Think Complexity

Version 2.1.0

Think Complexity

Version 2.1.0

Allen B. Downey

Green Tea Press

Needham, Massachusetts

Green Tea Press
9 Washburn Ave
Needham MA 02492

The LaTeX source for this book is available from

```
http://greenteapress.com/complexity
```

Contents

Preface **xi**

0.1 Who is this book for? xiii

0.2 Using the code xiii

1 Complexity Science **1**

1.1 Paradigm shift? 3

1.2 The axes of scientific models 4

1.3 A new kind of model 6

1.4 A new kind of engineering 7

1.5 A new kind of thinking 8

2 Graphs **11**

2.1 What is a graph? 11

2.2 NetworkX 13

2.3 Random graphs 16

2.4 Generating graphs 17

2.5 Connected graphs 18

2.6 Generating ER graphs 20

2.7 The probability of connectivity 22

2.8 Analysis of graph algorithms 24

2.9 Exercises 25

3 Small world graphs **27**

3.1 Stanley Milgram 27

3.2 Watts and Strogatz 28

3.3 Ring lattice 30

3.4 WS graphs 32

3.5 Clustering 33

3.6 Shortest path lengths 35

3.7 The WS experiment 36

3.8 What kind of explanation is *that*? 38

3.9 Breadth-First Search 39

3.10 Dijkstra's algorithm 41

3.11 Exercises 43

4 Scale-free networks **47**

4.1 Social network data 47

4.2 WS Model 50

4.3 Degree 51

4.4 Heavy-tailed distributions 53

4.5 Barabási-Albert model 55

4.6 Generating BA graphs 57

4.7 Cumulative distributions 59

4.8 Explanatory models 62

4.9 Exercises 64

5 Cellular Automatons **67**

5.1 A simple CA 67
5.2 Wolfram's experiment 68
5.3 Classifying CAs 70
5.4 Randomness 71
5.5 Determinism 72
5.6 Spaceships 74
5.7 Universality 76
5.8 Falsifiability 78
5.9 What is this a model of? 79
5.10 Implementing CAs 81
5.11 Cross-correlation 83
5.12 CA tables 84
5.13 Exercises 85

6 Game of Life **89**

6.1 Conway's GoL 89
6.2 Life patterns 91
6.3 Conway's conjecture 92
6.4 Realism 94
6.5 Instrumentalism 96
6.6 Implementing Life 97
6.7 Exercises 99

7 Physical modeling **103**

7.1 Diffusion 103
7.2 Reaction-diffusion 105
7.3 Percolation 109

7.4 Phase change 110
7.5 Fractals 113
7.6 Fractals and Percolation Models 115
7.7 Exercises 117

8 Self-organized criticality 119

8.1 Critical Systems 119
8.2 Sand Piles 120
8.3 Implementing the Sand Pile 121
8.4 Heavy-tailed distributions 125
8.5 Fractals 127
8.6 Spectral Density 131
8.7 Reductionism and Holism 134
8.8 SOC, causation and prediction 136
8.9 Exercises 138

9 Agent-based models 141

9.1 Schelling's Model 142
9.2 Implementation of Schelling's model 143
9.3 Segregation 145
9.4 Sugarscape 147
9.5 Wealth inequality 150
9.6 Implementing Sugarscape 151
9.7 Migration and Wave Behavior 154
9.8 Emergence 155
9.9 Exercises 157

10 Herds, Flocks, and Traffic Jams 159

10.1 Traffic jams 159

10.2 Random Noise 163

10.3 Boids 164

10.4 Free will 167

10.5 Exercises 169

11 Game Theory 171

11.1 The El Farol Problem 171

11.2 Prisoner's Dilemma 171

11.3 Game Theory 172

11.4 Exercises 173

A Analysis of algorithms 175

A.1 Order of growth 176

A.2 Analysis of basic Python operations 179

A.3 Analysis of search algorithms 182

A.4 Hashtables 183

A.5 Summing lists 188

A.6 `pyplot` 190

B Reading list 193

Preface

This book is primarily about complexity science, but it is also about data structures and algorithms, intermediate programming in Python, computational modeling, and the philosophy of science:

Complexity science Complexity is an interdisciplinary field—at the intersection of mathematics, computer science and natural science—that focuses on discrete models of physical and social systems. In particular, it focuses on **complex systems**, which are systems with many interacting components.

Data structures: A data structure is a collection of data elements organized in a way that supports particular operations. For example, a Python dictionary organizes key-value pairs in a way that provides fast mapping from keys to values, but mapping from values to keys is slower.

Algorithms: An algorithm is a process for performing a computation. Designing efficient programs often involves the co-evolution of data structures and the algorithms that use them. For example, in the first few chapters I present graphs, data structures that implement graphs, and graph algorithms based on those data structures.

Computational modeling: A model is a simplified description of a system used for simulation or analysis. Computational models are designed to take advantage of cheap, fast computation.

Philosophy of science: The experiments and results in this book raise questions relevant to the philosophy of science, including the nature of scientific laws, theory choice, realism and instrumentalism, holism and reductionism, and epistemology.

Complex systems include networks and graphs, cellular automatons, agent-based models and swarms, fractals and self-organizing systems, chaotic systems and cybernetic systems. These terms might not mean much to you at this point. We will get to them soon, but you can get a preview at http://en.wikipedia.org/wiki/Complex_systems.

I hope this book helps readers explore a set of topics and ideas they might not encounter otherwise, practice programming skills in Python, and learn more about data structures and algorithms (or review material that might have been less engaging the first time around).

Features of this book include:

Technical details There are many books about complex systems, but most are written for a popular audience. They usually skip the technical details, which is frustrating for people who can handle it. This book presents the mathematics, the code, and the other material you need to really understand this work.

Further reading Throughout the book, I include pointers to further reading, including original papers (most of which are available electronically) and related articles from Wikipedia and other sources. Some professors have an allergic reaction to Wikipedia, on the grounds that students depend too heavily on a suppposedly unreliable source. Since many of my references are Wikipedia articles, I want to explain my thinking. First, the articles on complexity science and related topics tend to be good; second, they are written at a level that is accessible after you have read this book (but sometimes not before); and finally, they are freely available to readers all over the world. If there is a danger in sending readers to these references, it is not that the material is unreliable, but that the readers won't come back!

Jupyter notebooks For each chapter I provide a Jupyter notebook that includes the code from the chapter, additional examples (and especially animations), and suggestions for experiments you can run with small changes in the code.

Exercises and solutions At the end of each chapter I suggest exercises you might want to work on, and I include solutions.

0.1 Who is this book for?

The examples and supporting code for this book are in Python. You should know core Python and you should be familiar with object-oriented features, at least using objects if not defining your own.

If you are not already familiar with Python, you might want to start with my other book, *Think Python*, which is an introduction to Python for people who have never programmed, or Mark Lutz's *Learning Python*, which might be better for people with programming experience.

I use NumPy and SciPy extensively. If you are familiar with them already, that's great, but I will also explain the functions and data structures I use.

I assume that the reader knows basic mathematics, including complex numbers. You don't need much calculus; if you understand the concepts of integration and differentiation, that will do. I use some linear algebra, but I will explain it as we go along.

0.2 Using the code

All code used in this book is available from https://github.com/AllenDowney/ThinkComplexity2. If you are not familiar with Git, it is a version control system that allows you to keep track of the files that make up a project. A collection of files under Git's control is called a "repository". GitHub is a hosting service that provides storage for Git repositories and a convenient web interface.

The GitHub homepage for my repository provides several ways to work with the code:

- You can create a copy of my repository on GitHub by pressing the Fork button. If you don't already have a GitHub account, you'll need to create one. After forking, you'll have your own repository on GitHub that you can use to keep track of code you write while working on this book. Then you can clone the repo, which means that you copy the files to your computer.

- Or you can clone my repository. You don't need a GitHub account to do this, but you won't be able to write your changes back to GitHub.
- If you don't want to use Git at all, you can download the files in a Zip file using the green button that says "Clone or download".

All of the code is written to work in both Python 2 and Python 3 with no translation.

I developed this book using Anaconda from Continuum Analytics, which is a free Python distribution that includes all the packages you'll need to run the code (and lots more). I found Anaconda easy to install. By default it does a user-level installation, not system-level, so you don't need administrative privileges. And it supports both Python 2 and Python 3. You can download Anaconda from http://continuum.io/downloads.

If you don't want to use Anaconda, you will need the following packages:

- NumPy for basic numerical computation, http://www.numpy.org/;
- SciPy for scientific computation, http://www.scipy.org/;
- matplotlib for visualization, http://matplotlib.org/.

Although these are commonly used packages, they are not included with all Python installations, and they can be hard to install in some environments. If you have trouble installing them, I recommend using Anaconda or one of the other Python distributions that include these packages.

The repository includes Python scripts and several Jupyter notebooks. If you have not used Jupyter before, you can read about it at http://jupyter.org.

There are three ways you can work with the Jupyter notebooks:

Run Jupyter on your computer If you installed Anaconda, you probably got Jupyter by default. To check, start the server from the command line, like this:

```
$ jupyter notebook
```

If it's not installed, you can install it using `conda`, which is the package manager used by Anaconda:

```
$ conda install jupyter
```

When you start the server, it should launch your default web browser or create a new tab in an open browser window.

Run Jupyter on Binder Binder is a service that runs Jupyter in a virtual machine. If you follow this link, http://mybinder.org/repo/AllenDowney/ThinkComplexity2, you should get a Jupyter home page with the notebooks for this book and the supporting data and scripts.

You can run the scripts and modify them to run your own code, but the virtual machine you run them in is temporary. Any changes you make will disappear, along with the virtual machine, if you leave it idle for more than about an hour.

View notebooks on GitHub GitHub provides a view of the notebooks you can can use to read the notebooks and see the results I generated, but you won't be able to modify or run the code. And in the later chapters you won't see the animations move.

Good luck, and have fun!

Allen B. Downey
Professor of Computer Science
Olin College of Engineering
Needham, MA

Contributor List

If you have a suggestion or correction, please send email to `downey@allendowney.com`. If I make a change based on your feedback, I will add you to the contributor list (unless you ask to be omitted).

If you include at least part of the sentence the error appears in, that makes it easy for me to search. Page and section numbers are fine, too, but not quite as easy to work with. Thanks!

- John Harley, Jeff Stanton, Colden Rouleau and Keerthik Omanakuttan are Computational Modeling students who pointed out typos.

- Jose Oscar Mur-Miranda found several typos.
- Phillip Loh, Corey Dolphin, Noam Rubin and Julian Ceipek found typos and made helpful suggestions.
- Sebastian Schöner sent two pages of corrections!
- Philipp Marek sent a number of corrections.

Other people who reported errors include Richard Hollands, Muhammad Najmi bin Ahmad Zabidi, Alex Hantman, and Jonathan Harford.

Chapter 1

Complexity Science

The thesis of this book is that complexity science is a "new kind of science", a phrase I'm borrowing from Stephen Wolfram.

In 2002 Wolfram published *A New Kind of Science*, where he presents his and others' work on cellular automatons and describes a scientific approach to the study of computational systems. We'll get back to Wolfram in Chapter 5, but for now I want to use his title for something a little broader.

I think complexity is new not because it applies the tools of science to a new subject, but because it uses different tools, allows different kinds of work, and ultimately changes what we mean by "science".

To demonstrate the difference, I'll start with an example of classical science: suppose someone asked you why planetary orbits are elliptical. You might invoke Newton's law of universal gravitation and use it to write a differential equation that describes planetary motion. Then you could solve the differential equation and show that the solution is an ellipse. QED!

Most people find this kind of explanation satisfying. It includes a mathematical derivation—so it has some of the rigor of a proof—and it explains a specific observation, elliptical orbits, by appealing to a general principle, gravitation.

Let me contrast that with a different kind of explanation. Suppose you move to a city like Detroit that is racially segregated, and you want to know why it's like that. If you do some research, you might find a paper by Thomas

Schelling called "Dynamic Models of Segregation", which proposes a simple model of racial segregation:

Here is my description of the model, from Chapter 9:

> The Schelling model of the city is an array of cells where each cell represents a house. The houses are occupied by two kinds of "agents", labeled red and blue, in roughly equal numbers. About 10% of the houses are empty.
>
> At any point in time, an agent might be happy or unhappy, depending on the other agents in the neighborhood. In one version of the model, agents are happy if they have at least two neighbors like themselves, and unhappy if they have one or zero.
>
> The simulation proceeds by choosing an agent at random and checking to see whether it is happy. If so, nothing happens; if not, the agent chooses one of the unoccupied cells at random and moves.

If you start with a simulated city that is entirely unsegregated and run the model for a short time, clusters of similar agents appear. As time passes, the clusters grow and coalesce until there are a small number of large clusters and most agents live in homogeneous neighborhoods.

The degree of segregation in the model is surprising, and it suggests an explanation of segregation in real cities. Maybe Detroit is segregated because people prefer not to be greatly outnumbered and will move if the composition of their neighborhoods makes them unhappy.

Is this explanation satisfying in the same way as the explanation of planetary motion? Many people would say not, but why?

Most obviously, the Schelling model is highly abstract, which is to say not realistic. It is tempting to say that people are more complicated than planets, but when you think about it, planets are just as complicated as people (especially the ones that *have* people).

Both systems are complicated, and both models are based on simplifications; for example, in the model of planetary motion we include forces between the planet and its sun, and ignore interactions between planets.

The important difference is that, for planetary motion, we can defend the model by showing that the forces we ignore are smaller than the ones we include. And we can extend the model to include other interactions and show that the effect is small. For Schelling's model it is harder to justify the simplifications.

To make matters worse, Schelling's model doesn't appeal to any physical laws, and it uses only simple computation, not mathematical derivation. Models like Schelling's don't look like classical science, and many people find them less compelling, at least at first. But as I will try to demonstrate, these models do useful work, including prediction, explanation, and design. One of the goals of this book is to explain how.

1.1 Paradigm shift?

When I describe this book to people, I am often asked if this new kind of science is a paradigm shift. I don't think so, and here's why.

Thomas Kuhn introduced the term "paradigm shift" in *The Structure of Scientific Revolutions* in 1962. It refers to a process in the history of science where the basic assumptions of a field change, or where one theory is replaced by another. He presents as examples the Copernican revolution, the displacement of phlogiston by the oxygen model of combustion, and the emergence of relativity.

The development of complexity science is not the replacement of an older model, but (in my opinion) a gradual shift in the criteria models are judged by, and in the kinds of models that are considered acceptable.

For example, classical models tend to be law-based, expressed in the form of equations, and solved by mathematical derivation. Models that fall under the umbrella of complexity are often rule-based, expressed as computations, and simulated rather than analyzed.

Not everyone finds these models satisfactory. For example, in *Sync*, Steven Strogatz writes about his model of spontaneous synchronization in some species of fireflies. He presents a simulation that demonstrates the phenomenon, but then writes:

> I repeated the simulation dozens of times, for other random initial conditions and for other numbers of oscillators. Sync every time. [...] The challenge now was to prove it. Only an ironclad proof would demonstrate, in a way that no computer ever could, that sync was inevitable; and the best kind of proof would clarify *why* it was inevitable.

Strogatz is a mathematician, so his enthusiasm for proofs is understandable, but his proof doesn't address what is, to me, the most interesting part the phenomenon. In order to prove that "sync was inevitable", Strogatz makes several simplifying assumptions, in particular that each firefly can see all the others.

In my opinion, it is more interesting to explain how an entire valley of fireflies can synchronize *despite the fact that they cannot all see each other*. How this kind of global behavior emerges from local interactions is the subject of Chapter 9. Explanations of these phenomena often use agent-based models, which explore (in ways that would be difficult or impossible with mathematical analysis) the conditions that allow or prevent synchronization.

I am a computer scientist, so my enthusiasm for computational models is probably no surprise. I don't mean to say that Strogatz is wrong, but rather that people have different opinions about what questions to ask and what tools to use to answer them. These opinions are based on value judgments, so there is no reason to expect agreement.

Nevertheless, there is rough consensus among scientists about which models are considered good science, and which others are fringe science, pseudoscience, or not science at all.

I claim, and this is a central thesis of this book, that the criteria this consensus is based on change over time, and that the emergence of complexity science reflects a gradual shift in these criteria.

1.2 The axes of scientific models

I have described classical models as based on physical laws, expressed in the form of equations, and solved by mathematical analysis; conversely, models

of complexity systems are often based on simple rules and implemented as computations.

We can think of this trend as a shift over time along two axes:

Equation-based → simulation-based

Analysis → computation

The new kind of science is different in several other ways. I present them here so you know what's coming, but some of them might not make sense until you have seen the examples later in the book.

Continuous → discrete Classical models tend to be based on continuous mathematics, like calculus; models of complex systems are often based on discrete mathematics, including graphs and cellular automatons.

Linear → non-linear Classical models are often linear, or use linear approximations to non-linear systems; complexity science is more friendly to non-linear models. One example is chaos theory[1].

Deterministic → stochastic Classical models are usually deterministic, which may reflect underlying philosophical determinism, discussed in Chapter 5; complex models often feature randomness.

Abstract → detailed In classical models, planets are point masses, planes are frictionless, and cows are spherical (see http://en.wikipedia.org/wiki/Spherical_cow). Simplifications like these are often necessary for analysis, but computational models can be more realistic.

One, two → many In celestial mechanics, the two-body problem can be solved analytically; the three-body problem cannot. Where classical models are often limited to small numbers of interacting elements, complexity science works with larger complexes (which is where the name comes from).

Homogeneous → composite In classical models, the elements tend to be interchangeable; complex models more often include heterogeneity.

[1] Chaos is not covered in this book, but you can read about it at http://en.wikipedia.org/wiki/Chaos.

These are generalizations, so we should not take them too seriously. And I don't mean to deprecate classical science. A more complicated model is not necessarily better; in fact, it is usually worse.

Also, I don't mean to say that these changes are abrupt or complete. Rather, there is a gradual migration in the frontier of what is considered acceptable, respectable work. Some tools that used to be regarded with suspicion are now common, and some models that were widely accepted are now regarded with scrutiny.

For example, when Appel and Haken proved the four-color theorem in 1976, they used a computer to enumerate 1,936 special cases that were, in some sense, lemmas of their proof. At the time, many mathematicians did not consider the theorem truly proved. Now computer-assisted proofs are common and generally (but not universally) accepted.

Conversely, a substantial body of economic analysis is based on a model of human behavior called "Economic man", or, with tongue in cheek, *Homo economicus*. Research based on this model was highly regarded for several decades, especially if it involved mathematical virtuosity. More recently, this model is treated with skepticism, and models that include imperfect information and bounded rationality are hot topics.

1.3 A new kind of model

Complex models are often appropriate for different purposes and interpretations:

Predictive → explanatory Schelling's model of segregation might shed light on a complex social phenomenon, but it is not useful for prediction. On the other hand, a simple model of celestial mechanics can predict solar eclipses, down to the second, years in the future.

Realism → instrumentalism Classical models lend themselves to a realist interpretation; for example, most people accept that electrons are real things that exist. Instrumentalism is the view that models can be useful even if the entities they postulate don't exist. George Box wrote what might be the motto of instrumentalism: "All models are wrong, but some are useful."

Reductionism → holism Reductionism is the view that the behavior of a system can be explained by understanding its components. For example, the periodic table of the elements is a triumph of reductionism, because it explains the chemical behavior of elements with a simple model of electrons in atoms. Holism is the view that some phenomena that appear at the system level do not exist at the level of components, and cannot be explained in component-level terms.

We get back to explanatory models in Chapter 4, instrumentalism in Chapter 6.1, and holism in Chapter 8.

1.4 A new kind of engineering

I have been talking about complex systems in the context of science, but complexity is also a cause, and effect, of changes in engineering and the organization of social systems:

Centralized → decentralized Centralized systems are conceptually simple and easier to analyze, but decentralized systems can be more robust. For example, in the World Wide Web clients send requests to centralized servers; if the servers are down, the service is unavailable. In peer-to-peer networks, every node is both a client and a server. To take down the service, you have to take down *every* node.

Isolation → interaction In classical engineering, the complexity of large systems is managed by isolating components and minimizing interactions. This is still an important engineering principle; nevertheless, the availability of cheap computation makes it increasingly feasible to design systems with complex interactions between components.

One-to-many → many-to-many In many communication systems, broadcast services are being augmented, and sometimes replaced, by services that allow users to communicate with each other and create, share, and modify content.

Top-down → bottom-up In social, political and economic systems, many activities that would normally be centrally organized now operate as grassroots movements. Even armies, which are the canonical example of

hierarchical structure, are moving toward devolved command and control.

Analysis → computation In classical engineering, the space of feasible designs is limited by our capability for analysis. For example, designing the Eiffel Tower was possible because Gustave Eiffel developed novel analytic techniques, in particular for dealing with wind load. Now tools for computer-aided design and analysis make it possible to build almost anything that can be imagined. Frank Gehry's Guggenheim Museum Bilbao is my favorite example.

Design → search Engineering is sometimes described as a search for solutions in a landscape of possible designs. Increasingly, the search process can be automated. For example, genetic algorithms explore large design spaces and discover solutions human engineers would not imagine (or like). The ultimate genetic algorithm, evolution, notoriously generates designs that violate the rules of human engineering.

1.5 A new kind of thinking

We are getting farther afield now, but the shifts I am postulating in the criteria of scientific modeling are related to 20th Century developments in logic and epistemology.

Aristotelian logic → many-valued logic In traditional logic, any proposition is either true or false. This system lends itself to math-like proofs, but fails (in dramatic ways) for many real-world applications. Alternatives include many-valued logic, fuzzy logic, and other systems designed to handle indeterminacy, vagueness, and uncertainty. Bart Kosko discusses some of these systems in *Fuzzy Thinking*.

Frequentist probability → Bayesianism Bayesian probability has been around for centuries, but was not widely used until recently, facilitated by the availability of cheap computation and the reluctant acceptance of subjectivity in probabilistic claims. Sharon Bertsch McGrayne presents this history in *The Theory That Would Not Die*.

Objective → subjective The Enlightenment, and philosophic modernism, are based on belief in objective truth; that is, truths that are independent of the people that hold them. 20th Century developments including quantum mechanics, Gödel's Incompleteness Theorem, and Kuhn's study of the history of science called attention to seemingly unavoidable subjectivity in even "hard sciences" and mathematics. Rebecca Goldstein presents the historical context of Gödel's proof in *Incompleteness*.

Physical law → theory → model Some people distinguish between laws, theories, and models, but I think they are the same thing. People who use "law" are likely to believe that it is objectively true and immutable; people who use "theory" concede that it is subject to revision; and "model" concedes that it is based on simplification and approximation.

Some concepts that are called "physical laws" are really definitions; others are, in effect, the assertion that a model predicts or explains the behavior of a system particularly well. We come back to the nature of physical laws in Section 4.8, Section 5.9 and Section 8.7.

Determinism → indeterminism Determinism is the view that all events are caused, inevitably, by prior events. Forms of indeterminism include randomness, probabilistic causation, and fundamental uncertainty. We come back to this topic in Section 5.5 and Section 10.4

These trends are not universal or complete, but the center of opinion is shifting along these axes. As evidence, consider the reaction to Thomas Kuhn's *The Structure of Scientific Revolutions*, which was reviled when it was published and now considered almost uncontroversial.

These trends are both cause and effect of complexity science. For example, highly abstracted models are more acceptable now because of the diminished expectation that there should be a unique, correct model for every system. Conversely, developments in complex systems challenge determinism and the related concept of physical law.

This chapter is an overview of the themes coming up in the book, but not all of it will make sense before you see the examples. When you get to the end of the book, you might find it helpful to read this chapter again.

Chapter 2

Graphs

The first three chapters of this book are about models that describe systems that are made up of components and connections between components. For example, in an ecological food web, the components are species and the connections represent predator-prey relationships.

In this chapter, I introduce NetworkX, a Python package for building and studying these models. We start with the Erdős-Rényi model, which has some interesting mathematical properties. In the next chapter we move on to models that are more useful for explaining real-world systems.

The code for this chapter is in `chap02.ipynb` in the repository for this book. More information about working with the code is in Section 0.2.

2.1 What is a graph?

To most people a graph is a visual representation of a data set, like a bar chart or a plot of stock prices over time. That's not what this chapter is about.

In this chapter, a **graph** is a representation of a system that contains discrete, interconnected elements. The elements are represented by **nodes** and the interconnections are represented by **edges**.

For example, you could represent a road map with a node for each city and an edge for each road between cities. Or you could represent a social network

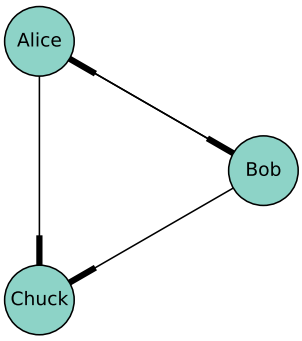

Figure 2.1: A directed graph that represents a social network.

using a node for each person, with an edge between two people if they are friends and no edge otherwise.

In some graphs, edges have attributes like length, cost, or weight. For example, in a road map, the length of an edge might represent the distance between two cities, or the travel time. In a social network there might be different kinds of edges to represent different kinds of relationships: friends, business associates, etc.

Edges may be **directed** or **undirected**, depending on whether the relationships they represent are asymmetric or symmetric. In a road map, you might represent a one-way street with a directed edge and a two-way street with an undirected edge. In some social networks, like Facebook, friendship is symmetric: if A is friends with B then B is friends with A. But in Twitter, for example, the "follows" relationship is not symmetric; if A follows B, that doesn't imply that B follows A. So you might use undirected edges to represent a Facebook network and directed edges for Twitter.

Graphs have interesting mathematical properties, and there is a branch of mathematics called **graph theory** that studies them.

Graphs are also useful, because there are many real world problems that can be solved using **graph algorithms**. For example, Dijkstra's shortest path algorithm is an efficient way to find the shortest path from a node to all other

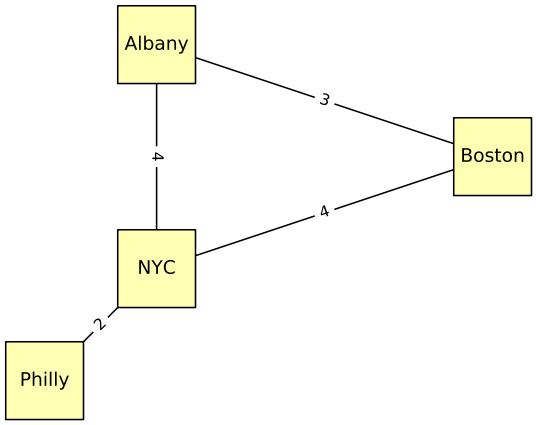

Figure 2.2: An undirected graph that represents cities and highways.

nodes in a graph. A **path** is a sequence of nodes with an edge between each consecutive pair.

Graphs are usually drawn with squares or circles for nodes and lines for edges. For example, the directed graph in Figure 2.1 might represent three people who "follow" each other on Twitter. The thick part of the line indicates edge direction. In this example, Alice and Bob follow each other and both follow Chuck, but Chuck follows no one.

The undirected graph in Figure 2.2 shows four cities in the northeast United States; the labels on the edges indicate driving time in hours. In this example the placement of the nodes corresponds roughly to the geography of the cities, but in general the layout of a graph is arbitrary.

2.2 NetworkX

To represent graphs, we'll use a package called NetworkX, which is the most commonly used network library in Python. You can read more about it at https://networkx.github.io/, but I'll explain it as we go along.

We can create a directed graph by importing NetworkX and instantiating `nx.DiGraph`:

```
import networkx as nx
G = nx.DiGraph()
```

It is conventional to import NetworkX as `nx`. At this point, `G` is a `DiGraph` object that contains no nodes and no edges. We can add nodes using the `add_node`method:

```
G.add_node('Alice')
G.add_node('Bob')
G.add_node('Chuck')
```

Now we can use the `nodes` method to get the list of nodes:

```
>>> G.nodes()
['Alice', 'Bob', 'Chuck']
```

Adding edges works pretty much the same way:

```
G.add_edge('Alice', 'Bob')
G.add_edge('Alice', 'Chuck')
G.add_edge('Bob', 'Alice')
G.add_edge('Bob', 'Chuck')
```

And we can use `edges` to get the list of edges:

```
>>> G.edges()
[('Alice', 'Bob'), ('Alice', 'Chuck'),
 ('Bob', 'Alice'), ('Bob', 'Chuck')]
```

NetworkX provides several functions for drawing graphs; `draw_circular`arranges the nodes in a circle and connects them with edges:

```
nx.draw_circular(G,
                 node_color=COLORS[0],
                 node_size=2000,
                 with_labels=True)
```

And that's the code I use to generate Figure 2.1. The option `with_labels`causes the nodes to be labeled; in the next example we'll see how to label the edges.

To generate Figure 2.2, I start with a dictionary that maps from each city name to its approximate longitude and latitude:

```
pos = dict(Albany=(-74, 43),
           Boston=(-71, 42),
           NYC=(-74, 41),
           Philly=(-75, 40))
```

Since this is an undirected graph, I instantiate `nx.Graph`:

```
G = nx.Graph()
```

Then I can use `add_nodes_from`to iterate the keys of `pos` and add them as nodes:

```
G.add_nodes_from(pos)
```

Next I'll make a dictionary that maps from each edge to the corresponding driving time:

```
drive_times = {('Albany', 'Boston'): 3,
               ('Albany', 'NYC'): 4,
               ('Boston', 'NYC'): 4,
               ('NYC', 'Philly'): 2}
```

Now I can use `add_edges_from`, which iterates the keys of `drive_times` and adds them as edges:

```
G.add_edges_from(drive_times)
```

Now instead of `draw_circular`, which arranges the nodes in a circle, I'll use `draw`, which takes `pos` as the second parameter:

```
nx.draw(G, pos,
        node_color=COLORS[1],
        node_shape='s',
        node_size=2500,
        with_labels=True)
```

`pos` is a dictionary that maps from each city to its coordinates; `draw` uses it to determine the locations of the nodes.

To add the edge labels, we use `draw_networkx_edge_labels`:

```
nx.draw_networkx_edge_labels(G, pos,
                             edge_labels=drive_times)
```

`drive_times` is a dictionary that maps from each edge, represented by a pair of city names, to the driving distance between them. And that's how I generated Figure 2.2.

In both of these examples, the nodes are strings, but in general they can be any hashable type.

2.3 Random graphs

A random graph is just what it sounds like: a graph with nodes and edges generated at random. Of course, there are many random processes that can generate graphs, so there are many kinds of random graphs.

One of the more interesting kinds is the Erdős-Rényi model, studied by Paul Erdős and Alfréd Rényi in the 1960s.

An Erdős-Rényi graph (ER graph) is characterized by two parameters: n is the number of nodes and p is the probability that there is an edge between any two nodes. See http://en.wikipedia.org/wiki/Erdos-Renyi_model.

Erdős and Rényi studied the properties of these random graphs; one of their surprising results is the existence of abrupt changes in the properties of random graphs as random edges are added.

One of the properties that displays this kind of transition is connectivity. An undirected graph is **connected** if there is a path from every node to every other node.

In an ER graph, the probability that the graph is connected is very low when p is small and nearly 1 when p is large. Between these two regimes, there is a rapid transition at a particular value of p, denoted p^*.

Erdős and Rényi showed that this critical value is $p^* = \ln n/n$, where n is the number of nodes. A random graph, $G(n, p)$, is unlikely to be connected if $p < p^*$ and very likely to be connected if $p > p^*$.

To test this claim, we'll develop algorithms to generate random graphs and check whether they are connected.

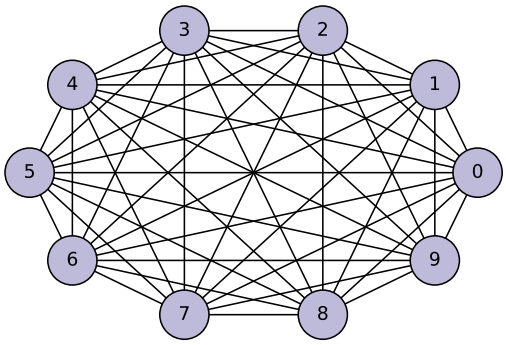

Figure 2.3: A complete graph with 10 nodes.

2.4 Generating graphs

I'll start by generating a **complete** graph, which is a graph where every node is connected to every other.

Here's a generator function that takes a list of nodes and enumerates all distinct pairs. If you are not familiar with generator functions, you might want to read Appendix **??** and then come back.

```
def all_pairs(nodes):
    for i, u in enumerate(nodes):
        for j, v in enumerate(nodes):
            if i>j:
                yield u, v
```

We can use `all_pairs` to construct a complete graph:

```
def make_complete_graph(n):
    G = nx.Graph()
    nodes = range(n)
    G.add_nodes_from(nodes)
    G.add_edges_from(all_pairs(nodes))
    return G
```

`make_complete_graph` takes the number of nodes, `n`, and returns a new `Graph` with `n` nodes and edges between all pairs of nodes.

The following code makes a complete graph with 10 nodes and draws it.

```
complete = make_complete_graph(10)
nx.draw_circular(complete,
                 node_color=COLORS[2],
                 node_size=1000,
                 with_labels=True)
```

Figure 2.3 shows the result. Soon we will modify this code to generate ER graphs, but first we'll develop functions to check whether a graph is connected.

2.5 Connected graphs

A graph is **connected** if there is a path from every node to every other node. See http://en.wikipedia.org/wiki/Connectivity_(graph_theory).

For many applications involving graphs, it is useful to check whether a graph is connected. Fortunately, there is a simple algorithm that does it.

You can start at any node and check whether you can reach all other nodes. If you can reach a node, v, you can reach any of the **neighbors** of v, which is any node connected by v by an edge.

The `Graph` class provides a method called `neighbors` that returns a list of neighbors for a given node. For example, in the complete graph we generated in the previous section:

```
>>> complete.neighbors(0)
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Suppose we start at node s. We can mark s as "seen", then we can mark its neighbors. Then we mark the neighbor's neighbors, and so on, until you can't reach any more nodes. If all nodes are seen, the graph is connected.

Here's what that looks like in Python:

```
def reachable_nodes(G, start):
    seen = set()
    stack = [start]
    while stack:
        node = stack.pop()
        if node not in seen:
            seen.add(node)
            stack.extend(G.neighbors(node))
    return seen
```

`reachable_nodes` takes a `Graph` and a starting node, `start`, and returns the set of nodes that can be reached from `start`.

Initially the set, `seen`, is empty, and we create a list called `stack` that keeps track of nodes we have discovered but not yet processed. Initially the stack contains a single node, `start`.

Now, each time through the loop, we

1. Remove one node from the stack.
2. If the node is already in `seen`, we go back to Step 1.
3. Otherwise, we add the node to `seen` and add its neighbors to the stack.

When the stack is empty, we can't reach any more nodes, so we break out of the loop and return `seen`.

As an example, we can find all nodes in the complete graph that are reachable from node 0:

```
>>> reachable_nodes(complete, 0)
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
```

Initially, the stack contains node 0 and `seen` is empty. The first time through the loop, node 0 is added to `seen` and all the other nodes are added to the stack (since they are all neighbors of node 0).

The next time through the loop, `pop` returns the last element in the stack, which is node 9. So node 9 gets added to `seen` and its neighbors get added to the stack.

Notice that the same node can appear more than once in the stack; in fact, a node with k neighbors will be added to the stack k times. Later we will look for ways to make this algorithm more efficient.

We can use `reachable_nodes` to write `is_connected`:

```
def is_connected(G):
    start = next(G.nodes_iter())
    reachable = reachable_nodes(G, start)
    return len(reachable) == len(G)
```

`is_connected` chooses a starting node by calling `nodes_iter`, which returns an iterator object, and passing the result to `next`, which returns the first node.

`reachable` gets the set of nodes that can be reached from `start`. If the size of this set is the same as the size of the graph, that means we can reach all nodes, which means the graph is connected.

A complete graph is, not surprisingly, connected:

```
>>> is_connected(complete)
True
```

In the next section we will generate ER graphs and check whether they are connected.

2.6 Generating ER graphs

The ER graph $G(n, p)$ contains n nodes, and each pair of nodes is connected by an edge with probability p. Generating an ER graph is similar to generating a complete graph.

The following generator function enumerates all possible edges and uses a helper function, `flip`, to choose which ones should be added to the graph:

```
def random_pairs(nodes, p):
    for i, u in enumerate(nodes):
        for j, v in enumerate(nodes):
            if i>j and flip(p):
                yield u, v
```

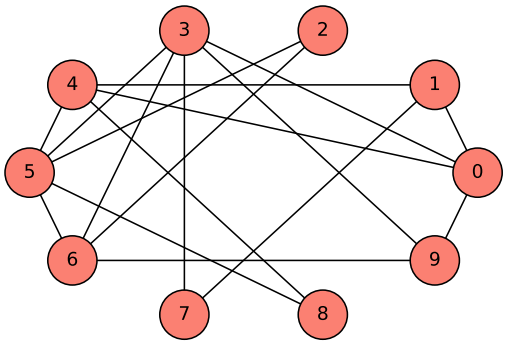

Figure 2.4: An ER graph with `n=10` and `p=0.3`.

`flip` returns `True` with the given probability, `p`, and `False` with the complementary probability `1-p`:

```
from numpy.random import random

def flip(p):
    return random() < p
```

Finally, `make_random_graph` generates and returns the ER graph $G(n, p)$.

```
def make_random_graph(n, p):
    G = nx.Graph()
    nodes = range(n)
    G.add_nodes_from(nodes)
    G.add_edges_from(random_pairs(nodes, p))
    return G
```

`make_random_graph` is almost identical to `make_complete_graph`; the only difference is that it uses `random_pairs` instead of `all_pairs`.

Here's an example with `p=0.3`:

```
random_graph = make_random_graph(10, 0.3)
```

Figure 2.4 shows the result. This graph turns out to be connected; in fact, most ER graphs with $n = 10$ and $p = 0.3$ are connected. In the next section, we'll see how many.

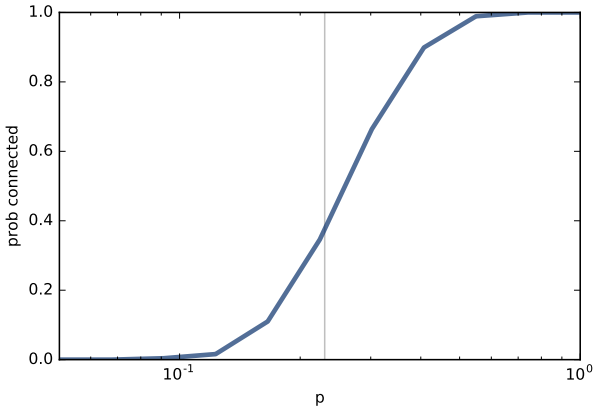

Figure 2.5: Probability of connectivity with $n = 10$ and a range of p. The vertical line shows the predicted critical value.

2.7 The probability of connectivity

For given values of n and p, we would like to know the probability that $G(n, p)$ is connected. We can estimate it by generating a large number of random graphs and counting how many are connected. Here's how:

```
def prob_connected(n, p, iters=100):
    count = 0
    for i in range(iters):
        random_graph = make_random_graph(n, p)
        if is_connected(random_graph):
            count += 1
    return count/iters
```

`iters` is the number of random graphs we generate. As we increase `iters`, the estimated probability gets more precise.

```
>>> prob_connected(10, 0.3, iters=10000)
0.6454
```

Out of 10000 ER graphs with these parameters, 6498 are connected, so we estimate that 65% of them are connected. So 0.3 is near the critical value where the probability of connectivity goes from near 0 to near 1. According to Erdős and Rényi, $p^* = \ln n/n = 0.23$.

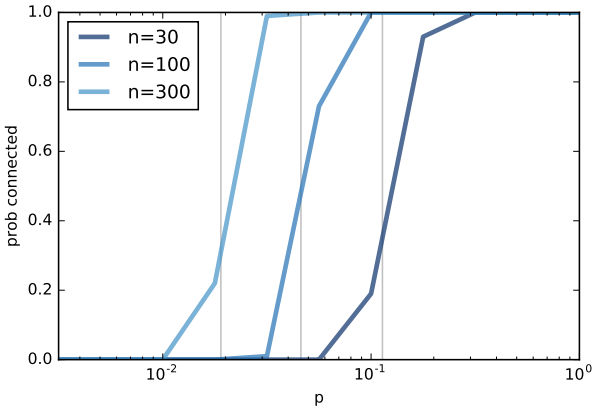

Figure 2.6: Probability of connectivity for several values of n and a range of p.

We can get a clearer view of the transition by estimating the probability of connectivity for a range of values of p:

```
import numpy as np

n = 10
ps = np.logspace(-2.5, 0, 11)
ys = [prob_connected(n, p) for p in ps]
```

This is the first example we've seen using NumPy. Following convention, I import NumPy as `np`. The function `logspace` returns an array of 11 values from $10^{-2.5}$ to $10^0 = 1$, equally spaced on a logarithmic scale.

To compute `ys`, I use a list comprehension that iterates the elements of `ps` and computes the probability that a random graph with each value of `p` is connected.

Figure 2.5 shows the results, with a vertical line at p^*. The transition from 0 to 1 occurs near the predicted critical value. With p on a log scale, the transition is roughly symmetric.

Figure 2.6 shows similar results for larger values of n. As n increases, the critical value gets smaller and the transition gets more abrupt.

These experiments are consistent with the results Erdős and Rényi proved in their papers.

2.8 Analysis of graph algorithms

I think chapter I presented an algorithm for checking whether a graph is connected; in the next few chapters, we will see a few more graph algorithms. And we will want to analyze the performance of those algorithms, figuring out how their run times grow as the size of the graphs increases.

If you are not already familiar with analysis of algorithms, you should read Appendix **??** before you continue.

The order of growth for graph algorithms is usually expressed as a function of n, the number of vertices, and m, the number of edges.

As an example, let's analyze `reachable_nodes` from Section 2.5:

```
def reachable_nodes(G, start):
    seen = set()
    stack = [start]
    while stack:
        node = stack.pop()
        if node not in seen:
            seen.add(node)
            stack.extend(G.neighbors(node))
    return seen
```

Each time through the loop, we pop a node off the stack; by default, `pop` removes and returns the last element of a list, which is a constant time operation.

Next we check whether the node is in `seen`, which is a set, so checking membership is constant time.

If the node is not already in `seen`, we add it, which is constant time, and then add the neighbors to the stack, which is linear in the number of neighbors.

To express the run time in terms of n and m, we can add up the total number of times each node is added `seen` and `stack`.

Each node is only added to `seen` once, so the total number of additions is n.

But nodes might be added to `stack` many times, depending on how many neighbors they have. If a node has k neighbors, it is added to `stack` k times. Of course, if it has k neighbors, that means it is connected to k edges.

So the total number of additions to `stack` is the total number of edges, m, doubled because we consider every edge twice.

Therefore, the order of growth for this function is $O(n + m)$, which is a convenient way to say that the run time grows in proportion to either n or m, whichever is "bigger."

If we know the relationship between n and m, we can simplify this expression. For example, in a complete graph the number of edges is $n(n-1)/2$, which is in $O(n^2)$. So for a complete graph, `reachable_nodes` is quadratic in n.

2.9 Exercises

The code for this chapter is in `chap02.ipynb`, which is a Jupyter notebook in the repository for this book. For more information about working with this code, see Section 0.2.

Exercise 2.1 Launch `chap02.ipynb` and run the code. There are a few short exercises embedded in the notebook that you might want to try.

Exercise 2.2 In Section 2.8 we analyzed the performance of `reachable_nodes` and classified it in $O(n + m)$, where n is the number of nodes and m is the number of edges. Continuing the analysis, what is the order of growth for `is_connected`?

```
def is_connected(G):
    start = next(G.nodes_iter())
    reachable = reachable_nodes(G, start)
    return len(reachable) == len(G)
```

Exercise 2.3 In my implementation of `reachable_nodes`, you might be bothered by the apparent inefficiency of adding *all* neighbors to the stack without checking whether they are already in `seen`. Write a version of this function that checks the neighbors before adding them to the stack. Does this "optimization" change the order of growth? Does it make the function faster?

Exercise 2.4 There are actually two kinds of ER graphs. The one we generated in this chapter, $G(n, p)$, is characterized by two parameters, the number of nodes and the probability of an edge between nodes.

An alternative definition, denoted $G(n, m)$, is also characterized by two parameters: the number of nodes, n, and the number of edges, m. Under this definition, the number of edges is fixed, but their location is random.

Repeat the experiments we did in this chapter using this alternative definition. Here are a few suggestions for how to proceed:

1. Write a function called `m_pairs` that takes a list of nodes and the number of edges, m, and returns a random selection of m edges. A simple way to do that is to generate a list of all possible edges and use `random.sample`.

2. Write a function called `make_m_graph` that takes n and m and returns a random graph with n nodes and m edges.

3. Make a version of `prob_connected` that uses `make_m_graph` instead of `make_random_graph`.

4. Compute the probability of connectivity for a range of values of m.

How do the results of this experiment compare to the results using the first type of ER graph?

Chapter 3

Small world graphs

Many networks in the real world, including social networks, have the "small world property", which is that the average distance between nodes, measured in number of edges on the shortest path, is much smaller than expected.

In this chapter, I present Stanley Milgram's famous Small World Experiment, which was the first scientific demonstration of the small world property in a real social network. Then we'll consider Watts-Strogatz graphs, which are intended as a model of small world graphs. I'll replicate the experiment Watts and Strogatz performed and explain what it is intended to show.

Along the way, we'll see two new graph algorithms: breadth-first search (BFS) and Dijkstra's algorithm for computing the shortest path between nodes in a graph.

The code for this chapter is in `chap03.ipynb` in the repository for this book. More information about working with the code is in Section 0.2.

3.1 Stanley Milgram

Stanley Milgram was an American social psychologist who conducted two of the most famous experiments in social science, the Milgram experiment, which studied people's obedience to authority (http://en.wikipedia.org/

wiki/Milgram_experiment) and the Small World Experiment, which studied the structure of social networks (http://en.wikipedia.org/wiki/Small_world_phenomenon).

In the Small World Experiment, Milgram sent a package to several randomly-chosen people in Wichita, Kansas, with instructions asking them to forward an enclosed letter to a target person, identified by name and occupation, in Sharon, Massachusetts (which is the town near Boston where I grew up). The subjects were told that they could mail the letter directly to the target person only if they knew him personally; otherwise they were instructed to send it, and the same instructions, to a relative or friend they thought would be more likely to know the target person.

Many of the letters were never delivered, but for the ones that were the average path length—the number of times the letters were forwarded—was about six. This result was taken to confirm previous observations (and speculations) that the typical distance between any two people in a social network is about "six degrees of separation."

This conclusion is surprising because most people expect social networks to be localized—people tend to live near their friends—and in a graph with local connections, path lengths tend to increase in proportion to geographical distance. For example, most of my friends live nearby, so I would guess that the average distance between nodes in a social network is about 50 miles. Wichita is about 1600 miles from Boston, so if Milgram's letters traversed typical links in the social network, they should have taken 32 hops, not six.

3.2 Watts and Strogatz

In 1998 Duncan Watts and Steven Strogatz published a paper in *Nature*, "Collective dynamics of 'small-world' networks", that proposed an explanation for the small world phenomenon. You can download it from http://www.nature.com/nature/journal/v393/n6684/abs/393440a0.html.

Watts and Strogatz start with two kinds of graph that were well understood: random graphs and regular graphs. In a random graph, nodes are connected at random. In a regular graph, every node has the same number of neighbors. They consider two properties of these graphs, clustering and path length:

Clustering is a measure of the "cliquishness" of the graph. In a graph, a **clique** is a subset of nodes that are all connected to each other; in a social network, a clique is a set of people who are all friends with each other. Watts and Strogatz defined a clustering coefficient that quantifies the likelihood that two nodes that are connected to the same node are also connected to each other.

Path length is a measure of the average distance between two nodes, which corresponds to the degrees of separation in a social network.

Watts and Strogatz show that regular graphs have high clustering and high path lengths, whereas random graphs with the same size usually have low clustering and low path lengths. So neither of these is a good model of social networks, which combine high clustering with short path lengths.

Their goal was to create a **generative model** of a social network. A generative model tries to explain a phenomenon by modeling the process that builds or leads to the phenomenon. Watts and Strogatz proposed this process for building small-world graphs:

1. Start with a regular graph with n nodes and each node connected to k neighbors.

2. Choose a subset of the edges and "rewire" them by replacing them with random edges.

The probability that an edge is rewired is a parameter, p, that controls how random the graph is. With $p = 0$, the graph is regular; with $p = 1$ it is random.

Watts and Strogatz found that small values of p yield graphs with high clustering, like a regular graph, and low path lengths, like a random graph.

In this chapter I replicate the Watts and Strogatz experiment in the following steps:

1. We'll start by constructing a ring lattice, which is a kind of regular graph.

2. Then we'll rewire it as Watts and Strogatz did.

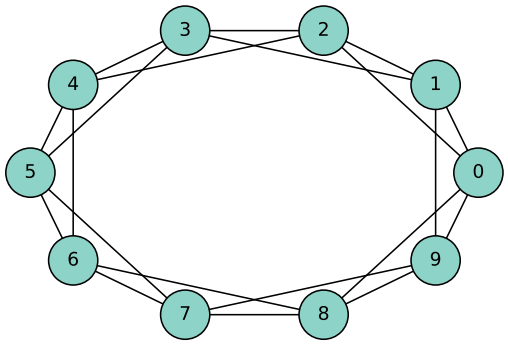

Figure 3.1: A ring lattice with $n = 10$ and $k = 4$.

3. We'll write a function to measure the degree of clustering and use a NetworkX function to compute path lengths.

4. Then we'll compute the degree of clustering and path length for a range of values of p.

5. Finally, I'll present an efficient algorithm for computing shortest paths, Dijkstra's algorithm.

3.3 Ring lattice

A **regular** graph is a graph where each node has the same number of neighbors; the number of neighbors is also called the **degree** of the node.

A ring lattice is a kind of regular graph, which Watts and Strogatz use as the basis of their model. In a ring lattice with n nodes, the nodes can be arranged in a circle with each node connected the k nearest neighbors.

For example, a ring lattice with $n = 3$ and $k = 2$ would add the following edges: $(0, 1)$, $(1, 2)$, and $(2, 0)$. Notice that the edges "wrap around" from the highest-numbered node back to 0.

More generally, we can enumerate the edges like this:

```
def adjacent_edges(nodes, halfk):
    n = len(nodes)
    for i, u in enumerate(nodes):
        for j in range(i+1, i+halfk+1):
            v = nodes[j % n]
            yield u, v
```

`adjacent_edges` takes a list of nodes and a parameter, `halfk`, which is half of k. It is a generator function that yields one edge at a time. It uses the modulus operator, `%`, to wrap around from the highest-numbered node to the lowest.

We can test it like this:

```
>>> nodes = range(3)
>>> for edge in adjacent_edges(nodes, 1):
...     print(edge)
(0, 1)
(1, 2)
(2, 0)
```

Now we can use `adjacent_edges` to make a ring lattice:

```
def make_ring_lattice(n, k):
    G = nx.Graph()
    nodes = range(n)
    G.add_nodes_from(nodes)
    G.add_edges_from(adjacent_edges(nodes, k//2))
    return G
```

Notice that `make_ring_lattice` uses floor division to compute `halfk`, so if `k` is odd, it will round down and generate a ring lattice with degree `k-1`. That's probably not what we want, but it is good enough for now.

We can test the function like this:

```
lattice = make_ring_lattice(10, 4)
```

Figure 3.1 shows the result.

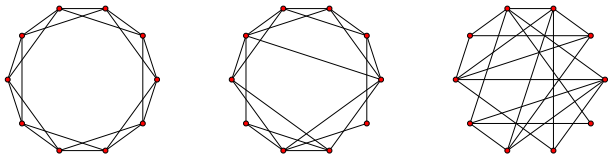

Figure 3.2: WS graphs with $n = 20$, $k = 4$, and $p = 0$ (left), $p = 0.2$ (middle), and $p = 1$ (right).

3.4 WS graphs

To make a Watts-Strogatz (WS) graph, we start with a ring lattice and "rewire" some of the edges. In their paper, Watts and Strogatz consider the edges in a particular order and rewire each one with probability p. If an edge is rewired, they leave the first node unchanged and choose the second node at random. They don't allow self loops or multiple edges; that is, you can't have a edge from a node to itself, and you can't have more than one edge between the same two nodes.

Here is my implementation of this process.

```
def rewire(G, p):
    nodes = set(G.nodes())
    for edge in G.edges():
        if flip(p):
            u, v = edge
            choices = nodes - {u} - set(G[u])
            new_v = choice(tuple(choices))
            G.remove_edge(u, v)
            G.add_edge(u, new_v)
```

The parameter `p` is the probability of rewiring an edge. The `for` loop enumerates the edges and uses `flip`, which returns `True` with probability `p`, to choose which ones get rewired.

If we are rewiring an edge from node `u` to node `v`, we have to choose a replacement for `v`, called `new_v`. To compute the possible choices, we start with

`nodes`, which is a set, and subtract off `u` and its neighbors, which avoids self loops and multiple edges.

Then we choose `new_v` from `choices`, remove the existing edge from `u` to `v`, and add a new edge from `u` to `new_v`.

As an aside, the expression `G[u]` returns a dictionary that contains the neighbors of `u` as keys. In this case it is a little faster than using `G.neighbors`.

This function does not consider the edges in the order specified by Watts and Strogatz, but it doesn't seem to affect the results.

Figure 3.2 shows WS graphs with $n = 20$, $k = 4$, and a range of values of p. When $p = 0$, the graph is a ring lattice. When $p = 1$, it is completely random. As we'll see, the interesting things happen in between.

3.5 Clustering

The next step is to compute the clustering coefficient, which quantifies the tendency for the nodes to form cliques. A **clique** is a set of nodes that are completely connected; that is, there are edges between all pairs of nodes in the set.

Suppose a particular node, u, has k neighbors. If all of the neighbors are connected to each other, there would be $k(k-1)/2$ edges among them. The fraction of those edges that actually exist is the local clustering coefficient for u, denoted C_u. It is called a "coefficient" because it is always between 0 and 1.

If we compute the average of C_u over all nodes, we get the "network average clustering coefficient", denoted $\bar{C}$.

Here is a function that computes it.

```
def node_clustering(G, u):
    neighbors = G[u]
    k = len(neighbors)
    if k < 2:
        return 0

    total = k * (k-1) / 2
    exist = 0
    for v, w in all_pairs(neighbors):
        if G.has_edge(v, w):
            exist +=1
    return exist / total
```

Again I use `G[u]`, which returns a dictionary with the neighbors of `node` as keys. If a node has fewer than 2 neighbors, the clustering coefficient is undefined, but for simplicity `node_clustering` returns 0.

Otherwise we compute the number of possible edges among the neighbors, `total`, and then count the number of those edges that actually exist. The result is the fraction of all edges that exist.

We can test the function like this:

```
>>> lattice = make_ring_lattice(10, 4)
>>> node_clustering(lattice, 1)
0.5
```

In a ring lattice with $k = 4$, the clusting coefficient for each node is 0.5 (if you are not convinced, take another look at Figure 3.1).

Now we can compute the network average clustering coefficient like this:

```
def clustering_coefficient(G):
    cc = np.mean([node_clustering(G, node) for node in G])
    return cc
```

`np.mean` is a NumPy function that computes the mean of the numbers in a list or array.

And we can test it like this:

```
>>> clustering_coefficient(lattice)
0.5
```

In this graph, the local clustering coefficient for all nodes is 0.5, so the average across nodes is 0.5. Of course, we expect this value to be different for WS graphs.

3.6 Shortest path lengths

The next step is to compute the characteristic path length, L, which is the average length of the shortest path between each pair of nodes. To compute it, I'll start with a function provided by NetworkX, `shortest_path_length`. I'll use it to replicate the Watts and Strogatz experiment, then I'll explain how it works.

Here's a function that takes a graph and returns a list of shortest path lengths, one for each pair of nodes.

```
def path_lengths(G):
    length_map = nx.shortest_path_length(G)
    lengths = [length_map[u][v] for u, v in all_pairs(G)]
    return lengths
```

The return value from `nx.shortest_path_length` is a dictionary of dictionaries. The outer dictionary maps from each node, `u`, to a dictionary that maps from each node, `v`, to the length of the shortest path from `u` to `v`.

With the list of lengths from `path_lengths`, we can compute L like this:

```
def characteristic_path_length(G):
    return np.mean(path_lengths(G))
```

And we can test it with a small ring lattice:

```
>>> lattice = make_ring_lattice(3, 2)
>>> characteristic_path_length(lattice)
1.0
```

In this example, all 3 nodes are connected to each other, so the mean path length is 1.

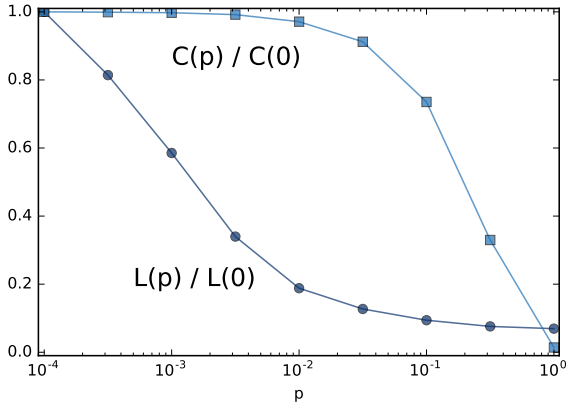

Figure 3.3: Clustering coefficient (C) and characteristic path length (L) for WS graphs with $n = 1000$, $k = 10$, and a range of p.

3.7 The WS experiment

Now we are ready to replicate the WS experiment, which shows that for a range of values of p, a WS graph has high clustering like a regular graph and short path lengths like a random graph.

I'll start with `run_one_graph`, which takes `n`, `k`, and `p`; it generates a WS graph with the given parameters and computes the mean path length, `mpl`, and clustering coefficient `cc`:

```
def run_one_graph(n, k, p):
    ws = make_ws_graph(n, k, p)
    mpl = characteristic_path_length(ws)
    cc = clustering_coefficient(ws)
    print(mpl, cc)
    return mpl, cc
```

Watts and Strogatz ran their experiment with `n=1000` and `k=10`. With these parameters, `run_one_graph` takes about one second on my computer; most of that time is spent computing the mean path length.

Now we need to compute these values for a range of `p`. I'll use the NumPy function `logspace` again to compute `ps`:

```
ps = np.logspace(-4, 0, 9)
```

For each value of `p`, I generate 3 random graphs and we'll average the results. Here's the function that runs the experiment:

```
def run_experiment(ps, n=1000, k=10, iters=3):
    res = {}
    for p in ps:
        print(p)
        res[p] = []
        for _ in range(iters):
            res[p].append(run_one_graph(n, k, p))
    return res
```

The result is a dictionary that maps from each value of `p` to a list of `(mpl, cc)` pairs.

The last step is to aggregate the results:

```
L = []
C = []
for p, t in sorted(res.items()):
    mpls, ccs = zip(*t)
    mpl = np.mean(mpls)
    cc = np.mean(ccs)
    L.append(mpl)
    C.append(cc)
```

Each time through the loop, we get a value of p and a list of `(mpl, cc)` pairs. We use `zip` to extracts two lists, `mpls` and `ccs`, then compute their means and append them to `L` and `C`, which are lists of path lengths and clustering coefficients.

In order to plot `L` and `C` on the same axes, we standardize them by dividing through by the first element:

```
L = np.array(L) / L[0]
C = np.array(C) / C[0]
```

Figure 3.3 shows the results. As p increases, the mean path length drops quickly, because even a small number of randomly rewired edges provide shortcuts between regions of the graph that are far apart in the lattice. On the other

hand, removing local links decreases the clustering coefficient, but much more slowly.

As a result, there is a wide range of p where a WS graph has the properties of a small world graph, high clustering and low path lengths.

And that's why Watts and Strogatz propose WS graphs as a model for real-world networks that exhibit the small world phenomenon.

3.8 What kind of explanation is *that*?

If you ask me why planetary orbits are elliptical, I might start by modeling a planet and a star as point masses; I would look up the law of universal gravitation at http://en.wikipedia.org/wiki/Newton's_law_of_universal_gravitation and use it to write a differential equation for the motion of the planet. Then I would either derive the orbit equation or, more likely, look it up at http://en.wikipedia.org/wiki/Orbit_equation. With a little algebra, I could derive the conditions that yield an elliptical orbit. Then I would argue that the objects we consider planets satisfy these conditions.

People, or at least scientists, are generally satisfied with this kind of explanation. One of the reasons for its appeal is that the assumptions and approximations in the model seem reasonable. Planets and stars are not really point masses, but the distances between them are so big that their actual sizes are negligible. Planets in the same solar system can affect each others' orbits, but the effect is usually small. And we ignore relativistic effects, again on the assumption that they are small.

This explanation is also appealing because it is equation-based. We can express the orbit equation in a closed form, which means that we can compute orbits efficiently. It also means that we can derive general expressions for the orbital velocity, orbital period, and other quantities.

Finally, I think this kind of explanation is appealing because it has the form of a mathematical proof. It starts from a set of axioms and derives the result by logic and analysis. But it is important to remember that the proof pertains to the model and not the real world. That is, we can prove that an idealized

model of a planet yields an elliptic orbit, but we can't prove that the model pertains to actual planets (in fact, it does not).

By comparison, Watts and Strogatz's explanation of the small world phenomenon may seem less satisfying. First, the model is more abstract, which is to say less realistic. Second, the results are generated by simulation, not by mathematical analysis. Finally, the results seem less like a proof and more like an example.

Many of the models in this book are like the Watts and Strogatz model: abstract, simulation-based and (at least superficially) less formal than conventional mathematical models. One of the goals of this book it to consider the questions these models raise:

- What kind of work can these models do: are they predictive, or explanatory, or both?
- Are the explanations these models offer less satisfying than explanations based on more traditional models? Why?
- How should we characterize the differences between these and more conventional models? Are they different in kind or only in degree?

Over the course of the book I will offer my answers to these questions, but they are tentative and sometimes speculative. I encourage you to consider them skeptically and reach your own conclusions.

3.9 Breadth-First Search

When we computed shortest paths, we used a function provided by NetworkX, but I have not explained how it works. To do that, I'll start with breadth-first search, which is the basis of Dijkstra's algorithm for computing shortest paths.

In Section 2.5 I presented `reachable_nodes`, which finds all the nodes that can be reached from a given starting node:

```
def reachable_nodes(G, start):
    seen = set()
    stack = [start]
    while stack:
        node = stack.pop()
        if node not in seen:
            seen.add(node)
            stack.extend(G.neighbors(node))
    return seen
```

I didn't say so at the time, but `reachable_nodes` performs a depth-first search (DFS). Now we'll modify it to perform breadth-first search (BFS).

To understand the difference, imagine you are exploring a castle. You start in a room with three doors marked A, B, and C. You open door C and discover another room, with doors marked D, E, and F.

Which door do you open next? If you are feeling adventurous, you might want to go deeper into the castle and choose D, E, or F. That would be a depth-first search.

But if you wanted to be more systematic, you might go back and explore A and B before D, E, and F. That would be a breadth-first search.

In `reachable_nodes`, we use `list.pop` to choose the next node to "explore". By default, `pop` returns the last element of the list, which is the last one we added. In the example, that would be door F.

If we want to perform a BFS instead, the simplest solution is to pop the first element off the stack:

```
            node = stack.pop(0)
```

That works, but it is slow. In Python, popping the last element of a list takes constant time, but popping the first element is linear in the length of the list. In the worst case, the length of the stack is $O(n)$, which makes this implementation of BFS $O(nm)$, which is much worse than what it should be, $O(n + m)$.

We can solve this problem with a double-ended queue, also known as a **deque**. The important feature of a deque is that you can add and remove elements

from the beginning or end in constant time. To see how it is implemented, see https://en.wikipedia.org/wiki/Double-ended_queue.

Python provides a `deque` in the `collections` module, so we can import it like this:

```
from collections import deque
```

And we can use it to write an efficient BFS:

```
def reachable_nodes_bfs(G, start):
    seen = set()
    queue = deque([start])
    while queue:
        node = queue.popleft()
        if node not in seen:
            seen.add(node)
            queue.extend(G.neighbors(node))
    return seen
```

The differences are:

- I replaced the list called `stack` with a deque called `queue`.
- I replaced `pop` with `popleft`, which removes and returns leftmost element of the queue, which was the first to be added.

This version is back to being $O(n + m)$. Now we're ready to find shortest paths.

3.10 Dijkstra's algorithm

Edsger W. Dijkstra was a Dutch computer scientist who invented an efficient shortest-path algorithm (see http://en.wikipedia.org/wiki/Dijkstra's_algorithm). He also invented the semaphore, which is a data structure used to coordinate programs that communicate with each other (see http://en.wikipedia.org/wiki/Semaphore_(programming) and Downey, *The Little Book of Semaphores*).

Dijkstra is famous (and notorious) as the author of a series of essays on computer science. Some, like "A Case against the GO TO Statement", have had a profound effect on programming practice. Others, like "On the Cruelty of Really Teaching Computing Science", are entertaining in their cantankerousness, but less effective.

Dijkstra's algorithm solves the "single source shortest path problem", which means that it finds the minimum distance from a given "source" node to every other node in the graph (or at least every connected node).

We start with a simplified version of the algorithm that considers all edges the same length. The more general version works with any non-negative edge lengths.

The simplified version is similar to the breadth-first search in Section **??** except that we replace the set called `seen` with a dictionary called `dist`, which maps from each node to its distance from the source:

```
def shortest_path_dijkstra(G, start):
    dist = {start: 0}
    queue = deque([start])
    while queue:
        node = queue.popleft()
        new_dist = dist[node] + 1

        neighbors = set(G[node]) - set(dist)
        for n in neighbors:
            dist[n] = new_dist

        queue.extend(neighbors)
    return dist
```

Here's how it works:

- Initially, `queue` contains a single element, `start`, and `dist` maps from `start` to distance 0 (which is the distance from `start` to itself).
- Each time through the loop, we use `popleft` to select nodes in the order they were added to the queue.
- Next we find all neighbors of `node` that are not already in `dist`.

- Since the distance from `start` to `node` is `dist[node]`, the distance to any of the undiscovered neighbors is `dist[node]+1`.
- For each neighbor, we add an entry to `dist`, then we add the neighbors to the queue.

This algorithm only works if we use BFS, not DFS. Why?

The first time through the loop `node` is `start`, and `new_dist` is 1. So the neighbors of `start` get distance 1 and they go in the queue.

When we process the neighbors of `start`, all of *their* neighbors get distance 2. We know that none of them can have distance 1, because if they did, we would have discovered them during the first iteration.

Similarly, when we process the nodes with distance 2, we give their neighbors distance 3. We know that none of them can have distance 1 or 2, because if they did, we would have discovered them during a previous iteration.

And so on. If you are familiar with proof by induction, you can see where this is going.

But this argument only works if we process all nodes with distance 1 before we start processing nodes with distance 2, and so on. And that's exactly what BFS does.

In the exercises at the end of this chapter, you'll write a version of Dijkstra's algorithm using DFS, so you'll have a chance to see what goes wrong.

3.11 Exercises

Exercise 3.1 In a ring lattice, every node has the same number of neighbors. The number of neighbors is called the **degree** of the node, and a graph where all nodes have the same degree is called a **regular graph**.

All ring lattices are regular, but not all regular graphs are ring lattices. In particular, if `k` is odd, we can't construct a ring lattice, but we might be able to construct a regular graph.

Write a function called `make_regular_graph` that takes `n` and `k` and returns a regular graph that contains `n` nodes, where every node has `k` neighbors. If it's not possible to make a regular graph with the given values of `n` and `k`, the function should raise a `ValueError`.

Exercise 3.2 My implementation of `reachable_nodes_bfs` is efficient in the sense that it is in $O(n + m)$, but it incurs a lot of overhead adding nodes to the queue and removing them. NetworkX provides a simple, fast implementation of BFS, available from the NetworkX repository on GitHub at https://github.com/networkx/networkx/blob/master/networkx/algorithms/components/connected.py.

Here is a version I modified to return a set of nodes:

```
def _plain_bfs(G, source):
    seen = set()
    nextlevel = {source}
    while nextlevel:
        thislevel = nextlevel
        nextlevel = set()
        for v in thislevel:
            if v not in seen:
                seen.add(v)
                nextlevel.update(G[v])
    return seen
```

Compare this function to `reachable_nodes_bfs` and see which is faster. Then see if you can modify this function to implement a faster version of `shortest_path_dijkstra`

Exercise 3.3 The following implementation of a BFS contains two performance errors. What are they? What is the actual order of growth for this algorithm?

```
def bfs(top_node, visit):
    """Breadth-first search on a graph, starting at top_node."""
    visited = set()
    queue = [top_node]
    while len(queue):
        curr_node = queue.pop(0)    # Dequeue
        visit(curr_node)            # Visit the node
        visited.add(curr_node)

        # Enqueue non-visited and non-enqueued children
        queue.extend(c for c in curr_node.children
                     if c not in visited and c not in queue)
```

Exercise 3.4 In Section 3.10, I claimed that Dijkstra's algorithm does not work unless it uses BFS. Write a version of `shortest_path_dijkstra` that uses DFS and test it on a few examples to see what goes wrong.

Exercise 3.5 A natural question about the Watts and Strogatz paper is whether the small world phenomenon is specific to their generative model or whether other similar models yield the same qualitative result (high clustering and low path lengths).

To answer this question, choose a variation of the Watts and Strogatz model and repeat the experiment. There are two kinds of variation you might consider:

- Instead of starting with a regular graph, start with another graph with high clustering. For example, you could put nodes at random locations in a 2-D space and connect each node to its nearest k neighbors.
- Experiment with different kinds of rewiring.

If a range of similar models yield similar behavior, we say that the results of the paper are **robust**.

Exercise 3.6 Dijkstra's algorithm solves the "single source shortest path" problem, but to compute the characteristic path length of a graph, we actually want to solve the "all pairs shortest path" problem.

Of course, one option is to run Dijkstra's algorithm n times, once for each starting node. And for some applications, that's probably good enough. But there are are more efficient alternatives.

Find an algorithm for the all-pairs shortest path problem and implement it. See https://en.wikipedia.org/wiki/Shortest_path_problem#All-pairs_shortest_paths.

Compare the run time of your implementation with running Dijkstra's algorithm n times. Which algorithm is better in theory? Which is better in practice? Which one does NetworkX use?

Chapter 4

Scale-free networks

In this chapter, we'll work with data from an online social network, and use a Watts-Strogatz graph to model it. The WS model has characteristics of a small world network, like the data, but it has low variability in the number of neighbors from node to node, unlike the data.

This discrepancy is the motivation for a network model developed by Barabási and Albert. The BA model captures the observed variability in the number of neighbors, and it has one of the small world properties, short path lengths, but it does not have the high clustering of a small world network.

The chapter ends with a discussion of WS and BA graphs as explanatory models for small world networks.

The code for this chapter is in `chap04.ipynb` in the respository for this book. More information about working with the code is in Section 0.2.

4.1 Social network data

Watts-Strogatz graphs are intended to model networks in the natural and social sciences. In their original paper, Watts and Strogatz looked at the network of film actors (connected if they have appeared in a movie together); the electrical power grid in the western United States; and the network of neurons in the brain of the roundworm *C. elegans*. They found that all of

these networks had the high connectivity and low path lengths characteristic of small world graphs.

In this section we'll perform the same analysis with a different dataset, a set Facebook users and their friends. If you are not familiar with Facebook, users who are connected to each other are called "friends", regardless of the nature of their relationship in the real world.

I'll use data from the Stanford Network Analysis Project (SNAP), which shares large datasets from online social networks and other sources. Specifically, I'll use their Facebook dataset[1], which includes 4039 users and 88,234 friend relationships among them. This dataset is in the repository for this book, but it is also available from the SNAP web site at https://snap.stanford.edu/data/egonets-Facebook.html

The data file contains one line per edge, with users identified by integers from 0 to 4038. Here's the code that reads the file:

```
def read_graph(filename):
    G = nx.Graph()
    array = np.loadtxt(filename, dtype=int)
    G.add_edges_from(array)
    return G
```

NumPy provides a function called `loadtext` that reads the given file and returns the contents as a NumPy array. The parameter `dtype` specifies the type of the elements of the array.

Then we can use `add_edges_from` to iterate the rows of the array and make edges. Here are the results:

```
>>> fb = read_graph('facebook_combined.txt.gz')
>>> n = len(fb)
>>> m = len(fb.edges())
>>> n, m
(4039, 88234)
```

The number of nodes and edges are consistent with the documentation of the dataset.

[1] J. McAuley and J. Leskovec. Learning to Discover Social Circles in Ego Networks. NIPS, 2012.

Now we can check whether this dataset has the characteristics of a small world graph: high clustering and low path lengths.

In Section 3.5 we wrote a function to compute the network average clustering coefficient. NetworkX provides a function called `average_clustering`, which does the same thing a little faster.

But for larger graphs, they are both too slow, taking time proportional to nk^2, where n is the number of nodes and k is the number of neighbors each node is connected to.

Fortunately, NetworkX provides a function that estimates the clustering coefficient by random sampling. You can invoke it like this:

```
from networkx.algorithms.approximation import average_clustering
average_clustering(G, trials=1000)
```

The following function does something similar for path lengths.

```
def random_path_lengths(G, nodes=None, trials=1000):
    if nodes is None:
        nodes = G.nodes()
    else:
        nodes = list(nodes)

    pairs = np.random.choice(nodes, (trials, 2))
    lengths = [nx.shortest_path_length(G, *pair)
               for pair in pairs]
    return lengths
```

`G` is a graph, `nodes` is the list of nodes we should sample from, and `trials` is the number of random paths to sample. If `nodes` is `None`, we sample from the entire graph.

`pairs` is a NumPy array of randomly chosen nodes with one row for each trial and two columns.

The list comprehension enumerates the rows in the array and computes the shortest distance between each pair of nodes. The result is a list of path lengths.

`estimate_path_length` generates a list of random path lengths and returns their mean:

```
def estimate_path_length(G, nodes=None, trials=1000):
    return np.mean(random_path_lengths(G, nodes, trials))
```

I'll use `average_clustering` to compute C:

```
C = average_clustering(fb)
```

And `estimate_path_lengths` to compute L:

```
L = estimate_path_lengths(fb)
```

The clustering coefficient is about 0.61, which is high, as we expect if this network has the small world property.

And the average path is 3.7, which is quite short in a network of more than 4000 users. It's a small world after all.

Now let's see if we can construct a WS graph that has the same characteristics as this network.

4.2 WS Model

In the Facebook dataset, the average number of edges per node is about 22. Since each edge is connected to two nodes, the average degree is twice the number of edges per node:

```
>>> k = int(round(2*m/n))
>>> k
44
```

We can make a WS graph with `n=4039` and `k=44`. When `p=0`, we get a ring lattice.

```
lattice = nx.watts_strogatz_graph(n, k, 0)
```

In this graph, clustering is high: `C` is 0.73, compared to 0.61 in the dataset. But `L` is 46, much higher than in the dataset!

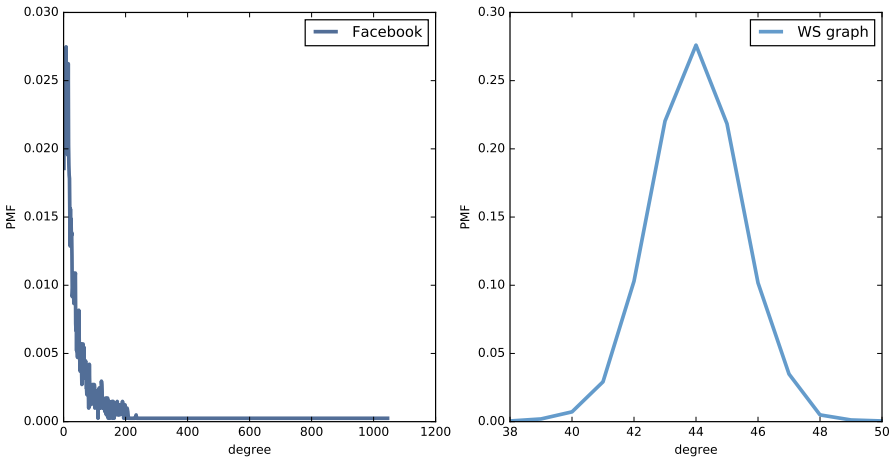

Figure 4.1: PMF of degree in the Facebook dataset and in the WS model.

With `p=1` we get a random graph:

```
random_graph = nx.watts_strogatz_graph(n, k, 1)
```

In the random graph, `L` is 2.6, even shorter than in the dataset (3.7), but `C` is only 0.011, so that's no good.

By trial and error, we find that when `p=0.05` we get a WS graph with high clustering and low path length:

```
ws = nx.watts_strogatz_graph(n, k, 0.05, seed=15)
```

In this graph `C` is 0.63, a bit higher than in the dataset, and `L` is 3.2, a bit lower than in the dataset. So this graph models the small world characteristics of the dataset well.

So far, so good.

4.3 Degree

Recall that the degree of a node is the number of neighbors it is connected to. If the WS graph is a good model for the Facebook network, it should have the

same total (or average) degree, and ideally the same variance in degree, across nodes.

This function returns a list of degrees in a graph, one for each node:

```
def degrees(G):
    return [G.degree(u) for u in G]
```

The mean degree in the dataset is 43.7; the mean degree in the WS model is 44. So far, so good.

However, the standard deviation of degrees in the WS model is 1.5; the standard deviation in the data is 52.4. Oops.

What's going on here? To get a better view, we have to look at the **distribution** of degrees, not just the mean and standard deviation.

I'll represent the distribution of degrees with a `Pmf` object, which is defined in the `thinkstats2` module. `Pmf` stands for "probability mass function"; if you are not familiar with this concept, you might want to read Chapter 3 of *Think Stats, 2nd edition* at http://greenteapress.com/thinkstats2/html/thinkstats2004.html.

Briefly, a `Pmf` maps from values to their probabilities. A `Pmf` of degrees is a mapping from each possible degree, d, to the fraction of nodes with degree d.

As an example, I'll construct a graph with nodes 1, 2, and 3 connected to a central node, 0:

```
G = nx.Graph()
G.add_edge(1, 0)
G.add_edge(2, 0)
G.add_edge(3, 0)
nx.draw(G)
```

Here's the list of degrees in this graph:

```
>>> degrees(G)
[3, 1, 1, 1]
```

Node 0 has degree 3, the others have degree 1. Now I can make a `Pmf` that represents this degree distribution:

```
>>> from thinkstats2 import Pmf
>>> Pmf(degrees(G))
Pmf({1: 0.75, 3: 0.25})
```

The result is a `Pmf` object that maps from each degree to a fraction or probability. In this example, 75% of the nodes have degree 1 and 25% have degree 3.

Now we can make a `Pmf` that contains node degrees from the dataset, and compute the mean and standard deviation:

```
>>> from thinkstats2 import Pmf
>>> pmf_fb = Pmf(degrees(fb))
>>> pmf_fb.Mean(), pmf_fb.Std()
(43.691, 52.414)
```

And the same for the WS model:

```
>>> pmf_ws = Pmf(degrees(ws))
>>> pmf_ws.mean(), pmf_ws.std()
(44.000, 1.465)
```

We can use the `thinkplot` module to plot the results:

```
thinkplot.Pdf(pmf_fb, label='Facebook')
thinkplot.Pdf(pmf_ws, label='WS graph')
```

Figure 4.1 shows the two distributions. They are very different.

In the WS model, most users have about 44 friends; the minimum is 38 and the maximum is 50. That's not much variation. In the dataset, there are many users with only 1 or 2 friends, but one has more than 1000!

Distributions like this, with many small values and a few very large values, are called **heavy-tailed**.

4.4 Heavy-tailed distributions

Heavy-tailed distributions are a common feature in many areas of complexity science and they will be a recurring theme of this book.

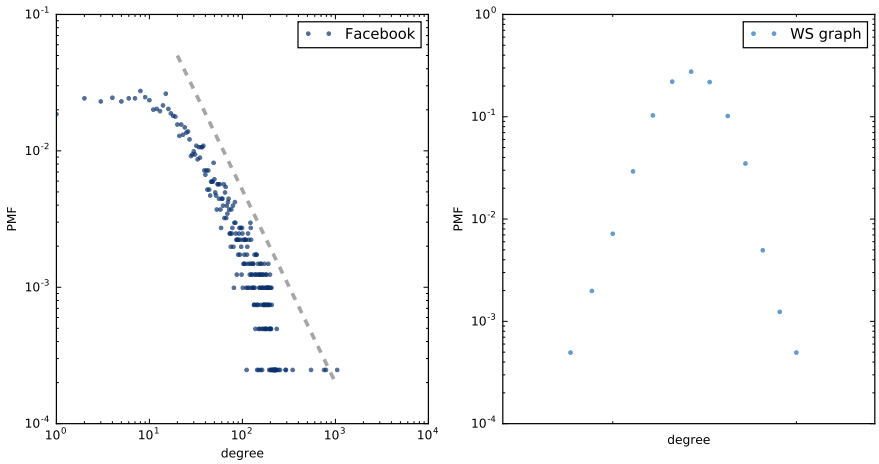

Figure 4.2: PMF of degree in the Facebook dataset and in the WS model, on a log-log scale.

We can get a clearer picture of a heavy-tailed distribution by plotting it on a log-log axis, as shown in Figure 4.2. This transformation emphasizes the tail of the distribution; that is, the probabilities of large values.

Under this transformation, the data fall approximately on a straight line, which suggests that there is a "power law" relationship between the largest values in the distribution and their probabilities. Mathematically,

$$\text{PMF}(k) \sim k^{-\alpha}$$

where $\text{PMF}(k)$ is the fraction of nodes with degree k, α is a parameter, and the symbol $\sim$ indicates that the PMF is asymptotic to $k^{-\alpha}$ as k increases.

If we take the log of both sides, we get

$$\log \text{PMF}(k) \sim -\alpha \log k$$

So if a distribution follows a power law and we plot $\text{PMF}(k)$ versus k on a log-log scale, we expect a straight line with slope $-\alpha$, at least for large values of k.

All power law distributions are heavy-tailed, but there are other heavy-tailed distributions that don't follow a power law. We will see more examples soon.

But first, we have a problem: the WS model has the high clustering and low path length we see in the data, but the degree distribution doesn't resemble the data at all. This discrepancy is the motivation for our next topic, the Barabási-Albert model.

4.5 Barabási-Albert model

In 1999 Barabási and Albert published a paper, "Emergence of Scaling in Random Networks", that characterizes the structure of several real-world networks, including graphs that represent the interconnectivity of movie actors, world-wide web (WWW) pages, and elements in the electrical power grid in the western United States. You can download the paper from http://www.sciencemag.org/content/286/5439/509.

They measure the degree of each node and compute PMF(k), the probability that a vertex has degree k. Then they plot PMF(k) versus k on a log-log scale. The plots fit a straight line, at least for large values of k; so Barabási and Albert conclude that these distributions are heavy-tailed.

They also propose a model that generates graphs with the same property. The essential features of the model, which distinguish it from the WS model, are:

Growth: Instead of starting with a fixed number of vertices, the BA model starts with a small graph and adds vertices one at a time.

Preferential attachment: When a new edge is created, it is more likely to connect to a vertex that already has a large number of edges. This "rich get richer" effect is characteristic of the growth patterns of some real-world networks.

Finally, they show that graphs generated by the Barabási-Albert (BA) model model have a distribution of degrees that obeys a power law.

Graphs with this property are sometimes called **scale-free networks**, for reasons I won't explain; if you are curious, you can read more at http://en.wikipedia.org/wiki/Scale-free_network.

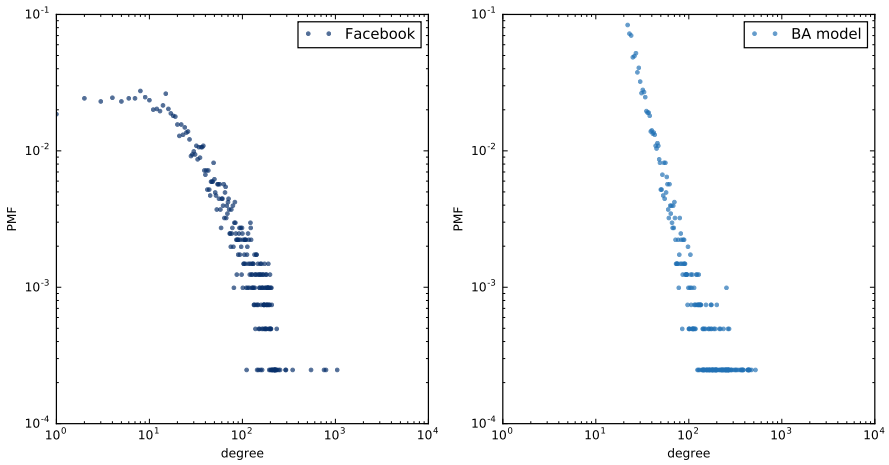

Figure 4.3: PMF of degree in the Facebook dataset and in the BA model, on a log-log scale.

NetworkX provides a function that generates BA graphs. We will use it first; then I'll show you how it works.

```
ba = nx.barabasi_albert_graph(n=4039, k=22)
```

The parameters are `n`, the number of nodes to generate, and `k`, the number of edges each node starts with when it is added to the graph. I chose `k=22` because that is the average number of edges per node in the dataset.

The resulting graph has 4039 nodes and 21.9 edges per node. Since every edge is connected to two nodes, the average degree is 43.8, very close to the average degree in the dataset, 43.7.

And the standard deviation of degree is 40.9, which is a bit less than in the dataset, 52.4, but it is much better than what we got from the WS graph, 1.5.

Figure 4.3 shows the degree distributions for the Facebook network and the BA model on a log-log scale. The model is not perfect; in particular, it deviates from the data when `k` is less than 10. But the tail looks like a straight line, which suggests that this process generates degree distributions that follow a power law.

	Facebook	WS model	BA model
C	0.61	0.63	0.037
L	3.69	3.23	2.51
mean degree	43.7	44	43.7
std degree	52.4	1.5	40.1
power law?	maybe	no	yes

Table 4.1: Characteristics of the Facebook network compared to two models.

So the BA model is better than the WS model at reproducing the degree distribution. But does it have the small world property?

In this example, the average path length, L, is 2.5, which is even more "small world" than the actual network, which has $L = 3.69$. So that's good, although maybe too good.

On the other hand, the clustering coefficient, C, is 0.037, not even close to the value in the dataset, 0.61. So that's a problem.

Table 4.1 summarizes these results. The WS model captures the small world characteristics, but not the degree distribution. The BA model captures the degree distribution, at least approximately, and the average path length, but not the clustering coefficient.

In the exercises at the end of this chapter, you can explore other models intended to capture all of these characteristics.

4.6 Generating BA graphs

In the previous sections we used a NetworkX function to generate BA graphs; now let's see how it works. Here is a version of `barabasi_albert_graph`, with some changes I made to make it easier to read:

```
def barabasi_albert_graph(n, k):

    G = nx.empty_graph(k)
    targets = list(range(k))
    repeated_nodes = []

    for source in range(k, n):

        G.add_edges_from(zip([source]*k, targets))

        repeated_nodes.extend(targets)
        repeated_nodes.extend([source] * k)

        targets = _random_subset(repeated_nodes, k)

    return G
```

The parameters are `n`, the number of nodes we want, and `k`, the number of edges each new node gets (which will turn out to be approximately the number of edges per node).

We start with a graph that has `k` nodes and no edges. Then we initialize two variables:

> The list of `k` nodes that will be connected to the next node. Initially `targets` contains the original `k` nodes; later it will contain a random subset of nodes.
>
> A list of existing nodes where each node appears once for every edge it is connected to. When we select from `repeated_nodes`, the probability of selecting any node is proportional to the number of edges it has.

Each time through the loop, we add edges from the source to each node in `targets`. Then we update `repeated_nodes` by adding each target once and the new node `k` times.

Finally, we choose a subject of the nodes to be targets for the next iteration. Here's the definition of `_random_subset`:

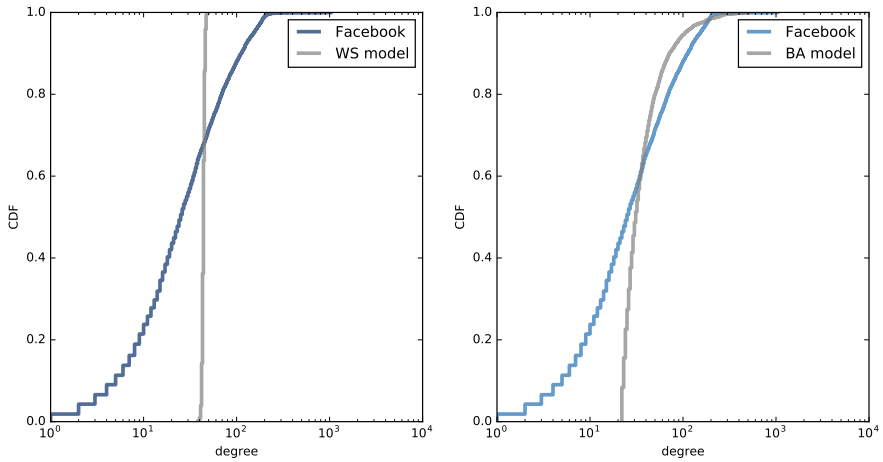

Figure 4.4: CDF of degree in the Facebook dataset along with the WS model (left) and the BA model (right), on a log-x scale.

```
def _random_subset(repeated_nodes, k):
    targets = set()
    while len(targets) < k:
        x = random.choice(repeated_nodes)
        targets.add(x)
    return targets
```

Each time through the loop, `_random_subset` chooses from `repeated_nodes` and adds the chosen node to `targets`. Because `targets` is a set, it automatically discards duplicates, so the loop only exits when we have selected `k` different nodes.

4.7 Cumulative distributions

Figure 4.3 represents the degree distribution by plotting the probability mass function (PMF) on a log-log scale. That's how Barabási and Albert present their results and it is the representation used most often in articles about power law distributions. But it is not the best way to look at data like this.

A better alternative is a **cumulative distribution function** (CDF), which maps from a value, x, to the fraction of values less than or equal to x.

Given a `Pmf`, the simplest way to compute a cumulative probability is to add up the probabilities for values up to and including x:

```
def cumulative_prob(pmf, x):
    ps = [pmf[value] for value in pmf if value<=x]
    return sum(ps)
```

For example, given the degree distribution in the dataset, `pmf_pf`, we can compute the fraction of users with 25 or fewer friends:

```
>>> cumulative_prob(pmf_fb, 25)
0.506
```

The result is close to 0.5, which means that the median number of friends is about 25.

CDFs are better for visualization because they are less noisy than PMFs. Once you get used to intepreting CDFs, they provide a clearer picture of the shape of a distribution than PMFs.

The `thinkstats` module provides a class called `Cdf` that represents a cumulative distribution function. We can use it to compute the CDF of degree in the dataset.

```
from thinkstats2 import Cdf
cdf_fb = Cdf(degrees(fb), label='Facebook')
```

And `thinkplot` provides a function called `Cdf` that plots cumulative distribution functions.

```
thinkplot.Cdf(cdf_fb)
```

Figure **??** shows the degree CDF for the Facebook dataset along with the WS model (left) and the BA model (right). The x-axis is on a log scale.

Clearly the CDF for the WS model is very different from the CDF from the data. The BA model is better, but still not very good, especially for small values.

In the tail of the distribution (values greater than 100) it looks like the BA model matches the dataset well enough, but it is hard to see. We can get

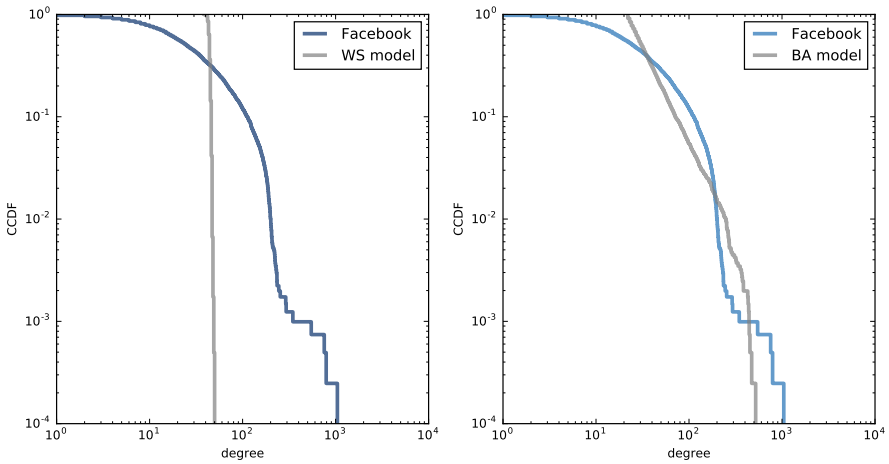

Figure 4.5: Complementary CDF of degree in the Facebook dataset along with the WS model (left) and the BA model (right), on a log-log scale.

a clearer view with one other view of the data: plotting the complementary CDF on a log-log scale.

The **complementary CDF** (CCDF) is defined

$$\text{CCDF}(x) = 1 - \text{CDF}(x)$$

If is useful because if the PMF follows a power law, the CCDF also follows a power law:

$$\text{CCDF}(x) = \left(\frac{x}{x_m}\right)^{-\alpha}$$

where x_m is the minimum possible value and α is a parameter that determines the shape of the distribution.

Taking the log of both sides yields:

$$\log \text{CCDF}(x) = -\alpha(\log x - \log x_m)$$

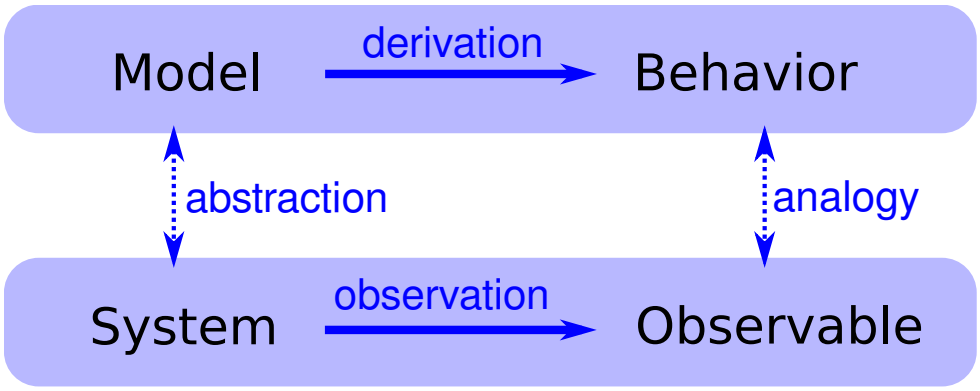

Figure 4.6: The logical structure of an explanatory model.

So if the distribution obeys a power law, we expect the CCDF on a log-log scale to be a straight line with slope $-\alpha$.

Figure 4.5 shows the CCDF of degree for the Facebook data, along with the WS model (left) and the BA model (right), on a log-log scale.

With this way of looking at the data, we can see that the BA model matches the tail of the distribution (values above 20) reasonably well. The WS model does not.

4.8 Explanatory models

We started the discussion of networks with Milgram's Small World Experiment, which shows that path lengths in social networks are surprisingly small; hence, "six degrees of separation".

When we see something surprising, it is natural to ask "Why?" but sometimes it's not clear what kind of answer we are looking for. One kind of answer is an **explanatory model** (see Figure 4.6). The logical structure of an explanatory model is:

> target system, S, we see something observable, O, that warrants explanation.

2. We construct a model, M, that is analogous to the system; that is, there is a correspondence between the elements of the model and the elements of the system.

3. By simulation or mathematical derivation, we show that the model exhibits a behavior, B, that is analogous to O.

4. We conclude that S exhibits O *because* S is similar to M, M exhibits B, and B is similar to O.

At its core, this is an argument by analogy, which says that if two things are similar in some ways, they are likely to be similar in other ways.

Argument by analogy can be useful, and explanatory models can be satisfying, but they do not constitute a proof in the mathematical sense of the word.

Remember that all models leave out, or "abstract away" details that we think are unimportant. For any system there are many possible models that include or ignore different features. And there might be models that exhibit different behaviors, B, B' and B", that are similar to O in different ways. In that case, which model explains O?

The small world phenomenon is an example: the Watts-Strogatz (WS) model and the Barabási-Albert (BA) model both exhibit elements of small world behavior, but they offer different explanations:

- The WS model suggests that social networks are "small" because they include both strongly-connected clusters and "weak ties" that connect clusters (see http://en.wikipedia.org/wiki/Mark_Granovetter#The_strength_of_weak_ties).

- The BA model suggests that social networks are small because they include nodes with high degree that act as hubs, and that hubs grow, over time, due to preferential attachment.

As is often the case in young areas of science, the problem is not that we have no explanations, but too many.

4.9 Exercises

Exercise 4.1 In Section 4.8 we discussed two explanations for the small world phenomenon, "weak ties" and "hubs". Are these explanations compatible; that is, can they both be right? Which do you find more satisfying as an explanation, and why?

Is there data you could collect, or experiments you could perform, that would provide evidence in favor of one model over the other?

Choosing among competing models is the topic of Thomas Kuhn's essay, "Objectivity, Value Judgment, and Theory Choice", which you can read at https://github.com/AllenDowney/ThinkComplexity2/blob/master/papers/kuhn.pdf.

What criteria does Kuhn propose for choosing among competing models? Do these criteria influence your opinion about the WS and BA models? Are there other criteria you think should be considered?

Exercise 4.2 NetworkX provides a function called `powerlaw_cluster_graph` that implements the "Holme and Kim algorithm for growing graphs with powerlaw degree distribution and approximate average clustering". Read the documentation of this function and see if you can use it to generate a graph that has the same number of nodes as the Facebook dataset, the same average degree, and the same clustering coefficient. How does the degree distribution in the model compare to the actual distribution?

Exercise 4.3 Data files from the Barabási and Albert paper are available from http://www3.nd.edu/~networks/resources.htm. Their actor collaboration data is included in the repository for this book in a file named `actor.dat.gz`. The following function reads the file and builds the graph.

```
import gzip

def read_actor_network(filename, n=None):
    G = nx.Graph()
    with gzip.open(filename) as f:
        for i, line in enumerate(f):
            nodes = [int(x) for x in line.split()]
            G.add_edges_from(thinkcomplexity.all_pairs(nodes))
            if n and i >= n:
                break
    return G
```

Compute the number of actors in the graph and the average degree. Plot the PMF of degree on a log-log scale. Also plot the CDF of degree on a log-x scale, to see the general shape of the distribution, and on a log-log scale, to see whether the tail follows a power law.

Note: The actor network is not connected, so you might want to use `nx.connected_component_s` to find connected subsets of the nodes.

Chapter 5

Cellular Automatons

A **cellular automaton** (CA) is a model of a world with very simple physics. "Cellular" means that the world is divided into discrete chunks, called cells. An "automaton" is a machine that performs computations—it could be a real machine, but more often the "machine" is a mathematical abstraction or a computer simulation.

This chapter presents experiments Steven Wolfram performed in the 1980s, showing that some cellular automatons display surprisingly complicated behavior, including the ability to perform arbitrary computations.

I discuss implications of these results, and at the end of the chapter I suggest methods for implementing CAs efficiently in Python.

The code for this chapter is in `chap05.ipynb` in the repository for this book. More information about working with the code is in Section 0.2.

5.1 A simple CA

Cellular automatons[1] are governed by rules that determine how the system evolves in time. Time is divided into discrete steps, and the rules specify how to compute the state of the world during the next time step based on the current state.

[1] You might also see the plural "automata".

As a trivial example, consider a cellular automaton (CA) with a single cell. The state of the cell is an integer represented with the variable x_i, where the subscript i indicates that x_i is the state of the system during time step i. As an initial condition, $x_0 = 0$.

Now all we need is a rule. Arbitrarily, I'll pick $x_i = x_{i-1} + 1$, which says that after each time step, the state of the CA gets incremented by 1. So far, we have a simple CA that performs a simple calculation: it counts.

But this CA is atypical; normally the number of possible states is finite. To bring it into line, I'll choose the smallest interesting number of states, 2, and another simple rule, $x_i = (x_{i-1}+1)\%2$, where % is the remainder (or modulus) operator.

The behavior of this CA is simple: it blinks. That is, the state of the cell switches between 0 and 1 after every time step.

Most CAs are **deterministic**, which means that rules do not have any random elements; given the same initial state, they always produce the same result. There are also nondeterministic CAs, but I will not address them here.

5.2 Wolfram's experiment

The CA in the previous section had only one cell, so we can think of it as zero-dimensional, and it wasn't very interesting. In the rest of this chapter, we explore one-dimensional (1-D) CAs, which turn out to be surprisingly interesting.

To say that a CA has dimensions is to say that the cells are arranged in a contiguous space so that some of them are considered "neighbors". In one dimension, there are three natural configurations:

Finite sequence: A finite number of cells arranged in a row. All cells except the first and last have two neighbors.

Ring: A finite number of cells arranged in a ring. All cells have two neighbors.

Infinite sequence: An infinite number of cells arranged in a row.

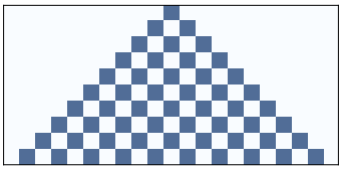

Figure 5.1: Rule 50 after 10 time steps.

The rules that determine how the system evolves in time are based on the notion of a "neighborhood", which is the set of cells that determines the next state of a given cell.

In the early 1980s Stephen Wolfram published a series of papers presenting a systematic study of 1-D CAs. He identified four general categories of behavior, each more interesting than the last.

Wolfram's experiments use a 3-cell neighborhood: the cell itself and its left and right neighbors.

In these experiments, the cells have two states, denoted 0 and 1, so the rules can be summarized by a table that maps from the state of the neighborhood (a tuple of 3 states) to the next state for the center cell. The following table shows an example:

prev	111	110	101	100	011	010	001	000
next	0	0	1	1	0	0	1	0

The first row shows the eight states a neighborhood can be in. The second row shows the state of the center cell during the next time step. As a concise encoding of this table, Wolfram suggested reading the bottom row as a binary number. Because 00110010 in binary is 50 in decimal, Wolfram calls this CA "Rule 50".

Figure 5.1 shows the effect of Rule 50 over 10 time steps. The first row shows the state of the system during the first time step; it starts with one cell "on" and the rest "off". The second row shows the state of the system during the next time step, and so on.

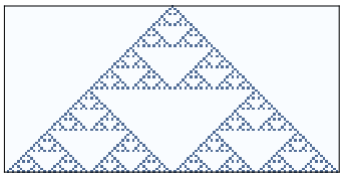

Figure 5.2: Rule 18 after 64 steps.

The triangular shape in the figure is typical of these CAs; is it a consequence of the shape of the neighborhood. In one time step, each cell influences the state of one neighbor in either direction. During the next time step, that influence can propagate one more cell in each direction. So each cell in the past has a "triangle of influence" that includes all of the cells that can be affected by it.

5.3 Classifying CAs

How many different CAs are there?

Since each cell is either on or off, we can specify the state of a cell with a single bit. In a neighborhood with three cells, there are 8 possible configurations, so there are 8 entries in the rule tables. And since each entry contains a single bit, we can specify a table using 8 bits. With 8 bits, we can specify 256 different rules.

One of Wolfram's first experiments with CA was to test all 256 possibilities and try to classify them.

Examining the results visually, he proposed that the behavior of CAs can be grouped into four classes. Class 1 contains the simplest (and least interesting) CAs, the ones that evolve from almost any starting condition to the same uniform pattern. As a trivial example, Rule 0 always generates an empty pattern after one time step.

Rule 50 is an example of Class 2. It generates a simple pattern with nested structure; that is, the pattern contains many smaller versions of itself. Rule

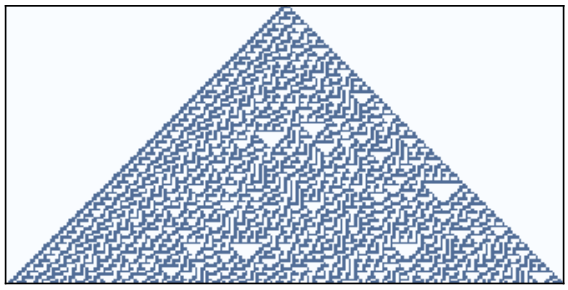

Figure 5.3: Rule 30 after 100 time steps.

18 makes the nested structure even clearer; Figure 5.2 shows what it looks like after 64 steps.

This pattern resembles the Sierpińskitriangle, which you can read about at http://en.wikipedia.org/wiki/Sierpinski_triangle.

Some Class 2 CAs generate patterns that are intricate and pretty, but compared to Classes 3 and 4, they are relatively simple.

5.4 Randomness

Class 3 contains CAs that generate randomness. Rule 30 is an example; Figure 5.3 shows what it looks like after 100 time steps.

Along the left side there is an apparent pattern, and on the right side there are triangles in various sizes, but the center seems quite random. In fact, if you take the center column and treat it as a sequence of bits, it is hard to distinguish from a truly random sequence. It passes many of the statistical tests people use to test whether a sequence of bits is random.

Programs that produce random-seeming numbers are called **pseudo-random number generators** (PRNGs). They are not considered truly random because:

- Many of them produce sequences with regularities that can be detected statistically. For example, the original implementation of `rand` in the C library used a linear congruential generator that yielded sequences with easily detectable serial correlations.

- Any PRNG that uses a finite amount of state (that is, storage) will eventually repeat itself. One of the characteristics of a generator is the **period** of this repetition.

- The underlying process is fundamentally deterministic, unlike some physical processes, like radioactive decay and thermal noise, that are considered to be fundamentally random.

Modern PRNGs produce sequences that are statistically indistinguishable from random, and they can be implemented with with periods so long that the universe will collapse before they repeat. The existence of these generators raises the question of whether there is any real difference between a good quality pseudo-random sequence and a sequence generated by a "truly" random process. In *A New Kind of Science*, Wolfram argues that there is not (pages 315–326).

5.5 Determinism

The existence of Class 3 CAs is surprising. To explain how surprising, let me start with philosophical **determinism** (see http://en.wikipedia.org/wiki/Determinism). Many philosophical stances are hard to define precisely because they come in a variety of flavors. I often find it useful to define them with a list of statements ordered from weak to strong:

D1: Deterministic models can make accurate predictions for some physical systems.

D2: Many physical systems can be modeled by deterministic processes, but some are intrinsically random.

D3: All events are caused by prior events, but many physical systems are nevertheless fundamentally unpredictable.

D4: All events are caused by prior events, and can (at least in principle) be predicted.

My goal in constructing this range is to make D1 so weak that virtually everyone would accept it, D4 so strong that almost no one would accept it, with intermediate statements that some people accept.

The center of mass of world opinion swings along this range in response to historical developments and scientific discoveries. Prior to the scientific revolution, many people regarded the working of the universe as fundamentally unpredictable or controlled by supernatural forces. After the triumphs of Newtonian mechanics, some optimists came to believe something like D4; for example, in 1814 Pierre-Simon Laplace wrote

> We may regard the present state of the universe as the effect of its past and the cause of its future. An intellect which at a certain moment would know all forces that set nature in motion, and all positions of all items of which nature is composed, if this intellect were also vast enough to submit these data to analysis, it would embrace in a single formula the movements of the greatest bodies of the universe and those of the tiniest atom; for such an intellect nothing would be uncertain and the future just like the past would be present before its eyes.

This "intellect" came to be called "Laplace's Demon". See http://en.wikipedia.org/wiki/Laplace's_demon. The word "demon" in this context has the sense of "spirit", with no implication of evil.

Discoveries in the 19th and 20th centuries gradually dismantled Laplace's hope. Thermodynamics, radioactivity, and quantum mechanics posed successive challenges to strong forms of determinism.

In the 1960s chaos theory showed that in some deterministic systems prediction is only possible over short time scales, limited by precision in the measurement of initial conditions.

Most of these systems are continuous in space (if not time) and nonlinear, so the complexity of their behavior is not entirely surprising. Wolfram's demonstration of complex behavior in simple cellular automatons is more surprising—and disturbing, at least to a deterministic world view.

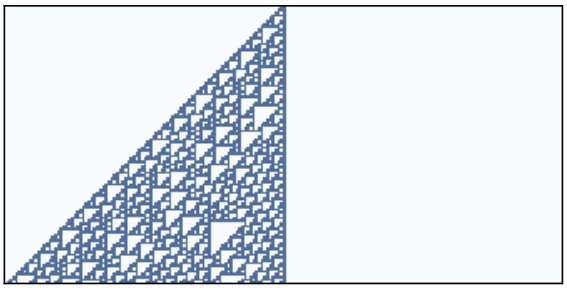

Figure 5.4: Rule 110 after 100 time steps.

So far I have focused on scientific challenges to determinism, but the longest-standing objection is the conflict between determinism and human free will. Complexity science provides a possible resolution of this apparent conflict; I'll come back to this topic in Section 10.4.

5.6 Spaceships

The behavior of Class 4 CAs is even more surprising. Several 1-D CAs, most notably Rule 110, are **Turing complete**, which means that they can compute any computable function. This property, also called **universality**, was proved by Matthew Cook in 1998. See http://en.wikipedia.org/wiki/Rule_110.

Figure 5.4 shows what Rule 110 looks like with an initial condition of a single cell and 100 time steps. At this time scale it is not apparent that anything special is going on. There are some regular patterns but also some features that are hard to characterize.

Figure 5.5 shows a bigger picture, starting with a random initial condition and 600 time steps:

After about 100 steps the background settles into a simple repeating pattern, but there are a number of persistent structures that appear as disturbances in

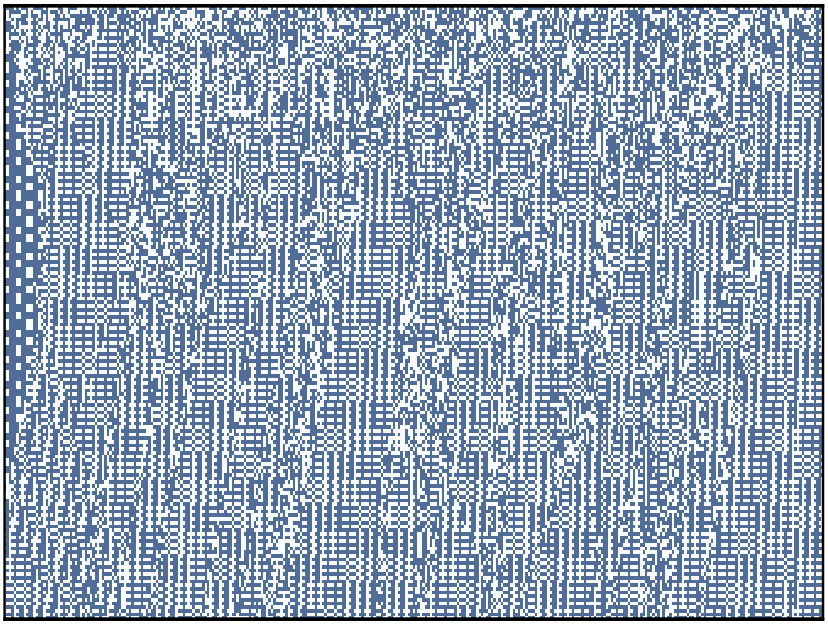

Figure 5.5: Rule 110 with random initial conditions and 600 time steps.

the background. Some of these structures are stable, so they appear as vertical lines. Others translate in space, appearing as diagonals with different slopes, depending on how many time steps they take to shift by one column. These structures are called **spaceships**.

Collisions between spaceships yield different results depending on the types of the spaceships and the phase they are in when they collide. Some collisions annihilate both ships; others leave one ship unchanged; still others yield one or more ships of different types.

These collisions are the basis of computation in a Rule 110 CA. If you think of spaceships as signals that propagate on wires, and collisions as gates that compute logical operations like AND and OR, you can see what it means for a CA to perform a computation.

5.7 Universality

To understand universality, we have to understand computability theory, which is about models of computation and what they compute.

One of the most general models of computation is the Turing machine, which is an abstract computer proposed by Alan Turing in 1936. A Turing machine is a 1-D CA, infinite in both directions, augmented with a read-write head. At any time, the head is positioned over a single cell. It can read the state of that cell (usually there are only two states) and it can write a new value into the cell.

In addition, the machine has a register, which records the state of the machine (one of a finite number of states), and a table of rules. For each machine state and cell state, the table specifies an action. Actions include modifying the cell the head is over and moving one cell to the left or right.

A Turing machine is not a practical design for a computer, but it models common computer architectures. For a given program running on a real computer, it is possible (at least in principle) to construct a Turing machine that performs an equivalent computation.

The Turing machine is useful because it is possible to characterize the set of functions that can be computed by a Turing machine, which is what Turing did. Functions in this set are called Turing computable.

To say that a Turing machine can compute any Turing-computable function is a **tautology**: it is true by definition. But Turing-computability is more interesting than that.

It turns out that just about every reasonable model of computation anyone has come up with is Turing complete; that is, it can compute exactly the same set of functions as the Turing machine. Some of these models, like lamdba calculus, are very different from a Turing machine, so their equivalence is surprising.

This observation led to the Church-Turing Thesis, which is essentially a definition of what it means to be computable. The "thesis" is that Turing-computability is the right, or at least natural, definition of computability, because it describes the power of such a diverse collection of models of computation.

The Rule 110 CA is yet another model of computation, and remarkable for its simplicity. That it, too, turns out to be universal lends support to the Church-Turing Thesis.

In *A New Kind of Science*, Wolfram states a variation of this thesis, which he calls the "principle of computational equivalence:"

> Almost all processes that are not obviously simple can be viewed as computations of equivalent sophistication.
>
> More specifically, the principle of computational equivalence says that systems found in the natural world can perform computations up to a maximal ("universal") level of computational power, and that most systems do in fact attain this maximal level of computational power. Consequently, most systems are computationally equivalent (see mathworld.wolfram.com/PrincipleofComputationalEquivalence.html).

Applying these definitions to CAs, Classes 1 and 2 are "obviously simple". It may be less obvious that Class 3 is simple, but in a way perfect randomness

is as simple as perfect order; complexity happens in between. So Wolfram's claim is that Class 4 behavior is common in the natural world, and that almost all of the systems that manifest it are computationally equivalent.

5.8 Falsifiability

Wolfram holds that his principle is a stronger claim than the Church-Turing Thesis because it is about the natural world rather than abstract models of computation. But saying that natural processes "can be viewed as computations" strikes me as a statement about theory choice more than a hypothesis about the natural world.

Also, with qualifications like "almost" and undefined terms like "obviously simple", his hypothesis may be **unfalsifiable**. Falsifiability is an idea from the philosophy of science, proposed by Karl Popper as a demarcation between scientific hypotheses and pseudoscience. A hypothesis is falsifiable if there is an experiment, at least in the realm of practicality, that would contradict the hypothesis if it were false.

For example, the claim that all life on earth is descended from a common ancestor is falsifiable because it makes specific predictions about similarities in the genetics of modern species (among other things). If we discovered a new species whose DNA was almost entirely different from ours, that would contradict (or at least bring into question) the theory of universal common descent.

On the other hand, "special creation", the claim that all species were created in their current form by a supernatural agent, is unfalsifiable because there is nothing that we could observe about the natural world that would contradict it. Any outcome of any experiment could be attributed to the will of the creator.

Unfalsifiable hypotheses can be appealing because they are impossible to refute. If your goal is never to be proved wrong, you should choose hypotheses that are as unfalsifiable as possible.

But if your goal is to make reliable predictions about the world—and this is at least one of the goals of science—unfalsifiable hypotheses are useless. The

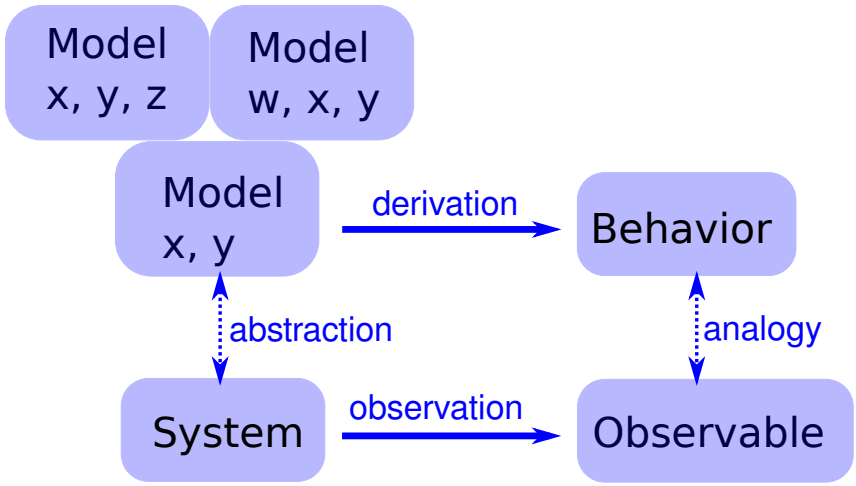

Figure 5.6: The logical structure of a simple physical model.

problem is that they have no consequences (if they had consequences, they would be falsifiable).

For example, if the theory of special creation were true, what good would it do me to know it? It wouldn't tell me anything about the creator except that he has an "inordinate fondness for beetles" (attributed to J. B. S. Haldane). And unlike the theory of common descent, which informs many areas of science and bioengineering, it would be of no use for understanding the world or acting in it.

5.9 What is this a model of?

Some cellular automatons are primarily mathematical artifacts. They are interesting because they are surprising, or useful, or pretty, or because they provide tools for creating new mathematics (like the Church-Turing thesis).

But it is not clear that they are models of physical systems. And if they are, they are highly abstracted, which is to say that they are not very detailed or realistic.

For example, some species of cone snail produce a pattern on their shells that resembles the patterns generated by cellular automatons (seee en.wikipedia.

org/wiki/Cone_snail). So it is natural to suppose that a CA is a model of the mechanism that produces patterns on shells as they grow. But, at least initially, it is not clear how the elements of the model (so-called cells, communication between neighbors, rules) correspond to the elements of a growing snail (real cells, chemical signals, protein interaction networks).

For conventional physical models, being realistic is a virtue, at least up to a point. If the elements of a model correspond to the elements of a physical system, there is an obvious analogy between the model and the system. In general, we expect a model that is more realistic to make better predictions and to provide more believable explanations.

Of course, this is only true up to a point. Models that are more detailed are harder to work with, and usually less amenable to analysis. At some point, a model becomes so complex that it is easier to experiment with the system.

At the other extreme, simple models can be compelling exactly because they are simple.

Simple models offer a different kind of explanation than detailed models. With a detailed model, the argument goes something like this: "We are interested in physical system S, so we construct a detailed model, M, and show by analysis and simulation that M exhibits a behavior, B, that is similar (qualitatively or quantitatively) to an observation of the real system, O. So why does O happen? Because S is similar to M, and B is similar to O, and we can prove that M leads to B."

With simple models we can't claim that S is similar to M, because it isn't. Instead, the argument goes like this: "There is a set of models that share a common set of features. Any model that has these features exhibits behavior B. If we make an observation, O, that resembles B, one way to explain it is to show that the system, S, has the set of features sufficient to produce B."

For this kind of argument, adding more features doesn't help. Making the model more realistic doesn't make the model more reliable; it only obscures the difference between the essential features that cause O and the incidental features that are particular to S.

Figure 5.6 shows the logical structure of this kind of model. The features x and y are sufficient to produce the behavior. Adding more detail, like features

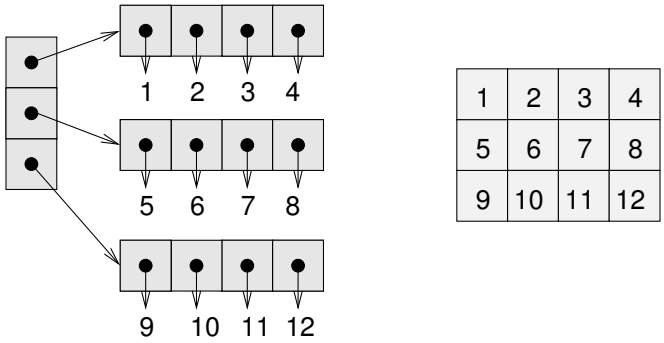

Figure 5.7: A list of lists (left) and a Numpy array (right).

w and z, might make the model more realistic, but that realism adds no explanatory power.

5.10 Implementing CAs

To generate the figures in this chapter, I wrote a Python class called `CA`, which represents a cellular automaton, and classes for plotting the results. In the next few sections I explain how they work.

To store the state of the CA, I use a NumPy array, which is a multi-dimensional data structure whose elements are all the same type. It is similar to a nested list, but usually smaller and faster. Figure 5.7 shows why. The diagram on the left shows a list of lists of integers; each dot represents a reference, which takes up 4–8 bytes. To access one of the integers, you have to follow two references.

The diagram on the right shows an array of the same integers. Because the elements are all the same size, they can be stored contiguously in memory. This arrangement saves space because it doesn't use references, and it saves time because the location of an element can be computed directly from the indices; there is no need to follow a series of references.

To explain how my code works, I'll start with a CA that computes the "parity" of the cells in each neighborhood. The parity of a number is 0 if the number is even and 1 if it is odd.

First I create an array of zeros with a single 1 in the middle of the first row.

```
>>> rows = 5
>>> cols = 11
>>> ca = np.zeros((rows, cols))
>>> ca[0, 5] = 1
print(ca)
[[ 0.  0.  0.  0.  0.  1.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]]
```

`plot_ca` displays the results graphically:

```
import matplotlib.pyplot as plt

def plot_ca(ca, rows, cols):
    cmap = plt.get_cmap('Blues')
    plt.imshow(array, interpolation='none', cmap=cmap)
```

Following convention, I import `pyplot` with the abbreviated name `plt`. `imshow` treats the array as an "image" and displays it. Using the colormap `'Blues'` draws the on cells in dark blue and the off cells in light blue.

Now, to compute the state of the CA during the next time step, we can use `step`:

```
def step(array, i):
    rows, cols = array.shape
    for j in range(1, cols):
        array[i, j] = sum(array[i-1, j-1:j+2]) % 2
```

The parameter `ca` is the NumPy array that represents the state of the CA. `rows` and `cols` are the dimensions of the array, and `i` is the index of the timestep we should compute. I use `i` to indicate rows of the array, which correspond to time, and `j` to indicate columns, which correspond to space.

Inside `step` we loop through the elements of row `i`. Each element is the sum of three elements from the previous row, mod 2.

5.11 Cross-correlation

The `step` function in the previous section is simple, but it is not very fast. In general, we can speed up operations like this if we can replace `for` loops with NumPy operations, because `for` loops incur a lot of overhead in the Python interpreter. In this section I'll show how we can speed up `step` using the NumPy function `correlate`.

First, instead of using a slice operator to select a neighborhood, we can use array multiplication. Specifically, we multiply `array` by a **window** that contains ones for the cells we want to select and zeros for the rest.

For example, the following window selects the first three elements:

```
>>> window = np.zeros(cols, dtype=np.int8)
>>> window[:3] = 1
>>> print(window)
[1 1 1 0 0 0 0 0 0 0]
```

If we multiply by the last row of `array`, we get the first three elements:

```
>>> print(array[4])
>>> print(window * array[4])
[0 1 0 0 0 1 0 0 0 1 0]
[0 1 0 0 0 0 0 0 0 0 0]
```

Now we can use `sum` and the modulus operator to compute the first element of the next row:

```
>>> sum(window * array[4]) % 2
1
```

If we shift the window to the right, it selects the next three elements, and so on. So we can rewrite `step` like this:

```
def step2(array, i):
    rows, cols = array.shape
    window = np.zeros(cols)
    window[:3] = 1
    for j in range(1, cols):
        array[i, j] = sum(window * array[i-1]) % 2
        window = np.roll(window, 1)
```

`roll` shifts the window to the right (it also wraps it around to the beginning, but that doesn't affect this function).

`step2` yields the same results as `step`. It is still not very fast, but it is a step in the right direction because the operation we just performed—multiplying by a window, summing the result, shifting the window, and repeating—is used for a variety of applications. It is called **cross-correlation**, and NumPy provides a function called `correlate` that computes it.

We can use it to write faster, simpler version of `step`:

```
def step3(array, i):
    window = np.array([1, 1, 1])
    array[i] = np.correlate(array[i-1], window, mode='same') % 2
```

When we use `np.correlate`, the window does not have to be the same size as `array`, so making the window is a little simpler.

The `mode` parameter determines the size of the result. You can read the details in the NumPy documentation, but when the mode is `'same'`, the result is the same size as the input.

5.12 CA tables

Now there's just one more step. The function we have so far works if the CA rule only depends on the sum of the neighbors, but most rules also depend on which neighbors are on and off. For example, `100` and `001` might yield different results.

We can make `step` more general using a window with elements `[4, 2, 1]`, which interprets the neighborhood as a binary number. For example, the neighborhood `100` yields 4; `010` yields 2, and `001` yields 1. Then we can take these results and look them up in the rule table.

Here's the more general version of `step`:

```
def step4(array, i):
    window = np.array([4, 2, 1])
    corr = np.correlate(array[i-1], window, mode='same')
    array[i] = table[corr]
```

The first two lines are almost the same. The last line looks up each element from `corr` in `table` and assigns the results to `array[i]`.

Finally, here's the function that computes the table:

```
def make_table(rule):
    rule = np.array([rule], dtype=np.uint8)
    table = np.unpackbits(rule)[::-1]
    return table
```

The parameter, `rule`, is an integer between 0 and 255. The first line put the rule into an array with a single element so we can use `unpackbits`, which converts the rule number to its binary representation. For example, here's the table for Rule 150:

```
>>> table = make_table(150)
>>> print(table)
[0 1 1 0 1 0 0 1]
```

In `thinkcomplexity.py`, you'll find definitions for `CA`, which encapsulates the code in this section, and two classes that draw CAs, `PyplotDrawer` and `EPSDrawer`.

5.13 Exercises

Exercise 5.1 The code for this chapter is in the Jupyter notebook `chap05.ipynb` in the repository for this book. Open this notebook, read the code, and run the cells. You can use this notebook to work on the exercises in this chapter. My solutions are in `chap05soln.ipynb`.

Exercise 5.2 This exercise asks you to experiment with Rule 110 and some of its spaceships.

1. Read the Wikipedia page about Rule 110, which describes its background pattern and spaceships: https://en.wikipedia.org/wiki/Rule_110.

2. Create a Rule 110 CA with an initial condition that yields the stable background pattern.

   Note that the CA class provides `start_string`, which allow you to initialize the state of the array using a string of `1`s and `0`s.

3. Modify the initial condition by adding different patterns in the center of the row and see which ones yield spaceships. You might want to enumerate all possible patterns of n bits, for some reasonable value of n. For each spaceship, can you find the period and rate of translation? What is the biggest spaceship you can find?

4. What happens when spaceships collide?

Exercise 5.3 The goal of this exercise is to implement a Turing machine.

1. Read about Turing machines at http://en.wikipedia.org/wiki/Turing_machine.

2. Write a class called `Turing` that implements a Turing machine. For the action table, use the rules for a 3-state busy beaver.

3. Write a class named `TuringDrawer` that generates an image that represents the state of the tape and the position and state of the head. For one example of what that might look like, see http://mathworld.wolfram.com/TuringMachine.html.

Exercise 5.4 This exercise asks you to implement and test several PRNGs. For testing, you will need to install `DieHarder`, which you can download from https://www.phy.duke.edu/~rgb/General/dieharder.php, or it might be available as a package for your operating system.

1. Write a program that implements one of the linear congruential generators described at http://en.wikipedia.org/wiki/Linear_congruential_generator). Test it using `DieHarder`.

2. Read the documentation of Python's `random` module. What PRNG does it use? Test it.

3. Implement a Rule 30 CA with a few hundred cells, run it for as many time steps as you can in a reasonable amount of time, and output the center column as a sequence of bits. Test it.

Exercise 5.5 Falsifiability is an appealing and useful idea, but among philosophers of science it is not generally accepted as a solution to the demarcation problem, as Popper claimed.

Read http://en.wikipedia.org/wiki/Falsifiability and answer the following questions.

1. What is the demarcation problem?
2. How, according to Popper, does falsifiability solve the demarcation problem?
3. Give an example of two theories, one considered scientific and one considered unscientific, that are successfully distinguished by the criterion of falsifiability.
4. Can you summarize one or more of the objections that philosophers and historians of science have raised to Popper's claim?
5. Do you get the sense that practicing philosophers think highly of Popper's work?

Chapter 6

Game of Life

In this chapter we consider two-dimensional cellular automatons, especially John Conway's Game of Life (GoL). Like some of the 1-D CAs in the previous chapter, GoL follows simple rules and produces surprisingly complicated behavior. And like Wolfram's Rule 110, GoL turns out to be universal; that is, it can compute any computable function, at least in theory.

Complex behavior in GoL raises issues in the philosophy of science, particularly related to scientific realism and instrumentalism. I discuss these issues and suggest additional reading.

At the end of the chapter, I demonstrate ways to implement GoL efficiently in Python.

The code for this chapter is in `chap06.ipynb` in the repository for this book. More information about working with the code is in Section 0.2.

6.1 Conway's GoL

One of the first cellular automatons to be studied, and probably the most popular of all time, is a 2-D CA called "The Game of Life", or GoL for short. It was developed by John H. Conway and popularized in 1970 in Martin Gardner's column in *Scientific American*. See http://en.wikipedia.org/wiki/Conway_Game_of_Life.

The cells in GoL are arranged in a 2-D **grid**, either infinite in both directions or wrapped around. A grid wrapped in both directions is called a **torus** because it is topographically equivalent to the surface of a doughnut. See http://en.wikipedia.org/wiki/Torus.

Each cell has two states—live and dead—and 8 neighbors—north, south, east, west, and the four diagonals. This set of neighbors is sometimes called a "Moore neighborhood".

Like the 1-D CAs in the previous chapters, the Game of Life evolves over time according to rules, which are like simple laws of physics.

In GoL, the next state of each cell depends on its current state and its number of live neighbors. If a cell is alive, it survives if it has 2 or 3 neighbors and dies otherwise. If a cell is dead, it stays dead unless it has exactly 3 neighbors.

The following table summarizes the rules:

Current state	Number of neighbors	Next state
live	2–3	live
live	0–1, 4–8	dead
dead	3	live
dead	0–2, 4–8	dead

This behavior is loosely analogous to real cell growth: cells that are isolated or overcrowded die; at moderate densities they flourish.

GoL is popular because:

- There are simple initial conditions that yield surprisingly complex behavior.
- There are many interesting stable patterns: some oscillate (with various periods) and some move like the spaceships in Wolfram's Rule 110 CA.
- And like Rule 110, GoL is Turing complete.

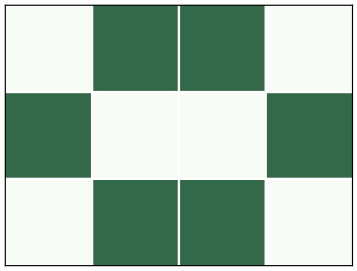

Figure 6.1: A stable pattern called a beehive.

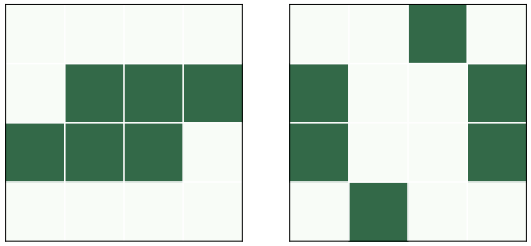

Figure 6.2: An oscillator called a toad.

- Another factor that generated interest was Conway's conjecture—that there is no initial condition that yields unbounded growth in the number of live cells—and the $50 bounty he offered to anyone who could prove or disprove it.
- Finally, the increasing availability of computers made it possible to automate the computation and display the results graphically.

6.2 Life patterns

If you run GoL from a random starting state, a number of stable patterns are likely to appear. Over time, people have identified these patterns and given them names.

For example, Figure 6.1 shows a stable pattern called a "beehive". Every cell

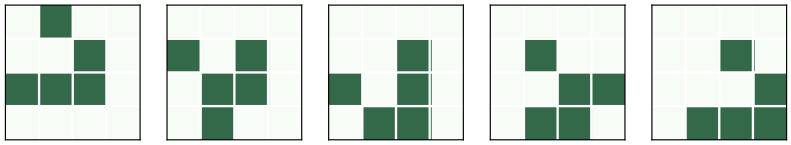

Figure 6.3: A spaceship called a glider.

in the beehive has 2 or 3 neighbors, so they all survive, and none of the dead cells adjacent to the beehive has 3 neighbors, so no new cells are born.

Other patterns "oscillate"; that is, they change over time but eventually return to their starting configuration (provided they don't collide with another pattern). For example, Figure 6.2 shows a pattern called a "toad", which is an oscillator that alternates between two states. The "period" of this oscillator is 2.

Finally, some patterns oscillate and return to the starting configuration, but shifted in space. Because these patterns seem to move, they are called "spaceships".

Figure 6.3 shows a spaceship called a "glider". After a period of 4 steps, the glider is back in the starting configuration, shifted one unit down and to the right.

Depending on the starting orientation, gliders can move along any of the four diagonals. There are other spaceships that move horizontally and vertically.

People have spent embarrassing amounts of time finding and naming these patterns. If you search the web, you will find many collections.

6.3 Conway's conjecture

From most initial conditions, GoL quickly reaches a stable state where the number of live cells is nearly constant (possibly with some oscillation).

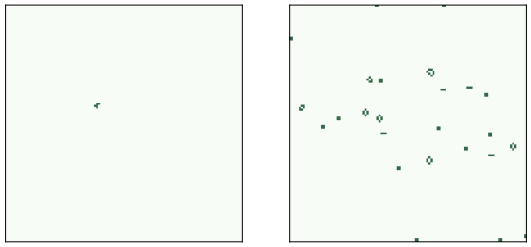

Figure 6.4: Starting and final configurations of the r-pentomino.

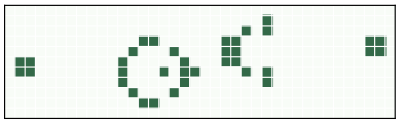

Figure 6.5: Gosper's glider gun, which produces a stream of gliders.

But there are a some simple starting conditions that take a long time to settle down and yield a surprising number of live cells. These patterns are called "Methuselahs" because they are so long-lived.

One of the simplest is the r-pentomino, which has only five cells, roughly in the shape of the letter "r". Figure 6.4 shows the initial configuration of the r-pentomino and the final configuration after 1103 steps.

This configuration is "final" in the sense that all remaining patterns are either stable, oscillators, or gliders that will never collide with another pattern. In total, the r-pentomino yields 6 gliders, 8 blocks, 4 blinkers, 4 beehives, 1 boat, 1 ship, and 1 loaf.

The existence of long-lived patterns prompted Conway to wonder if there are initial patterns that never stabilize. He conjectured that there were not, but he described two kinds of pattern that would prove him wrong, a "gun" and a "puffer train". A gun is a stable pattern that periodically produces a

spaceship—as the stream of spaceships moves out from the source, the number of live cells grows indefinitely. A puffer train is a translating pattern that leaves live cells in its wake.

It turns out that both of these patterns exist. A team led by Bill Gosper discovered the first, a glider gun now called Gosper's Gun, which is shown in Figure 6.5. Gosper also discovered the first puffer train.

There are many patterns of both types, but they are not easy to design or find. That is not a coincidence. Conway chose the rules of GoL so that his conjecture would not be obviously true or false. Of all possible rules for a 2-D CA, most yield simple behavior: most initial conditions stabilize quickly or grow unboundedly. By avoiding uninteresting CAs, Conway was also avoiding Wolfram's Class 1 and Class 2 behavior, and probably Class 3 as well.

If we believe Wolfram's Principle of Computational Equivalence, we expect GoL to be in Class 4, and it is. The Game of Life was proved Turing complete in 1982 (and again, independently, in 1983). Since then, several people have constructed GoL patterns that implement a Turing machine or another machine known to be Turing complete.

6.4 Realism

Stable patterns in GoL are hard not to notice, especially the ones that move. It is natural to think of them as persistent entities, but remember that a CA is made of cells; there is no such thing as a toad or a loaf. Gliders and other spaceships are even less real because they are not even made up of the same cells over time. So these patterns are like constellations of stars. We perceive them because we are good at seeing patterns, or because we have active imaginations, but they are not real.

Right?

Well, not so fast. Many entities that we consider "real" are also persistent patterns of entities at a smaller scale. Hurricanes are just patterns of air flow, but we give them personal names. And people, like gliders, are not made up of the same cells over time. But even if you replace every cell in your body, we consider you the same person.

This is not a new observation—about 2500 years ago Heraclitus pointed out that you can't step in the same river twice—but the entities that appear in the Game of Life are a useful test case for thinking about **philosophical realism**.

In the context of philosophy, realism is the view that entities in the world exist independent of human perception and conception. By "perception" I mean the information that we get from our senses, and by "conception" I mean the mental model we form of the world. For example, our vision systems perceive something like a 2-D projection of a scene, and our brains use that image to construct a 3-D model of the objects in the scene.

Scientific realism pertains to scientific theories and the entities they postulate. A theory postulates an entity if it is expressed in terms of the properties and behavior of the entity. For example, theories about electromagnetism are expressed in terms of electrical and magnetic fields. Some theories about economics are expressed in terms of supply, demand, and market forces. And theories about biology are expressed in terms of genes.

But are these entities real? That is, do they exist in the world independent of us and our theories?

Again, I find it useful to state philosophical positions in a range of strengths; here are four statements of scientific realism with increasing strength:

SR1: Scientific theories are true or false to the degree that they approximate reality, but no theory is exactly true. Some postulated entities may be real, but there is no principled way to say which ones.

SR2: As science advances, our theories become better approximations of reality. At least some postulated entities are known to be real.

SR3: Some theories are exactly true; others are approximately true. Entities postulated by true theories, and some entities in approximate theories, are real.

SR4: A theory is true if it describes reality correctly, and false otherwise. The entities postulated by true theories are real; others are not.

SR4 is so strong that it is probably untenable; by such a strict criterion, almost all current theories are known to be false. Most realists would accept something in the space between SR1 and SR3.

6.5 Instrumentalism

But SR1 is so weak that it verges on **instrumentalism**, which is the view that we can't say whether a theory is true or false because we can't know whether a theory corresponds to reality. Theories are instruments that we use for our purposes; a theory is useful, or not, to the degree that it is fit for its purpose.

To see whether you are comfortable with instrumentalism, consider the following statements:

> "Entities in the Game of Life aren't real; they are just patterns of cells that people have given cute names."
>
> "A hurricane is just a pattern of air flow, but it is a useful description because it allows us to make predictions and communicate about the weather."
>
> "Freudian entities like the Id and the Superego aren't real, but they are useful tools for thinking and communicating about psychology (or at least some people think so)."
>
> "Electric and magnetic fields are postulated entities in our best theories of electromagnetism, but they aren't real. We could construct other theories, without postulating fields, that would be just as useful."
>
> "Many of the things in the world that we identify as objects are arbitrary collections like constellations. For example, a mushroom is just the fruiting body of a fungus, most of which grows underground as a barely-contiguous network of cells. We focus on mushrooms for practical reasons like visibility and edibility."
>
> "Some objects have sharp boundaries, but many are fuzzy. For example, which molecules are part of your body: Air in your lungs? Food in your stomach? Nutrients in your blood? Nutrients in a cell? Water in a cell? Structural parts of a cell? Hair? Dead skin? Dirt? Bacteria on your skin? Bacteria in your gut? Mitochondria? How many of those molecules do you include when you weigh yourself? Conceiving the world in terms of discrete objects is useful, but the entities we identify are not real."

Give yourself one point for each statement you agree with. If you score 4 or more, you might be an instrumentalist!

If you are more comfortable with some of these statements than others, ask yourself why. What are the differences in these scenarios that influence your reaction? Can you make a principled distinction between them?

For more on instrumentalism, see http://en.wikipedia.org/wiki/Instrumentalism.

6.6 Implementing Life

The exercises at the end of this chapter ask you to experiment with and modify the Game of Life, and implement other 2-D cellular automata. This section explains my implementation of GoL, which you can use as a starting place for your experiments.

To represent the state of the cells, I use a NumPy array with type `uint8`, which is an 8-bit unsigned integer. As an example, the following line creates a 10 by 10 array initialized with random values of 0 and 1.

```
a = np.random.randint(2, size=(10, 10)).astype(np.uint8)
```

There are a few ways we can compute the GoL rules. The simplest is to use `for` loops to iterate through the rows and columns of the array:

```
b = np.zeros_like(a)
rows, cols = a.shape
for i in range(1, rows-1):
    for j in range(1, cols-1):
        state = a[i, j]
        neighbors = a[i-1:i+2, j-1:j+2]
        k = np.sum(neighbors) - state
        if state:
            if k==2 or k==3:
                b[i, j] = 1
        else:
            if k == 3:
                b[i, j] = 1
```

Initially, `b` is an array of zeros with the same size as `a`. Each time through the loop, `state` is the condition of the center cell and `neighbors` is the 3x3 neighborhood. `k` is the number of live neighbors (not including the center cell). The nested `if` statements evaluate the GoL rules and turn on cells in `b` accordingly.

This implementation is a straightforward translation of the rules, but it is verbose and slow. We can do better using cross-correlation, as we saw in Section 5.11. There, we used `np.correlate` to compute a 1-D correlation. Now, to perform 2-D correlation, we'll use `correlate2d` from `scipy.signal`, a SciPy module that provides functions related to signal processing:

```
from scipy.signal import correlate2d

kernel = np.array([[1, 1, 1],
                   [1, 0, 1],
                   [1, 1, 1]])

c = correlate2d(a, kernel, mode='same')
```

What we called a "window" in the context of 1-D correlation is called a "kernel" in the context of 2-D correlation, but the idea is the same: `correlate2d` multiplies the kernel and the array to select a neighborhood, then adds up the result. This kernel selects the 8 neighbors that surround the center cell.

`correlate2d` applies the kernel to each location in the array. With `mode='same'`, the result has the same size as `a`.

Now we can use logical operators to compute the rules:

```
b = (c==3) | (c==2) & a
b = b.astype(np.uint8)
```

The first line computes a boolean array with `True` where there should be a live cell and `False` elsewhere. Then `astype` converts the boolean array to an array of integers.

This version is faster, and probably good enough, but we can simplify it slightly by modifying the kernel:

```
kernel = np.array([[1, 1, 1],
                   [1,10, 1],
                   [1, 1, 1]])

c = correlate2d(a, kernel, mode='same')
b = (c==3) | (c==12) | (c==13)
b = b.astype(np.uint8)
```

This version of the kernel includes the center cell and gives it a weight of 10. If the center cell is 0, the result is between 0 and 8; if the center cell is 1, the result is between 10 and 18. Using this kernel, we can simplify the logical operations, selecting only cells with the values 3, 12, and 13.

That might not seem like a big improvement, but it allows one more simplification: with this kernel, we can use a table to look up cell values, as we did in Section 5.12.

```
table = np.zeros(20, dtype=np.uint8)
table[[3, 12, 13]] = 1
c = correlate2d(a, kernel, mode='same')
b = table[c]
```

`table` has zeros everywhere except locations 3, 12, and 13. When we use `c` as an index into `table`, NumPy performs element-wise lookup; that is, it takes each value from `c`, looks it up in `table`, and puts the result into `b`.

This version is faster and more concise than the others; the only drawback is that it takes more explaining.

`Life.py`, which is included in the repository for this book, provides a `Life` class that encapsulates this implementation of the rules. If you run `Life.py`, you should see an animation of a "puffer train", which is a spaceship that leaves a trail of detritus in its wake.

6.7 Exercises

Exercise 6.1 The code for this chapter is in the Jupyter notebook `chap06.ipynb` in the repository for this book. Open this notebook, read the code, and run the cells. You can use this notebook to work on the exercises in this chapter. My solutions are in `chap06soln.ipynb`.

Exercise 6.2 Start GoL in a random state and run it until it stabilizes. What stable patterns can you identify?

Exercise 6.3 Many named patterns are available in portable file formats. Modify `Life.py` to parse one of these formats and initialize the grid.

Exercise 6.4 One of the longest-lived small patterns is "rabbits", which starts with 9 live cells and takes 17 331 steps to stabilize. You can get the initial configuration in various formats from http://www.conwaylife.com/wiki/Rabbits. Load this configuration and run it.

Exercise 6.5 In my implementation, the `Life` class is based on a parent class called `Cell2D`, and `LifeViewer` is based on `Cell2DViewer`. You can use these base classes to implement other 2-D cellular automatons.

For example, one variation of GoL, called "Highlife", has the same rules as GoL, plus one additional rule: a dead cell with 6 neighbors comes to life.

Write a class named `Highlife` that inherits from `Cell2D` and implements this version of the rules. Also write a class named `HighlifeViewer` that inherits from `Cell2DViewer` and try different ways to visualize the results. As a simple example, use a different color map.

One of the more interesting patterns in Highlife is the replicator. Use `add_cells` to initialize Highlife with a replicator and see what it does.

Exercise 6.6 If you generalize the Turing machine to two dimensions, or add a read-write head to a 2-D CA, the result is a cellular automaton called a Turmite. It is named after a termite because of the way the read-write head moves, but spelled wrong as an homage to Alan Turing.

The most famous Turmite is Langton's Ant, discovered by Chris Langton in 1986. See http://en.wikipedia.org/wiki/Langton_ant.

The ant is a read-write head with four states, which you can think of as facing north, south, east or west. The cells have two states, black and white.

The rules are simple. During each time step, the ant checks the color of the cell it is on. If black, the ant turns to the right, changes the cell to white, and moves forward one space. If the cell is white, the ant turns left, changes the cell to black, and moves forward.

Given a simple world, a simple set of rules, and only one moving part, you might expect to see simple behavior—but you should know better by now. Starting with all white cells, Langton's ant moves in a seemingly random pattern for more than 10 000 steps before it enters a cycle with a period of 104 steps. After each cycle, the ant is translated diagonally, so it leaves a trail called the "highway".

Write an implementation of Langton's Ant.

Chapter 7

Physical modeling

The cellular automatons we have seen so far are not physical models; that is, they are not intended to describe systems in the real world. But some CAs are intended as physical models.

In this chapter we consider a CA that models chemicals that diffuse (spread out) and react with each other, which is a process Alan Turing proposed to explain how some animal patterns develop.

And we'll experiment with a CA that models percolation of liquid through porous material, like water through coffee grounds. This model is the first of several models that exhibit **phase change** behavior and **fractal geometry**, and I'll explain what both of those mean.

The code for this chapter is in `chap07.ipynb` in the repository for this book. More information about working with the code is in Section 0.2.

7.1 Diffusion

In 1952 Alan Turing published a paper called "The chemical basis of morphogenesis", which describes the behavior of systems involving two chemicals that diffuse in space and react with each other. He showed that these systems produce a wide range of patterns, depending on the diffusion and reaction rates, and conjectured that systems like this might be an important mechanism in

biological growth processes, particularly the development of animal coloration patterns.

Turing's model is based on differential equations, but it can also be implemented using a cellular automaton.

But before we get to Turing's model, we'll start with something simpler: a diffusion system with just one chemical. We'll use a 2-D CA where the state of each cell is a continuous quantity (usually between 0 and 1) that represents the concentration of the chemical.

We'll model the diffusion process by comparing each cell with the average of its neighbors. If the concentration of the center cell exceeds the neighborhood average, the chemical flows from the center to the neighbors. If the concentration of the center cell is lower, the chemical flows the other way.

The following kernel computes the difference between each cell and the average of its neighbors:

```
    kernel = np.array([[0, 1, 0],
                       [1,-4, 1],
                       [0, 1, 0]])
```

Using `np.correlate2d`, we can apply this kernel to each cell in an array:

```
    c = correlate2d(array, kernel, mode='same')
```

We'll use a diffusion constant, `r`, that relates the difference in concentration to the rate of flow:

```
    array += r * c
```

Figure 7.1 show results for a CA with `n=9`, `r=0.1` and initial concentration 0 everywhere except for an "island" in the middle. The figure shows the starting configuration and the state of the CA after 5 and 10 steps. The chemical spreads from the center outward, continuing until the concentration is the same everywhere.

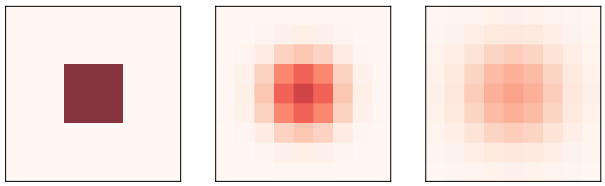

Figure 7.1: A simple diffusion model after 0, 5, and 10 steps.

7.2 Reaction-diffusion

Now let's add a second chemical. I'll define a new object, `ReactionDiffusion`, that contains two arrays, one for each chemical:

```
class ReactionDiffusion(Cell2D):

    def __init__(self, n, m, params):
        self.params = params
        self.array = np.ones((n, m), dtype=float)
        self.array2 = np.zeros((n, m), dtype=float)
        island(self.array2, val=0.1, noise=0.1)
```

`n` and `m` are the number of rows and columns in the array. `params` is a tuple of parameters, which I explain below.

`array` represents the concentration of the first chemical, `A`, which is initially 1 everywhere.

`array2` represents the concentration of `B`, which is initially 0 everywhere except an island in the middle, which is initialized by `island`:

```
def island(a, val, noise):
    n, m = a.shape
    r = min(n, m) // 20
    a[n//2-r:n//2+r, m//2-r:m//2+r] = val
    a += noise * np.random.random((n, m))
```

The radius of the island, `r`, is one twentieth of `n` or `m`, whichever is smaller. The height of the island is `val`, which is 0.1 in the example. Also, random uniform noise, with values from 0 to `noise`, is added to the entire array.

And here is the `step` function that updates the arrays:

```
    def step(self):
        """Executes one time step."""
        A = self.array
        B = self.array2
        ra, rb, f, k = self.params

        cA = correlate2d(A, self.kernel, **self.options)
        cB = correlate2d(B, self.kernel, **self.options)

        reaction = A * B**2
        self.array += ra * cA - reaction + f * (1-A)
        self.array2 += rb * cB + reaction - (f+k) * B
```

The parameters are

> The diffusion rate of `A` (analogous to `r` in the previous section).
>
> The diffusion rate of `B`. In most versions of this model, `rb` is about half of `ra`.
>
> The "feed" rate, which controls how quickly `A` is added to the system.
>
> The "kill" rate, which controls how quickly `B` is removed from the system.

Now let's look more closely at the update statements:

```
        reaction = A * B**2
        self.array += ra * cA - reaction + f * (1-A)
        self.array2 += rb * cB + reaction - (f+k) * B
```

The arrays `cA` and `cB` are the result of applying a diffusion kernel to `A` and `B`. Multiplying by `ra` and `rb` yields the rate of diffusion into or out of each cell.

The term `A * B**2` represents the rate that `A` and `B` react with each other. Assuming that the reaction consumes `A` and produces `B`, we subtract this term in the first equation and add it in the second.

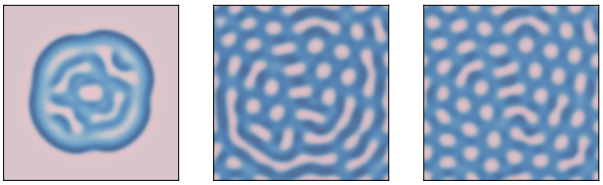

Figure 7.2: Reaction-diffusion model with parameters `f=0.035` and `k=0.057` after 1000, 2000, and 4000 steps.

The term `f * (1-A)` determines the rate that `A` is added to the system. Where `A` is near 0, the maximum feed rate is `f`. Where `A` approaches 1, the feed rate drops off to zero.

Finally, the term `(f+k) * B` determines the rate the `B` is removed from the system. As `B` approaches 0, this rate goes to zero.

As long as the rate parameters are not too high, the values of `A` and `B` usually stay between 0 and 1.

With different parameters, this model can produce patterns similar to the stripes and spots on a variety of animals. In some cases, the similarity is striking, especially when the feed and kill parameters vary in space.

For all simulations in this section, `ra=0.5` and `rb=0.25`.

Figure 7.2 shows results with `f=0.035` and `k=0.057`, with the concentration of `B` shown in darker colors. With these parameters, the system evolves toward a stable configuration with light spots of `A` on a dark background of `B`.

Figure 7.3 shows results with `f=0.055` and `k=0.062` which yields a coral-like pattern of `B` on a background of `A`.

Figure 7.4 shows results with `f=0.039` and `k=0.065`. These parameters produce spots of `B` that grow and divide in a process that resembles mitosis, ending with a stable pattern of equally-spaced spots.

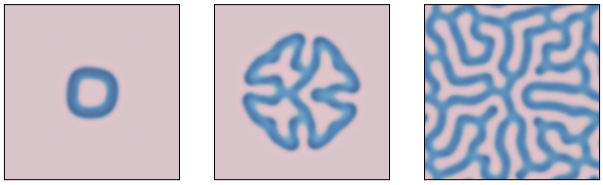

Figure 7.3: Reaction-diffusion model with parameters `f=0.055` and `k=0.062` after 1000, 2000, and 4000 steps.

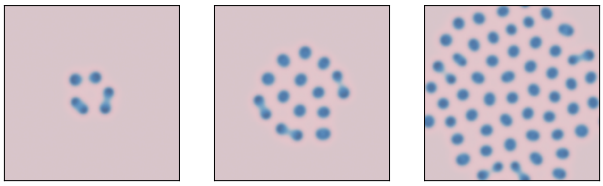

Figure 7.4: A reaction-diffusion model with parameters `f=0.039` and `k=0.065` after 1000, 2000, and 4000 steps.

Since 1952, observations and experiments have provides some support for Turing's conjecture. At this point it seems likely, but not yet proven, that many animal patterns are actually formed by reaction-diffusion processes of some kind.

7.3 Percolation

Percolation is a process in which a fluid flows through a semi-porous material. Examples include oil in rock formations, water in paper, and hydrogen gas in micropores. Percolation models are also used to study systems that are not literally percolation, including epidemics and networks of electrical resistors. See http://en.wikipedia.org/wiki/Percolation_theory.

Percolation models are often represented using random graphs like the ones we saw in Chapter 2, but they can also be represented using cellular automatons. In the next few sections we'll explore a 2-D CA that simulates percolation.

In this model:

- Initially, each cell is either "porous" with probability `p` or "non-porous", and all cells are considered "dry" except the top row, which is "wet".
- During each time step, if a porous cell has at least one wet neighbor, it becomes wet. Non-porous cells stay dry.
- The simulation runs until it reaches a "fixed point" where no more cells change state.

If there is a path of wet cells from the top to the bottom row, we say that the CA has a "percolating cluster".

One of the primary questions of interest regarding percolation is the probability of finding a percolating cluster and how it depends on `p`. This question might remind you of Section 2.3, where we computed the probability that a random Erdős-Rényi graph is connected. We will see several connections between that model and this one.

I define a new class to represent a percolation model:

```
class Percolation(Cell2D):

    def __init__(self, n, m, p):
        self.p = p
        self.array = np.random.choice([0, 1], (n, m), p=[1-p, p])
        self.array[0] = 5
```

`n` and `m` are the number of rows and columns in the CA. `p` is the probability that a cell is porous.

The state of the CA is stored in `array`, which is initialized using `np.random.choice` to choose 1 (porous) with probability `p`, and 0 (non-porous) with probability `1-p`. The state of the top row is set to 5, which represents a wet cell.

During each time step, we check whether any porous cell has a wet neighbor, using a 4-cell neighborhood (not including the diagonals). Here is the kernel:

```
    kernel = np.array([[0, 1, 0],
                       [1, 0, 1],
                       [0, 1, 0]])
```

And here is the `step` function:

```
    def step(self):
        a = self.array
        c = correlate2d(a, self.kernel, mode='same')
        self.array[(a==1) & (c>=5)] = 5
```

`correlate2d` adds up the state of the neighbors, which can only exceed 5 if at least one neighbor is wet. The last line finds cells that are porous, `a==1`, with at least one wet neighbor, `c>=5`, and sets their state to 5, which is wet.

Figure 7.5 shows the first few steps of a percolation model with `n=10` and `p=0.5`. Non-porous cells are white, porous cells are lightly shaded, and wet cells are dark.

7.4 Phase change

Now let's test whether a CA contains a percolating cluster:

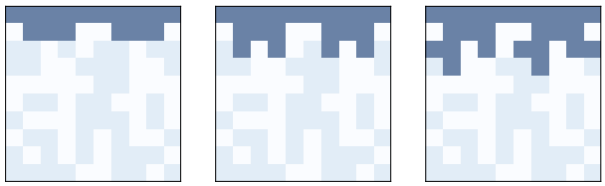

Figure 7.5: The first three steps of a percolation model with `n=10` and `p=0.5`.

```
def test_perc(perc):
    num_wet = perc.num_wet()

    num_steps = 0
    while True:
        perc.step()
        num_steps += 1

        if perc.bottom_row_wet():
            return True, num_steps

        new_num_wet = perc.num_wet()
        if new_num_wet == num_wet:
            return False, num_steps

        num_wet = new_num_wet
```

`test_perc` takes a `Percolation` object as a parameter. Each time through the loop, it advances the CA one time step. It checks the bottom row to see if any cells are wet; if so, it returns `True`, to indicate that there is a percolating cluster, and `num_steps`, which is the number of time steps it took to get to the bottom.

During each time step, it also computes the number of wet cells and checks whether the number increased since the last step. If not, we have reached a fixed point without finding a percolating cluster, so we return `False`.

To estimate the probability of a percolating cluster, we generate many random initial configurations and test them:

```
def estimate_prob_percolating(p=0.5, n=100, iters=100):
    count = 0
    for i in range(iters):
        perc = Percolation(n, p=p)
        flag, _ = test_perc(perc)
        if flag:
            count += 1

    return count / iters
```

`estimate_prob_percolating` makes 100 CAs with the given values of `p` and `n` and calls `test_perc` to see how many of them have a percolating cluster. The return value is the fraction of CAs that do.

When `p=0.55`, the probability of a percolating cluster is near 0. At `p=0.60`, it is about 70%, and at `p=0.65` it is near 1. This rapid transition indicates that there is a critical value of `p` near 0.6.

We can estimate the critical value more precisely using a random walk. Starting from an initial value of `p`, we construct a `Percolation` object and check whether it has a percolating cluster. If so, `p` is probably too high, so we decrease it. If not, `p` is probably too low, so we increase it.

Here's the code:

```
def find_critical(p=0.6, n=100, iters=100):
    ps = [p]
    for i in range(iters):
        perc = Percolation(n=n, p=p)
        flag, _ = test_perc(perc)
        if flag:
            p -= 0.005
        else:
            p += 0.005
        ps.append(p)
    return ps
```

`find_critical` starts with the given value of `p` and adjusts up and down, returning the list of values. With `n=100` the average of `ps` is about 0.59, and this critical value seems to be the same for values of `n` from 50 to 400.

The rapid change in behavior near the critical value is called a **phase change** by analogy with phase changes in physical systems, like the way water changes from liquid to solid at its freezing point.

A wide variety of systems display a common set of behaviors and characteristics when they are at or near a critical point. These behaviors are known collectively as **critical phenomena**. In the next section, we explore one of them: fractal geometry.

7.5 Fractals

To understand fractals, we have to start with dimensions.

For simple geometric objects, dimension is defined in terms of scaling behavior. For example, if the side of a square has length l, its area is l^2. The exponent, 2, indicates that a square is 2-dimensional. Similarly, if the side of a cube has length l, its volume is l^3, which indicates that a cube is 3-dimensional.

More generally, we can estimate the dimension of an object by measuring its "size" (by some definition) as a function of a linear measure.

As an example, I'll estimate the dimension of a 1-D cellular automaton by measuring its area (total number of "on" cells) as a function of the number of rows.

Figure 7.6 shows three 1-D CAs like the ones we saw in Section 5.2. Rule 20 (left) generates a set of cells that seems linear, so we expect it to be one-dimensional. Rule 50 (center) produces something like a triangle, so we expect it to be 2-D. Rule 18 (right) also produces something like a triangle, but the density is not uniform, so its scaling behavior is not obvious.

I'll estimate the dimension of these CAs with the following function, which counts the number of on cells after each time step. It returns a list of tuples, where each tuple contains i and i^2, for purposes of comparison, and the total number of cells.

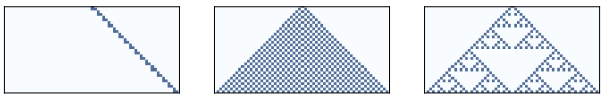

Figure 7.6: One-dimensional CAs with rules 20, 50, and 18, after 32 time steps.

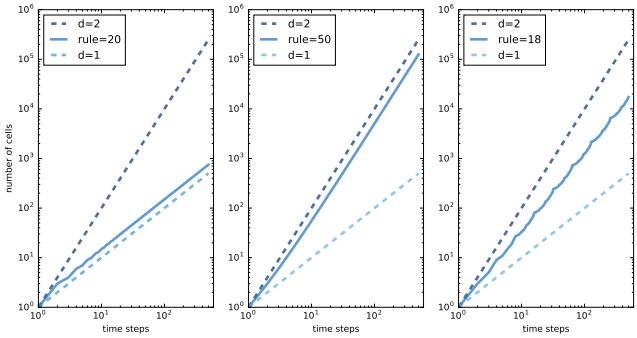

Figure 7.7: Number of "on" cells versus number of time steps for rules 20, 50, and 18.

```
def count_cells(rule, n=500):
    ca = Cell1D(rule, n)
    ca.start_single()

    res = []
    for i in range(1, n):
        cells = np.sum(ca.array)
        res.append((i, i**2, cells))
        ca.step()

    return res
```

Figure 7.7 shows the results plotted on a log-log scale.

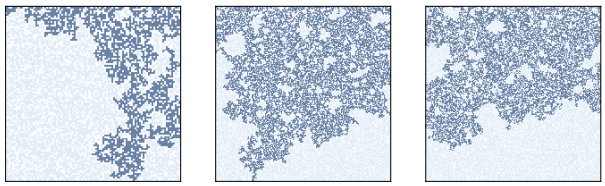

Figure 7.8: Percolation models with `p=0.6` and `n=100`, `200`, and `300`.

In each figure, the top dashed line shows $y = i^2$. Taking the log of both sides, we have $\log y = 2 \log i$. Since the figure is on a log-log scale, the slope of this line is 2.

Similarly, the bottom dashed line shows $y = i$. On a log-log scale, the slope of this line is 1.

Rule 20 (left) produces 3 cells every 2 time steps, so the total number of cells after i steps is $y = 1.5i$. Taking the log of both sides, we have $\log y = \log 1.5 + \log i$, so on a log-log scale, we expect a line with slope 1. In fact, the estimated slope of the line is 1.01.

Rule 50 (center) produces $i + 1$ new cells during the ith time step, so the total number of cells after i steps is $y = i^2 + i$. If we ignore the second term and take the log of both sides, we have $\log y \sim 2 \log i$, so as i gets large, we expect to see a line with slope 2. In fact, the estimated slope is 1.97.

Finally, for Rule 18 (right), the estimated slope is about 1.57, which is clearly not 1, 2, or any other integer. This suggests that the pattern generated by Rule 18 has a "fractional dimension"; that is, it is a fractal.

7.6 Fractals and Percolation Models

Now let's get back to percolation models. Figure **??** shows clusters of wet cells in percolation simulations with `p=0.6` and `n=100`, `200`, and `300`. Informally, they resemble fractal patterns seen in nature and in mathematical models.

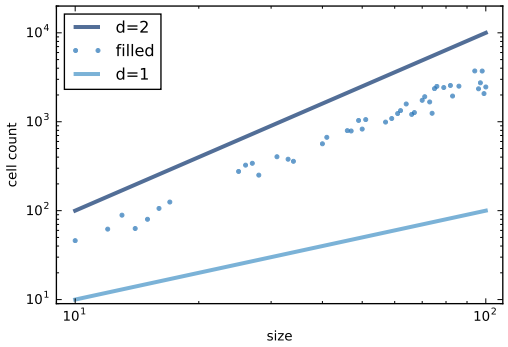

Figure 7.9: Number of cells in the percolating cluster versus CA size.

To estimate their fractal dimension, we can run CAs with a range of sizes, count the number of wet cells in each percolating cluster, and then see how the cell counts scale as we increase the size of the CA.

The following loop runs the simulations:

```
    for size in sizes:
        perc = Percolation(size, p=p)
        flag, _ = test_perc(perc)
        if flag:
            num_filled = perc.num_wet() - size
            res.append((size, size**2, num_filled))
```

The result is a list of tuples where each tuple contains `size` and `size**2`, for comparison, and the number of cells in the percolating cluster (not including the initial wet cells in the top row).

Figure 7.9 shows the results for a range of sizes from 10 to 100. The dots show the number of cells in each percolating cluster. The slope of a line fitted to these dots is about 1.85, which indicates that the percolating cluster is, in fact, fractal when `p` is near the critical value.

When `p` is larger than the critical value, nearly every porous cell gets filled, so the number of wet cells is just `p * size^2`, which has dimension 2.

When `p` is substantially smaller than the critical value, the number of wet cells is proportional to the linear size of the CA, so it has dimension 1.

7.7 Exercises

Exercise 7.1 In Section 7.6 we showed that the Rule 18 CA produces a fractal. Can you find other 1-D CAs that produce fractals?

Note: the `Cell1D` object in `Cell1D.py` does not wrap around from the left edge to the right, which creates artifacts at the boundaries for some rules. You might want to use `Wrap1D`, which is a child class of `Cell1D` that wraps around. It is also defined in `Cell1D.py`.

Exercise 7.2 In 1990 Bak, Chen and Tang proposed a cellular automaton that is an abstract model of a forest fire. Each cell is in one of three states: empty, occupied by a tree, or on fire.

The rules of the CA are:

1. An empty cell becomes occupied with probability p.
2. A cell with a tree burns if any of its neighbors is on fire.
3. A cell with a tree spontaneously burns, with probability f, even if none of its neighbors is on fire.
4. A cell with a burning tree becomes an empty cell in the next time step.

Write a program that implements this model. You might want to inherit from `Cell2D`. Typical values for the parameters are $p = 0.01$ and $f = 0.001$, but you might want to experiment with other values.

Starting from a random initial condition, run the CA until it reaches a steady state where the number of trees no longer increases or decreases consistently.

In steady state, is the geometry of the forest fractal? What is its fractal dimension?

Chapter 8

Self-organized criticality

In the previous chapter we saw an example of a system with a critical point and we explored one of the common properties of critical systems, fractal geometry.

In this chapter, we explore two other properties of critical systems: heavy-tailed distributions, which we saw in Chapter 4.4 and pink noise, which I'll explain in this chapter.

These properties are interesting in part because they appear frequently in nature; that is, many natural systems produce fractal-like geometry, heavy-tailed distributions, and pink noise.

This observation raises a natural question: why do so many natural systems have properties of critical systems? A possible answer is **self-organized criticality** (SOC), which is the tendency of some systems to evolve toward and stay in a critical state.

In this chapter I'll present a **sand pile model**, which was the first system shown to exhibit SOC.

The code for this chapter is in `chap08.ipynb` in the repository for this book. More information about working with the code is in Section 0.2.

8.1 Critical Systems

Many critical systems demonstrate common behaviors:

- Fractal geometry: For example, freezing water tends to form fractal patterns, including snowflakes and other crystal structures. Fractals are characterized by self-similarity; that is, parts of the pattern are similar to scaled copies of the whole.
- Heavy-tailed distributions of some physical quantities: For example, in freezing water the distribution of crystal sizes is characterized by a power law.
- Variations in time that exhibit **pink noise**. Complex signals can be decomposed into their frequency components. In pink noise, low-frequency components have more power than high-frequency components. Specifically, the power at frequency f is proportional to $1/f$.

Critical systems are usually unstable. For example, to keep water in a partially frozen state requires active control of the temperature. If the system is near the critical temperature, a small deviation tends to move the system into one phase or the other.

Many natural systems exhibit characteristic behaviors of criticality, but if critical points are unstable, they should not be common in nature. This is the puzzle Bak, Tang and Wiesenfeld address. Their solution is called self-organized criticality (SOC), where "self-organized" means that from any initial condition, the system moves toward a critical state, and stays there, without external control.

8.2 Sand Piles

The sand pile model was proposed by Bak, Tang and Wiesenfeld in 1987. It is not meant to be a realistic model of a sand pile, but rather an abstraction that models physical systems with (1) a large number of elements that (2) interact with their neighbors.

The sand pile model is a 2-D cellular automaton where the state of each cell represents the slope of a part of a sand pile. During each time step, each cell is checked to see whether it exceeds a critical value, K, which is usually 3. If so, it "topples" and transfers sand to four neighboring cells; that is, the slope of the cell is decreased by 4, and each of the neighbors is increased by 1. At

the perimeter of the grid, all cells are kept at slope 0, so the excess spills over the edge.

Bak, Tang and Wiesenfeld initialize all cells a level level greater than `K` and run the model until it stabilizes. Then they observe the effect of small perturbations; they choose a cell at random, increment its value by 1, and run the model, again, until it stabilizes.

For each perturbation, they measure `T`, the number of time steps the pile takes to stabilize, and `S`, the total number of cells that topple[1].

Most of the time, dropping an single grain causes no cells to topple, so `T=1` and `S=0`. But occasionally a single grain can cause an **avalanche** that affects a substantial fraction of the grid. The distributions of `T` and `S` turn out to be heavy-tailed, which supports the claim that the system is in a critical state.

They conclude that the sand pile model exhibits "self-organized criticality", which means that from the initial condition it evolves toward a critical state without the need for external control or what they call "fine tuning" of any parameters. And the model stays in a critical state as more grains are added.

In the next few sections I replicate their experiments and interpret the results.

8.3 Implementing the Sand Pile

To implement the sand pile model, I define a class called `SandPile` that inherits from `Cell2D`, which is defined in `Cell2D.py`.

```
class SandPile(Cell2D):

    def __init__(self, n, m, level=9):
        self.array = np.ones((n, m)) * level
```

All values in the array are initialized to `level`, which is generally greater than the toppling threshold, `K`.

Here's the `step` method that finds all cells above `K` and topples them:

[1]The original paper uses a different definition of `S`, but most later work uses this definition.

```
    kernel = np.array([[0, 1, 0],
                       [1,-4, 1],
                       [0, 1, 0]], dtype=np.int32)

    def step(self, K=3):
        toppling = self.array > K
        c = correlate2d(toppling, self.kernel, mode='same')
        self.array += c
        return num_toppled
```

To explain how that works, I'll start with a small pile with just two cells ready to topple:

```
pile = SandPile(n=3, m=5, level=0)
pile.array[1, 1] = 4
pile.array[1, 3] = 4
```

Initially, `pile.array` looks like this:

```
[[0 0 0 0 0]
 [0 4 0 4 0]
 [0 0 0 0 0]]
```

Now we can select the cells that are above the toppling threshold:

```
toppling = pile.array > K
```

The result is an array of booleans, but we can use it as if it were an array of integers like this:

```
[[0 0 0 0 0]
 [0 1 0 1 0]
 [0 0 0 0 0]]
```

If we correlate this array with the kernel, it makes copies of the kernel at each location where `toppling` is 1.

```
c = correlate2d(toppling, kernel, mode='same')
```

And here's the result:

```
[[ 0  1  0  1  0]
 [ 1 -4  2 -4  1]
 [ 0  1  0  1  0]]
```

Notice that where the copies of the kernel overlap, they add up.

This array contains the change for each cell, which we use to update the original array:

```
pile.array += c
```

And here's the result.

```
[[0 1 0 1 0]
 [1 0 2 0 1]
 [0 1 0 1 0]]
```

So that's how `step` works.

By default, `correlate2d` considers the boundary of the array to be fixed at zero, so any grains of sand that go over the edge disappear.

`SandPile` also provides `run`, which calls step until no more cells topple:

```
    def run(self):
        total = 0
        for i in itertools.count(1):
            num_toppled = self.step()
            total += num_toppled
            if num_toppled == 0:
                return i, total
```

The return value is a tuple that contains the number of time steps and the total number of cells that toppled.

If you are not familiar with `itertools.count`, it is an infinite generator that counts up from the given initial value, so the `for` loop runs until `step` returns 0.

Finally, the `drop` method chooses a random cell and adds a grain of sand:

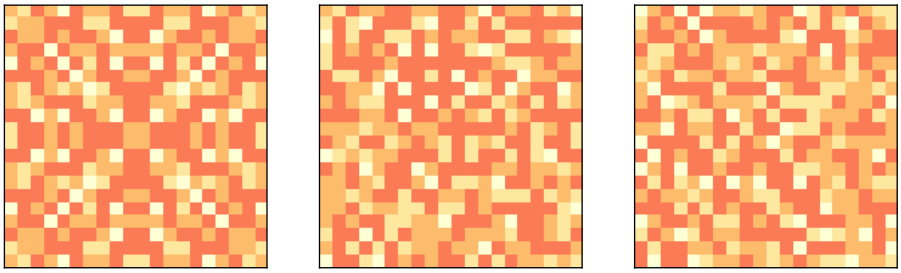

Figure 8.1:

```
    def drop(self):
        a = self.array
        n, m = a.shape
        index = np.random.randint(n), np.random.randint(m)
        a[index] += 1
```

Let's look at a bigger example, with `n=20`:

```
pile = SandPile(n=20, level=10)
pile.run()
```

With an initial level of `10`, this sand pile takes 332 time steps to reach equilibrium, with a total of 53,336 topplings. Figure 8.1 (left) show the configuration, after this initial run. Notice that it has the repeating elements that are characteristic of fractals. We'll come back to that soon.

Figure 8.1 (middle) shows the configuration of the sand pile after dropping 200 grains onto random cells, each time running until the pile reaches equilibrium. The symmetry of the initial configuration has been broken; the configuration looks random.

Finally Figure 8.1 (right) shows the configuration after 400 drops. It looks similar to the configuration after 200 drops. In fact, the pile is now in a steady

state where its statistical properties don't change over time. I'll explain some of those statistical properties in the next section.

8.4 Heavy-tailed distributions

If the sand pile model is in a critical state, we expect to find heavy-tailed distributions for quantities like the duration and size of avalanches. So let's take a look.

I'll make a larger sand pile, with `n=50`, an initial level of `30`, and run until equilibrium:

```
pile2 = SandPile(n=50, level=30)
pile2.run()
```

Next, I'll run 100,000 random drops

```
iters = 100000
res = [pile2.drop_and_run() for _ in range(iters)]
```

As the name suggests, `drop_and_run` calls `drop` and `run` and returns the duration of the avalanche and total number of cells that toppled.

So `res` is a list of `(T, S)` tuples, where `T` is duration, in time steps, and `S` is cells toppled. We can use `np.transpose` to unpack `res` into two NumPy arrays:

```
T, S = np.transpose(res)
```

A large majority of drops have duration 1 and no toppled cells, so we'll filter those out.

```
T = T[T>1]
S = S[S>0]
```

The distributions of `T` and `S` have many small values and a few very large ones. I'll use the `Hist` class from `thinkstats2` to make a histogram of the values; that is, a map from each value to the number of times it occurs.

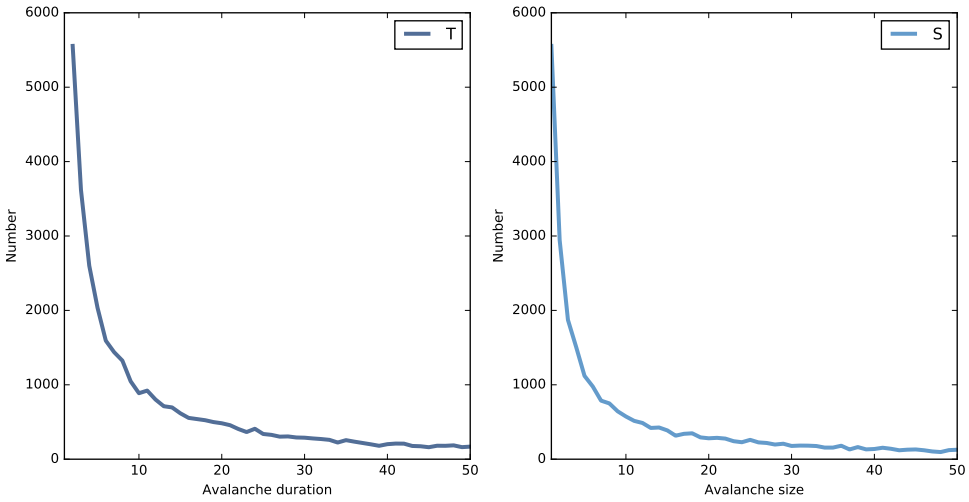

Figure 8.2:

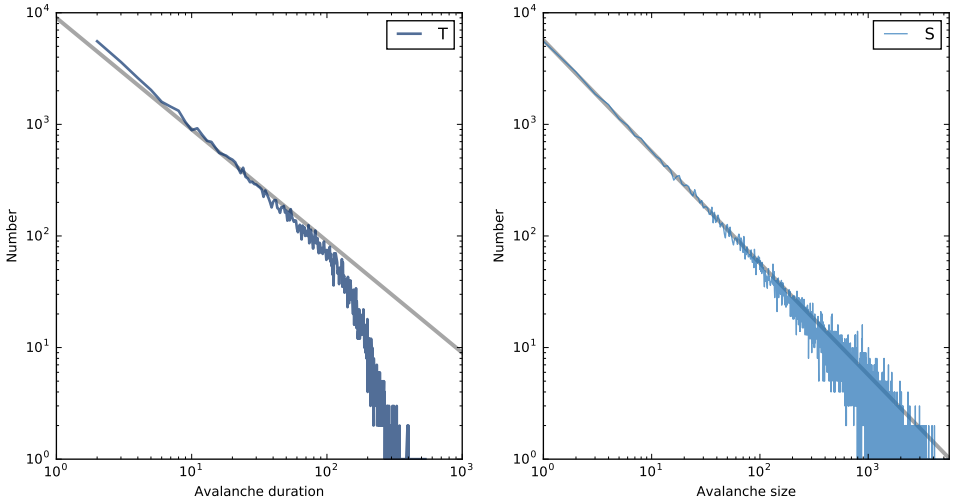

Figure 8.3:

```
from thinkstats2 import Hist

histT = Hist(T)
histS = Hist(S)
```

Figure 8.2 shows the results for values less than 50. But as we saw in Section 4.4, we can get a clearer picture of these distributions by plotting them on a log-log scale, as shown in Figure 8.3.

For values between 1 and 100, the distributions are nearly straight on a log-log scale, which is indicative of a heavy tail. The gray lines in the figure have slope -1, which suggests that these distributions follow a power law with parameter $\alpha = 1$.

For values greater than 100, the distributions fall away more quickly than the power law model, which means there are fewer very large values than the model predicts. One explanation is that this effect is due to the finite size of the sand pile, so we might expect larger piles to fit the power law better.

Another possibility, which you can explore in one of the exercises at the end of this chapter, is that these distributions do not strictly obey a power law. But even if they are not power-law distributions, they are still heavy-tailed as we expect for a system in a critical state.

8.5 Fractals

Another property of critical systems is fractal geometry. The initial configuration in Figure 8.1 (left) resembles a fractal, but you can't always tell by looking. A more reliable way to identify a fractal is to estimate its fractal dimension, as we saw in Section 7.5 and Section 7.6.

I'll start by making a bigger sand pile, with `n=131` and initial level `22`.

```
pile3 = SandPile(n=131, level=22)
pile3.run()
```

By the way, it takes 28,379 steps for this pile to reach equilibrium, with more than 200 million cells toppled.

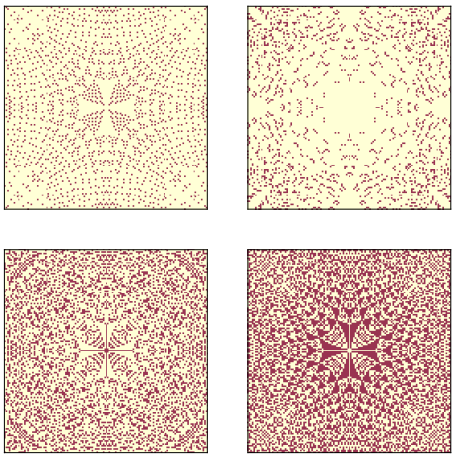

Figure 8.4:

To see the resulting pattern more clearly, I select cells with levels 0, 1, 2, and 3, and plot them separately:

```
def draw_four(viewer, vals=range(4)):
    thinkplot.preplot(rows=2, cols=2)
    a = viewer.viewee.array

    for i, val in enumerate(vals):
        thinkplot.subplot(i+1)
        viewer.draw_array(a==vals[i], vmax=1)
```

`draw_four` takes a `SandPileViewer` object, which is defined in `Sand.py`. The parameter `vals` is the list of values we want to plot; the default values are 0, 1, 2, and 3.

Here's how it's used:

```
viewer3 = SandPileViewer(pile3)
draw_four(viewer3)
```

Figure 8.4 shows the results. Now for each of these patterns we can estimate the fractal dimension using a **box-counting algorithm**: we'll count the num-

ber of cells in a small box at the center of the pile, and then see how the number of cell increases as the box gets bigger. Here's my implementation:

```
def count_cells(a):
    n, m = a.shape
    end = min(n, m)

    res = []
    for i in range(1, end, 2):
        top = (n-i) // 2
        left = (m-i) // 2
        box = a[top:top+i, left:left+i]
        total = np.sum(box)
        res.append((i, i**2, total))

    return np.transpose(res)
```

The parameter, `a`, is a NumPy array of booleans or 0s and 1s. The size of the box is initially 1. Each time through the loop, it increases by 2 until it reaches `end`, which is the smaller of `n` and `m`.

Each time through the loop, `box` is a set of cells with width and height `i`, centered in the array. `total` is the number of "on" cells in the box.

The result is a list of tuples, where each tuple contains `i` and `i**2`, for purposes of comparison, and the number of cells in the box.

Finally, we use `np.transpose` to make a NumPy array with 3 rows containing `i`, `i**2`, and `total`.

To estimate the fractal dimension, we extract the rows:

```
steps, steps2, cells = res
```

Then we can plot the results:

```
    thinkplot.plot(steps, steps2, linestyle='dashed')
    thinkplot.plot(steps, cells)
    thinkplot.plot(steps, steps, linestyle='dashed')
```

And use `linregress` to fit a line on a log-log scale.

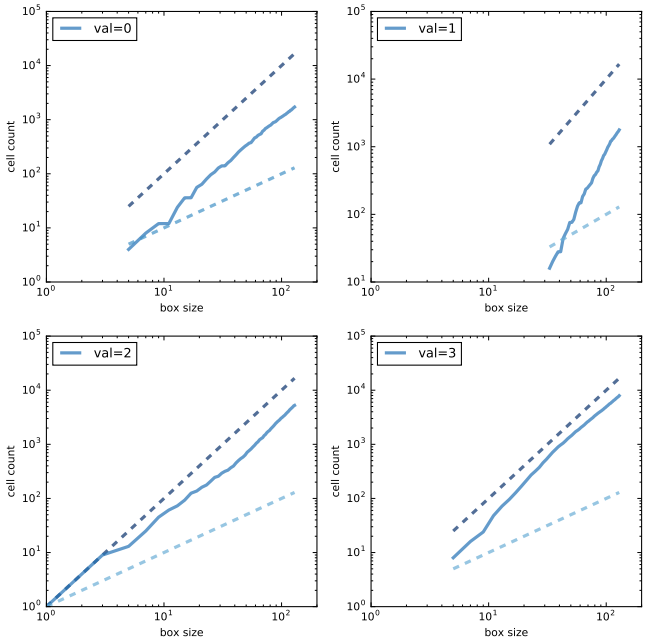

Figure 8.5:

```
from scipy.stats import linregress

params = linregress(np.log(steps), np.log(cells))
slope = params[0]
```

Figure 8.5 shows the results. Notice that only `val=2` (lower left) starts from box size 1 because the center cell has value 2; the other lines start at the first box size that contains a nonzero cell count.

On a log-log scale, the cell counts form nearly straight lines, which indicates that we are measuring fractal dimension over a valid range of box sizes.

And the estimated fractal dimensions are:

```
0  1.871
1  3.502
2  1.781
3  2.084
```

The fractal dimension for values 0, 1, and 2 seems to be clearly non-integer, which indicates that the image is fractal.

Strictly, the fractal dimension for value 3 is indistinguishable from 2, but given the results for the other values, the apparent curvature of the line, and the appearance of the pattern, it seems likely that it is also fractal.

One of the exercises at the end of this chapter asks you to run this analysis again with different values of `n` and `level` to see if the estimated dimensions are consistent.

8.6 Spectral Density

The title of the original paper that presented the sand pile model is "Self-Organized Criticality: An Explanation of $1/f$ Noise". As the subtitle suggests, Bak, Tang and Wiesenfeld were trying to explain why many natural and engineered systems exhibit $1/f$ noise, which is also known as "flicker noise" and "pink noise".

To understand pink noise, we have to take a detour to understand signals, spectral analysis, and noise.

Signal: A **signal** is any quantity that varies in time. One example is sound, which is variation in air density. In this section we'll explore how avalanche durations and sizes vary over time steps.

Spectral analysis: Any signal can be decomposed into a set of frequency components with different volume or power. For example, the sound of a violin playing the A above middle C contains a dominant component at frequency 440 Hz, but it also contains less powerful components at 880 Hz, 1320 Hz, and other integer multiples of the fundamental. **Spectral analysis** is the process of finding the components that make up a signal and their powers, which is called its **spectrum**.

Noise: In common use, "noise" is usually an unwanted sound, but in the context of signal processing, it is a signal that contains many frequency components.

There are many kinds of noise. For example, "white noise" is a signal that has components with equal power over a wide range of frequencies.

Other kinds of noise have different relationships between frequency and power. In "red noise", the power at frequency f is $1/f^2$, which we can write like this:

$$P(f) = 1/f^2$$

We can generalize that by replacing the exponent 2 with a parameter β:

$$P(f) = 1/f^\beta$$

When $\beta = 0$, this equation describes white noise; when $\beta = 2$ it describes red noise. When the parameter is near 1, we call the result $1/f$ noise. More generally, noise with any value between 0 and 2 is called "pink", because it's between white and red.

So how does this apply to the sand pile model? Suppose that every time a cell topples, it makes a sound. If we recorded a sand pile model while its running, what would it sound like?

As my implementation of `SandPile` runs, it records the number of cells that topple during each time step, recording the results in a list called `toppled_seq`. For example, after running the model in Section 8.4, we can extract the resulting signal:

```
signal = pile2.toppled_seq
```

To compute the spectrum of a signal (again, that's the frequencies it contains and their powers) we can use the Fast Fourier Transform (FFT).

The only problem is that the spectrum of a noise signal tends to be noisy. However, we can smooth it out by breaking a long signal into segments, computing the FFT of each segment, and then computing the *average* power at each frequency.

One version of this algorithm is called "Welch's method" and SciPy provides an implementation. We can use it like this:

```
from scipy.signal import welch

nperseg = 2048
freqs, spectrum = welch(signal, nperseg=nperseg, fs=nperseg)
```

`nperseg` is the length of the segments the signal is broken into. With longer segments we get more frequency components, but since there are fewer segments to average, the results are noisier.

`fs` is the "sampling frequency", which is the number of data points in the signal per unit of time. By setting `fs=nperseg`, we get a range of frequencies from 0 to `nperseg/2`, but the units of time in the model are arbitrary, so the frequencies don't mean very much.

The return values, `freqs` and `powers`, are NumPy arrays containing the frequencies of the components and their corresponding powers.

If the signal is pink noise, we expect

$$P(f) = 1/f^{\beta}$$

Taking the log of both sides yields

$$\log P(f) = -\beta \log f$$

So if we plot `powers` versus `freqs` on a log-log scale, we expect a straight line with slope $-\beta$.

Figure 8.6 shows the result. For frequencies between 10 and 1000 (in arbitrary units), the spectrum falls on a straight line. The gray line has slope -1.58, which corresponds to the parameter, $\beta = 1.58$ reported by Bak, Tang, and Wiesenfeld.

This result confirms that the sand pile model generates pink noise.

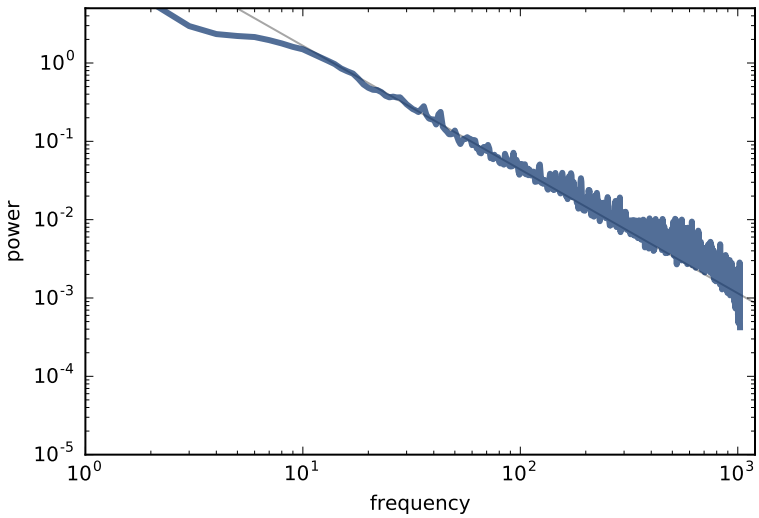

Figure 8.6:

8.7 Reductionism and Holism

The original paper by Bak, Tang and Wiesenfeld is one of the most frequently-cited papers in the last few decades. Other systems have been shown to be self-organized critical, and the sand-pile model, in particular, has been studied in detail.

As it turns out, the sand pile model is not a very good model of a sand pile. Sand is dense and not very sticky, so momentum has a non-negligible effect on the behavior of avalanches. As a result, there are fewer very large and very small avalanches than the model predicts, and the distribution might not be heavy-tailed.

Bak has suggested that this observation misses the point. The sand pile model is not meant to be a realistic model of a sand pile; it is meant to be a simple model for a broad category of systems.

To understand this point, it is useful to think about two kinds of models, **reductionist** and **holistic**. A reductionist model describes a system by describing its parts and their interactions. When a reductionist model is used as an explanation, it depends on an analogy between the components of the model and the components of the system.

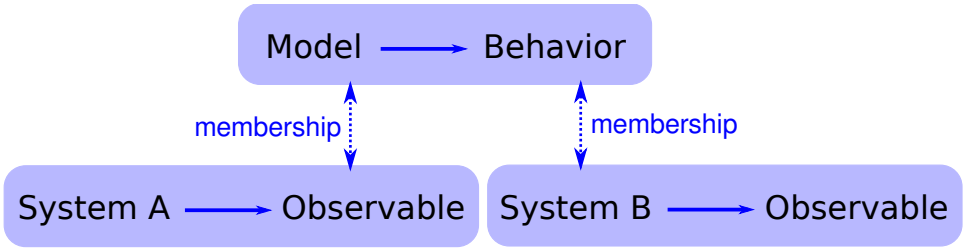

Figure 8.7: The logical structure of a holistic model.

For example, to explain why the ideal gas law holds, we can model the molecules that make up a gas with point masses and model their interactions as elastic collisions. If you simulate or analyze this model, you find that it obeys the ideal gas law. This model is satisfactory to the degree that molecules in a gas behave like molecules in the model. The analogy is between the parts of the system and the parts of the model.

Holistic models are more focused on similarities between systems and less interested in analogous parts. A holistic approach to modeling often consists of two steps, not necessarily in this order:

- Identify a kind of behavior that appears in a variety of systems.
- Find a simple model that demonstrates that behavior.

For example, in *The Selfish Gene*, Richard Dawkins suggests that genetic evolution is just one example of an evolutionary system. He identifies the essential elements of the category—discrete replicators, variability, and differential reproduction—and proposes that any system with these elements displays similar behavior, including complexity without design.

As another example of an evolutionary system, he proposes "memes", which are thoughts or behaviors that are replicated by transmission from person to person[2]. As memes compete for the resource of human attention, they evolve in ways that are similar to genetic evolution.

Critics of the meme model have pointed out that memes are a poor analogy for genes. Memes differ from genes in many obvious ways. But Dawkins

[2]This use of "meme" is original to Dawkins, and predates the distantly-related use of the word on the Internet by about 20 years.

has argued that these differences are beside the point because memes are not *supposed* to be analogous to genes. Rather, memes and genes are examples of the same category: evolutionary systems. The differences between them emphasize the real point, which is that evolution is a general model that applies to many seemingly disparate systems. The logical structure of this argument is shown in Figure 8.7.

Bak has made a similar argument that self-organized criticality is a general model for a broad category of system:

> Since these phenomena appear everywhere, they cannot depend on any specific detail whatsoever... If the physics of a large class of problems is the same, this gives [the theorist] the option of selecting the *simplest* possible [model] belonging to that class for detailed study.[3]

Many natural systems demonstrate behaviors characteristic of critical systems. Bak's explanation for this prevalence is that these systems are examples of the broad category of self-organized criticality. There are two ways to support this argument. One is to build a realistic model of a particular system and show that the model exhibits SOC. The second is to show that SOC is a feature of many diverse models, and to identify the essential characteristics those models have in common.

The first approach, which I characterize as reductionist, can explain the behavior of a particular system. The second, holistic, approach, explains the prevalence of criticality in natural systems. They are different models with different purposes.

For reductionist models, realism is the primary virtue, and simplicity is secondary. For holistic models, it is the other way around.

8.8 SOC, causation and prediction

If a stock market index drops by a fraction of a percent in a day, there is no need for an explanation. But if it drops 10%, people want to know why.

[3]Bak, *How Nature Works*, Springer-Verlag 1996, page 43.

Pundits on television are willing to offer explanations, but the real answer may be that there is no explanation.

Day-to-day variability in the stock market shows evidence of criticality: the distribution of value changes is heavy-tailed and the time series exhibits pink noise. If the stock market is a critical system, we should expect occasional large changes as part of the ordinary behavior of the market.

The distribution of earthquake sizes is also heavy-tailed, and there are simple models of the dynamics of geological faults that might explain this behavior. If these models are right, they imply that large earthquakes are unexceptional; that is, they do not require explanation any more than small earthquakes do.

Similarly, Charles Perrow has suggested that failures in large engineered systems, like nuclear power plants, are like avalanches in the sand pile model. Most failures are small, isolated and harmless, but occasionally a coincidence of bad fortune yields a catastrophe. When big accidents occur, investigators go looking for the cause, but if Perrow's "normal accident theory" is correct, there may be no special cause of large failures.

These conclusions are not comforting. Among other things, they imply that large earthquakes and some kinds of accidents are fundamentally unpredictable. It is impossible to look at the state of a critical system and say whether a large avalanche is "due". If the system is in a critical state, then a large avalanche is always possible. It just depends on the next grain of sand.

In a sand pile model, what is the cause of a large avalanche? Philosophers sometimes distinguish the **proximate** cause, which is most immediately responsible, from the **ultimate** cause, which is, for whatever reason, considered the true cause.

In the sand pile model, the proximate cause of an avalanche is a grain of sand, but the grain that causes a large avalanche is identical to every other grain, so it offers no special explanation. The ultimate cause of a large avalanche is the structure and dynamics of the systems as a whole: large avalanches occur because they are a property of the system.

Many social phenomena, including wars, revolutions, epidemics, inventions and terrorist attacks, are characterized by heavy-tailed distributions. If the reason for these distributions is that social systems are critical, that suggests

that major historical events may be fundamentally unpredictable and unexplainable.

8.9 Exercises

The code for this chapter is in `chap08.ipynb`, which is a Jupyter notebook in the repository for this book. For more information about working with this code, see Section 0.2.

Exercise 8.1 Launch `chap08.ipynb` and run the code. There are a few short exercises embedded in the notebook that you might want to try.

Exercise 8.2 To test whether the distributions of `T` and `S` are heavy-tailed, we plotted their histograms on a log-log scale, which is what Bak, Tang and Wiesenfeld show in their paper. But as we saw in Section 4.7, this visualization can obscure the shape of the distribution. Using the same data, make a plot that shows the cumulative distributions (CDFs) of `S` and `T`. What can you say about their shape? Do they follow a power law? Are they heavy tailed?

You might find it helpful to plot the CDFs on a log-x scale and on a log-log scale.

Exercise 8.3 In Section 8.5 we showed that the initial configuration of the sand pile model produces fractal patterns. But after we drop a large number of random grains, the patterns look more random.

Starting with the example in Section 8.5, run the sand pile model for a while and then compute fractal dimensions for each of the 4 levels. Is the sand pile model fractal in steady state?

Exercise 8.4 Another version of the sand pile model, called the "single source" model, starts from a different initial condition: instead of all cells at the same level, all cells are set to 0 except the center cell, which is set to a large value. Write a function that creates a `SandPile` object, sets up the single source initial condition, and runs until the pile reaches equilibrium. Does the result appear to be fractal?

You can read more about this version of the sand pile model at http://math.cmu.edu/~wes/sandgallery.html.

Exercise 8.5 In a 1989 paper, Bak, Chen and Creutz suggest that the Game of Life is an SOC system.[4]

To replicate their tests, run the GoL CA until it stabilizes, then choose a random cell and flip it. Run the CA until it stabilizes again, keeping track of `T`, the number of time steps it takes, and `S`, the number of cells affected. Repeat for a large number of trials and plot the distributions of `T` and `S`. Also, record the number of cells that change state during each time step and see if the resulting time series resembles pink noise.

Exercise 8.6 In *The Fractal Geometry of Nature*, Benoit Mandelbrot proposes what he calls a "heretical" explanation for the prevalence of heavy-tailed distributions in natural systems. It may not be, as Bak suggests, that many systems can generate this behavior in isolation. Instead there may be only a few, but there may be interactions between systems that cause the behavior to propagate.

To support this argument, Mandelbrot points out:

- The distribution of observed data is often "the joint effect of a fixed underlying *true distribution* and a highly variable *filter*."
- Heavy-tailed distributions are robust to filtering; that is, "a wide variety of filters leave their asymptotic behavior unchanged."

What do you think of this argument? Would you characterize it as reductionist or holist?

Exercise 8.7 Read about the "Great Man" theory of history at http://en.wikipedia.org/wiki/Great_man_theory. What implication does self-organized criticality have for this theory?

[4] "Self-organized criticality in the Game of Life", available from http://www.nature.com/nature/journal/v342/n6251/abs/342780a0.html.

Chapter 9

Agent-based models

The models we have seen so far might be characterized as "rule-based" in the sense that they involve systems governed by simple rules. In this and the following chapters, we explore **agent-based models**.

Agent-based models include **agents** that are intended to model people and other entities that gather information about the world, make decisions, and take actions.

The agents are usually situated in space or in a network, and interact with each other locally. They usually have imperfect, local information.

Often there are differences among agents, unlike previous models where all components are identical. And agent-based models often include randomness, either among the agents or in the world.

Since the 1970s, agent-based modeling has become an important tool in economics and other social sciences, and in some natural sciences.

Agent-based models are useful for modeling the dynamics of systems that are not in equilibrium (although they are also used to study equilibrium). And they are particularly useful for understanding relationships between individual decisions and system behavior.

The code for this chapter is in `chap09.ipynb`, which is a Jupyter notebook in the repository for this book. For more information about working with this code, see Section 0.2.

9.1 Schelling's Model

In 1971 Thomas Schelling published "Dynamic Models of Segregation", which proposes a simple model of racial segregation. The Schelling model of the world is a grid; each cell represents a house. The houses are occupied by two kinds of people, called **agents**, labeled red and blue, in roughly equal numbers. About 10% of the houses are empty.

At any point in time, an agent might be happy or unhappy, depending on the other agents in the neighborhood. The neighborhood of each house is the set of eight adjacent cells. In one version of the model, agents are happy if they have at least two neighbors like themselves, and unhappy if they have one or zero.

The simulation proceeds by choosing an agent at random and checking to see whether they are happy. If so, nothing happens; if not, the agent chooses one of the unoccupied cells at random and moves.

You might not be surprised to hear that this model leads to some segregation, but you might be surprised by the degree. Fairly quickly, clusters of similar agents appear. The clusters grow and coalesce over time until there are a small number of large clusters and most agents live in homogeneous neighborhoods.

If you did not know the process and only saw the result, you might assume that the agents were racist, but in fact all of them would be perfectly happy in a mixed neighborhood. Since they prefer not to be greatly outnumbered, they might be considered xenophobic at worst. Of course, these agents are a wild simplification of real people, so it may not be appropriate to apply these descriptions at all.

Racism is a complex human problem; it is hard to imagine that such a simple model could shed light on it. But in fact it provides a strong argument about the relationship between a system and its parts: if you observe segregation in a real city, you cannot conclude that individual racism is the immediate cause, or even that the people in the city are racists.

Of course, we have to keep in mind the limitations of this argument: Schelling's model demonstrates a possible cause of segregation, but says nothing about actual causes.

9.2 Implementation of Schelling's model

To implement Schelling's model, I wrote yet another class that inherits from `Cell2D`:

```
class Schelling(Cell2D):

    def __init__(self, n, m=None, p=0.5):
        self.p = p
        m = n if m is None else m
        choices = [0, 1, 2]
        probs = [0.1, 0.45, 0.45]
        self.array = np.random.choice(choices, (n, m), p=probs)
```

The parameters `n` and `m` are the dimensions of the grid, and `p` is the threshold on the fraction of similar neighbors. For example, if `p=0.5`, an agent will be unhappy if fewer than 50% of their neighbors are the same color.

`array` is a NumPy array where each cell is 0 if empty, 1 if occupied by a red agent, and 2 if occupied by a blue agent. Initially 10% of the cells are empty, 45% red, and 45% blue.

The `step` function for Schelling's model is substantially more complicated than previous step functions. If you are not interested in the details, you can skip to the next section. But if you stick around, you might pick up some NumPy tips.

First, I'll make logical arrays indicating which cells are red, blue, and occupied:

```
        a = self.array
        red = a==1
        blue = a==2
        occupied = a!=0
```

I'll use `np.correlate2d` to count, for each cell, the number of neighboring cells that are red and the number that are occupied.

```
    options = dict(mode='same', boundary='wrap')

    kernel = np.array([[1, 1, 1],
                       [1, 0, 1],
                       [1, 1, 1]], dtype=np.int8)

    num_red = correlate2d(red, kernel, **options)
    num_neighbors = correlate2d(occupied, kernel, **options)
```

Now for each cell we can compute the fraction of neighbors that are red and the fraction that are the same color:

```
    frac_red = num_red / num_neighbors
    frac_blue = 1 - frac_red
    frac_same = np.where(red, frac_red, frac_blue)
```

`frac_red` is just the ratio of `num_red` and `num_neighbors`, and `frac_blue` is the complement of `frac_red`.

`frac_same` is a little bit more complicated. The function `np.where` is like an element-wise `if` expression. The first parameter is a condition that selects elements from the second or third parameter.

In this case, wherever `red` is `True`, `frac_same` gets the corresponding element of `frac_red`. Where `red` is `False`, `frac_same` gets the corresponding element of `frac_blue`.

Now we can identify the locations of the unhappy agents:

```
    unhappy_locs = locs_where(occupied & (frac_same < self.p))
```

The result, `unhappy_locs`, is a NumPy array where each row is the coordinates of an occupied cell where `frac_same` is below the threshold `p`.

`locs_where` is a wrapper function for `np.nonzero`:

```
def locs_where(condition):
    return np.transpose(np.nonzero(condition))
```

`np.nonzero` takes an array and returns the coordinates of all non-zero cells, but the results are in the form of two tuples. `np.transpose` converts the results to a more useful form, an array where each row is a coordinate pair.

Similarly, `empty_locs` is an array that contains the coordinates of the empty cells, shuffled:

```
    empty_locs = locs_where(a==0)
```

Now we get to the core of the simulation. We loop through the unhappy agents and move them:

```
    for source in unhappy_locs:
        i = np.random.randint(len(empty_locs))
        dest = tuple(empty_locs[i])
        a[dest] = a[tuple(source)]
        a[tuple(source)] = 0
        empty_locs[i] = source
```

`i` is an index used to choose a random empty cell.

`dest` is a tuple containing the coordinates of the empty cell.

In order to move an agent, we copy the value from `source` to `dest`, and then set the value of `source` to 0 (since it is now empty).

Finally, we replace the entry in `empty_locs` with `source`, so the cell that just became empty can be chosen by the next agent.

9.3 Segregation

Now let's see what happens when we run the model. I'll start with `n=100` and `p=0.3`, and run for 10 steps.

```
grid = Schelling(n=100, p=0.3)
for i in range(10):
    grid.step()
```

Figure 9.1 shows the initial configuration (left), the state of the simulation after 2 steps (middle) and after 10 steps (right).

Clusters form quickly, with red and blue agents moving into segregated clusters separated by boundaries of empty cells.

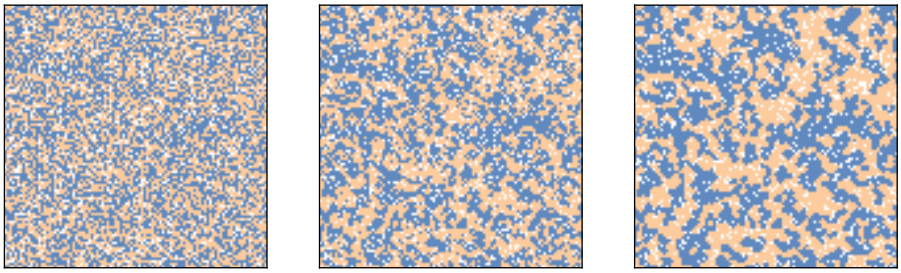

Figure 9.1: Schelling's segregation model with `n=100`, initial condition (left), after 2 steps (middle) and after 10 steps (right).

For each configuration, we can compute the degree of segregation, which is the fraction of neighbors who are the same color, averaged across cells:

```
    np.sum(frac_same) / np.sum(occupied)
```

In Figure 9.1, the degree average fraction of similar neighbors is 55% in the initial configuration, 71% after two steps, and 80% after 10 steps!

Remember that when `p=0.3` the agents would be happy if 3 of 8 neighbors were their own color, but they end up living in neighborhood where 6 or 7 of their neighbors are their own color, typically.

Figure 9.2 shows how the degree of segregation increases and where it levels off for several values of `p`. When `p=0.4`, the degree of segregation in steady state is about 88%, and a majority of agents have no neighbors with a different color.

These results are surprising to many people, and they make a striking example of the complex and unpredictable relationship between individual decisions and system behavior.

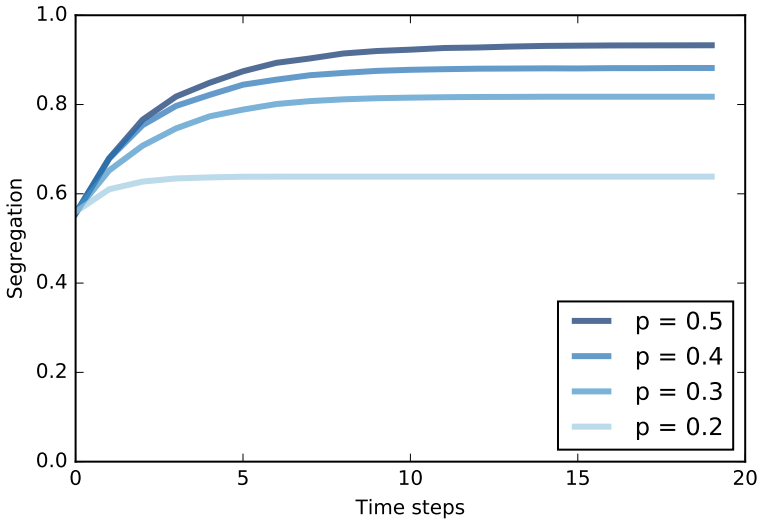

Figure 9.2: Degree of segregation in Schelling's model, over time, for a range of `p`.

9.4 Sugarscape

In 1996 Joshua Epstein and Robert Axtell proposed Sugarscape, an agent-based model of an "artificial society" intended to support experiments related to economics and other social sciences.

Sugarscape is a versatile model that has been adapted for a wide variety of topics. As examples, I will replicate the first few experiments from Epstein and Axtell's book, *Growing Artificial Societies*.

In its simplest form, Sugarscape is a model of a simple economy where agents move around on a 2D grid, harvesting and accumulating "sugar", which represents economic wealth. Some parts of the grid produce more sugar than others, and some agents are better at finding it than others.

This version of Sugarscape is often used to explore and explain the distribution of wealth, in particular the tendency toward inequality.

In the Sugarscape grid, each cell has a capacity, which is the maximum amount of sugar it can hold. In the original configuration, there are two high-sugar

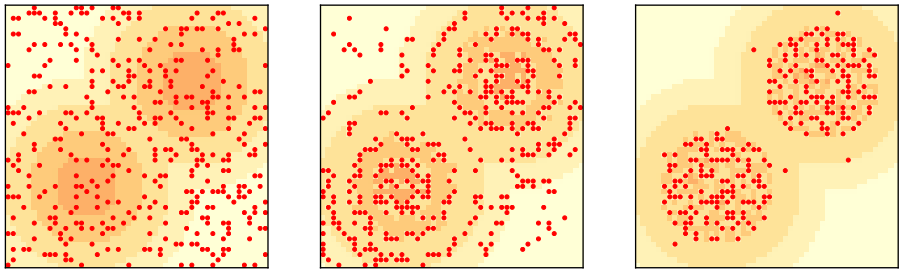

Figure 9.3: Replication of the original Sugarscape model: initial configuration (left), after 2 steps (middle) and 100 steps (right).

regions, with capacity 4, surrounding by concentric rings with capacities 3, 2, and 1.

Figure 9.3 (left) shows the initial configuration, with the darkest areas indicating cells with the highest capacity, and small circles representing the agents.

Initially there are 400 agents placed at random locations. Each agent has three randomly-chosen attributes:

Sugar: Each agent starts with an endowment of sugar chosen from a uniform distribution between 5 and 25 units.

Metabolism: Each agent has some amount of sugar they must consumer per time step, chosen uniformly between 1 and 4.

Vision: Each agent can "see" the amount of sugar in nearby cells and move to the cell with the most, but some agents can see farther than others. The distance agents see is chosen uniformly between 1 and 6.

During each time step, agents move one at a time in a random order. Each agent follows these rules:

- The agent surveys `k` cells in each of the 4 compass directions, where `k` is the range of the agent's vision.

- It chooses the unoccupied cell with the most sugar. In case of a tie, it chooses the closer cell; among cells at the same distance, it chooses randomly.

- The agent moves to the selected cell and harvests the sugar, adding the harvest to its accumulated wealth and leaving the cell empty.

- The agent consumes some part of its wealth, depending on its metabolism. If the resulting total is negative, the agent "starves" and is removed.

After all agents have executed these steps, the cells grow back some sugar, typically 1 unit, but the total sugar in each cell is bounded by its capacity.

Figure 9.3 (middle) shows the state of the model after two steps. Most agents are moving toward the areas with the most sugar. Agents with high vision also move the fastest; agents with low vision tend to get stuck on the plateaus, wandering randomly until they get close enough to see the next level.

Agents born in the areas with the least sugar are likely to starve unless they also have high vision and a high initial endowment.

Within the high-sugar areas, agents compete with each other to find and harvest sugar as it grows back. Agents with high metabolism or low vision are the most likely to starve.

When sugar grows back at 1 unit per time step, there is not enough sugar to sustain the 400 agents we started with. The population drops quickly at first, then more slowly, and levels off around 250.

Figure 9.3 (right) shows the state of the model after 100 time steps, with about 250 agents. The agents who survive tend to be the lucky ones, born with high vision and/or low metabolism. Having survived to this point, they are likely to survive forever, accumulating unbounded stockpiles of sugar.

9.5 Wealth inequality

In its current form, Sugarscape models a simple ecology, and could be used to explore the relationship between the parameters of the model, like the growth rate and the attributes of the agents, and the carrying capacity of the system (the number of agents that survive in steady state). And it models a form of natural selection, where agents with higher "fitness" are more likely to survive.

The model also demonstrates a kind of wealth inequality, with some agents accumulating sugar faster than others. But it would be hard to say anything specific about the distribution of wealth because it is not "stationary"; that is, the distribution changes over time and does not reach a steady state.

However, if we give the agents finite lifespans, the model produces a stationary distribution of wealth. And then we can run experiments to see what effect the parameters and rules have on this distribution.

In this version of the model, agents have an age that gets incremented each time step, and a random lifespan that is uniform from 60 to 100. If an agent's age exceeds its lifespan, it dies.

When an agent dies, from starvation or old age, it is replaced by a new agent with random attributes, so the total population is constant.

Starting with 250 agents, which is close to carrying capacity, I ran the model for 500 steps. After each 100 steps, I plot the distribution of sugar accumulated by the agents. Figure 9.4 shows the results on a linear scale (left) and a log-x scale (right).

After about 200 steps (which is twice the longest lifespan) the distribution doesn't change much. And it is skewed to the right.

Most agents have little accumulated wealth: the 25th percentile is about 10 and the median is about 20. But a few agents have accumulated much more: the 75th percentile is about 40, and the highest value is more than 150.

On a log scale the shape of the distribution resembles a Gaussian or normal distribution, although the right tail is truncated. If it were actually normal on a log scale, the distribution would be lognormal, which is a heavy-tailed distribution. And in fact, the distribution of wealth in practically every country, and in the world, is a heavy-tailed distribution.

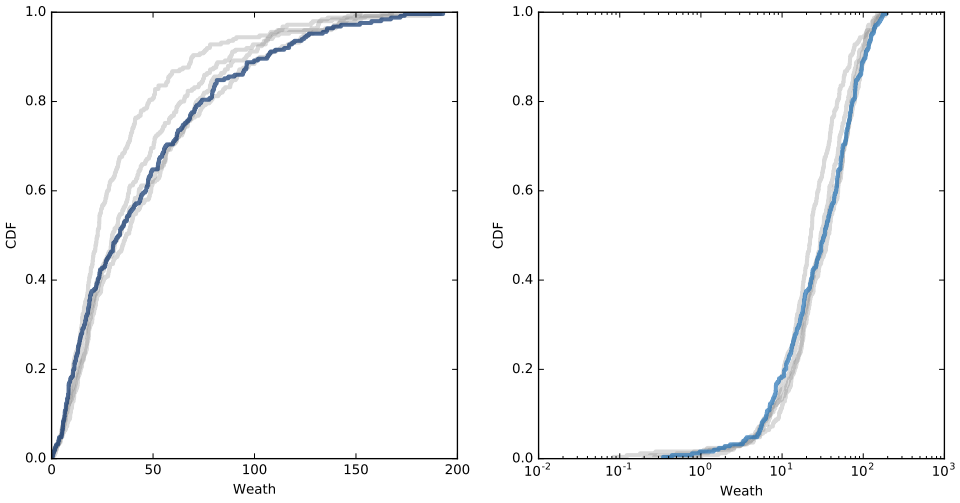

Figure 9.4: Distribution of sugar (wealth) after 100, 200, 300, and 400 steps (gray lines) and 500 steps (dark line). Linear scale (left) and log-x scale (right).

It would be too much to claim that Sugarscape explains why wealth distributions are heavy-tailed, but the prevalence of inequality in variations of Sugarscape suggests that inequality is characteristic of many economies, even very simple ones. And experiments with rules that model taxation and other income transfers suggest that it is not easy to avoid or mitigate.

9.6 Implementing Sugarscape

Sugarscape is more complicated than the previous models, so I won't present the entire implementation here. I will outline the structure of the code and you can see the details in the Jupyter notebook for this chapter, `chap09.ipynb`, which is in the repository for this book. And if you are not interested in the details, you can skip to the next section.

Here is the `Agent` class with the `step` method:

```
class Agent:

    def step(self, env):
        self.loc = env.look_around(self.loc, self.vision)
        self.sugar += env.harvest(self.loc) - self.metabolism
        self.age += 1
```

During each step, the agent moves, harvests sugar, and increments `age`.

The parameter `env` is a reference to the environment, which is a `Sugarscape` object. It provides methods `look_around` and `harvest`:

- `look_around` takes the location of the agent, which is a tuple of coordinates, and the range of the agent's vision, which is an integer. It returns the agent's new location, which is the visible cell with the most sugar.

- pyharvest takes the (new) location of the agent, and removes and returns the sugar at that location.

And here's the `Sugarscape` class and its `step` method (without replacement):

```
class Sugarscape(Cell2D):

    def step(self):

        # loop through the agents in random order
        random_order = np.random.permutation(self.agents)
        for agent in random_order:

            # mark the current cell unoccupied
            self.occupied.remove(agent.loc)

            # execute one step
            agent.step(self)

            # if the agent is dead, remove from the list
            if agent.is_starving():
                self.agents.remove(agent)
            else:
                # otherwise mark its cell occupied
                self.occupied.add(agent.loc)

        # grow back some sugar
        self.grow()
        return len(self.agents)
```

`Sugarscape` inherits from `Cell2D`, so it is similar to the other grid-based models we've seen.

The attributes include `agents`, which is a list of `Agent` objects, and `occupied`, which is a set of tuples, where each tuples contains the coordinates of a cell occupied by an agent.

During each step, the `Sugarscape` loops through the agents in random order. It invokes `step` on each agent and then checks whether it is dead. After all agents have moved, some of the sugar grows back.

If you are interested in learning more about NumPy, you might want to look more closely at `make_visible_locs`, which builds an array where each row contains the coordinates of a cell visible to an agent, sorted by distance but with cells at the same distance appearing in random order.

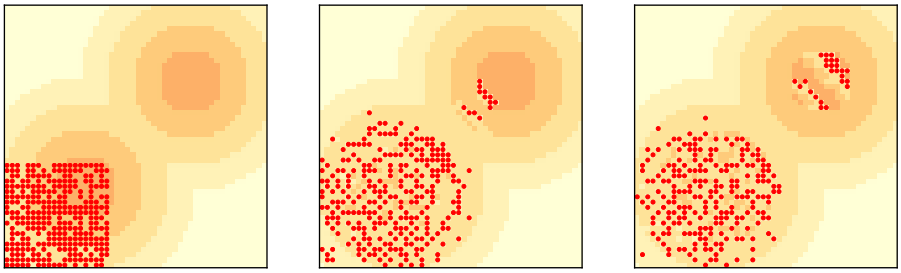

Figure 9.5: Wave behavior in Sugarscape: initial configuration (left), after 6 steps (middle) and after 12 steps (right).

And you might want to look at `Sugarscape.make_capacity`, which initializes the capacity of the cells. It demonstrates a use of `np.meshgrid`, which is often useful but takes some time to understand.

9.7 Migration and Wave Behavior

Although the purpose of Sugarscape is not primarily to explore the movement of agents in space, Epstein and Axtell observed some interesting patterns when agents migrate.

If we start with all agents in the lower-left corner, they quickly move toward the closest "peak" of high-capacity cells. But if there are more agents than a single peak can support, they quickly exhaust the sugar and agents are forced to move into lower-capacity areas.

The ones with the longest vision range cross the valley between the peaks first and propagate toward the northeast in a pattern that resembles a wave front. Because they leave a stripe of empty cells behind them, other agents don't follow until the sugar grows back.

The result is a series of discrete waves of migration, where each wave resembles a coherent object, like the spaceships we saw in the Rule 110 CA and Game of Life (see Section 5.6 and Section 6.2).

Figure 9.5 shows the initial condition (left) and the state of the model after 6 steps (middle) and 12 steps (right). You can see the first two waves reaching and moving through the second peak, leaving a stripe of empty cells behind. You can also see an animated version of this model, where the wave patterns are more clearly visible.

Although these waves are made up of agents, we can think of them as entities of their own, in the same way we think of gliders in Game of Life.

An interesting property of these waves is that they move diagonally, which might be surprising, because the agents themselves only move north or east, never northeast. Outcomes like this – groups or "aggregates" with properties and behaviors that the agents don't have – are common in agent-based models. We will see more examples in the next chapter.

9.8 Emergence

The examples in this chapter demonstrate one of the most important ideas in complexity science: emergence. An **emergent property** is a characteristic of a system that results from the interaction of its components, not from their properties.

To clarify what emergence is, it helps to consider what it isn't. For example, a brick wall is hard because bricks and mortar are hard, so that's not an emergent property. As another example, some rigid structures are built from flexible components, so that seems like a kind of emergence. But it is at best a weak kind, because structural properties follow from well understood laws of mechanics.

In contrast, the segregation we see in Schelling's model is an emergent property because it is not caused by racist agents. Even when the agents are only mildly xenophobic, the outcome of the system is substantially different from the intention of the agent's decisions.

The distribution of wealth in Sugarscape might be an emergent property, but it is a weak example because we could reasonably predict it based on the distributions of vision, metabolism, and lifespan. The wave behavior we saw in the last example might be a stronger example, since the wave displays a capability – diagonal movement – that the agents clearly do not have.

Emergent properties are surprising: it is hard to predict the behavior of the system even if we know all the rules. That difficulty is not an accident; in fact, it may be the defining characteristic of emergence.

As Wolfram discusses in *A New Kind of Science*, conventional science is based on the axiom that if you know the rules that govern a system, you can predict its behavior. What we call "laws" are often computational shortcuts that allow us to predict the outcome of a system without building or observing it.

But many cellular automatons are **computationally irreducible**, which means that there are no shortcuts. The only way to get the outcome is to implement the system.

The same may be true of complex systems in general. For physical systems with more than a few components, there is usually no model that yields an analytic solution. Numerical methods provide a kind of computational shortcut, but there is still a qualitative difference.

Analytic solutions often provide a constant-time algorithm for prediction; that is, the run time of the computation does not depend on t, the time scale of prediction. But numerical methods, simulation, analog computation, and similar methods take time proportional to t. And for many systems, there is a bound on t beyond which we can't compute reliable predictions at all.

These observations suggest that emergent properties are fundamentally unpredictable, and that for complex systems we should not expect to find natural laws in the form of computational shortcuts.

To some people, "emergence" is another name for ignorance; by this reckoning, a property is emergent if we don't have a reductionist explanation for it, but if we come to understand it better in the future, it would no longer be emergent.

The status of emergent properties is a topic of debate, so it is appropriate to be skeptical. When we see an apparently emergent property, we should not

assume that there can never be a reductionist explanation. But neither should we assume that there has to be one.

The examples in this book and the principle of computational equivalence give good reasons to believe that at least some emergent properties can never be "explained" by a classical reductionist model.

You can read more about emergence at http://en.wikipedia.org/wiki/Emergence.

9.9 Exercises

Exercise 9.1 Bill Bishop, author of *The Big Sort*, argues that American society is increasingly segregated by political opinion, as people choose to live among like-minded neighbors.

The mechanism Bishop hypothesizes is not that people, like the agents in Schelling's model, are more likely to move if they are isolated, but that when they move for any reason, they are likely to choose a neighborhood with people like themselves.

Modify your implementation of Schelling's model to simulate this kind of behavior and see if it yields similar degrees of segregation.

There are several ways you can model Bishop's hypothesis. In my implementation, a random selection of agents moves during each step. Each agent considers `k` randomly-chosen empty locations and chooses the one with the highest fraction of similar neighbors. How does the degree of segregation depend on `k`?

Exercise 9.2 In the first version of SugarScape, we never add agents, so once the population falls, it never recovers. In the second version, we only replace agents when they die, so the population is constant. Now let's see what happens if we add some "population pressure".

Write a version of SugarScape that adds a new agent at the end of every step. Add code to compute the average vision and the average metabolism of the agents at the end of each step. Run the model for a few hundred steps

and plot the population over time, as well as the average vision and average metabolism.

You should be able to implement this model by inheriting from `SugarScape` and overriding `__init__` and `step`.

Exercise 9.3 In the philosophy of mind, *Strong AI* is the theory that an appropriately-programmed computer could have a mind in the same sense that humans have minds.

John Searle presented a thought experiment called "The Chinese Room", intended to show that *Strong AI* is false. You can read about it at http://en.wikipedia.org/wiki/Chinese_room.

What is the **system reply** to the Chinese Room argument? How does what you have learned about emergence influence your reaction to the system response?

Chapter 10

Herds, Flocks, and Traffic Jams

NOTE: THIS CHAPTER IS NOT DONE!

The code for this chapter is in `chap10.ipynb`, which is a Jupyter notebook in the repository for this book. For more information about working with this code, see Section 0.2.

10.1 Traffic jams

What causes traffic jams? In some cases there is an obvious cause, like an accident, a speed trap, or something else that disturbs the flow of traffic. But other times traffic jams appear for no apparent reason.

Agent-based models can help explain spontaneous traffic jams. As an example, I implemented a simple highway simulation, based on a model in Resnick, *Turtles, Termites and Traffic Jams*.

Here's the class that represents the "highway":

```
class Highway:

    def __init__(self, n=10, length=1000, eps=0):
        self.length = length
        self.eps = eps

        # create the drivers
        locs = np.linspace(0, length, n, endpoint=False)
        self.drivers = [Driver(loc) for loc in locs]

        # and link them up
        for i in range(n):
            j = (i+1) % n
            self.drivers[i].next = self.drivers[j]
```

`n` is the number of cars.

`length` is the length of the highway, which is 1000 by default (in arbitrary units).

`eps` is the amount of random noise we'll add to the system.

`locs` contains the locations of the drivers; initially they are equally spaced along the highway.

Finally, we link the drivers so that each driver contains a reference to the next driver in front. The highway is circular, so the last driver contains a reference to the first.

The `step` method is simple; it just moves each of the drivers:

```
    def step(self):
        for driver in self.drivers:
            self.move(driver)
```

And here's the `move` method:

```
    def move(self, driver):
        d = self.distance(driver)

        # let the driver choose acceleration
        acc = driver.choose_acceleration(d)
        acc = min(acc, self.max_acc)
        acc = max(acc, self.min_acc)
        speed = driver.speed + acc

        # add random noise to speed
        speed *= np.random.uniform(1-self.eps, 1+self.eps)

        # keep it nonnegative and under the speed limit
        speed = max(speed, 0)
        speed = min(speed, self.speed_limit)

        # if current speed would collide, stop
        if speed > d:
            speed = 0

        # update speed and loc
        driver.speed = speed
        driver.loc += speed
```

`d` is the distance between `driver` and the next driver ahead. This distance is passed to `choose_acceleration`, which specifies the behavior of the driver. This is the only decision the driver gets to make; everything else is determined by the "physics" of the simulation.

- `acc` is acceleration, which limited by `min_acc` and `max_acc`. In my implementation, cars can accelerate with `max_acc=1` and brake with `min_acc=-10`.

- `speed` is the old speed plus the requested acceleration, but then we make some adjustments. First, we add random noise to `speed`, because the world is not perfect. `eps` determines the magnitude of the noise, which is a percentage applied to `speed`; for example, if `eps` is 0.02, `speed` is muliplied by a random number between 98% and 102%.

- Then speed is bounded between 0 and `speed_limit`, which is 40 in my implementation, so cars are not allowed to go backward or speed.
- If the requested speed would cause a collision with the next car, `speed` is set to 0.
- Finally, we update the `speed` and `loc` attributes of `driver`.

Here's the definition for the `Driver` class:

```
class Driver:

    def __init__(self, loc, speed=0):
        self.loc = loc
        self.speed = speed

    def choose_acceleration(self, d):
        return 1
```

The attributes `loc` and `speed` are the location and speed of the driver.

This implementation of `choose_acceleration` is very simple: it always accelerates at the maximum rate.

Since the cars start out equally spaced, we expect them all to accelerate until they reach the speed limit, or until their speed exceeds the space between them. At that point, at least one "collision" will occur, causing some cars to stop.

Figure 10.1 shows a few steps in this process, starting with 30 cars and `eps=0.02`. On the left is the configuration after 16 time steps, with the cars arranged in a circle. Because of random noise, some cars are going faster than others, and the spacing has become uneven.

During the next time step (center) two cars collide, indicated by the `x` marks.

During the next time step (right) two cars collide with the stopped cars, and we can see the initial formation of a traffic jam. Once a jam forms, it tends to persist, with additional cars approaching from behind and colliding, and with the cars in the front accelerating away.

Under some conditions, the jam itself propagates backwards, as you can see if you watch the animations in the notebook for this chapter.

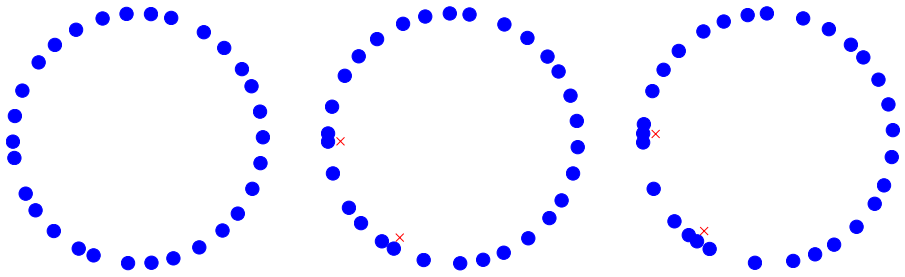

Figure 10.1:

10.2 Random Noise

As the number of cars increases, traffic jams become more severe. Figure 10.2 shows the average speed cars are able to achieve, as a function of the number of cars.

The top line shows results with `eps=0`; that is, with no random variation in speed. With 25 or fewer cars, the spacing between cars is greater than 40, which allows cars to reach and maintain the maximum speed, which is 40. With more than 25 cars, traffic jams form and the average speed drops quickly.

This effect is a direct result of the physics of the simulation, so it should not be surprising. If the length of the road is 1000, the spacing between `n` cars is `1000/n`. And since cars can't move faster than the space in front of them, the highest average speed we expect is `1000/n` or 40, whichever is less.

But that's the best case scenario. With just a small amount of randomness, things get much worse.

Figure 10.2 also shows results with `eps=0.001` and `eps=0.01`, which correspond to errors in speed of 0.1% and 1%.

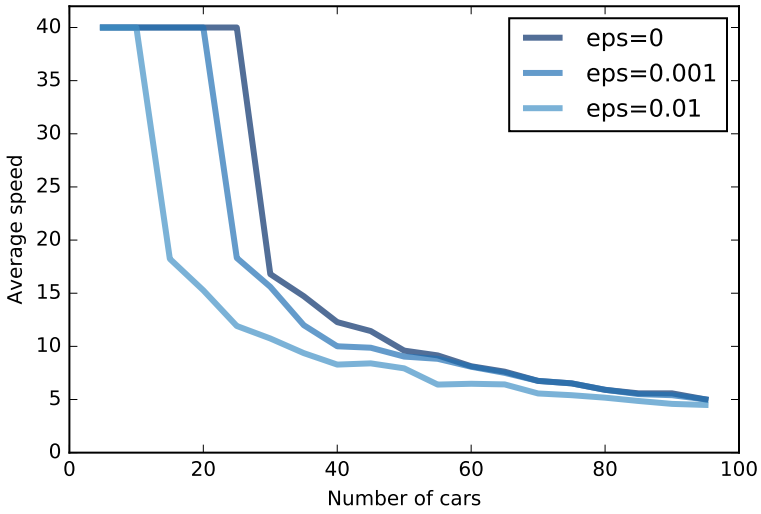

Figure 10.2:

With even a small amount of noise, the capacity of the highway drops from 25 to 20 (by "capacity" I mean the maximum number of cars that can reach and sustain the speed limit). And with 1% errors, the capacity drops to 10. Ugh.

As one of the exercises at the end of this chapter, you'll have a chance to design a better driver; that is, you will experiment with different strategies in `choose_acceleration` and see if you can find driver behaviors that improve average speed.

10.3 Boids

In 1987 Craig Reynolds published "Flocks, herds and schools: A distributed behavioral model", which describes an agent-based model of herd behavior. You can download his paper from http://www.red3d.com/cwr/papers/1987/boids.html.

Agents in this models are called "boids", which is both a contraction of "bird-oid" and an accented pronunciation of "bird" (although boids are also used to model fish and herding land animals).

Each agent simulates three behaviors:

Collision avoidance: Avoid obstacles, including other birds.

Flock centering: Move toward the center of the flock.

Velocity matching: Align velocity with neighboring birds.

Boids make decisions based on local information only; each boid only sees (or pays attention to) other boids in its field of vision and range.

The `Visual` package, also known as VPython, is well-suited for implementing boids. It provides simple 3-D graphics as well as vector objects and operations that are useful for the computations.

You can download my implementation from thinkcomplex.com/Boids.py. It is based in part on the description of boids in Flake, *The Computational Beauty of Nature*.

The program defines two classes: `Boid`, which implements the boid algorithm, and `World`, which contains a list of Boids and a "carrot" the Boids are attracted to.

The boid algorithm uses `get_neighbors` to find other boids in the field of view:

```
    def get_neighbors(self, others, radius, angle):
        boids = []
        for other in others:
            if other is self:
               continue

            offset = other.pos - self.pos

            # if not in range, skip it
            if offset.mag > radius:
                continue

            # if not within viewing angle, skip it
            if self.vel.diff_angle(offset) > angle:
                continue

            # otherwise add it to the list
            boids.append(other)

        return boids
```

`get_neighbors` uses vector subtraction to compute the vector from `self` to `other`. The magnitude of this vector is the distance to the other boid. `diff_angle` computes the angle between the velocity of `self`, which is also the line of sight, and the other boid.

`center` finds the center of mass of the boids in the field of view and returns a vector pointing toward it:

```
    def center(self, others):
        close = self.get_neighbors(others, r_center, a_center)
        t = [other.pos for other in close]
        if t:
            center = sum(t)/len(t)
            toward = vector(center - self.pos)
            return limit_vector(toward)
        else:
            return null_vector
```

Similarly, `avoid` finds the center of mass of any obstacles in range and returns a

vector pointing away from it, `copy` returns the difference between the current heading and the average heading of the neighbors, and `love` computes the heading toward the carrot.

`set_goal` computes the weighed sum of these goals and sets the overall goal:

```
    def set_goal(self, boids, carrot):
        self.goal = (w_avoid * self.avoid(boids, carrot) +
                     w_center * self.center(boids) +
                     w_copy * self.copy(boids) +
                     w_love * self.love(carrot))
```

Finally, `move` updates the velocity, position and attitude of the boid:

```
    def move(self, mu=0.1):
        self.vel = (1-mu) * self.vel + mu * self.goal
        self.vel.mag = 1

        self.pos += dt * self.vel
        self.axis = b_length * self.vel.norm()
```

The new velocity is the weighted sum of the old velocity and the goal. The parameter `mu` determines how quickly the birds can change speed and direction. The time step, `dt`, determines how far the boids move.

Many parameters influence flock behavior, including the range, angle and weight for each behavior, and the maneuverability, `mu`.

These parameters determine the ability of the boids to form and maintain a flock and the patterns of motion and organization in the flock. For some settings, the boids resemble a flock of birds; other settings resemble a school of fish or a cloud flying insects.

10.4 Free will

Many complex systems have properties, as a whole, that their components do not:

- The Rule 30 cellular automaton is deterministic, and the rules that govern its evolution are completely known. Nevertheless, it generates a sequence that is statistically indistiguishable from random.
- The agents in Schelling's model are not racist, but the outcome of their interactions is as if they were.
- Agents in Sugarscape form waves that move diagonally even though the agents cannot.
- Traffic jams move backward even though the cars in them are moving forward.
- The behavior of flocks and herds emerges from local interactions between their members.

These examples suggest an approach to several old and challenging questions, including the problems of consciousness and free will.

Free will is the ability to make choices, but if our bodies and brains are governed by deterministic physical laws, our choices would be determined. Arguments about free will are innumerable; I will only mention two:

- William James proposed a two-stage model in which possible actions are generated by a random process and then selected by a deterministic process. In that case our actions are fundamentally unpredictable because the process that generates them includes a random element.
- David Hume suggested that our perception of making choices is an illusion; in that case, our actions are deterministic because the system that produces them is deterministic.

These arguments reconcile the conflict in opposite ways, but they agree that there is a conflict: the system cannot have free will if the parts are deterministic.

The complex systems in this book suggest the alternative that free will, at the level of options and decisions, is compatible with determinism at the level of neurons (or some lower level). In the same way that a traffic jam moves backward while the cars move forward, a person can have free will even though neurons don't.

10.5 Exercises

Exercise 10.1 In the traffic jam simulation, define a class, `BetterDriver`, that inherits from `Driver` and overrides `choose_acceleration`. See if you can define driving rules that do better than the basic implementation in `Driver`. You might try to achieve higher average speed, or a lower number of collisions.

Exercise 10.2 Run my implementation of the boid algorithm and experiment with different parameters. What happens if you "turn off" one of the behaviors by setting the weight to 0?

To generate more bird-like behavior, Flake suggests adding a fourth behavior to maintain a clear line of sight; in other words, if there is another bird directly ahead, the boid should move away laterally. What effect do you expect this rule to have on the behavior of the flock? Implement it and see.

Exercise 10.3 Read more about free will at http://en.wikipedia.org/wiki/Free_will. The view that free will is compatible with determinism is called **compatibilism**. One of the strongest challenges to compatibilism is the "consequence argument". What is the consequence argument? What response can you give to the consequence argument based on what you have read in this book?

Chapter 11

Game Theory

NOTE: THIS CHAPTER IS NOT DONE!

11.1 The El Farol Problem

11.2 Prisoner's Dilemma

The Prisoner's Dilemma is a topic of study in game theory, but it's not the fun kind of game. Instead, it is the kind of game that sheds light on human motivation and behavior.

Here is the presentation of the dilemma from http://en.wikipedia.org/wiki/Prisoner's_dilemma.

> Two suspects [Alice and Bob] are arrested by the police. The police have insufficient evidence for a conviction, and, having separated the prisoners, visit each of them to offer the same deal. If one testifies against the other (defects) and the other remains silent (cooperates), the defector goes free and the silent accomplice receives the full one-year sentence. If both remain silent, both prisoners are sentenced to only one month in jail for a minor charge. If each betrays the other, each receives a three-month sentence. Each prisoner must choose to betray the other or to remain silent. Each

> one is assured that the other would not know about the betrayal before the end of the investigation. How should the prisoners act?

Notice that in this context, "cooperate" means to keep silent, not to cooperate with police.

It is tempting to say that the players should cooperate with each other, since they would both be better off. But neither player knows what the other will do. Looking at it from Bob's point of view:

- If Alice remains silent, Bob is better off defecting.
- If Alice defects, Bob is better off defecting.

Either way, Bob is better off defecting. And from her point of view, Alice reaches the same conclusion. So if both players do the math, and no other factors come into play, we expect them to defect and be worse off for it.

This result is saddening because it is an example of how good intentions can lead to bad outcomes, and, unfortunately, it applies to other scenarios in real life, not just hypothetical prisoners.

But in real scenarios, the game is often iterated; that is, the same players face each other over and over, so they have the opportunity to learn, react, and communicate, at least implicitly.

The iterated version of the game is not as easy to analyze; it is not obvious what the optimal strategy is or even whether one exists.

So in the late 1970s Robert Axelrod organized a tournament to compare strategies. He invited participants to submit strategies in the form of computer programs, then played the programs against each other and kept score.

I won't tell you the outcome, and if you don't know you should resist the temptation to look it up. Instead, I encourage you to run your own tournament.

11.3 Game Theory

NOT DONE.

You have reached the end of the book. Congratulations! When you first read Chapter 1, some of the topics might not have made sense. You might find it helpful to read that chapter again now. Then get to work on your case study!

11.4 Exercises

Exercise 11.1 Download thinkcomplex.com/Referee.py, which runs the tournament, and thinkcomplex.com/PlayerFlipper.py, which implements a simple player strategy.

Here is the code from `PlayerFlipper.py`:

```
def move(history):
    mine, theirs = history
    if len(mine) % 2 == 0:
        return 'C'
    else:
        return 'D'
```

Any file that matches the pattern `Player*.py` is recognized as a player. The file should contain a definition for `move`, which takes the history of the match so far and returns a string: `'D'` for defect and `'C'` for cooperate.

`history` is a pair of lists: the first list contains the player's previous responses in order; the second contains the opponent's responses.

`PlayerFlipper` checks whether the number of previous rounds is even or odd and returns `'C'` or `'D'` respectively.

Write a `move` function in a file like `PlayerFlipper.py`, but replace "Flipper" with a name that summarizes your strategy.

Run `Referee.py` and see how your strategy does.

After you run your own tournament, you can read about the results of Axelrod's tournament in his book, *The Evolution of Cooperation*.

Appendix A

Analysis of algorithms

Analysis of algorithms is the branch of computer science that studies the performance of algorithms, especially their run time and space requirements. See http://en.wikipedia.org/wiki/Analysis_of_algorithms.

The practical goal of algorithm analysis is to predict the performance of different algorithms in order to guide design decisions.

During the 2008 United States Presidential Campaign, candidate Barack Obama was asked to perform an impromptu analysis when he visited Google. Chief executive Eric Schmidt jokingly asked him for "the most efficient way to sort a million 32-bit integers". Obama had apparently been tipped off, because he quickly replied, "I think the bubble sort would be the wrong way to go." See http://www.youtube.com/watch?v=k4RRi_ntQc8.

This is true: bubble sort is conceptually simple but slow for large datasets. The answer Schmidt was probably looking for is "radix sort" (see http://en.wikipedia.org/wiki/Radix_sort)[1].

So the goal of algorithm analysis is to make meaningful comparisons between algorithms, but there are some problems:

[1] But if you get a question like this in an interview, I think a better answer is, "The fastest way to sort a million integers is to use whatever sort function is provided by the language I'm using. Its performance is good enough for the vast majority of applications, but if it turned out that my application was too slow, I would use a profiler to see where the time was being spent. If it looked like a faster sort algorithm would have a significant effect on performance, then I would look around for a good implementation of radix sort."

- The relative performance of the algorithms might depend on characteristics of the hardware, so one algorithm might be faster on Machine A, another on Machine B. The general solution to this problem is to specify a **machine model** and analyze the number of steps, or operations, an algorithm requires under a given model.

- Relative performance might depend on the details of the dataset. For example, some sorting algorithms run faster if the data are already partially sorted; other algorithms run slower in this case. A common way to avoid this problem is to analyze the **worst case** scenario. It is also sometimes useful to analyze average case performance, but it is usually harder, and sometimes it is not clear what set of cases to average over.

- Relative performance also depends on the size of the problem. A sorting algorithm that is fast for small lists might be slow for long lists. The usual solution to this problem is to express run time (or number of operations) as a function of problem size, and to compare the functions **asymptotically** as the problem size increases.

The good thing about this kind of comparison that it lends itself to simple classification of algorithms. For example, if I know that the run time of Algorithm A tends to be proportional to the size of the input, n, and Algorithm B tends to be proportional to n^2, then I expect A to be faster than B for large values of n.

This kind of analysis comes with some caveats, but we'll get to that later.

A.1 Order of growth

Suppose you have analyzed two algorithms and expressed their run times in terms of the size of the input: Algorithm A takes $100n + 1$ steps to solve a problem with size n; Algorithm B takes $n^2 + n + 1$ steps.

The following table shows the run time of these algorithms for different problem sizes:

Input size	Run time of Algorithm A	Run time of Algorithm B
10	1 001	111
100	10 001	10 101
1 000	100 001	1 001 001
10 000	1 000 001	$> 10^{10}$

At $n = 10$, Algorithm A looks pretty bad; it takes almost 10 times longer than Algorithm B. But for $n = 100$ they are about the same, and for larger values A is much better.

The fundamental reason is that for large values of n, any function that contains an n^2 term will grow faster than a function whose leading term is n. The **leading term** is the term with the highest exponent.

For Algorithm A, the leading term has a large coefficient, 100, which is why B does better than A for small n. But regardless of the coefficients, there will always be some value of n where $an^2 > bn$.

The same argument applies to the non-leading terms. Even if the run time of Algorithm A were $n + 1000000$, it would still be better than Algorithm B for sufficiently large n.

In general, we expect an algorithm with a smaller leading term to be a better algorithm for large problems, but for smaller problems, there may be a **crossover point** where another algorithm is better. The location of the crossover point depends on the details of the algorithms, the inputs, and the hardware, so it is usually ignored for purposes of algorithmic analysis. But that doesn't mean you can forget about it.

If two algorithms have the same leading order term, it is hard to say which is better; again, the answer depends on the details. So for algorithmic analysis, functions with the same leading term are considered equivalent, even if they have different coefficients.

An **order of growth** is a set of functions whose asymptotic growth behavior is considered equivalent. For example, $2n$, $100n$ and $n + 1$ belong to the same

order of growth, which is written $O(n)$ in **Big-Oh notation** and often called **linear** because every function in the set grows linearly with n.

All functions with the leading term n^2 belong to $O(n^2)$; they are **quadratic**, which is a fancy word for functions with the leading term n^2.

The following table shows some of the orders of growth that appear most commonly in algorithmic analysis, in increasing order of badness.

Order of growth	Name
$O(1)$	constant
$O(\log_b n)$	logarithmic (for any b)
$O(n)$	linear
$O(n \log_b n)$	"en log en"
$O(n^2)$	quadratic
$O(n^3)$	cubic
$O(c^n)$	exponential (for any c)

For the logarithmic terms, the base of the logarithm doesn't matter; changing bases is the equivalent of multiplying by a constant, which doesn't change the order of growth. Similarly, all exponential functions belong to the same order of growth regardless of the base of the exponent. Exponential functions grow very quickly, so exponential algorithms are only useful for small problems.

Exercise A.1 Read the Wikipedia page on Big-Oh notation at http://en.wikipedia.org/wiki/Big_O_notation and answer the following questions:

1. What is the order of growth of $n^3 + n^2$? What about $1000000n^3 + n^2$? What about $n^3 + 1000000n^2$?

2. What is the order of growth of $(n^2 + n) \cdot (n + 1)$? Before you start multiplying, remember that you only need the leading term.

3. If f is in $O(g)$, for some unspecified function g, what can we say about $af + b$?

4. If f_1 and f_2 are in $O(g)$, what can we say about $f_1 + f_2$?

5. If f_1 is in $O(g)$ and f_2 is in $O(h)$, what can we say about $f_1 + f_2$?

6. If f_1 is in $O(g)$ and f_2 is $O(h)$, what can we say about $f_1 * f_2$?

Programmers who care about performance often find this kind of analysis hard to swallow. They have a point: sometimes the coefficients and the non-leading terms make a real difference. And sometimes the details of the hardware, the programming language, and the characteristics of the input make a big difference. And for small problems asymptotic behavior is irrelevant.

But if you keep those caveats in mind, algorithmic analysis is a useful tool. At least for large problems, the "better" algorithms is usually better, and sometimes it is *much* better. The difference between two algorithms with the same order of growth is usually a constant factor, but the difference between a good algorithm and a bad algorithm is unbounded!

A.2 Analysis of basic Python operations

Most arithmetic operations are constant time; multiplication usually takes longer than addition and subtraction, and division takes even longer, but these run times don't depend on the magnitude of the operands. Very large integers are an exception; in that case the run time increases with the number of digits.

Indexing operations—reading or writing elements in a sequence or dictionary—are also constant time, regardless of the size of the data structure.

A `for` loop that traverses a sequence or dictionary is usually linear, as long as all of the operations in the body of the loop are constant time. For example, adding up the elements of a list is linear:

```
total = 0
for x in t:
    total += x
```

The built-in function `sum` is also linear because it does the same thing, but it tends to be faster because it is a more efficient implementation; in the language of algorithmic analysis, it has a smaller leading coefficient.

If you use the same loop to "add" a list of strings, the run time is quadratic because string concatenation is linear.

The string method `join` is usually faster because it is linear in the total length of the strings.

As a rule of thumb, if the body of a loop is in $O(n^a)$ then the whole loop is in $O(n^{a+1})$. The exception is if you can show that the loop exits after a constant number of iterations. If a loop runs k times regardless of n, then the loop is in $O(n^a)$, even for large k.

Multiplying by k doesn't change the order of growth, but neither does dividing. So if the body of a loop is in $O(n^a)$ and it runs n/k times, the loop is in $O(n^{a+1})$, even for large k.

Most string and tuple operations are linear, except indexing and `len`, which are constant time. The built-in functions `min` and `max` are linear. The run-time of a slice operation is proportional to the length of the output, but independent of the size of the input.

All string methods are linear, but if the lengths of the strings are bounded by a constant—for example, operations on single characters—they are considered constant time.

Most list methods are linear, but there are some exceptions:

- Adding an element to the end of a list is constant time on average; when it runs out of room it occasionally gets copied to a bigger location, but the total time for n operations is $O(n)$, so we say that the "amortized" time for one operation is $O(1)$.
- Removing an element from the end of a list is constant time.
- Sorting is $O(n \log n)$.

Most dictionary operations and methods are constant time, but there are some exceptions:

- The run time of `copy` is proportional to the number of elements, but not the size of the elements (it copies references, not the elements themselves).

- The run time of `update` is proportional to the size of the dictionary passed as a parameter, not the dictionary being updated.

- `keys`, `values` and `items` are linear because they return new lists; `iterkeys`, `itervalues` and `iteritems` are constant time because they return iterators. But if you loop through the iterators, the loop will be linear. Using the "iter" functions saves some overhead, but it doesn't change the order of growth unless the number of items you access is bounded.

The performance of dictionaries is one of the minor miracles of computer science. We will see how they work in Section A.4.

Exercise A.2 Read the Wikipedia page on sorting algorithms at http://en.wikipedia.org/wiki/Sorting_algorithm and answer the following questions:

1. What is a "comparison sort?" What is the best worst-case order of growth for a comparison sort? What is the best worst-case order of growth for any sort algorithm?

2. What is the order of growth of bubble sort, and why does Barack Obama think it is "the wrong way to go?"

3. What is the order of growth of radix sort? What preconditions do we need to use it?

4. What is a stable sort and why might it matter in practice?

5. What is the worst sorting algorithm (that has a name)?

6. What sort algorithm does the C library use? What sort algorithm does Python use? Are these algorithms stable? You might have to Google around to find these answers.

7. Many of the non-comparison sorts are linear, so why does does Python use an $O(n \log n)$ comparison sort?

A.3 Analysis of search algorithms

A **search** is an algorithm that takes a collection and a target item and determines whether the target is in the collection, often returning the index of the target.

The simplest search algorithm is a "linear search", which traverses the items of the collection in order, stopping if it finds the target. In the worst case it has to traverse the entire collection, so the run time is linear.

The `in` operator for sequences uses a linear search; so do string methods like `find` and `count`.

If the elements of the sequence are in order, you can use a **bisection search**, which is $O(\log n)$. Bisection search is similar to the algorithm you probably use to look a word up in a dictionary (a real dictionary, not the data structure). Instead of starting at the beginning and checking each item in order, you start with the item in the middle and check whether the word you are looking for comes before or after. If it comes before, then you search the first half of the sequence. Otherwise you search the second half. Either way, you cut the number of remaining items in half.

If the sequence has 1,000,000 items, it will take about 20 steps to find the word or conclude that it's not there. So that's about 50,000 times faster than a linear search.

Exercise A.3 Write a function called `bisection` that takes a sorted list and a target value and returns the index of the value in the list, if it's there, or `None` if it's not.

Or you could read the documentation of the `bisect` module and use that!

Bisection search can be much faster than linear search, but it requires the sequence to be in order, which might require extra work.

There is another data structure, called a **hashtable** that is even faster—it can do a search in constant time—and it doesn't require the items to be sorted. Python dictionaries are implemented using hashtables, which is why most dictionary operations, including the `in` operator, are constant time.

A.4 Hashtables

To explain how hashtables work and why their performance is so good, I start with a simple implementation of a map and gradually improve it until it's a hashtable.

I use Python to demonstrate these implementations, but in real life you wouldn't write code like this in Python; you would just use a dictionary! So for the rest of this chapter, you have to imagine that dictionaries don't exist and you want to implement a data structure that maps from keys to values. The operations you have to implement are:

> Add a new item that maps from key `k` to value `v`. With a Python dictionary, `d`, this operation is written `d[k] = v`.
>
> Look up and return the value that corresponds to key `target`. With a Python dictionary, `d`, this operation is written `d[target]` or `d.get(target)`.

For now, I assume that each key only appears once. The simplest implementation of this interface uses a list of tuples, where each tuple is a key-value pair.

```
class LinearMap(object):

    def __init__(self):
        self.items = []

    def add(self, k, v):
        self.items.append((k, v))

    def get(self, k):
        for key, val in self.items:
            if key == k:
                return val
        raise KeyError
```

`add` appends a key-value tuple to the list of items, which takes constant time.

`get` uses a `for` loop to search the list: if it finds the target key it returns the corresponding value; otherwise it raises a `KeyError`. So `get` is linear.

An alternative is to keep the list sorted by key. Then `get` could use a bisection search, which is $O(\log n)$. But inserting a new item in the middle of a list is linear, so this might not be the best option. There are other data structures (see http://en.wikipedia.org/wiki/Red-black_tree) that can implement `add` and `get` in log time, but that's still not as good as constant time, so let's move on.

One way to improve `LinearMap` is to break the list of key-value pairs into smaller lists. Here's an implementation called `BetterMap`, which is a list of 100 LinearMaps. As we'll see in a second, the order of growth for `get` is still linear, but `BetterMap` is a step on the path toward hashtables:

```
class BetterMap(object):

    def __init__(self, n=100):
        self.maps = []
        for i in range(n):
            self.maps.append(LinearMap())

    def find_map(self, k):
        index = hash(k) % len(self.maps)
        return self.maps[index]

    def add(self, k, v):
        m = self.find_map(k)
        m.add(k, v)

    def get(self, k):
        m = self.find_map(k)
        return m.get(k)
```

`__init__` makes a list of `n` `LinearMaps`.

`find_map` is used by `add` and `get` to figure out which map to put the new item in, or which map to search.

`find_map` uses the built-in function `hash`, which takes almost any Python object and returns an integer. A limitation of this implementation is that it only works with hashable keys. Mutable types like lists and dictionaries are unhashable.

Hashable objects that are considered equal return the same hash value, but the converse is not necessarily true: two different objects can return the same hash value.

`find_map` uses the modulus operator to wrap the hash values into the range from 0 to `len(self.maps)`, so the result is a legal index into the list. Of course, this means that many different hash values will wrap onto the same index. But if the hash function spreads things out pretty evenly (which is what hash functions are designed to do), then we expect $n/100$ items per LinearMap.

Since the run time of `LinearMap.get` is proportional to the number of items, we expect BetterMap to be about 100 times faster than LinearMap. The order of growth is still linear, but the leading coefficient is smaller. That's nice, but still not as good as a hashtable.

Here (finally) is the crucial idea that makes hashtables fast: if you can keep the maximum length of the LinearMaps bounded, `LinearMap.get` is constant time. All you have to do is keep track of the number of items and when the number of items per LinearMap exceeds a threshold, resize the hashtable by adding more LinearMaps.

Here is an implementation of a hashtable:

```
class HashMap(object):

    def __init__(self):
        self.maps = BetterMap(2)
        self.num = 0

    def get(self, k):
        return self.maps.get(k)

    def add(self, k, v):
        if self.num == len(self.maps.maps):
            self.resize()

        self.maps.add(k, v)
        self.num += 1

    def resize(self):
        new_maps = BetterMap(self.num * 2)

        for m in self.maps.maps:
            for k, v in m.items:
                new_maps.add(k, v)

        self.maps = new_maps
```

Each `HashMap` contains a `BetterMap`; `__init__` starts with just 2 LinearMaps and initializes `num`, which keeps track of the number of items.

`get` just dispatches to `BetterMap`. The real work happens in `add`, which checks the number of items and the size of the `BetterMap`: if they are equal, the average number of items per LinearMap is 1, so it calls `resize`.

`resize` make a new `BetterMap`, twice as big as the previous one, and then "rehashes" the items from the old map to the new.

Rehashing is necessary because changing the number of LinearMaps changes the denominator of the modulus operator in `find_map`. That means that some objects that used to wrap into the same LinearMap will get split up (which is what we wanted, right?).

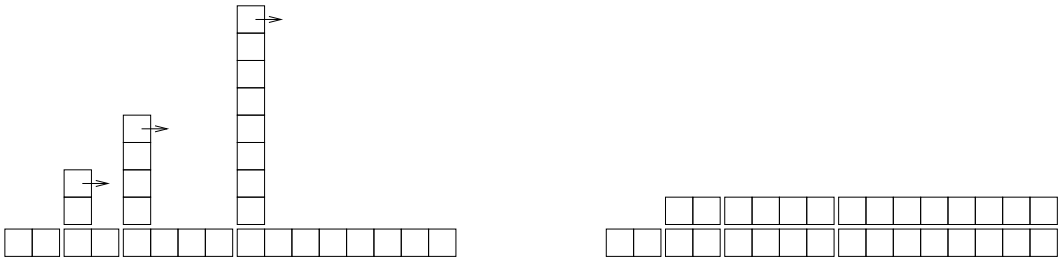

Figure A.1: The cost of a hashtable add.

Rehashing is linear, so `resize` is linear, which might seem bad, since I promised that `add` would be constant time. But remember that we don't have to resize every time, so `add` is usually constant time and only occasionally linear. The total amount of work to run `add` n times is proportional to n, so the average time of each `add` is constant time!

To see how this works, think about starting with an empty HashTable and adding a sequence of items. We start with 2 LinearMaps, so the first 2 adds are fast (no resizing required). Let's say that they take one unit of work each. The next add requires a resize, so we have to rehash the first two items (let's call that 2 more units of work) and then add the third item (one more unit). Adding the next item costs 1 unit, so the total so far is 6 units of work for 4 items.

The next `add` costs 5 units, but the next three are only one unit each, so the total is 14 units for the first 8 adds.

The next `add` costs 9 units, but then we can add 7 more before the next resize, so the total is 30 units for the first 16 adds.

After 32 adds, the total cost is 62 units, and I hope you are starting to see a pattern. After n adds, where n is a power of two, the total cost is $2n-2$ units, so the average work per add is a little less than 2 units. When n is a power of two, that's the best case; for other values of n the average work is a little higher, but that's not important. The important thing is that it is $O(1)$.

Figure A.1 shows how this works graphically. Each block represents a unit of work. The columns show the total work for each add in order from left to right: the first two `adds` cost 1 units, the third costs 3 units, etc.

The extra work of rehashing appears as a sequence of increasingly tall towers with increasing space between them. Now if you knock over the towers, amortizing the cost of resizing over all adds, you can see graphically that the total cost after n adds is $2n - 2$.

An important feature of this algorithm is that when we resize the HashTable it grows geometrically; that is, we multiply the size by a constant. If you increase the size arithmetically—adding a fixed number each time—the average time per `add` is linear.

You can download my implementation of HashMap from thinkcomplex.com/Map.py, but remember that there is no reason to use it; if you want a map, just use a Python dictionary.

Exercise A.4 My implementation of `HashMap` accesses the attributes of `BetterMap` directly, which shows poor object-oriented design.

1. The special method `__len__` is invoked by the built-in function `len`. Write a `__len__` method for `BetterMap` and use it in `add`.

2. Use a generator to write `BetterMap.iteritems`, and use it in `resize`.

Exercise A.5 A drawbacks of hashtables is that the elements have to be hashable, which usually means they have to be immutable. That's why, in Python, you can use tuples but not lists as keys in a dictionary. An alternative is to use a tree-based map.

Write an implementation of the map interface called `TreeMap` that uses a red-black tree to perform `add` and `get` in log time.

A.5 Summing lists

Suppose you have a bunch of lists and you want to join them up into a single list. There are three ways you might do that in Python:

- You could use the `+=` operator:

```
total = []
for x in t:
    total += x
```

- Or the `extend` method:

```
total = []
for x in t:
    total.extend(x)
```

- Or the built-in function `sum`:

```
total = sum(t, [])
```

  The second argument to `sum` is the initial value for the total.

Without knowing how `+=` and `extend` and `sum` are implemented, it is hard to analyze their performance. For example, if `total += x` creates a new list every time, the loop is quadratic; but if it modifies `total`, it's linear.

To find out, we could read the source code, but as an exercise, let's see if we can figure it out by measuring run times.

A simple way to measure the run time of a program is to use the function `times` in the `os` module, which returns a tuple of floats indicating the time your process has used (see the documentation for details). I use a function, `etime`, which returns the sum of "user time" and "system time" which is usually what we care about for performance measurement:

```
import os

def etime():
    """See how much user and system time this process has used
    so far and return the sum."""

    user, sys, chuser, chsys, real = os.times()
    return user+sys
```

To measure the elapsed time of a function you can call `etime` twice and compute the difference:

```
    start = etime()

    # put the code you want to measure here

    end = etime()
    elapsed = end - start
```

Alternatively, if you use IPython, you can use the `timeit` command. See ipython.scipy.org.

If an algorithm is quadratic, we expect the run time, t as a function of input size, n, to look like this:

$$t = an^2 + bn + c$$

Where a, b and c are unknown coefficients. If you take the log of both sides you get:

$$\log t \sim \log a + 2 \log n$$

For large values of n, the non-leading terms are insignificant and this approximation is pretty good. So if we plot t versus n on a log-log scale, we expect a straight line with slope 2.

Similarly if the algorithm is linear, we expect a line with slope 1.

I wrote three functions that concatenate lists: `sum_plus` uses `+=`; `sum_extend` uses `list.extend`; and `sum_sum` uses `sum`. I timed them for a range of `n` and plotted the results on a log-log scale. Figures A.2 and A.3 show the results.

In Figure A.2 I fit a line with slope 1 to the curves. The data fit this line well, so we conclude that these implementations are linear. The implementation for `+=` is faster by a constant factor because it takes some time to look up the `extend` method each time through the loop.

In Figure A.3 the data fit a line with slope 2, so the implementation of `sum` is quadratic.

A.6 `pyplot`

To make the figures in this section I used `pyplot`, which is part of `matplotlib`. If `matplotlib` is not part of your Python installation, you might have to install it, or you can use another library to make plots.

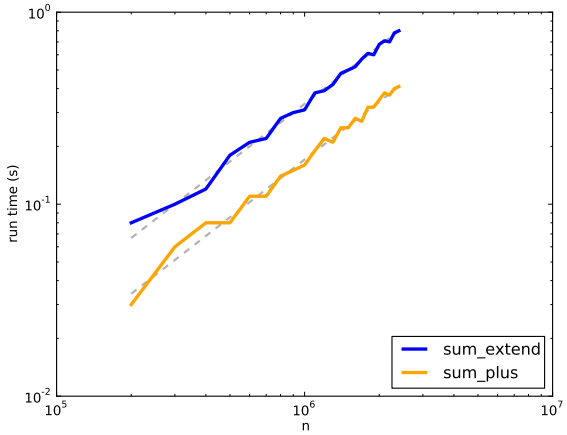

Figure A.2: Runtime versus `n`. The dashed lines have slope 1.

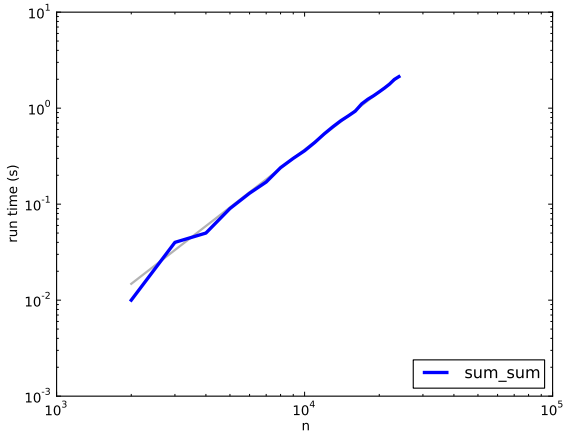

Figure A.3: Runtime versus `n`. The dashed line has slope 2.

Here's an example that makes a simple plot:

```
import matplotlib.pyplot as pyplot

pyplot.plot(xs, ys)
scale = 'log'
pyplot.xscale(scale)
pyplot.yscale(scale)
pyplot.title('')
pyplot.xlabel('n')
pyplot.ylabel('run time (s)')
pyplot.show()
```

The import statement makes `matplotlib.pyplot` accessible with the shorter name `pyplot`.

`plot` takes a list of x-values and a list of y-values and plots them. The lists have to have the same length. `xscale` and `yscale` make the axes either linear or logarithmic.

`title`, `xlabel` and `ylabel` are self-explanatory. Finally, `show` displays the plot on the screen. You could also use `savefig` to save the plot in a file.

Documentation of `pyplot` is at http://matplotlib.sourceforge.net/.

Exercise A.6 Test the performance of `LinearMap`, `BetterMap` and `HashMap`; see if you can characterize their order of growth.

You can download my map implementations from thinkcomplex.com/Map.py, and the code I used in this section from thinkcomplex.com/listsum.py.

You will have to find a range of `n` that is big enough to show asymptotic behavior, but small enough to run quickly.

Appendix B

Reading list

The following are selected books that provide an introduction to complexity science, and a pleasant way to get a big picture of the field.

- Axelrod, *Complexity of Cooperation.*
- Axelrod, *The Evolution of Cooperation.*
- Bak, *How Nature Works.*
- Barabasi, *Linked.*
- Buchanan, *Nexus.*
- Epstein and Axtell, *Growing Artificial Societies: Social Science from the Bottom Up.*
- Fisher, *The Perfect Swarm.*
- Flake, *The Computational Beauty of Nature.*
- Goodwin, *How the Leopard Changed Its Spots.*
- Holland, *Hidden Order.*
- Johnson, *Emergence.*
- Kelly, *Out of Control.*

- Kluger, *Simplexity*.
- Levy, *Artificial Life*.
- Lewin, *Complexity: Life at the Edge of Chaos*.
- Mitchell, *Complexity: A Guided Tour*.
- Mitchell Waldrop: Complexity, *the emerging science at the edge of order and chaos*.
- Resnick, *Turtles, Termites, and Traffic Jams*.
- Rucker, *The Lifebox, The Seashell, and the Soul*.
- Sawyer, *Social Emergence: Societies As Complex Systems*.
- Schelling, *Micromotives and Macrobehaviors*.
- Schiff, *Cellular Automata: A Discrete View of the World*.
- Strogatz, *Sync*.
- Watts, *Six Degrees*.
- Wolfram, *A New Kind Of Science*.