac{\left(\frac{\sqrt{3}}{2}\right)}{\left(\frac{1}{2}\right)} \approx 1.732$$

Calculating the defining ratios when horizontal and vertical components have switched

Exercise 2.34 The cosine of 50° is 0.643. What is $\sin(50^\circ)$ and what is $\tan(50^\circ)$? Draw a picture to help you calculate the answer.

Solution Given that the cosine of 50° is 0.643, the following triangle is valid:



(continued)

That is, it has the right ratio of the two known side lengths: $0.643 / 1 = 0.643$. To find the unknown side length, we can use the Pythagorean theorem:

$$\sqrt{0.643^2 + x^2} = 1$$

$$0.643^2 + x^2 = 1$$

$$0.413 + x^2 = 1$$

$$x^2 = 0.587$$

$$x = 0.766$$

With the known side lengths, $\sin(50^\circ) = 0.766/1 = 0.766$. Also, $\tan(50^\circ) = 0.766/0.643 = 1.192$.

Exercise 2.35 What is 116.57° in radians? Use Python to compute the tangent of this angle and confirm that it is close to -2 as we saw previously.

Solution $116.57^\circ \cdot (1 \text{ radian}/57.296^\circ) = 2.035$ radians:

```
>>> from math import tan
>>> tan(2.035)
-1.9972227673316139
```

Exercise 2.36 Locate the angle $10\pi/6$. Do you expect the values of $\cos(10\pi/6)$ and $\sin(10\pi/6)$ to be positive or negative? Use Python to calculate their values and confirm.

Solution A whole circle is 2π radians, so the angle $\pi/6$ is one twelfth of a circle. You can picture cutting a pizza in 12 slices, and counting counterclockwise from the positive x -axis; the angle $10\pi/6$ is two slices short of a full rotation. This means that it points down and to the right. The cosine should be positive, and the sine should be negative because the distance in this direction corresponds with a positive horizontal displacement and a negative vertical displacement:

```
>>> from math import pi, cos, sin
>>> sin(10*pi/6)
-0.8660254037844386
>>> cos(10*pi/6)
0.5000000000000001
```

Exercise 2.37 The following list comprehension creates 1,000 points in polar coordinates:

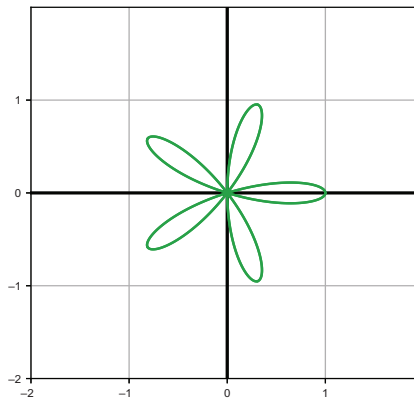
```
[(cos(5*x*pi/500.0), 2*pi*x/1000.0) for x in range(0,1000)]
```

In Python code, convert these to Cartesian coordinates and connect them in a closed loop with line segments to draw a picture.

Solution Including the setup and the original list of data, the code is as follows:

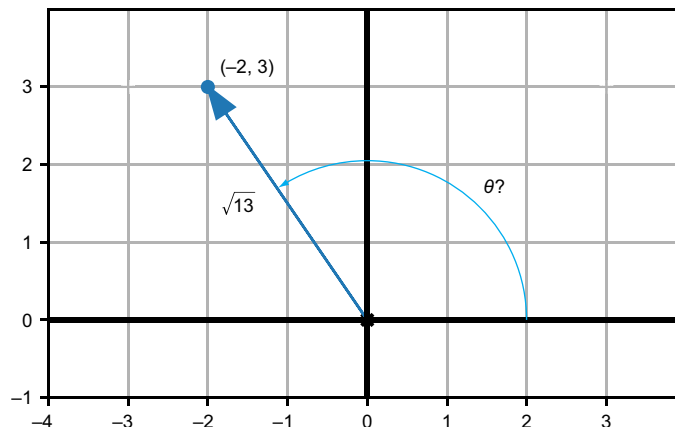
```
polar_coords = [(cos(x*pi/100.0), 2*pi*x/1000.0) for x in range(0,1000)]
vectors = [to_cartesian(p) for p in polar_coords]
draw(Polygon(*vectors, color=green))
```

And the result is a five-leaved flower:



The plot of the 1,000 connected points is a flower shape.

Exercise 2.38 Find the angle to get to the point $(-2, 3)$ by “guess-and-check.”



What is the angle to get to the point $(-2, 3)$?

(continued)

Hint We can tell visually that the answer is between $\pi/2$ and π . On that interval, the values of sine and cosine always decrease as the angle increases.

Solution Here's an example of guessing and checking between $\pi/2$ and π , looking for an angle with tangent close to $-3/2 = -1.5$:

```
>>> from math import tan, pi
>>> pi, pi/2
(3.141592653589793, 1.5707963267948966)
>>> tan(1.8)
-4.286261674628062
>>> tan(2.5)
-0.7470222972386603
>>> tan(2.2)
-1.3738230567687946
>>> tan(2.1)
-1.7098465429045073
>>> tan(2.15)
-1.5289797578045665
>>> tan(2.16)
-1.496103541616277
>>> tan(2.155)
-1.5124173422757465
>>> tan(2.156)
-1.5091348993879299
>>> tan(2.157)
-1.5058623488727219
>>> tan(2.158)
-1.5025996395625054
>>> tan(2.159)
-1.4993467206361923
```

The value must be between 2.158 and 2.159.

Exercise 2.39 Find another point in the plane with the same tangent as θ , namely $-3/2$. Use Python's implementation of the *arctangent* function, `math.atan`, to find the value of this angle.

Solution Another point with tangent $-3/2$ is $(3, -2)$. Python's `math.atan` finds the angle to this point:

```
>>> from math import atan
>>> atan(-3/2)
-0.982793723247329
```

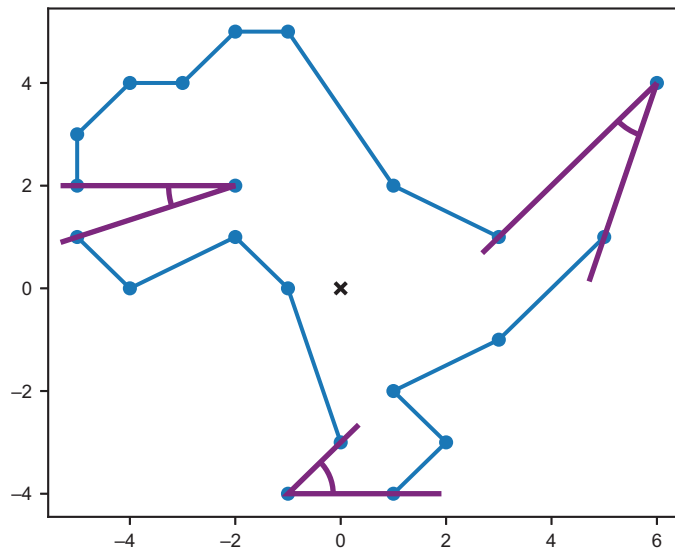
This is slightly less than a quarter turn in the clockwise direction.

Exercise 2.40 Without using Python, what are the polar coordinates corresponding to the Cartesian coordinates $(1, 1)$ and $(1, -1)$? Once you’ve found the answers, use `to_polar` to check your work.

Solution In polar coordinates, $(1, 1)$ becomes $(\sqrt{2}, \pi/4)$ and $(1, -1)$ becomes $(\sqrt{2}, -\pi/4)$.

With some care, you can find any angle on a shape made up of known vectors. The angle between two vectors is either a sum or difference of angles these make with the x -axis. You measure some trickier angles in the next mini-project.

Exercise 2.41—Mini Project What is the angle of the Dinosaur’s mouth? What is the angle of the dinosaur’s toe? Of the point of its tail?



Some angles we can measure or calculate on our dinosaur.

2.4 Transforming collections of vectors

Collections of vectors store spatial data like drawings of dinosaurs regardless of what coordinate system we use: polar or Cartesian. It turns out that when we want to manipulate vectors, one coordinate system can be better than another. We already saw that moving (or translating) a collection of vectors is easy with Cartesian coordinates. It

turns out to be much less natural in polar coordinates. Because polar coordinates have angles built in, these make it simple to carry out rotations.

In polar coordinates, adding to the angle rotates a vector further counterclockwise, while subtracting from it rotates the vector clockwise. The polar coordinate $(1, 2)$ is at distance 1 and at an angle of 2 radians. (Remember that we are working in radians if there is no degree symbol!) Starting with the angle 2 and adding or subtracting 1 takes the vector either 1 radian counterclockwise or clockwise, respectively (figure 2.35).

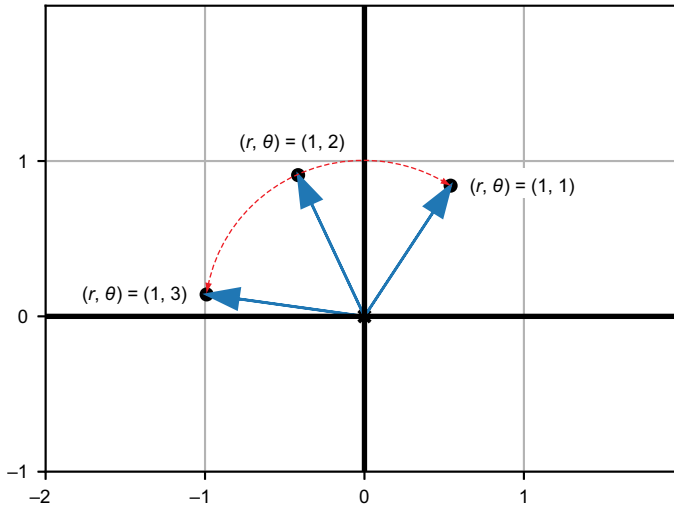


Figure 2.35 Adding or subtracting from the angle rotates the vector about the origin.

Rotating a number of vectors simultaneously has the effect of rotating the figure these represent about the origin. The `draw` function only understands Cartesian coordinates, so we need to convert from polar to Cartesian before using it. Likewise, we have only seen how to rotate vectors in polar coordinates, so we need to convert Cartesian coordinates to polar coordinates before executing a rotation. Using this approach, we can rotate the dinosaur like this:

```
rotation_angle = pi/4
dino_polar = [to_polar(v) for v in dino_vectors]
dino_rotated_polar = [(l, angle + rotation_angle) for l, angle in dino_polar]
dino_rotated = [to_cartesian(p) for p in dino_rotated_polar]
draw(
    Polygon(*dino_vectors, color=gray),
    Polygon(*dino_rotated, color=red)
)
```


The result of this code is a gray copy of the original dinosaur, plus a superimposed red copy that's rotated by $\pi/4$, or an eighth of a full revolution counterclockwise (figure 2.36).

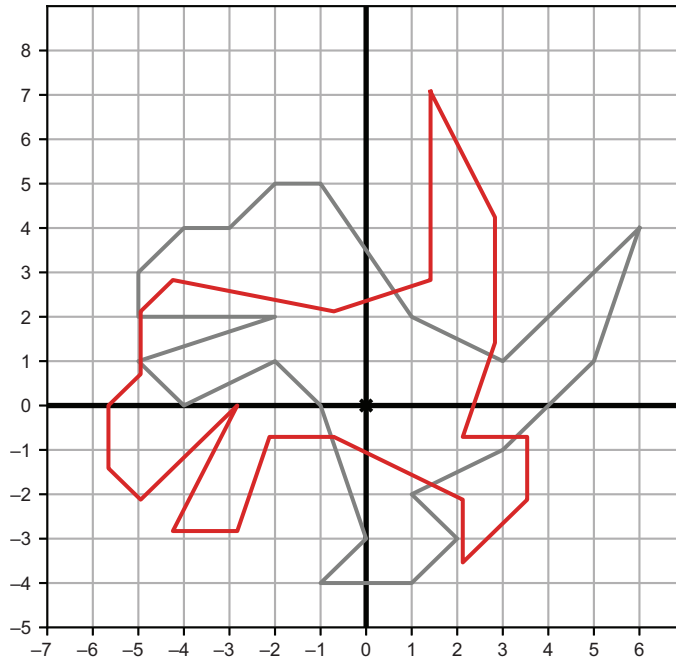


Figure 2.36 The original dinosaur in gray and a rotated copy in red

As an exercise at the end of this section, you can write a general-purpose `rotate` function that rotates a list of vectors by the same specified angle. I'm going to use such a function in the next few examples, and you can either use the implementation I provide in the source code or one you come up with yourself.

2.4.1 Combining vector transformations

So far, we've seen how to translate, rescale, and rotate vectors. Applying any of these transformations to a collection of vectors achieves the same effect on the shape that these define in the plane. The full power of these vector transformations comes when we apply them in sequence.

For instance, we could first rotate and *then* translate the dinosaur. Using the `translate` function from the exercise in section 2.2.4 and the `rotate` function, we can write such a transformation concisely (see the result in figure 2.37):

```
new_dino = translate((8,8), rotate(5 * pi/3, dino_vectors))
```

The rotation comes first, turning the dinosaur counterclockwise by $5\pi/3$, which is most of a full counterclockwise revolution. Then the dinosaur is translated up and to

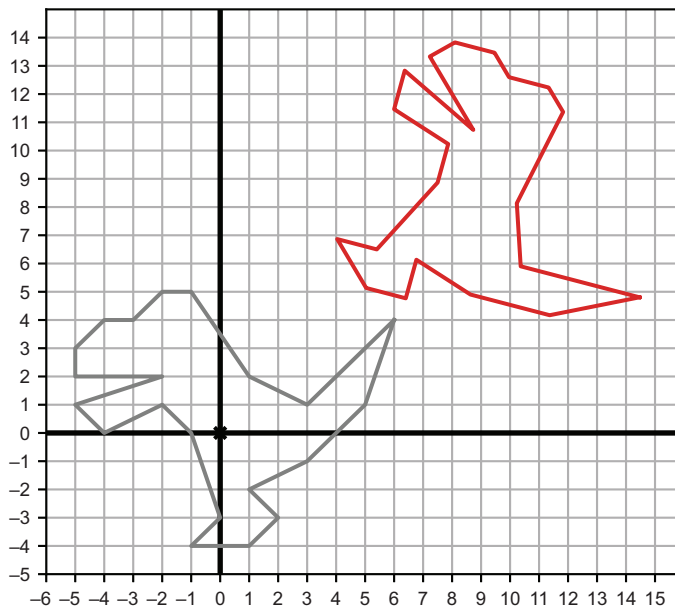


Figure 2.37 The original dinosaur in gray and a red copy that's rotated and then translated

the right by 8 units each. As you can imagine, combining rotations and translations appropriately can move the dinosaur (or any shape) to any desired location and orientation in the plane. Whether we're animating our dinosaur in a movie or in a game, the flexibility to move it around with vector transformations lets us give it life programmatically.

Our applications will soon take us past cartoon dinosaurs; there are plenty of other operations on vectors and many generalize to higher dimensions. Real-world data sets often live in dozens or hundreds of dimensions, so we'll apply the same kinds of transformations to these as well. It's often useful to both translate and rotate data sets to make their important features clearer. We won't be able to picture rotations in 100 dimensions, but we can always think of two dimensions as a trusty metaphor.

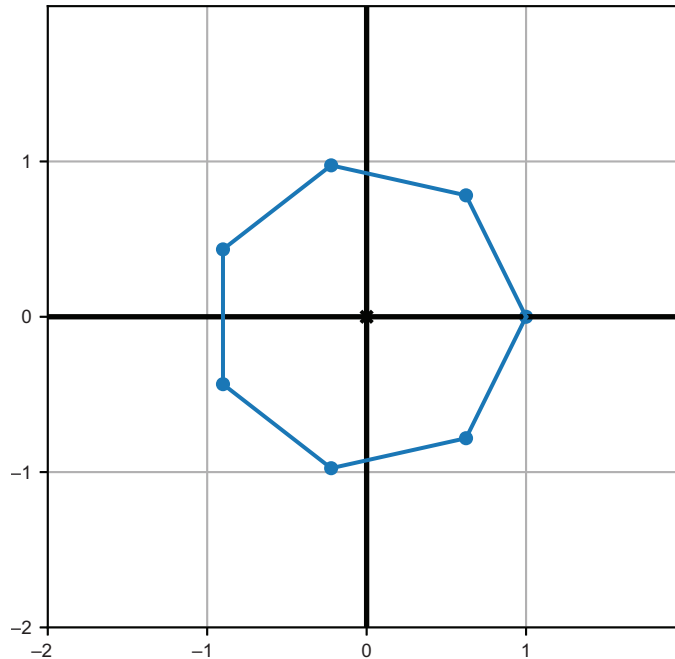
2.4.2 Exercises

Exercise 2.42 Create a `rotate(angle, vectors)` function that takes an array of input vectors in Cartesian coordinates and rotates those by the specified angle (counterclockwise or clockwise, according to whether the angle is positive or negative).

Solution

```
def rotate(angle, vectors):
    polars = [to_polar(v) for v in vectors]
    return [to_cartesian((l, a+angle)) for l,a in polars]
```

Exercise 2.43 Create a function `regular_polygon(n)` that returns Cartesian coordinates for the vertices of a regular n -sided polygon (that is, having all angles and side lengths equal). For instance, `regular_polygon(7)` produces vectors defining the following heptagon:



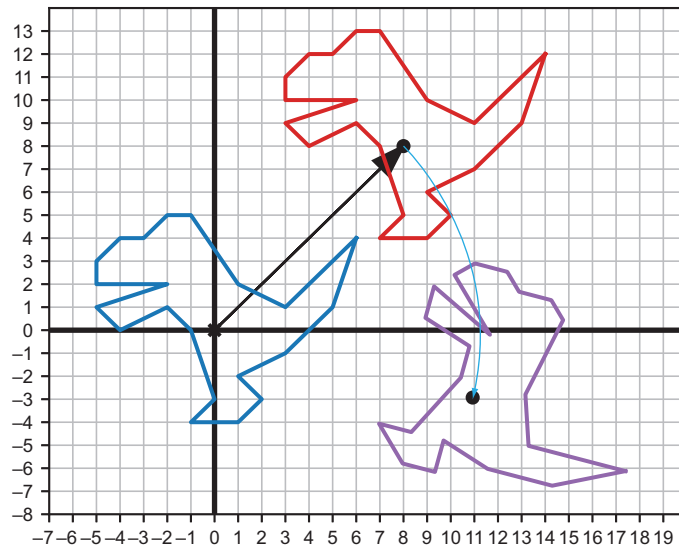
A regular heptagon with points at seven evenly-spaced angles around the origin

Hint In this picture, I used the vector $(1, 0)$ and copies that are rotated by seven evenly-spaced angles about the origin.

Solution

```
def regular_polygon(n):
    return [to_cartesian((1, 2*pi*k/n)) for k in range(0,n)]
```

Exercise 2.44 What is the result of first translating the dinosaur by the vector $(8, 8)$ and then rotating it by $5\pi/3$? Is the result the same as rotating and then translating?

*(continued)***Solution**

First translating and then rotating the dinosaur

The result is *not* the same. In general, applying rotations and translations in different orders yields different results.

2.5 Drawing with Matplotlib

As promised, I'll conclude by showing you how to build "from scratch" the drawing functions used in this chapter from the Matplotlib library. After installing Matplotlib with pip, you can import it (and some of its submodules); for example,

```
import matplotlib
from matplotlib.patches import Polygon
from matplotlib.collections import PatchCollection
```

The Polygon, Points, Arrow, and Segment classes are not that interesting; they simply hold the data passed to them in their constructors. For instance, the Points class contains only a constructor that receives and stores a list of vectors and a color keyword argument:

```
class Points():
    def __init__(self, *vectors, color=black):
        self.vectors = list(vectors)
        self.color = color
```

The draw function starts by figuring out how big the plot should be and then draws each of the objects it is passed one-by-one. For instance, to draw dots on the plane represented by a `Points` object, draw uses Matplotlib's scatter-plotting functionality:

```
def draw(*objects, ...
# ...
for object in objects:
# ...
    elif type(object) == Points:
        xs = [v[0] for v in object.vectors]
        ys = [v[1] for v in object.vectors]
        plt.scatter(xs, ys, color=object.color)
# ...
```

Some setup happens here, which is not shown.

Iterates over the objects passed in

If the current object is an instance of the `Points` class, draws dots for all of its vectors using Matplotlib's scatter function

Arrows, segments, and polygons are handled in much the same way using different pre-built Matplotlib functions to make the geometric objects appear on the plot. You can find all of these implemented in the source code file `vector_drawing.py`. We'll use Matplotlib throughout this book to plot data and mathematical functions, and I'll provide periodic refreshers on its functionality as we use it.

Now that you've mastered two dimensions, you're ready to add another one. With the third dimension, we can fully describe the world we live in. In the next chapter, you'll see how to model three-dimensional objects in code.

Summary

- Vectors are mathematical objects that live in multi-dimensional spaces. These can be geometric spaces like the two-dimensional (2D) plane of a computer screen or the three-dimensional (3D) world we inhabit.
- You can think of vectors equivalently as arrows having a specified length and direction, or as points in the plane relative to a reference point called the *origin*. Given a point, there is a corresponding arrow that shows how to get to that point from the origin.
- You can connect collections of points in the plane to form interesting shapes like a dinosaur.
- In 2D, coordinates are pairs of numbers that help us measure the location of points in the plane. Written as a tuple (x, y) , the x and y values tell us how far horizontally and vertically to travel to get to the point.
- We can store points as coordinate tuples in Python and choose from a number of libraries to draw the points on the screen.
- Vector addition has the effect of translating (or moving) a first vector in the direction of a second added vector. Thinking of a collection of vectors as paths to travel, their vector sum gives the overall direction and distance traveled.
- Scalar multiplication of a vector by a numeric factor yields a vector that is longer by that factor and points in the same direction as the original.

- Subtracting one vector from a second gives the relative position of the second vector from the first.
- Vectors can be specified by their length and direction (as an angle). These two numbers define the polar coordinates of a given 2D vector.
- The trigonometric functions sine, cosine, and tangent are used to convert between ordinary (Cartesian) coordinates and polar coordinates.
- It's easy to rotate shapes defined by collections of vectors in polar coordinates. You only need to add or subtract the given rotation angle from the angle of each vector. Rotating and translating shapes in the plane lets us place them anywhere and in any orientation.