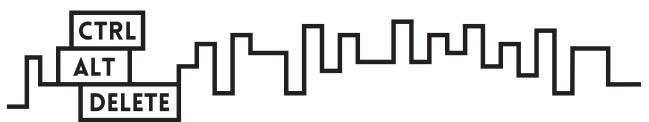


MADE
WITH
CREATIVE
COMMONS



MADE WITH CREATIVE COMMONS

PAUL STACEY AND SARAH HINCHLIFF PEARSON



Made With Creative Commons

by Paul Stacey & Sarah Hinchliff Pearson

© 2017, by Creative Commons.

Published under a Creative Commons Attribution-ShareAlike license (CC BY-SA), version 4.0.

ISBN 978-87-998733-3-3

Cover and interior design by Klaus Nielsen, vinterstille.dk

Content editing by Grace Yaginuma

Illustrations by Bryan Mathers, bryanmathers.com

Downloadable e-book available at madewith.cc

Publisher:

Ctrl+Alt+Delete Books

Husumgade 10, 5.

2200 Copenhagen N

Denmark

www.cadb.dk

hey@cadb.dk

Printer:

Drukarnia POZKAL Spółka z o.o. Spółka komandytowa

88-100 Inowrocław,

ul. Cegielna 10/12,

Poland

This book is published under a CC BY-SA license, which means that you can copy, redistribute, remix, transform, and build upon the content for any purpose, even commercially, as long as you give appropriate credit, provide a link to the license, and indicate if changes were made. If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original. License details: creativecommons.org/licenses/by-sa/4.0/

Made With Creative Commons is published with the kind support of Creative Commons and backers of our crowdfunding-campaign on the Kickstarter.com platform.

“I don’t know a whole lot about non-fiction journalism. . . The way that I think about these things, and in terms of what I can do is. . . essays like this are occasions to watch somebody reasonably bright but also reasonably average pay far closer attention and think at far more length about all sorts of different stuff than most of us have a chance to in our daily lives.”

- DAVID FOSTER WALLACE

CONTENTS

Foreword	xi
Introduction	xv

PART 1: THE BIG PICTURE

1 The New World of Digital Commons by Paul Stacey	3
The Commons, the Market, and the State	4
The Four Aspects of a Resource	5
A Short History of the Commons	7
The Digital Revolution	10
The Birth of Creative Commons	10
The Changing Market	11
Benefits of the Digital Commons	13
Our Case Studies	14
2 How to Be Made with Creative Commons by Sarah Hinchliff Pearson	19
Problem Zero: Getting Discovered	22
Making Money	26
Making Human Connections	30
3 The Creative Commons Licenses	39

PART 2: THE CASE STUDIES

Arduino	47
Ártica	51
Blender Institute	55
Cards Against Humanity	59
The Conversation	63
Cory Doctorow	67
Figshare	71
Figure.nz	75
Knowledge Unlatched	79
Lumen Learning	83
Jonathan Mann	87

Noun Project	91
Open Data Institute	95
Opendedesk.	99
OpenStax	105
Amanda Palmer	109
PLOS (Public Library of Science)	113
Rijksmuseum	117
Shareable	121
Siyavula	125
SparkFun	131
TeachAIDS.	135
Tribe of Noise.	139
Wikimedia Foundation	143
Bibliography	147
Acknowledgments	151

FOREWORD

Three years ago, just after I was hired as CEO of Creative Commons, I met with Cory Doctorow in the hotel bar of Toronto’s Gladstone Hotel. As one of CC’s most well-known proponents—one who has also had a successful career as a writer who shares his work using CC—I told him I thought CC had a role in defining and advancing open business models. He kindly disagreed, and called the pursuit of viable business models through CC “a red herring.”

He was, in a way, completely correct—those who make things with Creative Commons have ulterior motives, as Paul Stacey explains in this book: “Regardless of legal status, they all have a social mission. Their primary reason for being is to make the world a better place, not to profit. Money is a means to a social end, not the end itself.”

In the case study about Cory Doctorow, Sarah Hinchliff Pearson cites Cory’s words from his book *Information Doesn’t Want to Be Free*: “Entering the arts because you want to get rich is like buying lottery tickets because you want to get rich. It might work, but it almost certainly won’t. Though, of course, someone always wins the lottery.”

Today, copyright is like a lottery ticket—everyone has one, and almost nobody wins. What they don’t tell you is that if you choose to share your work, the returns can be significant and long-lasting. This book is filled with stories of those who take much greater risks than the two dollars we pay for a lottery ticket, and instead reap the rewards that come from pursuing their passions and living their values.

So it’s not about the money. Also: it is. Finding the means to continue to create and share often requires some amount of income. Max Temkin of Cards Against Humanity says it best

in their case study: “We don’t make jokes and games to make money—we make money so we can make more jokes and games.”

Creative Commons’ focus is on building a vibrant, usable commons, powered by collaboration and gratitude. Enabling communities of collaboration is at the heart of our strategy. With that in mind, Creative Commons began this book project. Led by Paul and Sarah, the project set out to define and advance the best open business models. Paul and Sarah were the ideal authors to write **Made with Creative Commons**.

Paul dreams of a future where new models of creativity and innovation overpower the inequality and scarcity that today define the worst parts of capitalism. He is driven by the power of human connections between communities of creators. He takes a longer view than most, and it’s made him a better educator, an insightful researcher, and also a skilled gardener. He has a calm, cool voice that conveys a passion that inspires his colleagues and community.

Sarah is the best kind of lawyer—a true advocate who believes in the good of people, and the power of collective acts to change the world. Over the past year I’ve seen Sarah struggle with the heartbreak that comes from investing so much into a political campaign that didn’t end as she’d hoped. Today, she’s more determined than ever to live with her values right out on her sleeve. I can always count on Sarah to push Creative Commons to focus on our impact—to make the main thing *the main thing*. She’s practical, detail-oriented, and clever. There’s no one on my team that I enjoy debating more.

As coauthors, Paul and Sarah complement each other perfectly. They researched, analyzed, argued, and worked as a team, sometimes together and sometimes independently. They dove into the research and writing with passion and curiosity, and a deep respect for what goes into building the commons and sharing with the world. They remained open to new ideas, including the possibility that their initial theories would need refinement or might be completely wrong. That's courageous, and it has made for a better book that is insightful, honest, and useful.

From the beginning, CC wanted to develop this project with the principles and values of open collaboration. The book was funded, developed, researched, and written in the open. It is being shared openly under a CC BY-SA license for anyone to use, remix, or adapt with attribution. It is, in itself, an example of an open business model.

For 31 days in August of 2015, Sarah took point to organize and execute a Kickstarter campaign to generate the core funding for the book. The remainder was provided by CC's generous donors and supporters. In the end, it became one of the most successful book projects on Kickstarter, smashing through two stretch goals and engaging over 1,600 donors—the majority of them new supporters of Creative Commons.

Paul and Sarah worked openly throughout the project, publishing the plans, drafts, case studies, and analysis, early and often, and they engaged communities all over the world to help write this book. As their opinions diverged and their interests came into focus, they divided their voices and decided to keep them separate in the final product. Working in this way requires both humility and self-confidence, and without question it has made *Made with Creative Commons* a better project.

Those who work and share in the commons are not typical creators. They are part of something greater than themselves, and what they offer us all is a profound gift. What they receive in return is gratitude and a community.

Jonathan Mann, who is profiled in this book, writes a song a day. When I reached out to ask him to write a song for our Kickstarter (and to offer himself up as a Kickstarter benefit), he agreed immediately. Why would he agree to do that? Because the commons has collaboration at its core, and community as a key value, and because the CC licenses have helped so many to share in the ways that they choose with a global audience.

Sarah writes, “Endeavors that are **Made with Creative Commons** thrive when community is built around what they do. This may mean a community collaborating together to create something new, or it may simply be a collection of like-minded people who get to know each other and rally around common interests or beliefs. To a certain extent, simply being **Made with Creative Commons** automatically brings with it some element of community, by helping connect you to like-minded others who recognize and are drawn to the values symbolized by using CC.” Amanda Palmer, the other musician profiled in the book, would surely add this from her case study: “There is no more satisfying end goal than having someone tell you that what you do is genuinely of value to them.”

This is not a typical business book. For those looking for a recipe or a roadmap, you might be disappointed. But for those looking to pursue a social end, to build something great through collaboration, or to join a powerful and growing global community, they're sure to be satisfied. *Made with Creative Commons* offers a world-changing set of clearly articulated values and principles, some essential tools for exploring your own business opportunities, and two dozen doses of pure inspiration.

In a 1996 *Stanford Law Review* article “*The Zones of Cyberspace*”, CC founder Lawrence Lessig wrote, “Cyberspace is a place. People live there. They experience all the sorts of things that they experience in real space, there. For

some, they experience more. They experience this not as isolated individuals, playing some high tech computer game; they experience it in groups, in communities, among strangers, among people they come to know, and sometimes like."

I'm incredibly proud that Creative Commons is able to publish this book for the many communities that we have come to know and like. I'm grateful to Paul and Sarah for their creativity and insights, and to the global communities that have helped us bring it to you. As CC board member Johnathan Nightingale often says, "It's all made of people."

That's the true value of things that are **Made with Creative Commons.**

*Ryan Merkley
CEO, Creative Commons*

INTRODUCTION

This book shows the world how sharing can be good for business—but with a twist.

We began the project intending to explore how creators, organizations, and businesses make money to sustain what they do when they share their work using Creative Commons licenses. Our goal was not to identify a formula for business models that use Creative Commons but instead gather fresh ideas and dynamic examples that spark new, innovative models and help others follow suit by building on what already works. At the onset, we framed our investigation in familiar business terms. We created a blank “open business model canvas,” an interactive online tool that would help people design and analyze their business model.

Through the generous funding of Kickstarter backers, we set about this project first by identifying and selecting a diverse group of creators, organizations, and businesses who use Creative Commons in an integral way—what we call being **Made with Creative Commons**. We interviewed them and wrote up their stories. We analyzed what we heard and dug deep into the literature.

But as we did our research, something interesting happened. Our initial way of framing the work did not match the stories we were hearing.

Those we interviewed were not typical businesses selling to consumers and seeking to maximize profits and the bottom line. Instead, they were sharing to make the world a better place, creating relationships and community around the works being shared, and generating revenue not for unlimited growth but to sustain the operation.

They often didn’t like hearing what they do described as an open business model. Their endeavor was something more than that. Something different. Something that generates not just economic value but social and cultural value. Something that involves human connection. Being **Made with Creative Commons** is not “business as usual.”

We had to rethink the way we conceived of this project. And it didn’t happen overnight. From the fall of 2015 through 2016, we documented our thoughts in blog posts on *Medium* and with regular updates to our Kickstarter backers. We shared drafts of case studies and analysis with our Kickstarter cocreators, who provided invaluable edits, feedback, and advice. Our thinking changed dramatically over the course of a year and a half.

Throughout the process, the two of us have often had very different ways of understanding and describing what we were learning. Learning from each other has been one of the great joys of this work, and, we hope, something that has made the final product much richer than it ever could have been if either of us undertook this project alone. We have preserved our voices throughout, and you’ll be able to sense our different but complementary approaches as you read through our different sections.

While we recommend that you read the book from start to finish, each section reads more or less independently. The book is structured into two main parts.

Part one, the overview, begins with a big-picture framework written by Paul. He provides some historical context for the digital commons, describing the three ways society

has managed resources and shared wealth—the commons, the market, and the state. He advocates for thinking beyond business and market terms and eloquently makes the case for sharing and enlarging the digital commons.

The overview continues with Sarah's chapter, as she considers what it means to be successfully **Made with Creative Commons**. While making money is one piece of the pie, there is also a set of public-minded values and the kind of human connections that make sharing truly meaningful. This section outlines the ways the creators, organizations, and businesses we interviewed bring in revenue, how they further the public interest and live out their values, and how they foster connections with the people with whom they share.

And to end part one, we have a short section that explains the different Creative Commons licenses. We talk about the misconception that the more restrictive licenses—the ones that are closest to the all-rights-reserved model of traditional copyright—are the only ways to make money.

Part two of the book is made up of the twenty-four stories of the creators, businesses, and organizations we interviewed. While both of us participated in the interviews, we divided up the writing of these profiles.

Of course, we are pleased to make the book available using a Creative Commons Attribution-ShareAlike license. Please copy, distribute, translate, localize, and build upon this work.

Writing this book has transformed and inspired us. The way we now look at and think about what it means to be **Made with Creative Commons** has irrevocably changed. We hope this book inspires you and your enterprise to use Creative Commons and in so doing contribute to the transformation of our economy and world for the better.

Paul and Sarah

Part 1

THE BIG PICTURE

THE NEW WORLD OF DIGITAL COMMONS



PAUL STACEY

Jonathan Rowe eloquently describes the commons as "the air and oceans, the web of species, wilderness and flowing water—all are parts of the commons. So are language and knowledge, sidewalks and public squares, the stories of childhood and the processes of democracy. Some parts of the commons are gifts of nature, others the product of human endeavor. Some are new, such as the Internet; others are as ancient as soil and calligraphy."¹

In **Made with Creative Commons**, we focus on our current era of digital commons, a commons of human-produced works. This commons cuts across a broad range of areas including cultural heritage, education, research, technology, art, design, literature, entertainment, business, and data. Human-produced works in all these areas are increasingly digital. The Internet is a kind of global, digital commons. The individuals, organizations, and businesses we profile in our case studies use

Creative Commons to share their resources online over the Internet.

The commons is not just about shared resources, however. It's also about the social practices and values that manage them. A resource is a noun, but *to common*—to put the resource into the commons—is a verb.² The creators, organizations, and businesses we profile are all engaged with commoning. Their use of Creative Commons involves them in the social practice of commoning, managing resources in a collective manner with a community of users.³ Commoning is guided by a set of values and norms that balance the costs and benefits of the enterprise with those of the community. Special regard is given to equitable access, use, and sustainability.

The Commons, the Market, and the State

Historically, there have been three ways to manage resources and share wealth: the commons (managed collectively), the state (i.e., the government), and the market—with the last two being the dominant forms today.⁴

The organizations and businesses in our case studies are unique in the way they participate in the commons while still engaging with the market and/or state. The extent of engagement with market or state varies. Some operate primarily as a commons with minimal or no reliance on the market or state.⁵ Others are very much a part of the market or state, depending on them for financial sustainability. All operate as hybrids, blending the norms of the commons with those of the market or state.

Fig. 1. is a depiction of how an enterprise can have varying levels of engagement with commons, state, and market.

Some of our case studies are simply commons and market enterprises with little or no engagement with the state. A depiction of those case studies would show the state sphere as

tiny or even absent. Other case studies are primarily market-based with only a small engagement with the commons. A depiction of those case studies would show the market sphere as large and the commons sphere as small. The extent to which an enterprise sees itself as being primarily of one type or another affects the balance of norms by which they operate.

All our case studies generate money as a means of livelihood and sustainability. Money is primarily of the market. Finding ways to generate revenue while holding true to the core values of the commons (usually expressed in mission statements) is challenging. To manage interaction and engagement between the commons and the market requires a deft touch, a strong sense of values, and the ability to blend the best of both.

The state has an important role to play in fostering the use and adoption of the commons. State programs and funding can deliberately contribute to and build the commons. Beyond money, laws and regulations regarding property, copyright, business, and finance can all be designed to foster the commons.

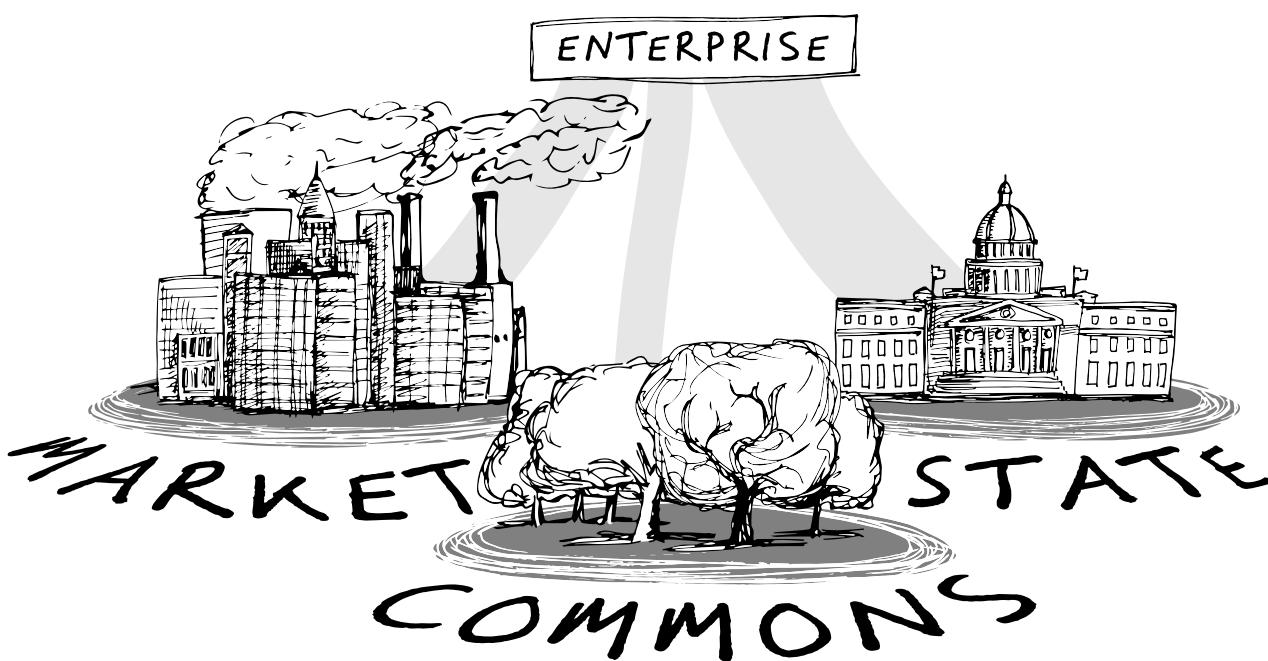


Fig. 1. Enterprise engagement with commons, state, and market.

It's helpful to understand how the commons, market, and state manage resources differently, and not just for those who consider themselves primarily as a commons. For businesses or governmental organizations who want to engage in and use the commons, knowing how the commons operates will help them understand how best to do so. Participating in and using the commons the same way you do the market or state is not a strategy for success.

The Four Aspects of a Resource

As part of her Nobel Prize-winning work, Elinor Ostrom developed a framework for analyzing how natural resources are managed in a commons.⁶ Her framework considered things like the biophysical characteristics of common resources, the community's actors and the interactions that take place between them, rules-in-use, and outcomes. That framework has been simplified and generalized to apply to the commons, the market, and the state for this chapter.

To compare and contrast the ways in which the commons, market, and state work, let's consider four aspects of resource management:



Fig. 2. Four aspects of resource management.

ment: resource characteristics, the people involved and the process they use, the norms and rules they develop to govern use, and finally actual resource use along with outcomes of that use (see Fig. 2).

Characteristics

Resources have particular characteristics or attributes that affect the way they can be used. Some resources are natural; others are human produced. And—significantly for today's commons—resources can be physical or digital, which affects a resource's inherent potential.

Physical resources exist in limited supply. If I have a physical resource and give it to you, I no longer have it. When a resource is removed and used, the supply becomes scarce or depleted. Scarcity can result in competing rivalry for the resource. **Made with Creative Commons** enterprises are usually digitally based but some of our case studies also produce resources in physical form. The costs of producing and distributing a physical good usually require them to engage with the market.

Physical resources are depletable, exclusive, and rivalrous. Digital resources, on the other hand, are nondepletable, nonexclusive, and nonrivalrous. If I share a digital resource with you, we both have the resource. Giving it to you does not mean I no longer have it. Digital resources can be infinitely stored, copied, and distributed without becoming depleted, and at close to zero cost. Abundance rather than scarcity is an inherent characteristic of digital resources.

The nondepletable, nonexclusive, and nonrivalrous nature of digital resources means the rules and norms for managing them can (and ought to) be different from how physical resources are managed. However, this is not always the case. Digital resources are frequently made artificially scarce. Placing digital resources in the commons makes them free and abundant.

Our case studies frequently manage hybrid resources, which start out as digital with the possibility of being made into a physical resource. The digital file of a book can be print-

ed on paper and made into a physical book. A computer-rendered design for furniture can be physically manufactured in wood. This conversion from digital to physical invariably has costs. Often the digital resources are managed in a free and open way, but money is charged to convert a digital resource into a physical one.

Beyond this idea of physical versus digital, the commons, market, and state conceive of resources differently (see Fig. 3). The market sees resources as private goods—commodities for sale—from which value is extracted. The state sees resources as public goods that provide value to state citizens. The commons sees resources as common goods, providing a common wealth extending beyond state boundaries, to be passed on in undiminished or enhanced form to future generations.

People and processes

In the commons, the market, and the state, different people and processes are used to manage resources. The processes used define both who has a say and how a resource is managed.

In the state, a government of elected officials is responsible for managing resources on behalf of the public. The citizens who produce and use those resources are not directly involved; instead, that responsibility is given over to the government. State ministries and departments staffed with public servants set budgets, implement programs, and manage

resources based on government priorities and procedures.

In the market, the people involved are producers, buyers, sellers, and consumers. Businesses act as intermediaries between those who produce resources and those who consume or use them. Market processes seek to extract as much monetary value from resources as possible. In the market, resources are managed as commodities, frequently mass-produced, and sold to consumers on the basis of a cash transaction.

In contrast to the state and market, resources in a commons are managed more directly by the people involved.⁷ Creators of human produced resources can put them in the commons by personal choice. No permission from state or market is required. Anyone can participate in the commons and determine for themselves the extent to which they want to be involved—as a contributor, user, or manager. The people involved include not only those who create and use resources but those affected by outcome of use. Who you are affects your say, actions you can take, and extent of decision making. In the commons, the community as a whole manages the resources. Resources put into the commons using Creative Commons require users to give the original creator credit. Knowing the person behind a resource makes the commons less anonymous and more personal.



Fig. 3. How the market, commons, and state conceive of resources.

Norms and rules

The social interactions between people, and the processes used by the state, market, and commons, evolve social norms and rules. These norms and rules define permissions, allocate entitlements, and resolve disputes.

State authority is governed by national constitutions. Norms related to priorities and decision making are defined by elected officials and parliamentary procedures. State rules are expressed through policies, regulations, and laws. The state influences the norms and rules of the market and commons through the rules it passes.

Market norms are influenced by economics and competition for scarce resources. Market rules follow property, business, and financial laws defined by the state.

As with the market, a commons can be influenced by state policies, regulations, and laws. But the norms and rules of a commons are largely defined by the community. They weigh individual costs and benefits against the costs and benefits to the whole community. Consideration is given not just to economic efficiency but also to equity and sustainability.⁹

Goals

The combination of the aspects we've discussed so far—the resource's inherent characteristics, people and processes, and norms and rules—shape how resources are used. Use is also influenced by the different goals the state, market, and commons have.

In the market, the focus is on maximizing the utility of a resource. What we pay for the goods we consume is seen as an objective measure of the utility they provide. The goal then becomes maximizing total monetary value in the economy.¹⁰ Units consumed translates to sales, revenue, profit, and growth, and these are all ways to measure goals of the market.

The state aims to use and manage resources in a way that balances the economy with the social and cultural needs of its citizens. Health care, education, jobs, the environment, transportation, security, heritage, and justice are all facets of a healthy society, and the state

applies its resources toward these aims. State goals are reflected in quality of life measures.

In the commons, the goal is maximizing access, equity, distribution, participation, innovation, and sustainability. You can measure success by looking at how many people access and use a resource; how users are distributed across gender, income, and location; if a community to extend and enhance the resources is being formed; and if the resources are being used in innovative ways for personal and social good.

As hybrid combinations of the commons with the market or state, the success and sustainability of all our case study enterprises depends on their ability to strategically utilize and balance these different aspects of managing resources.

A Short History of the Commons

Using the commons to manage resources is part of a long historical continuum. However, in contemporary society, the market and the state dominate the discourse on how resources are best managed. Rarely is the commons even considered as an option. The commons has largely disappeared from consciousness and consideration. There are no news reports or speeches about the commons.

But the more than 1.1 billion resources licensed with Creative Commons around the world are indications of a grassroots move toward the commons. The commons is making a resurgence. To understand the resilience of the commons and its current renewal, it's helpful to know something of its history.

For centuries, indigenous people and pre-industrialized societies managed resources, including water, food, firewood, irrigation, fish, wild game, and many other things collectively as a commons.¹¹ There was no market, no global economy. The state in the form of rulers influenced the commons but by no means controlled it. Direct social participation in a commons was the primary way in which resources were managed and needs met. (Fig. 4 illustrates the commons in relation to the state and the market.)

LONG AGO:

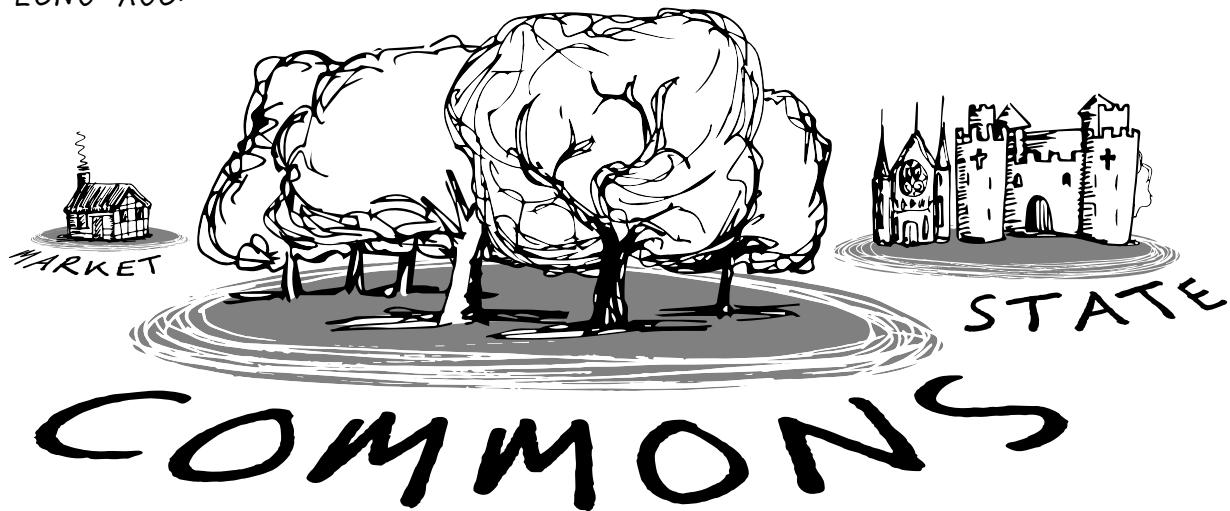


Fig. 4. In preindustrialized society.

This is followed by a long history of the state (a monarchy or ruler) taking over the commons for their own purposes. This is called enclosure of the commons.¹² In olden days, "commoners" were evicted from the land, fences and hedges erected, laws passed, and security set up to forbid access.¹³ Gradually, resources became the property of the state and the state became the primary means by which resources were managed. (See Fig. 5).

Holdings of land, water, and game were distributed to ruling family and political appointees. Commoners displaced from the land

migrated to cities. With the emergence of the industrial revolution, land and resources became commodities sold to businesses to support production. Monarchies evolved into elected parliaments. Commoners became labourers earning money operating the machinery of industry. Financial, business, and property laws were revised by governments to support markets, growth, and productivity. Over time ready access to market produced goods resulted in a rising standard of living, improved health, and education. Fig. 6 shows how today the market

STATE TAKEOVER OF THE COMMONS:



Fig. 5. The commons is gradually superseded by the state.

is the primary means by which resources are managed.

However, the world today is going through turbulent times. The benefits of the market have been offset by unequal distribution and overexploitation.

Overexploitation was the topic of Garrett Hardin's influential essay "The Tragedy of the Commons," published in *Science* in 1968. Hardin argues that everyone in a commons seeks to maximize personal gain and will continue to do so even when the limits of the commons are reached. The commons is then tragically depleted to the point where it can no longer support anyone. Hardin's essay became widely accepted as an economic truism and a justification for private property and free markets.

However, there is one serious flaw with Hardin's "The Tragedy of the Commons"—it's fiction. Hardin did not actually study how real commons work. Elinor Ostrom won the 2009 Nobel Prize in economics for her work studying different commons all around the world. Ostrom's work shows that natural resource commons can be successfully managed by local communities without any regulation by central authorities or without privatization. Government and privatization are not the only two choices. There is a third way: management by the people, where those that are directly impacted are

directly involved. With natural resources, there is a regional locality. The people in the region are the most familiar with the natural resource, have the most direct relationship and history with it, and are therefore best situated to manage it. Ostrom's approach to the governance of natural resources broke with convention; she recognized the importance of the commons as an alternative to the market or state for solving problems of collective action.¹⁴

Hardin failed to consider the actual social dynamic of the commons. His model assumed that people in the commons act autonomous-ly, out of pure self-interest, without interaction or consideration of others. But as Ostrom found, in reality, managing common resources together forms a community and encourages discourse. This naturally generates norms and rules that help people work collectively and ensure a sustainable commons. Paradoxically, while Hardin's essay is called The Tragedy of the Commons it might more accurately be titled The Tragedy of the Market.

Hardin's story is based on the premise of depletable resources. Economists have focused almost exclusively on scarcity-based markets. Very little is known about how abundance works.¹⁵ The emergence of information technology and the Internet has led to an explosion in digital resources and new means of sharing

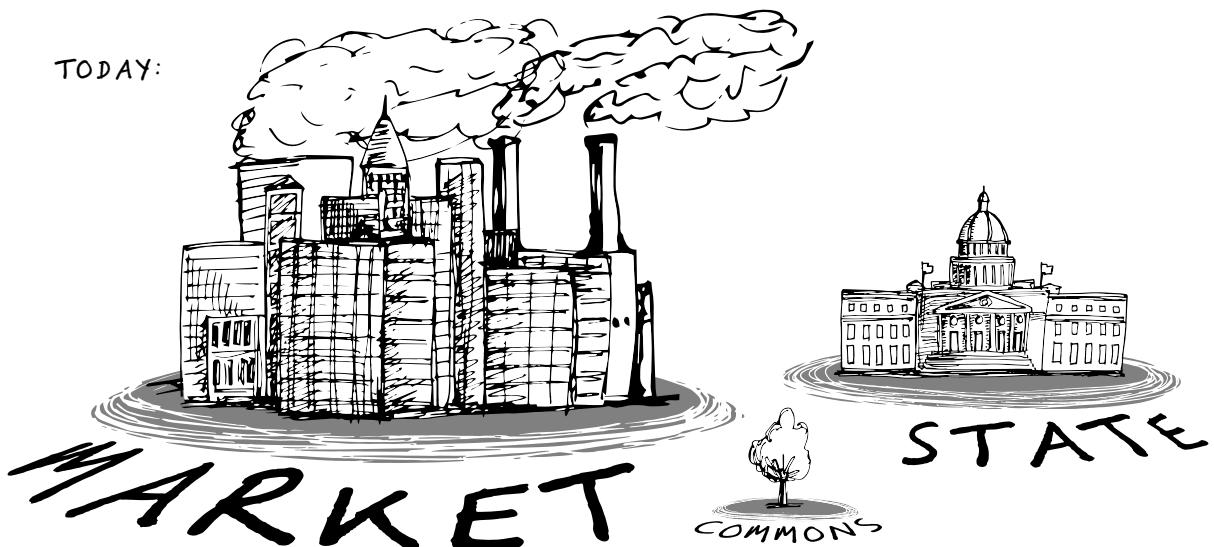


Fig. 6. How the market, the state, and the commons look today.

and distribution. Digital resources can never be depleted. An absence of a theory or model for how abundance works, however, has led the market to make digital resources artificially scarce and makes it possible for the usual market norms and rules to be applied.

When it comes to use of state funds to create digital goods, however, there is really no justification for artificial scarcity. The norm for state funded digital works should be that they are freely and openly available to the public that paid for them.

The Digital Revolution

In the early days of computing, programmers and developers learned from each other by sharing software. In the 1980s, the free-software movement codified this practice of sharing into a set of principles and freedoms:

- The freedom to run a software program as you wish, for any purpose.
- The freedom to study how a software program works (because access to the source code has been freely given), and change it so it does your computing as you wish.
- The freedom to redistribute copies.
- The freedom to distribute copies of your modified versions to others.¹⁶

These principles and freedoms constitute a set of norms and rules that typify a digital commons.

In the late 1990s, to make the sharing of source code and collaboration more appealing to companies, the open-source-software initiative converted these principles into licenses and standards for managing access to and distribution of software. The benefits of open source—such as reliability, scalability, and quality verified by independent peer review—became widely recognized and accepted. Customers liked the way open source gave them control without being locked into a closed, proprietary technology. Free and

open-source software also generated a network effect where the value of a product or service increases with the number of people using it.¹⁷ The dramatic growth of the Internet itself owes much to the fact that nobody has a proprietary lock on core Internet protocols.

While open-source software functions as a commons, many businesses and markets did build up around it. Business models based on the licenses and standards of open-source software evolved alongside organizations that managed software code on principles of abundance rather than scarcity. Eric Raymond's essay "The Magic Cauldron" does a great job of analyzing the economics and business models associated with open-source software.¹⁸ These models can provide examples of sustainable approaches for those **Made with Creative Commons**.

It isn't just about an abundant availability of digital assets but also about abundance of participation. The growth of personal computing, information technology, and the Internet made it possible for mass participation in producing creative works and distributing them. Photos, books, music, and many other forms of digital content could now be readily created and distributed by almost anyone. Despite this potential for abundance, by default these digital works are governed by copyright laws. Under copyright, a digital work is the property of the creator, and by law others are excluded from accessing and using it without the creator's permission.

But people like to share. One of the ways we define ourselves is by sharing valuable and entertaining content. Doing so grows and nourishes relationships, seeks to change opinions, encourages action, and informs others about who we are and what we care about. Sharing lets us feel more involved with the world.¹⁹

The Birth of Creative Commons

In 2001, Creative Commons was created as a nonprofit to support all those who wanted to share digital content. A suite of Creative Commons licenses was modeled on those of open-source software but for use with digital con-

tent rather than software code. The licenses give everyone from individual creators to large companies and institutions a simple, standardized way to grant copyright permissions to their creative work.

Creative Commons licenses have a three-layer design. The norms and rules of each license are first expressed in full legal language as used by lawyers. This layer is called the *legal code*. But since most creators and users are not lawyers, the licenses also have a *commons deed*, expressing the permissions in plain language, which regular people can read and quickly understand. It acts as a user-friendly interface to the legal-code layer beneath. The third layer is the machine-readable one, making it easy for the Web to know a work is Creative Commons-licensed by expressing permissions in a way that software systems, search engines, and other kinds of technology can understand.²⁰ Taken together, these three layers ensure creators, users, and even the Web itself understand the norms and rules associated with digital content in a commons.

In 2015, there were over one billion Creative Commons licensed works in a global commons. These works were viewed online 136 billion times. People are using Creative Commons licenses all around the world, in thirty-four languages. These resources include photos, artwork, research articles in journals, educational resources, music and other audio tracks, and videos.

Individual artists, photographers, musicians, and filmmakers use Creative Commons, but so do museums, governments, creative industries, manufacturers, and publishers. Millions of websites use CC licenses, including major platforms like Wikipedia and Flickr and smaller ones like blogs.²¹ Users of Creative Commons are diverse and cut across many different sectors. (Our case studies were chosen to reflect that diversity.)

Some see Creative Commons as a way to share a gift with others, a way of getting known, or a way to provide social benefit. Others are simply committed to the norms associated with a commons. And for some, partic-

ipation has been spurred by the free-culture movement, a social movement that promotes the freedom to distribute and modify creative works. The free-culture movement sees a commons as providing significant benefits compared to restrictive copyright laws. This ethos of free exchange in a commons aligns the free-culture movement with the free and open-source software movement.

Over time, Creative Commons has spawned a range of open movements, including open educational resources, open access, open science, and open data. The goal in every case has been to democratize participation and share digital resources at no cost, with legal permissions for anyone to freely access, use, and modify.

The state is increasingly involved in supporting open movements. The Open Government Partnership was launched in 2011 to provide an international platform for governments to become more open, accountable, and responsive to citizens. Since then, it has grown from eight participating countries to seventy.²² In all these countries, government and civil society are working together to develop and implement ambitious open-government reforms. Governments are increasingly adopting Creative Commons to ensure works funded with taxpayer dollars are open and free to the public that paid for them.

The Changing Market

Today's market is largely driven by global capitalism. Law and financial systems are structured to support extraction, privatization, and corporate growth. A perception that the market is more efficient than the state has led to continual privatization of many public natural resources, utilities, services, and infrastructures.²³ While this system has been highly efficient at generating consumerism and the growth of gross domestic product, the impact on human well-being has been mixed. Offsetting rising living standards and improvements to health and education are ever-increasing wealth inequality, social inequality, poverty,

deterioration of our natural environment, and breakdowns of democracy.²⁴

In light of these challenges there is a growing recognition that GDP growth should not be an end in itself, that development needs to be socially and economically inclusive, that environmental sustainability is a requirement not an option, and that we need to better balance the market, state and community.²⁵

These realizations have led to a resurgence of interest in the commons as a means of enabling that balance. City governments like Bologna, Italy, are collaborating with their citizens to put in place regulations for the care and regeneration of urban commons.²⁶ Seoul and Amsterdam call themselves “sharing cities,” looking to make sustainable and more efficient use of scarce resources. They see sharing as a way to improve the use of public spaces, mobility, social cohesion, and safety.²⁷

The market itself has taken an interest in the sharing economy, with businesses like Airbnb providing a peer-to-peer marketplace for short-term lodging and Uber providing a platform for ride sharing. However, Airbnb and Uber are still largely operating under the usual norms and rules of the market, making them less like a commons and more like a traditional business seeking financial gain. Much of the sharing economy is not about the commons or building an alternative to a corporate-driven market economy; it’s about extending the deregulated free market into new areas of our lives.²⁸ While none of the people we interviewed for our case studies would describe themselves as part of the sharing economy, there are in fact some significant parallels. Both the sharing economy and the commons make better use of asset capacity. The sharing economy sees personal residents and cars as having latent spare capacity with rental value. The equitable access of the commons broadens and diversifies the number of people who can use and derive value from an asset.

One way **Made with Creative Commons** case studies differ from those of the sharing economy is their focus on *digital* resources. Digital resources function under different

economic rules than physical ones. In a world where prices always seem to go up, information technology is an anomaly. Computer-processing power, storage, and bandwidth are all rapidly increasing, but rather than costs going up, costs are coming down. Digital technologies are getting faster, better, and cheaper. The cost of anything built on these technologies will always go down until it is close to zero.²⁹

Those that are **Made with Creative Commons** are looking to leverage the unique inherent characteristics of digital resources, including lowering costs. The use of digital-rights-management technologies in the form of locks, passwords, and controls to prevent digital goods from being accessed, changed, replicated, and distributed is minimal or nonexistent. Instead, Creative Commons licenses are used to put digital content out in the commons, taking advantage of the unique economics associated with being digital. The aim is to see digital resources used as widely and by as many people as possible. Maximizing access and participation is a common goal. They aim for abundance over scarcity.

The incremental cost of storing, copying, and distributing digital goods is next to zero, making abundance possible. But imagining a market based on abundance rather than scarcity is so alien to the way we conceive of economic theory and practice that we struggle to do so.³⁰ Those that are **Made with Creative Commons** are each pioneering in this new landscape, devising their own economic models and practice.

Some are looking to minimize their interactions with the market and operate as autonomously as possible. Others are operating largely as a business within the existing rules and norms of the market. And still others are looking to change the norms and rules by which the market operates.

For an ordinary corporation, making social benefit a part of its operations is difficult, as it’s legally required to make decisions that financially benefit stockholders. But new forms of business are emerging. There are benefit corporations and social enterprises, which

broaden their business goals from making a profit to making a positive impact on society, workers, the community, and the environment.³¹ Community-owned businesses, worker-owned businesses, cooperatives, guilds, and other organizational forms offer alternatives to the traditional corporation. Collectively, these alternative market entities are changing the rules and norms of the market.³²

"A book on open business models" is how we described it in this book's Kickstarter campaign. We used a handbook called *Business Model Generation* as our reference for defining just what a business model is. Developed over nine years using an "open process" involving 470 coauthors from forty-five countries, it is useful as a framework for talking about business models.³³

It contains a "business model canvas," which conceives of a business model as having nine building blocks.³⁴ This blank canvas can serve as a tool for anyone to design their own business model. We remixed this business model canvas into an *open* business model canvas, adding three more building blocks relevant to hybrid market, commons enterprises: *social good*, *Creative Commons license*, and "*type of open environment that the business fits in.*"³⁵ This enhanced canvas proved useful when we analyzed businesses and helped start-ups plan their economic model.

In our case study interviews, many expressed discomfort over describing themselves as an open business model—the term *business model* suggested primarily being situated in the market. Where you sit on the commons-to-market spectrum affects the extent to which you see yourself as a business in the market. The more central to the mission shared resources and commons values are, the less comfort there is in describing yourself, or depicting what you do, as a *business*. Not all who have endeavors **Made with Creative Commons** use business speak; for some the process has been experimental, emergent, and organic rather than carefully planned using a predefined model.

The creators, businesses, and organizations we profile all engage with the market to generate revenue in some way. The ways in which this is done vary widely. Donations, pay what you can, memberships, "digital for free but physical for a fee," crowdfunding, matchmaking, value-add services, patrons...the list goes on and on. (Initial description of how to earn revenue available through reference note. For latest thinking see How to Bring In Money in the next section.)³⁶ There is no single magic bullet, and each endeavor has devised ways that work for them. Most make use of more than one way. Diversifying revenue streams lowers risk and provides multiple paths to sustainability.

Benefits of the Digital Commons

While it may be clear why commons-based organizations want to interact and engage with the market (they need money to survive), it may be less obvious why the market would engage with the commons. The digital commons offers many benefits.

The commons speeds dissemination. The free flow of resources in the commons offers tremendous economies of scale. Distribution is decentralized, with all those in the commons empowered to share the resources they have access to. Those that are **Made with Creative Commons** have a reduced need for sales or marketing. Decentralized distribution amplifies supply and know-how.

The commons ensures access to all. The market has traditionally operated by putting resources behind a paywall requiring payment first before access. The commons puts resources in the open, providing access up front without payment. Those that are **Made with Creative Commons** make little or no use of digital rights management (DRM) to manage resources. Not using DRM frees them of the costs of acquiring DRM technology and staff resources to engage in the punitive practices associated with restricting access. The way the commons provides access to everyone levels the playing field and promotes inclusiveness, equity, and fairness.

The commons maximizes participation. Resources in the commons can be used and contributed to by everyone. Using the resources of others, contributing your own, and mixing yours with others to create new works are all dynamic forms of participation made possible by the commons. Being **Made with Creative Commons** means you're engaging as many users with your resources as possible. Users are also authoring, editing, remixing, curating, localizing, translating, and distributing. The commons makes it possible for people to directly participate in culture, knowledge building, and even democracy, and many other socially beneficial practices.

The commons spurs innovation. Resources in the hands of more people who can use them leads to new ideas. The way commons resources can be modified, customized, and improved results in derivative works never imagined by the original creator. Some endeavors that are **Made with Creative Commons** deliberately encourage users to take the resources being shared and innovate them. Doing so moves research and development (R&D) from being solely inside the organization to being in the community.³⁷ Community-based innovation will keep an organization or business on its toes. It must continue to contribute new ideas, absorb and build on top of the innovations of others, and steward the resources and the relationship with the community.

The commons boosts reach and impact. The digital commons is global. Resources may be created for a local or regional need, but they go far and wide generating a global impact. In the digital world, there are no borders between countries. When you are **Made with Creative Commons**, you are often local and global at the same time: Digital designs being globally distributed but made and manufactured locally. Digital books or music being globally distributed but readings and concerts performed locally. The digital commons magnifies impact by connecting creators to those who use and build on their work both locally and globally.

The commons is generative. Instead of extracting value, the commons adds value. Dig-

itized resources persist without becoming depleted, and through use are improved, personalized, and localized. Each use adds value. The market focuses on generating value for the business and the customer. The commons generates value for a broader range of beneficiaries including the business, the customer, the creator, the public, and the commons itself. The generative nature of the commons means that it is more cost-effective and produces a greater return on investment. Value is not just measured in financial terms. Each new resource added to the commons provides value to the public and contributes to the overall value of the commons.

The commons brings people together for a common cause. The commons vests people directly with the responsibility to manage the resources for the common good. The costs and benefits for the individual are balanced with the costs and benefits for the community and for future generations. Resources are not anonymous or mass produced. Their provenance is known and acknowledged through attribution and other means. Those that are **Made with Creative Commons** generate awareness and reputation based on their contributions to the commons. The reach, impact, and sustainability of those contributions rest largely on their ability to forge relationships and connections with those who use and improve them. By functioning on the basis of social engagement, not monetary exchange, the commons unifies people.

The benefits of the commons are many. When these benefits align with the goals of individuals, communities, businesses in the market, or state enterprises, choosing to manage resources as a commons ought to be the option of choice.

Our Case Studies

The creators, organizations, and businesses in our case studies operate as nonprofits, for-profits, and social enterprises. Regardless of legal status, they all have a social mission. Their primary reason for being is to make the world a better place, not to profit. Money is a

means to a social end, not the end itself. They factor public interest into decisions, behavior, and practices. Transparency and trust are really important. Impact and success are measured against social aims expressed in mission statements, and are not just about the financial bottom line.

The case studies are based on the narratives told to us by founders and key staff. Instead of solely using financials as the measure of success and sustainability, they emphasized their mission, practices, and means by which they measure success. Metrics of success are a blend of how social goals are being met and how sustainable the enterprise is.

Our case studies are diverse, ranging from publishing to education and manufacturing. All of the organizations, businesses, and creators in the case studies produce digital resources. Those resources exist in many forms including books, designs, songs, research, data, cultural works, education materials, graphic icons, and video. Some are digital representations of physical resources. Others are born digital but can be made into physical resources.

They are creating new resources, or using the resources of others, or mixing existing resources together to make something new. They, and their audience, all play a direct, participatory role in managing those resources, including their preservation, curation, distribution, and enhancement. Access and participation is open to all regardless of monetary means.

And as users of Creative Commons licenses, they are automatically part of a global community. The new digital commons is global. Those we profiled come from nearly every continent in the world. To build and interact within this global community is conducive to success.

Creative Commons licenses may express legal rules around the use of resources in a commons, but success in the commons requires more than following the letter of the law and acquiring financial means. Over and over we heard in our interviews how success and sustainability are tied to a set of beliefs, values, and principles that underlie their actions:

Give more than you take. Be open and inclusive. Add value. Make visible what you are using from the commons, what you are adding, and what you are monetizing. Maximize abundance. Give attribution. Express gratitude. Develop trust; don't exploit. Build relationship and community. Be transparent. Defend the commons.

The new digital commons is here to stay. **Made With Creative Commons** case studies show how it's possible to be part of this commons while still functioning within market and state systems. The commons generates benefits neither the market nor state can achieve on their own. Rather than the market or state dominating as primary means of resource management, a more balanced alternative is possible.

Enterprise use of Creative Commons has only just begun. The case studies in this book are merely starting points. Each is changing and evolving over time. Many more are joining and inventing new models. This overview aims to provide a framework and language for thinking and talking about the new digital commons. The remaining sections go deeper providing further guidance and insights on how it works.

Notes

- 1 Jonathan Rowe, *Our Common Wealth* (San Francisco: Berrett-Koehler, 2013), 14.
- 2 David Bollier, *Think Like a Commoner: A Short Introduction to the Life of the Commons* (Gabriola Island, BC: New Society, 2014), 176.
- 3 Ibid., 15.
- 4 Ibid., 145.
- 5 Ibid., 175.
- 6 Daniel H. Cole, "Learning from Lin: Lessons and Cautions from the Natural Commons for the Knowledge Commons," in *Governing Knowledge Commons*, eds. Brett M. Frischmann, Michael J. Madison, and Katherine J. Strandburg (New York: Oxford University Press, 2014), 53.
- 7 Max Haiven, *Crises of Imagination, Crises of Power: Capitalism, Creativity and the Commons* (New York: Zed Books, 2014), 93.
- 8 Cole, "Learning from Lin," in Frischmann, Madison, and Strandburg, *Governing Knowledge Commons*, 59.
- 9 Bollier, *Think Like a Commoner*, 175.
- 10 Joshua Farley and Ida Kubiszewski, "The Economics of Information in a Post-Carbon Economy," in *Free Knowledge: Confronting the Commodification of Human Discovery*, eds. Patricia W. Elliott and Daryl H. Hepting (Regina, SK: University of Regina Press, 2015), 201–4.
- 11 Rowe, *Our Common Wealth*, 19; and Heather Menzies, *Reclaiming the Commons for the Common Good: A Memoir and Manifesto* (Gabriola Island, BC: New Society, 2014), 42–43.
- 12 Bollier, *Think Like a Commoner*, 55–78.
- 13 Fritjof Capra and Ugo Mattei, *The Ecology of Law: Toward a Legal System in Tune with Nature and Community* (Oakland, CA: Berrett-Koehler, 2015), 46–57; and Bollier, *Think Like a Commoner*, 88.
- 14 Brett M. Frischmann, Michael J. Madison, and Katherine J. Strandburg, "Governing Knowledge Commons," in Frischmann, Madison, and Strandburg *Governing Knowledge Commons*, 12.
- 15 Farley and Kubiszewski, "Economics of Information," in Elliott and Hepting, *Free Knowledge*, 203.
- 16 "What Is Free Software?" GNU Operating System, the Free Software Foundation's Licensing and Compliance Lab, accessed December 30, 2016, www.gnu.org/philosophy/free-sw.
- 17 Wikipedia, s.v. "Open-source software," last modified November 22, 2016.
- 18 Eric S. Raymond, "The Magic Cauldron," in *The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary*, rev. ed. (Sebastopol, CA: O'Reilly Media, 2001), www.catb.org/esr/writings/cathedral-bazaar/.
- 19 New York Times Customer Insight Group, *The Psychology of Sharing: Why Do People Share Online?* (New York: New York Times Customer Insight Group, 2011), www.iab.net/media/file/POSWhitePaper.pdf.
- 20 "Licensing Considerations," Creative Commons, accessed December 30, 2016, creativecommons.org/share-your-work/licensing-considerations/.
- 21 Creative Commons, *2015 State of the Commons* (Mountain View, CA: Creative Commons, 2015), stateof.creativecommons.org/2015/.

- 22 Wikipedia, s.v. "Open Government Partnership," last modified September 24, 2016, en.wikipedia.org/wiki/Open_Government_Partnership.
- 23 Capra and Mattei, *Ecology of Law*, 114.
- 24 Ibid., 116.
- 25 The Swedish International Development Cooperation Agency, "Stockholm Statement" accessed February 15, 2017, sida.se/globalassets/sida/eng/press/stockholm-statement.pdf
- 26 City of Bologna, *Regulation on Collaboration between Citizens and the City for the Care and Regeneration of Urban Commons*, trans. LabGov (LABoratory for the GOVernance of Commons) (Bologna, Italy: City of Bologna, 2014), www.labgov.it/wp-content/uploads/sites/9/Bologna-Regulation-on-collaboration-between-citizens-and-the-city-for-the-care-and-regeneration-of-urban-commons1.pdf.
- 27 The Seoul Sharing City website is english. sharehub.kr; for Amsterdam Sharing City, go to www.sharenl.nl/amsterdam-sharing-city/.
- 28 Tom Slee, *What's Yours Is Mine: Against the Sharing Economy* (New York: OR Books, 2015), 42.
- 29 Chris Anderson, *Free: How Today's Smartest Businesses Profit by Giving Something for Nothing*, Reprint with new preface. (New York: Hyperion, 2010), 78.
- 30 Jeremy Rifkin, *The Zero Marginal Cost Society: The Internet of Things, the Collaborative Commons, and the Eclipse of Capitalism* (New York: Palgrave Macmillan, 2014), 273.
- 31 Gar Alperovitz, *What Then Must We Do?*
- 32 Marjorie Kelly, *Owning Our Future: The Emerging Ownership Revolution; Journeys to a Generative Economy* (San Francisco: Berrett-Koehler, 2012), 8–9.
- 33 Alex Osterwalder and Yves Pigneur, *Business Model Generation* (Hoboken, NJ: John Wiley and Sons, 2010). A preview of the book is available at strategyzer.com/books/business-model-generation.
- 34 This business model canvas is available to download at strategyzer.com/canvas/business-model-canvas.
- 35 We've made the "Open Business Model Canvas," designed by the coauthor Paul Stacey, available online at docs.google.com/drawings/d/1QOIDa2qak7wZSSOa4Wv6qVMO77IwkKHN7CYyq0wHivs/edit. You can also find the accompanying Open Business Model Canvas Questions at docs.google.com/drawings/d/1kACK7TkoJgsM18HUWCbX9xuQ0Byna4pISVZXZGTtays/edit.
- 36 A more comprehensive list of revenue streams is available in this post I wrote on Medium on March 6, 2016. "What Is an Open Business Model and How Can You Generate Revenue?", available at medium.com/made-with-creative-commons/what-is-an-open-business-model-and-how-can-you-generate-revenue-5854d2659b15.
- 37 Henry Chesbrough, *Open Innovation: The New Imperative for Creating and Profiting from Technology* (Boston: Harvard Business Review Press, 2006), 31–44.
- Straight Talk about the Next American Revolution: Democratizing Wealth and Building a Community-Sustaining Economy from the Ground Up* (White River Junction, VT: Chelsea Green, 2013), 39.

HOW TO BE MADE WITH CREATIVE COMMONS

2

SARAH HINCHLIFF PEARSON

When we began this project in August 2015, we set out to write a book about business models that involve Creative Commons licenses in some significant way—what we call being **Made with Creative Commons**. With the help of our Kickstarter backers, we chose twenty-four endeavors from all around the world that are **Made with Creative Commons**. The mix is diverse, from an individual musician to a university-textbook publisher to an electronics manufacturer. Some make their own content and share under Creative Commons licensing. Others are platforms for CC-licensed creative work made by others. Many sit somewhere in between, both using and contributing creative work that's shared with the public. Like all who

use the licenses, these endeavors share their work—whether it's open data or furniture designs—in a way that enables the public not only to access it but also to make use of it.

We analyzed the revenue models, customer segments, and value propositions of each endeavor. We searched for ways that putting their content under Creative Commons licenses helped boost sales or increase reach. Using traditional measures of economic success, we tried to map these business models in a way that meaningfully incorporated the impact of Creative Commons. In our interviews, we dug into the motivations, the role of CC licenses, modes of revenue generation, definitions of success.

In fairly short order, we realized the book we set out to write was quite different from the one that was revealing itself in our interviews and research.

It isn't that we were wrong to think you can make money while using Creative Commons licenses. In many instances, CC *can* help make you more money. Nor were we wrong that there are business models out there that others who want to use CC licensing as part of their livelihood or business could replicate. What we didn't realize was just how misguided it would be to write a book about being **Made with Creative Commons** using only a business lens.

According to the seminal handbook *Business Model Generation*, a business model "describes the rationale of how an organization creates, delivers, and captures value."¹ Thinking about sharing in terms of creating and capturing value always felt inappropriately transactional and out of place, something we heard time and time again in our interviews. And as Cory Doctorow told us in our interview with him, "*Business model* can mean anything you want it to mean."

Eventually, we got it. Being **Made with Creative Commons** is more than a business model. While we will talk about specific revenue models as one piece of our analysis (and in more detail in the case studies), we scrapped that as our guiding rubric for the book.

Admittedly, it took me a long time to get there. When Paul and I divided up our writing after finishing the research, my charge was to distill everything we learned from the case studies and write up the practical lessons and takeaways. I spent months trying to jam what we learned into the business-model box, convinced there must be some formula for the way things interacted. But there is no formula. You'll probably have to discard that way of thinking before you read any further.

In every interview, we started from the same simple questions. Amid all the diversity among

the creators, organizations, and businesses we profiled, there was one constant. Being **Made with Creative Commons** may be good for business, but that is not why they do it. Sharing work with Creative Commons is, at its core, a moral decision. The commercial and other self-interested benefits are secondary. Most decided to use CC licenses first and found a revenue model later. This was our first hint that writing a book solely about the impact of sharing on business might be a little off track.

But we also started to realize something about what it means to be **Made with Creative Commons**. When people talked to us about how and why they used CC, it was clear that it meant something more than using a copyright license. It also represented a set of values. There is symbolism behind using CC, and that symbolism has many layers.

At one level, being **Made with Creative Commons** expresses an affinity for the value of Creative Commons. While there are many different flavors of CC licenses and nearly infinite ways to be **Made with Creative Commons**, the basic value system is rooted in a fundamental belief that knowledge and creativity are building blocks of our culture rather than just commodities from which to extract market value. These values reflect a belief that the common good should always be part of the equation when we determine how to regulate our cultural outputs. They reflect a belief that everyone has something to contribute, and that no one can own our shared culture. They reflect a belief in the promise of sharing.

Whether the public makes use of the opportunity to copy and adapt your work, sharing with a Creative Commons license is a symbol of how you want to interact with the people who consume your work. Whenever you create something, "all rights reserved" under copyright is automatic, so the copyright symbol (©) on the work does not necessarily come across as a marker of distrust or excessive protectionism. But using a CC license *can* be a symbol of the opposite—of wanting a real human relationship, rather than an impersonal

market transaction. It leaves open the possibility of connection.

Being **Made with Creative Commons** not only demonstrates values connected to CC and sharing. It also demonstrates that something other than profit drives what you do. In our interviews, we always asked what success looked like for them. It was stunning how rarely money was mentioned. Most have a deeper purpose and a different vision of success.

The driving motivation varies depending on the type of endeavor. For individual creators, it is most often about personal inspiration. In some ways, this is nothing new. As Doctorow has written, "Creators usually start doing what they do for love."² But when you share your creative work under a CC license, that dynamic is even more pronounced. Similarly, for technological innovators, it is often less about creating a specific new thing that will make you rich and more about solving a specific problem you have. The creators of Arduino told us that the key question when creating something is "Do you as the creator want to use it? It has to have personal use and meaning."

Many that are **Made with Creative Commons** have an express social mission that underpins everything they do. In many cases, sharing with Creative Commons expressly advances that social mission, and using the licenses can be the difference between legitimacy and hypocrisy. Noun Project co-founder Edward Boatman told us they could not have stated their social mission of sharing with a straight face if they weren't willing to show the world that it was OK to share their content using a Creative Commons license.

This dynamic is probably one reason why there are so many nonprofit examples of being **Made with Creative Commons**. The content is the result of a labor of love or a tool to drive social change, and money is like gas in the car, something that you need to keep going but not an end in itself. Being **Made with Creative Commons** is a different vision of a business or livelihood, where profit is not paramount, and producing social good and human connection are integral to success.

Even if profit isn't the end goal, you have to bring in money to be successfully **Made with Creative Commons**. At a bare minimum, you have to make enough money to keep the lights on.

The costs of doing business vary widely for those made with CC, but there is generally a much lower threshold for sustainability than there used to be for any creative endeavor. Digital technology has made it easier than ever to create, and easier than ever to distribute. As Doctorow put it in his book *Information Doesn't Want to Be Free*, "If analog dollars have turned into digital dimes (as the critics of ad-supported media have it), there is the fact that it's possible to run a business that gets the same amount of advertising as its forebears at a fraction of the price."

Some creation costs are the same as they always were. It takes the same amount of time and money to write a peer-reviewed journal article or paint a painting. Technology can't change that. But other costs are dramatically reduced by technology, particularly in production-heavy domains like filmmaking.³ CC-licensed content and content in the public domain, as well as the work of volunteer collaborators, can also dramatically reduce costs if they're being used as resources to create something new. And, of course, there is the reality that some content would be created whether or not the creator is paid because it is a labor of love.

Distributing content is almost universally cheaper than ever. Once content is created, the costs to distribute copies digitally are essentially zero.⁴ The costs to distribute physical copies are still significant, but lower than they have been historically. And it is now much easier to print and distribute physical copies on-demand, which also reduces costs. Depending on the endeavor, there can be a whole host of other possible expenses like marketing and promotion, and even expenses associated with the various ways money is being made, like touring or custom training.

It's important to recognize that the biggest impact of technology on creative endeavors is that creators can now foot the costs of creation and distribution themselves. People now often have a direct route to their potential public without necessarily needing intermediaries like record labels and book publishers. Doctorow wrote, "If you're a creator who never got the time of day from one of the great imperial powers, this is your time. Where once you had no means of reaching an audience without the assistance of the industry-dominating megacompanies, now you have *hundreds* of ways to do it without them."⁵ Previously, distribution of creative work involved the costs associated with sustaining a monolithic entity, now creators can do the work themselves. That means the financial needs of creative endeavors can be a lot more modest.

Whether for an individual creator or a larger endeavor, it usually isn't enough to break even if you want to make what you're doing a livelihood. You need to build in some support for the general operation. This extra bit looks different for everyone, but importantly, in nearly all cases for those **Made with Creative Commons**, the definition of "enough money" looks a lot different than it does in the world of venture capital and stock options. It is more about sustainability and less about unlimited growth and profit. SparkFun founder Nathan Seidle told us, "*Business model* is a really grandiose word for it. It is really just about keeping the operation going day to day."

This book is a testament to the notion that it is possible to make money while using CC licenses and CC-licensed content, but we are still very much at an experimental stage. The creators, organizations, and businesses we profile in this book are blazing the trail and adapting in real time as they pursue this new way of operating.

There are, however, plenty of ways in which CC licensing can be good for business in fairly

predictable ways. The first is how it helps solve "problem zero."

Problem Zero: Getting Discovered

Once you create or collect your content, the next step is finding users, customers, fans—in other words, your people. As Amanda Palmer wrote, "It has to start with the art. The songs had to touch people initially, and mean something, for anything to work at all."⁶ There isn't any magic to finding your people, and there is certainly no formula. Your work has to connect with people and offer them some artistic and/or utilitarian value. In some ways, this is easier than ever. Online we are not limited by shelf space, so there is room for every obscure interest, taste, and need imaginable. This is what Chris Anderson dubbed the Long Tail, where consumption becomes less about mainstream mass "hits" and more about micromarkets for every particular niche. As Anderson wrote, "We are all different, with different wants and needs, and the Internet now has a place for all of them in the way that physical markets did not."⁷ We are no longer limited to what appeals to the masses.

While finding "your people" online is theoretically easier than in the analog world, as a practical matter it can still be difficult to actually get noticed. The Internet is a firehose of content, one that only grows larger by the minute. As a content creator, not only are you competing for attention against more content creators than ever before, you are competing against creativity generated *outside* the market as well.⁸ Anderson wrote, "The greatest change of the past decade has been the shift in time people spend consuming amateur content instead of professional content."⁹ To top it all off, you have to compete against the rest of their lives, too—"friends, family, music playlists, soccer games, and nights on the town."¹⁰ Somehow, some way, you have to get noticed by the right people.

When you come to the Internet armed with an all-rights-reserved mentality from the start, you are often restricting access to your work before there is even any demand for it. In

many cases, requiring payment for your work is part of the traditional copyright system. Even a tiny cost has a big effect on demand. It's called the *penny gap*—the large difference in demand between something that is available at the price of one cent versus the price of zero.¹¹ That doesn't mean it is wrong to charge money for your content. It simply means you need to recognize the effect that doing so will have on demand. The same principle applies to restricting access to copy the work. If your problem is how to get discovered and find "your people," prohibiting people from copying your work and sharing it with others is counterproductive.

Of course, it's not that being discovered by people who like your work will make you rich—far from it. But as Cory Doctorow says, "Recognition is one of many necessary preconditions for artistic success."¹²

Choosing not to spend time and energy restricting access to your work and policing infringement also builds goodwill. Lumen Learning, a for-profit company that publishes online educational materials, made an early decision not to prevent students from accessing their content, even in the form of a tiny paywall, because it would negatively impact student success in a way that would undermine the social mission behind what they do. They believe this decision has generated an immense amount of goodwill within the community.

It is not just that restricting access to your work may undermine your social mission. It also may alienate the people who most value your creative work. If people like your work, their natural instinct will be to share it with others. But as David Bollier wrote, "Our natural human impulses to imitate and share—the essence of culture—have been criminalized."¹³

The fact that copying can carry criminal penalties undoubtedly deters copying it, but copying with the click of a button is too easy and convenient to ever fully stop it. Try as the copyright industry might to persuade us otherwise, copying a copyrighted work just doesn't feel like stealing a loaf of bread. And, of course, that's because it isn't. Sharing a creative work

has no impact on anyone else's ability to make use of it.

If you take some amount of copying and sharing your work as a given, you can invest your time and resources elsewhere, rather than wasting them on playing a cat and mouse game with people who want to copy and share your work. Lizzy Jongma from the Rijksmuseum said, "We could spend a lot of money trying to protect works, but people are going to do it anyway. And they will use bad-quality versions." Instead, they started releasing high-resolution digital copies of their collection into the public domain and making them available for free on their website. For them, sharing was a form of quality control over the copies that were inevitably being shared online. Doing this meant forgoing the revenue they previously got from selling digital images. But Lizzy says that was a small price to pay for all of the opportunities that sharing unlocked for them.

Being **Made with Creative Commons** means you stop thinking about ways to artificially make your content scarce, and instead leverage it as the potentially abundant resource it is.¹⁴ When you see information abundance as a feature, not a bug, you start thinking about the ways to use the idling capacity of your content to your advantage. As my friend and colleague Eric Steuer once said, "Using CC licenses shows you get the Internet."

Cory Doctorow says it costs him nothing when other people make copies of his work, and it opens the possibility that he might get something in return.¹⁵ Similarly, the makers of the Arduino boards knew it was impossible to stop people from copying their hardware, so they decided not to even try and instead look for the benefits of being open. For them, the result is one of the most ubiquitous pieces of hardware in the world, with a thriving online community of tinkerers and innovators that have done things with their work they never could have done otherwise.

There are all kinds of ways to leverage the power of sharing and remix to your benefit. Here are a few.

Use CC to grow a larger audience

Putting a Creative Commons license on your content won't make it automatically go viral, but eliminating legal barriers to copying the work certainly can't hurt the chances that your work will be shared. The CC license symbolizes that sharing is welcome. It can act as a little tap on the shoulder to those who come across the work—a nudge to copy the work if they have any inkling of doing so. All things being equal, if one piece of content has a sign that says Share and the other says Don't Share (which is what “©” means), which do you think people are more likely to share?

The Conversation is an online news site with in-depth articles written by academics who are experts on particular topics. All of the articles are CC-licensed, and they are copied and re-shared on other sites by design. This proliferating effect, which they track, is a central part of the value to their academic authors who want to reach as many readers as possible.

The idea that more eyeballs equates with more success is a form of the *max strategy*, adopted by Google and other technology companies. According to Google's Eric Schmidt, the idea is simple: "Take whatever it is you are doing and do it at the max in terms of distribution. The other way of saying this is that since marginal cost of distribution is free, you might as well put things everywhere."¹⁶ This strategy is what often motivates companies to make their products and services free (i.e., no cost), but the same logic applies to making content freely shareable. Because CC-licensed content is free (as in cost) *and* can be freely copied, CC licensing makes it even more accessible and likely to spread.

If you are successful in reaching more users, readers, listeners, or other consumers of your work, you can start to benefit from the bandwagon effect. The simple fact that there are other people consuming or following your work spurs others to want to do the same.¹⁷ This is, in part, because we simply have a tendency to engage in herd behavior, but it is also because a large following is at least a partial indicator of quality or usefulness.¹⁸

Use CC to get attribution and name recognition

Every Creative Commons license requires that credit be given to the author, and that reusers supply a link back to the original source of the material. CC0, not a license but a tool used to put work in the public domain, does not make attribution a legal requirement, but many communities still give credit as a matter of best practices and social norms. In fact, it is social norms, rather than the threat of legal enforcement, that most often motivate people to provide attribution and otherwise comply with the CC license terms anyway. This is the mark of any well-functioning community, within both the marketplace and the society at large.¹⁹ CC licenses reflect a set of wishes on the part of creators, and in the vast majority of circumstances, people are naturally inclined to follow those wishes. This is particularly the case for something as straightforward and consistent with basic notions of fairness as providing credit.

The fact that the name of the creator follows a CC-licensed work makes the licenses an important means to develop a reputation or, in corporate speak, a brand. The drive to associate your name with your work is not just based on commercial motivations, it is fundamental to authorship. Knowledge Unlatched is a non-profit that helps to subsidize the print production of CC-licensed academic texts by pooling contributions from libraries around the United States. The CEO, Frances Pinter, says that the Creative Commons license on the works has a huge value to authors because reputation is the most important currency for academics. Sharing with CC is a way of having the most people see and cite your work.

Attribution can be about more than just receiving credit. It can also be about establishing provenance. People naturally want to know where content came from—the source of a work is sometimes just as interesting as the work itself. Opendesk is a platform for furniture designers to share their designs. Consumers who like those designs can then get matched with local makers who turn the de-

signs into real-life furniture. The fact that I, sitting in the middle of the United States, can pick out a design created by a designer in Tokyo and then use a maker within my own community to transform the design into something tangible is part of the power of their platform. The provenance of the design is a special part of the product.

Knowing the source of a work is also critical to ensuring its credibility. Just as a trademark is designed to give consumers a way to identify the source and quality of a particular good and service, knowing the author of a work gives the public a way to assess its credibility. In a time when online discourse is plagued with misinformation, being a trusted information source is more valuable than ever.

Use CC-licensed content as a marketing tool

As we will cover in more detail later, many endeavors that are **Made with Creative Commons** make money by providing a product or service *other* than the CC-licensed work. Sometimes that other product or service is completely unrelated to the CC content. Other times it's a physical copy or live performance of the CC content. In all cases, the CC content can attract people to your other product or service.

Knowledge Unlatched's Pinter told us she has seen time and again how offering CC-licensed content—that is, digitally for free—actually increases sales of the printed goods because it functions as a marketing tool. We see this phenomenon regularly with famous artwork. The *Mona Lisa* is likely the most recognizable painting on the planet. Its ubiquity has the effect of catalyzing interest in seeing the painting in person, and in owning physical goods with the image. Abundant copies of the content often entice more demand, not blunt it. Another example came with the advent of the radio. Although the music industry did not see it coming (and fought it!), free music on the radio functioned as advertising for the paid version people bought in music stores.²⁰ Free can be a form of promotion.

In some cases, endeavors that are **Made with Creative Commons** do not even need dedicated marketing teams or marketing budgets. Cards Against Humanity is a CC-licensed card game available as a free download. And because of this (thanks to the CC license on the game), the creators say it is one of the best-marketed games in the world, and they have never spent a dime on marketing. The textbook publisher OpenStax has also avoided hiring a marketing team. Their products are free, or cheaper to buy in the case of physical copies, which makes them much more attractive to students who then demand them from their universities. They also partner with service providers who build atop the CC-licensed content and, in turn, spend money and resources marketing those services (and by extension, the OpenStax textbooks).

Use CC to enable hands-on engagement with your work

The great promise of Creative Commons licensing is that it signifies an embrace of remix culture. Indeed, this is the great promise of digital technology. The Internet opened up a whole new world of possibilities for public participation in creative work.

Four of the six CC licenses enable reusers to take apart, build upon, or otherwise adapt the work. Depending on the context, adaptation can mean wildly different things—translating, updating, localizing, improving, transforming. It enables a work to be customized for particular needs, uses, people, and communities, which is another distinct value to offer the public.²¹ Adaptation is more game changing in some contexts than others. With educational materials, the ability to customize and update the content is critically important for its usefulness. For photography, the ability to adapt a photo is less important.

This is a way to counteract a potential downside of the abundance of free and open content described above. As Anderson wrote in *Free*, “People often don’t care as much about things they don’t pay for, and as a result they don’t think as much about how they consume

them.”²² If even the tiny act of volition of paying one penny for something changes our perception of that thing, then surely the act of remixing it enhances our perception exponentially.²³ We know that people will pay more for products they had a part in creating.²⁴ And we know that creating something, no matter what quality, brings with it a type of creative satisfaction that can never be replaced by consuming something created by someone else.²⁵

Actively engaging with the content helps us avoid the type of aimless consumption that anyone who has absentmindedly scrolled through their social-media feeds for an hour knows all too well. In his book, *Cognitive Surplus*, Clay Shirky says, “To participate is to act as if your presence matters, as if, when you see something or hear something, your response is part of the event.”²⁶ Opening the door to your content can get people more deeply tied to your work.

Use CC to differentiate yourself

Operating under a traditional copyright regime usually means operating under the rules of establishment players in the media. Business strategies that are embedded in the traditional copyright system, like using digital rights management (DRM) and signing exclusivity contracts, can tie the hands of creators, often at the expense of the creator’s best interest.²⁷ Being **Made with Creative Commons** means you can function without those barriers and, in many cases, use the increased openness as a competitive advantage. David Harris from OpenStax said they specifically pursue strategies they know that traditional publishers cannot. “Don’t go into a market and play by the incumbent rules,” David said. “Change the rules of engagement.”

Making Money

Like any moneymaking endeavor, those that are **Made with Creative Commons** have to generate some type of value for their audience or customers. Sometimes that value is subsidized by funders who are not actually beneficiaries of that value. Funders, whether

philanthropic institutions, governments, or concerned individuals, provide money to the organization out of a sense of pure altruism. This is the way traditional nonprofit funding operates.²⁸ But in many cases, the revenue streams used by endeavors that are **Made with Creative Commons** are directly tied to the value they generate, where the recipient is paying for the value they receive like any standard market transaction. In still other cases, rather than the quid pro quo exchange of money for value that typically drives market transactions, the recipient gives money out of a sense of reciprocity.

Most who are **Made with Creative Commons** use a variety of methods to bring in revenue, some market-based and some not. One common strategy is using grant funding for content creation when research-and-development costs are particularly high, and then finding a different revenue stream (or streams) for ongoing expenses. As Shirky wrote, “The trick is in knowing when markets are an optimal way of organizing interactions and when they are not.”²⁹

Our case studies explore in more detail the various revenue-generating mechanisms used by the creators, organizations, and businesses we interviewed. There is nuance hidden within the specific ways each of them makes money, so it is a bit dangerous to generalize too much about what we learned. Nonetheless, zooming out and viewing things from a higher level of abstraction can be instructive.

Market-based revenue streams

In the market, the central question when determining how to bring in revenue is what value people are willing to pay for.³⁰ By definition, if you are **Made with Creative Commons**, the content you provide is available for free and not a market commodity. Like the ubiquitous freemium business model, any possible market transaction with a consumer of your content has to be based on some added value you provide.³¹

In many ways, this is the way of the future for all content-driven endeavors. In the market,

value lives in things that are scarce. Because the Internet makes a universe of content available to all of us for free, it is difficult to get people to pay for content online. The struggling newspaper industry is a testament to this fact. This is compounded by the fact that at least some amount of copying is probably inevitable. That means you may end up competing with free versions of your own content, whether you condone it or not.³² If people can easily find your content for free, getting people to buy it will be difficult, particularly in a context where access to content is more important than owning it. In *Free*, Anderson wrote, "Copyright protection schemes, whether coded into either law or software, are simply holding up a price against the force of gravity."

Of course, this doesn't mean that content-driven endeavors have no future in the traditional marketplace. In *Free*, Anderson explains how when one product or service becomes free, as information and content largely have in the digital age, other things become more valuable. "Every abundance creates a new scarcity," he wrote. You just have to find some way *other* than the content to provide value to your audience or customers. As Anderson says, "It's easy to compete with Free: simply offer something better or at least different from the free version."³³

In light of this reality, in some ways endeavors that are **Made with Creative Commons** are at a level playing field with all content-based endeavors in the digital age. In fact, they may even have an advantage because they can use the abundance of content to derive revenue from something scarce. They can also benefit from the goodwill that stems from the values behind being **Made with Creative Commons**.

For content creators and distributors, there are nearly infinite ways to provide value to the consumers of your work, above and beyond the value that lives within your free digital content. Often, the CC-licensed content functions

as a marketing tool for the paid product or service.

Here are the most common high-level categories.

MARKET-BASED

Providing a custom service to consumers of your work

In this age of information abundance, we don't lack for content. The trick is finding content that matches our needs and wants, so customized services are particularly valuable. As Anderson wrote, "Commodity information (everybody gets the same version) wants to be free. Customized information (you get something unique and meaningful to you) wants to be expensive."³⁴ This can be anything from the artistic and cultural consulting services provided by Ártica to the custom-song business of Jonathan "Song-A-Day" Mann.

MARKET-BASED

Charging for the physical copy

In his book about maker culture, Anderson characterizes this model as giving away the bits and selling the atoms (where *bits* refers to digital content and *atoms* refer to a physical object).³⁵ This is particularly successful in domains where the digital version of the content isn't as valuable as the analog version, like book publishing where a significant subset of people still prefer reading something they can hold in their hands. Or in domains where the content isn't useful until it is in physical form, like furniture designs. In those situations, a significant portion of consumers will pay for the convenience of having someone else put the physical version together for them. Some endeavors squeeze even more out of this revenue stream by using a Creative Commons license that only allows noncommercial uses, which means no one else can sell physical copies of their work in competition with them. This strategy of reserving commercial rights can be particularly important for items like books,

where every printed copy of the same work is likely to be the same quality, so it is harder to differentiate one publishing service from another. On the other hand, for items like furniture or electronics, the provider of the physical goods can compete with other providers of the same works based on quality, service, or other traditional business principles.

Charging for the in-person version

As anyone who has ever gone to a concert will tell you, experiencing creativity in person is a completely different experience from consuming a digital copy on your own. Far from acting as a substitute for face-to-face interaction, CC-licensed content can actually create demand for the in-person version of experience. You can see this effect when people go view original art in person or pay to attend a talk or training course.

Selling merchandise

In many cases, people who like your work will pay for products demonstrating a connection to your work. As a child of the 1980s, I can personally attest to the power of a good concert T-shirt. This can also be an important revenue stream for museums and galleries.

Sometimes the way to find a market-based revenue stream is by providing value to people *other* than those who consume your CC-licensed content. In these revenue streams, the free content is being subsidized by an entirely different category of people or businesses. Often, those people or businesses are paying to access your main audience. The fact that the content is free increases the size of the audience, which in turn makes the offer more valuable to the paying customers. This is a variation of a traditional business model built on free called *multi-sided platforms*.³⁶ Access to your audience isn't the only thing people are

willing to pay for—there are other services you can provide as well.

MARKET-BASED

Charging advertisers or sponsors

The traditional model of subsidizing free content is advertising. In this version of multi-sided platforms, advertisers pay for the opportunity to reach the set of eyeballs the content creators provide in the form of their audience.³⁷ The Internet has made this model more difficult because the number of potential channels available to reach those eyeballs has become essentially infinite.³⁸ Nonetheless, it remains a viable revenue stream for many content creators, including those who are **Made with Creative Commons**. Often, instead of paying to display advertising, the advertiser pays to be an official sponsor of particular content or projects, or of the overall endeavor.

MARKET-BASED

Charging your content creators

Another type of multisided platform is where the content creators themselves pay to be featured on the platform. Obviously, this revenue stream is only available to those who rely on work created, at least in part, by others. The most well-known version of this model is the “author-processing charge” of open-access journals like those published by the Public Library of Science, but there are other variations. The Conversation is primarily funded by a university-membership model, where universities pay to have their faculties participate as writers of the content on the Conversation website.

MARKET-BASED

Charging a transaction fee

This is a version of a traditional business model based on brokering transactions between parties.³⁹ Curation is an important element of this model. Platforms like the Noun Project add value by wading through CC-licensed content to curate a high-quality set and then derive revenue when creators of that content make

transactions with customers. Other platforms make money when service providers transact with their customers; for example, Opendesk makes money every time someone on their site pays a maker to make furniture based on one of the designs on the platform.

Providing a service to your creators

As mentioned above, endeavors can make money by providing customized services to their users. Platforms can undertake a variation of this service model directed at the creators that provide the content they feature. The data platforms Figure.NZ and Figshare both capitalize on this model by providing paid tools to help their users make the data they contribute to the platform more discoverable and reusable.

Licensing a trademark

Finally, some that are **Made with Creative Commons** make money by selling use of their trademarks. Well known brands that consumers associate with quality, credibility, or even an ethos can license that trademark to companies that want to take advantage of that goodwill. By definition, trademarks are scarce because they represent a particular source of a good or service. Charging for the ability to use that trademark is a way of deriving revenue from something scarce while taking advantage of the abundance of CC content.

Reciprocity-based revenue streams

Even if we set aside grant funding, we found that the traditional economic framework of understanding the market failed to fully capture the ways the endeavors we analyzed were making money. It was not simply about monetizing scarcity.

Rather than devising a scheme to get people to pay money in exchange for some direct value provided to them, many of the revenue streams were more about providing value,

building a relationship, and then eventually finding some money that flows back out of a sense of reciprocity. While some look like traditional nonprofit funding models, they aren't charity. The endeavor exchange value with people, just not necessarily synchronously or in a way that requires that those values be equal. As David Bollier wrote in *Think Like a Commoner*, "There is no self-serving calculation of whether the value given and received is strictly equal."

This should be a familiar dynamic—it is the way you deal with your friends and family. We give without regard for what and when we will get back. David Bollier wrote, "Reciprocal social exchange lies at the heart of human identity, community and culture. It is a vital brain function that helps the human species survive and evolve."

What is rare is to incorporate this sort of relationship into an endeavor that also engages with the market.⁴⁰ We almost can't help but think of relationships in the market as being centered on an even-steven exchange of value.⁴¹

Memberships and individual donations

While memberships and donations are traditional nonprofit funding models, in the **Made with Creative Commons** context, they are directly tied to the reciprocal relationship that is cultivated with the beneficiaries of their work. The bigger the pool of those receiving value from the content, the more likely this strategy will work, given that only a small percentage of people are likely to contribute. Since using CC licenses can grease the wheels for content to reach more people, this strategy can be more effective for endeavors that are **Made with Creative Commons**. The greater the argument that the content is a public good or that the entire endeavor is furthering a social mission, the more likely this strategy is to succeed.

The pay-what-you-want model

In the pay-what-you-want model, the beneficiary of Creative Commons content is invited to give—at any amount they can and feel is appropriate, based on the public and personal value they feel is generated by the open content. Critically, these models are not touted as “buying” something free. They are similar to a tip jar. People make financial contributions as an act of gratitude. These models capitalize on the fact that we are naturally inclined to give money for things we value in the marketplace, even in situations where we could find a way to get it for free.

Crowdfunding

Crowdfunding models are based on recouping the costs of creating and distributing content before the content is created. If the endeavor is **Made with Creative Commons**, anyone who wants the work in question could simply wait until it’s created and then access it for free. That means, for this model to work, people have to care about more than just receiving the work. They have to want you to succeed. Amanda Palmer credits the success of her crowdfunding on Kickstarter and Patreon to the years she spent building her community and creating a connection with her fans. She wrote in *The Art of Asking*, “Good art is made, good art is shared, help is offered, ears are bent, emotions are exchanged, the compost of real, deep connection is sprayed all over the fields. Then one day, the artist steps up and asks for something. And if the ground has been fertilized enough, the audience says, without hesitation: of course.”

Other types of crowdfunding rely on a sense of responsibility that a particular community may feel. Knowledge Unlatched pools funds from major U.S. libraries to subsidize CC-licensed academic work that will be, by definition,

available to everyone for free. Libraries with bigger budgets tend to give more out of a sense of commitment to the library community and to the idea of open access generally.

Making Human Connections

Regardless of how they made money, in our interviews, we repeatedly heard language like “persuading people to buy” and “inviting people to pay.” We heard it even in connection with revenue streams that sit squarely within the market. Cory Doctorow told us, “I have to convince my readers that the right thing to do is to pay me.” The founders of the for-profit company Lumen Learning showed us the letter they send to those who opt not to pay for the services they provide in connection with their CC-licensed educational content. It isn’t a cease-and-desist letter; it’s an invitation to pay because it’s the right thing to do. This sort of behavior toward what could be considered nonpaying customers is largely unheard of in the traditional marketplace. But it seems to be part of the fabric of being **Made with Creative Commons**.

Nearly every endeavor we profiled relied, at least in part, on people being invested in what they do. The closer the Creative Commons content is to being “the product,” the more pronounced this dynamic has to be. Rather than simply selling a product or service, they are making ideological, personal, and creative connections with the people who value what they do.

It took me a very long time to see how this avoidance of thinking about what they do in pure market terms was deeply tied to being **Made with Creative Commons**.

I came to the research with preconceived notions about what Creative Commons is and what it means to be **Made with Creative Commons**. It turned out I was wrong on so many counts.

Obviously, being **Made with Creative Commons** means using Creative Commons licenses. That much I knew. But in our interviews, people spoke of so much more than copyright

permissions when they explained how sharing fit into what they do. I was thinking about sharing too narrowly, and as a result, I was missing vast swaths of the meaning packed within Creative Commons. Rather than parsing the specific and narrow role of the copyright license in the equation, it is important not to disaggregate the rest of what comes with sharing. *You have to widen the lens.*

Being **Made with Creative Commons** is not just about the simple act of licensing a copyrighted work under a set of standardized terms, but also about community, social good, contributing ideas, expressing a value system, working together. These components of sharing are hard to cultivate if you think about what you do in purely market terms. Decent social behavior isn't as intuitive when we are doing something that involves monetary exchange. It takes a conscious effort to foster the context for real sharing, based not strictly on impersonal market exchange, but on connections with the people with whom you share—connections with you, with your work, with your values, with each other.

The rest of this section will explore some of the common strategies that creators, companies, and organizations use to remind us that there are humans behind every creative endeavor. To remind us we have obligations to each other. To remind us what sharing really looks like.

Be human

Humans are social animals, which means we are naturally inclined to treat each other well.⁴² But the further removed we are from the person with whom we are interacting, the less caring our behavior will be. While the Internet has democratized cultural production, increased access to knowledge, and connected us in extraordinary ways, it can also make it easy forget we are dealing with another human.

To counteract the anonymous and impersonal tendencies of how we operate online, individual creators and corporations who use Creative Commons licenses work to demonstrate their humanity. For some, this means

pouring their lives out on the page. For others, it means showing their creative process, giving a glimpse into how they do what they do. As writer Austin Kleon wrote, "*Our work doesn't speak for itself.* Human beings want to know where things came from, how they were made, and who made them. The stories you tell about the work you do have a huge effect on how people feel and what they understand about your work, and how people feel and what they understand about your work affects how they value it."⁴³

A critical component to doing this effectively is not worrying about being a "brand." That means not being afraid to be vulnerable. Amanda Palmer says, "When you're afraid of someone's judgment, you can't connect with them. You're too preoccupied with the task of impressing them." Not everyone is suited to live life as an open book like Palmer, and that's OK. There are a lot of ways to be human. The trick is just avoiding pretense and the temptation to artificially craft an image. People don't just want the glossy version of you. They can't relate to it, at least not in a meaningful way.

This advice is probably even more important for businesses and organizations because we instinctively conceive of them as nonhuman (though in the United States, corporations are people!). When corporations and organizations make the people behind them more apparent, it reminds people that they are dealing with something other than an anonymous corporate entity. In business-speak, this is about "humanizing your interactions" with the public.⁴⁴ But it can't be a gimmick. You can't fake being human.

Be open and accountable

Transparency helps people understand who you are and why you do what you do, but it also inspires trust. Max Temkin of Cards Against Humanity told us, "One of the most surprising things you can do in capitalism is just be honest with people." That means sharing the good and the bad. As Amanda Palmer wrote, "You can fix almost anything by authentically communicating."⁴⁵ It isn't about trying to satis-

fy everyone or trying to sugarcoat mistakes or bad news, but instead about explaining your rationale and then being prepared to defend it when people are critical.⁴⁶

Being accountable does not mean operating on consensus. According to James Surowiecki, consensus-driven groups tend to resort to lowest-common-denominator solutions and avoid the sort of candid exchange of ideas that cultivates healthy collaboration.⁴⁷ Instead, it can be as simple as asking for input and then giving context and explanation about decisions you make, even if soliciting feedback and inviting discourse is time-consuming. If you don't go through the effort to actually respond to the input you receive, it can be worse than not inviting input in the first place.⁴⁸ But when you get it right, it can guarantee the type of diversity of thought that helps endeavors excel. And it is another way to get people involved and invested in what you do.

Design for the good actors

Traditional economics assumes people make decisions based solely on their own economic self-interest.⁴⁹ Any relatively introspective human knows this is a fiction—we are much more complicated beings with a whole range of needs, emotions, and motivations. In fact, we are hardwired to work together and ensure fairness.⁵⁰ Being **Made with Creative Commons** requires an assumption that people will largely act on those social motivations, motivations that would be considered “irrational” in an economic sense. As Knowledge Unlatched’s Pinter told us, “It is best to ignore people who try to scare you about free riding. That fear is based on a very shallow view of what motivates human behavior.” There will always be people who will act in purely selfish ways, but endeavors that are **Made with Creative Commons** design for the good actors.

The assumption that people will largely do the right thing can be a self-fulfilling prophecy. Shirky wrote in *Cognitive Surplus*, “Systems that assume people will act in ways that create public goods, and that give them opportunities and rewards for doing so, often let them work

together better than neoclassical economics would predict.”⁵¹ When we acknowledge that people are often motivated by something other than financial self-interest, we design our endeavors in ways that encourage and accentuate our social instincts.

Rather than trying to exert control over people’s behavior, this mode of operating requires a certain level of trust. We might not realize it, but our daily lives are already built on trust. As Surowiecki wrote in *The Wisdom of Crowds*, “It’s impossible for a society to rely on law alone to make sure citizens act honestly and responsibly. And it’s impossible for any organization to rely on contracts alone to make sure that its managers and workers live up to their obligation.” Instead, we largely trust that people—mostly strangers—will do what they are supposed to do.⁵² And most often, they do.

Treat humans like, well, humans

For creators, treating people as humans means not treating them like fans. As Kleon says, “If you want fans, you have to be a fan first.”⁵³ Even if you happen to be one of the few to reach celebrity levels of fame, you are better off remembering that the people who follow your work are human, too. Cory Doctorow makes a point to answer every single email someone sends him. Amanda Palmer spends vast quantities of time going online to communicate with her public, making a point to listen just as much as she talks.⁵⁴

The same idea goes for businesses and organizations. Rather than automating its customer service, the music platform Tribe of Noise makes a point to ensure its employees have personal, one-on-one interaction with users.

When we treat people like humans, they typically return the gift in kind. It’s called karma. But social relationships are fragile. It is all too easy to destroy them if you make the mistake of treating people as anonymous customers or free labor.⁵⁵ Platforms that rely on content from contributors are especially at risk of creating an exploitative dynamic. It is important to find ways to acknowledge and pay back the

value that contributors generate. That does not mean you can solve this problem by simply paying contributors for their time or contributions. As soon as we introduce money into a relationship—at least when it takes a form of paying monetary value in exchange for other value—it can dramatically change the dynamic.⁵⁶

State your principles and stick to them

Being **Made with Creative Commons** makes a statement about who you are and what you do. The symbolism is powerful. Using Creative Commons licenses demonstrates adherence to a particular belief system, which generates goodwill and connects like-minded people to your work. Sometimes people will be drawn to endeavors that are **Made with Creative Commons** as a way of demonstrating their own commitment to the Creative Commons value system, akin to a political statement. Other times people will identify and feel connected with an endeavor's separate social mission. Often both.

The expression of your values doesn't have to be implicit. In fact, many of the people we interviewed talked about how important it is to state your guiding principles up front. Lumen Learning attributes a lot of their success to having been outspoken about the fundamental values that guide what they do. As a for-profit company, they think their expressed commitment to low-income students and open licensing has been critical to their credibility in the OER (open educational resources) community in which they operate.

When your end goal is not about making a profit, people trust that you aren't just trying to extract value for your own gain. People notice when you have a sense of purpose that transcends your own self-interest.⁵⁷ It attracts committed employees, motivates contributors, and builds trust.

Build a community

Endeavors that are **Made with Creative Commons** thrive when community is built around what they do. This may mean a community collaborating together to create something new,

or it may simply be a collection of like-minded people who get to know each other and rally around common interests or beliefs.⁵⁸ To a certain extent, simply being **Made with Creative Commons** automatically brings with it some element of community, by helping connect you to like-minded others who recognize and are drawn to the values symbolized by using CC.

To be sustainable, though, you have to work to nurture community. People have to care—about you and each other. One critical piece to this is fostering a sense of belonging. As Jono Bacon writes in *The Art of Community*, "If there is no belonging, there is no community." For Amanda Palmer and her band, that meant creating an accepting and inclusive environment where people felt a part of their "weird little family."⁵⁹ For organizations like Red Hat, that means connecting around common beliefs or goals. As the CEO Jim Whitehurst wrote in *The Open Organization*, "Tapping into passion is especially important in building the kinds of participative communities that drive open organizations."⁶⁰

Communities that collaborate together take deliberate planning. Surowiecki wrote, "It takes a lot of work to put the group together. It's difficult to ensure that people are working in the group's interest and not in their own. And when there's a lack of trust between the members of the group (which isn't surprising given that they don't really know each other), considerable energy is wasted trying to determine each other's bona fides."⁶¹ Building true community requires giving people within the community the power to create or influence the rules that govern the community.⁶² If the rules are created and imposed in a top-down manner, people feel like they don't have a voice, which in turn leads to disengagement.

Community takes work, but working together, or even simply being connected around common interests or values, is in many ways what sharing is about.

Give more to the commons than you take

Conventional wisdom in the marketplace dictates that people should try to extract as much money as possible from resources. This is essentially what defines so much of the so-called sharing economy. In an article on the *Harvard Business Review* website called “The Sharing Economy Isn’t about Sharing at All,” authors Giana Eckhardt and Fleura Bardhi explained how the anonymous market-driven transactions in most sharing-economy businesses are purely about monetizing access.⁶³ As Lisa Gansky put it in her book *The Mesh*, the primary strategy of the sharing economy is to sell the same product multiple times, by selling access rather than ownership.⁶⁴ That is not sharing.

Sharing requires adding as much or more value to the ecosystem than you take. You can’t simply treat open content as a free pool of resources from which to extract value. Part of giving back to the ecosystem is contributing content back to the public under CC licenses. But it doesn’t have to just be about creating content; it can be about adding value in other ways. The social blogging platform *Medium* provides value to its community by incentivizing good behavior, and the result is an online space with remarkably high-quality user-generated content and limited trolling.⁶⁵ Opendesk contributes to its community by committing to help its designers make money, in part by actively curating and displaying their work on its platform effectively.

In all cases, it is important to openly acknowledge the amount of value you add versus that which you draw on that was created by others. Being transparent about this builds credibility and shows you are a contributing player in the commons. When your endeavor is making money, that also means apportioning financial compensation in a way that reflects the value contributed by others, providing more to contributors when the value they add outweighs the value provided by you.

Involve people in what you do

Thanks to the Internet, we can tap into the talents and expertise of people around the

globe. Chris Anderson calls it the Long Tail of talent.⁶⁶ But to make collaboration work, the group has to be effective at what it is doing, and the people within the group have to find satisfaction from being involved.⁶⁷ This is easier to facilitate for some types of creative work than it is for others. Groups tied together online collaborate best when people can work independently and asynchronously, and particularly for larger groups with loose ties, when contributors can make simple improvements without a particularly heavy time commitment.⁶⁸

As the success of Wikipedia demonstrates, editing an online encyclopedia is exactly the sort of activity that is perfect for massive co-creation because small, incremental edits made by a diverse range of people acting on their own are immensely valuable in the aggregate. Those same sorts of small contributions would be less useful for many other types of creative work, and people are inherently less motivated to contribute when it doesn’t appear that their efforts will make much of a difference.⁶⁹

It is easy to romanticize the opportunities for global cocreation made possible by the Internet, and, indeed, the successful examples of it are truly incredible and inspiring. But in a wide range of circumstances—perhaps more often than not—community cocreation is not part of the equation, even within endeavors built on CC content. Shirky wrote, “Sometimes the value of professional work trumps the value of amateur sharing or a feeling of belonging.⁷⁰ The textbook publisher OpenStax, which distributes all of its material for free under CC licensing, is an example of this dynamic. Rather than tapping the community to help cocreate their college textbooks, they invest a significant amount of time and money to develop professional content. For individual creators, where the creative work is the basis for what they do, community cocreation is only rarely a part of the picture. Even musician Amanda Palmer, who is famous for her openness and involvement with her fans, said, “The only department where I wasn’t

open to input was the writing, the music itself.”⁷¹

While we tend to immediately think of co-creation and remixing when we hear the word *collaboration*, you can also involve others in your creative process in more informal ways, by sharing half-baked ideas and early drafts, and interacting with the public to incubate ideas and get feedback. So-called “making in public” opens the door to letting people feel more invested in your creative work.⁷² And it shows a nonterritorial approach to ideas and information. Stephen Covey (of *The 7 Habits of Highly Effective People* fame) calls this the *abundance mentality*—treating ideas like something plentiful—and it can create an environment where collaboration flourishes.⁷³

There is no one way to involve people in what you do. The key is finding a way for people to contribute on their terms, compelled by their own motivations.⁷⁴ What that looks like varies wildly depending on the project. Not every endeavor that is **Made with Creative Commons** can be *Wikipedia*, but every endeavor can find ways to invite the public into what they do. The goal for any form of collaboration is to move away from thinking of consumers as passive recipients of your content and transition them into active participants.⁷⁵

Notes

- 1 Alex Osterwalder and Yves Pigneur, *Business Model Generation* (Hoboken, NJ: John Wiley and Sons, 2010), 14. A preview of the book is available at strategyzer.com/books/business-model-generation.
- 2 Cory Doctorow, *Information Doesn't Want to Be Free: Laws for the Internet Age* (San Francisco, CA: McSweeney's, 2014) 68.
- 3 Ibid., 55.
- 4 Chris Anderson, *Free: How Today's Smartest Businesses Profit by Giving Something for Nothing*, reprint with new preface (New York: Hyperion, 2010), 224.
- 5 Doctorow, *Information Doesn't Want to Be Free*, 44.
- 6 Amanda Palmer, *The Art of Asking: Or How I Learned to Stop Worrying and Let People Help* (New York: Grand Central, 2014), 121.
- 7 Chris Anderson, *Makers: The New Industrial Revolution* (New York: Signal, 2012), 64.
- 8 David Bollier, *Think Like a Commoner: A Short Introduction to the Life of the Commons* (Gabriola Island, BC: New Society, 2014), 70.
- 9 Anderson, *Makers*, 66.
- 10 Bryan Kramer, *Shareology: How Sharing Is Powering the Human Economy* (New York: Morgan James, 2016), 10.
- 11 Anderson, *Free*, 62.
- 12 Doctorow, *Information Doesn't Want to Be Free*, 38.
- 13 Bollier, *Think Like a Commoner*, 68.
- 14 Anderson, *Free*, 86.
- 15 Doctorow, *Information Doesn't Want to Be Free*, 144.
- 16 Anderson, *Free*, 123.
- 17 Ibid., 132.
- 18 Ibid., 70.
- 19 James Surowiecki, *The Wisdom of Crowds* (New York: Anchor Books, 2005), 124. Surowiecki says, "The measure of success of laws and contracts is how rarely they are invoked."
- 20 Anderson, *Free*, 44.
- 21 Osterwalder and Pigneur, *Business Model Generation*, 23.
- 22 Anderson, *Free*, 67.
- 23 Ibid., 58.
- 24 Anderson, *Makers*, 71.
- 25 Clay Shirky, *Cognitive Surplus: How Technology Makes Consumers into Collaborators* (London: Penguin Books, 2010), 78.
- 26 Ibid., 21.
- 27 Doctorow, *Information Doesn't Want to Be Free*, 43.
- 28 William Landes Foster, Peter Kim, and Barbara Christiansen, "Ten Nonprofit Funding Models," *Stanford Social Innovation Review*, Spring 2009, ssir.org/articles/entry/ten_nonprofit_funding_models.
- 29 Shirky, *Cognitive Surplus*, 111.
- 30 Osterwalder and Pigneur, *Business Model Generation*, 30.

- 31 Jim Whitehurst, *The Open Organization: Igniting Passion and Performance* (Boston: Harvard Business Review Press, 2015), 202.
- 32 Anderson, *Free*, 71.
- 33 Ibid., 231.
- 34 Ibid., 97.
- 35 Anderson, *Makers*, 107.
- 36 Osterwalder and Pigneur, *Business Model Generation*, 89.
- 37 Ibid., 92.
- 38 Anderson, *Free*, 142.
- 39 Osterwalder and Pigneur, *Business Model Generation*, 32.
- 40 Bollier, *Think Like a Commoner*, 150.
- 41 Ibid., 134.
- 42 Dan Ariely, *Predictably Irrational: The Hidden Forces That Shape Our Decisions*, rev. ed. (New York: Harper Perennial, 2010), 109.
- 43 Austin Kleon, *Show Your Work: 10 Ways to Share Your Creativity and Get Discovered* (New York: Workman, 2014), 93.
- 44 Kramer, *Shareology*, 76.
- 45 Palmer, *Art of Asking*, 252.
- 46 Whitehurst, *Open Organization*, 145.
- 47 Surowiecki, *Wisdom of Crowds*, 203.
- 48 Whitehurst, *Open Organization*, 80.
- 49 Bollier, *Think Like a Commoner*, 25.
- 50 Ibid., 31.
- 51 Shirky, *Cognitive Surplus*, 112.
- 52 Surowiecki, *Wisdom of Crowds*, 124.
- 53 Kleon, *Show Your Work*, 127.
- 54 Palmer, *Art of Asking*, 121.
- 55 Ariely, *Predictably Irrational*, 87.
- 56 Ibid., 105.
- 57 Ibid., 36.
- 58 Jono Bacon, *The Art of Community*, 2nd ed. (Sebastopol, CA: O'Reilly Media, 2012), 36.
- 59 Palmer, *Art of Asking*, 98.
- 60 Whitehurst, *Open Organization*, 34.
- 61 Surowiecki, *Wisdom of Crowds*, 200.
- 62 Bollier, *Think Like a Commoner*, 29.
- 63 Giana Eckhardt and Fleura Bardhi, "The Sharing Economy Isn't about Sharing at All," *Harvard Business Review* (website), January 28, 2015, hbr.org/2015/01/the-sharing-economy-isnt-about-sharing-at-all.
- 64 Lisa Gansky, *The Mesh: Why the Future of Business Is Sharing*, reprint with new epilogue (New York: Portfolio, 2012).
- 65 David Lee, "Inside Medium: An Attempt to Bring Civility to the Internet," *BBC News*, March 3, 2016, www.bbc.com/news/technology-35709680.
- 66 Anderson, *Makers*, 148.
- 67 Shirky, *Cognitive Surplus*, 164.

- 68 Whitehurst, foreword to *Open Organization*.
- 69 Shirky, *Cognitive Surplus*, 144.
- 70 Ibid., 154.
- 71 Palmer, *Art of Asking*, 163.
- 72 Anderson, *Makers*, 173.
- 73 Tom Kelley and David Kelley, *Creative Confidence: Unleashing the Potential within Us All* (New York: Crown, 2013), 82.
- 74 Whitehurst, foreword to *Open Organization*.
- 75 Rachel Botsman and Roo Rogers, *What's Mine Is Yours: The Rise of Collaborative Consumption* (New York: Harper Business, 2010), 188.

THE CREATIVE COMMONS LICENSES

3

All of the Creative Commons licenses grant a basic set of permissions. At a minimum, a CC-licensed work can be copied and shared in its original form for noncommercial purposes so long as attribution is given to the creator. There are six licenses in the CC license suite that build on that basic set of permissions, ranging from the most restrictive (allowing only those basic permissions to share unmodified copies for noncommercial purposes) to the most permissive (reusers can do anything they want with the work, even for commercial purposes, as long as they give the creator credit). The licenses are built on copyright and do not cover other types of rights that creators might have in their works, like patents or trademarks.

Here are the six licenses:



The Attribution license (CC BY) lets others distribute, remix, tweak, and build upon your work, even commercially, as long as they credit you for the original creation. This is the most accommodating of

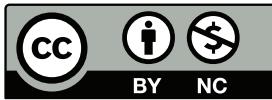
licenses offered. Recommended for maximum dissemination and use of licensed materials.



The Attribution-ShareAlike license (CC BY-SA) lets others remix, tweak, and build upon your work, even for commercial purposes, as long as they credit you and license their new creations under identical terms. This license is often compared to "copyleft" free and open source software licenses. All new works based on yours will carry the same license, so any derivatives will also allow commercial use.



The Attribution-NoDerivs license (CC BY-ND) allows for redistribution, commercial and noncommercial, as long as it is passed along unchanged with credit to you.



The Attribution-Non-Commercial license (CC BY-NC) lets others remix, tweak, and build upon your work noncommercially. Although their new works must also acknowledge you, they don't have to license their derivative works on the same terms.



The Attribution-Non-Commercial-ShareAlike license (CC BY-NC-SA) lets others remix, tweak, and build upon your work noncommercially, as long as they credit you and license their new creations under the same terms.



The Attribution-Non-Commercial-NoDerivs license (CC BY-NC-ND) is the most restrictive of our six main licenses, only allowing others to download your works and share them with others as long as they credit you, but they can't change them or use them commercially.

In addition to these six licenses, Creative Commons has two public-domain tools—one for creators and the other for those who manage collections of existing works by authors whose terms of copyright have expired:



CC0 enables authors and copyright owners to dedicate their works to the worldwide public domain ("no rights reserved").



The Creative Commons Public Domain Mark facilitates the labeling and discovery of works that are already free of known copyright restrictions.

In our case studies, some use just one Creative Commons license, others use several. Attribution (found in thirteen case studies) and Attribution-ShareAlike (found in eight studies) were the most common, with the other

licenses coming up in four or so case studies, including the public-domain tool CC0. Some of the organizations we profiled offer both digital content and software: by using open-source-software licenses for the software code and Creative Commons licenses for digital content, they amplify their involvement with and commitment to sharing.

There is a popular misconception that the three NonCommercial licenses offered by CC are the only options for those who want to make money off their work. As we hope this book makes clear, there are many ways to make endeavors that are **Made with Creative Commons** sustainable. Reserving commercial rights is only one of those ways. It is certainly true that a license that *allows* others to make commercial use of your work (CC BY, CC BY-SA, and CC BY-ND) forecloses some traditional revenue streams. If you apply an Attribution (CC BY) license to your book, you can't force a film company to pay you royalties if they turn your book into a feature-length film, or prevent another company from selling physical copies of your work.

The decision to choose a NonCommercial and/or NoDerivs license comes down to how much you need to retain control over the creative work. The NonCommercial and NoDerivs licenses are ways of reserving some significant portion of the exclusive bundle of rights that copyright grants to creators. In some cases, reserving those rights is important to how you bring in revenue. In other cases, creators use a NonCommercial or NoDerivs license because they can't give up on the dream of hitting the creative jackpot. The music platform Tribe of Noise told us the NonCommercial licenses were popular among their users because people still held out the dream of having a major record label discover their work.

Other times the decision to use a more restrictive license is due to a concern about the integrity of the work. For example, the non-profit TeachAIDS uses a NoDerivs license for its educational materials because the medical subject matter is particularly important to get right.

There is no one right way. The NonCommercial and NoDerivs restrictions reflect the values and preferences of creators about how their creative work should be reused, just as the ShareAlike license reflects a different set of values, one that is less about controlling access to their own work and more about ensuring that whatever gets created with their work is available to all on the same terms. Since the beginning of the commons, people have been setting up structures that helped regulate the way in which shared resources were used. The CC licenses are an attempt to standardize norms across all domains.

Note

For more about the licenses including examples and tips on sharing your work in the digital commons, start with the Creative Commons page called "Share Your Work" at creativecommons.org/share-your-work/.

Part 2

THE CASE STUDIES

The twenty-four case studies in this section were chosen from hundreds of nominations received from Kickstarter backers, Creative Commons staff, and the global Creative Commons community. We selected eighty potential candidates that represented a mix of industries, content types, revenue streams, and parts of the world. Twelve of the case studies were selected from that group based on votes cast by Kickstarter backers, and the other twelve were selected by us.

We did background research and conducted interviews for each case study, based on the same set of basic questions about the endeavor. The idea for each case study is to tell the story about the endeavor and the role sharing plays within it, largely the way in which it was told to us by those we interviewed.

ARDUINO



Arduino is a for-profit open-source electronics platform and computer hardware and software company. Founded in 2005 in Italy.

www.arduino.cc

Revenue model: charging for physical copies (sales of boards, modules, shields, and kits), licensing a trademark (fees paid by those who want to sell Arduino products using their name)

Interview date: February 4, 2016

Interviewees: David Cuartielles and Tom Igoe, cofounders

Profile written by Paul Stacey

In 2005, at the Interaction Design Institute Ivrea in northern Italy, teachers and students needed an easy way to use electronics and programming to quickly prototype design ideas. As musicians, artists, and designers, they needed a platform that didn't require engineering expertise. A group of teachers and students, including Massimo Banzi, David Cuartielles, Tom Igoe, Gianluca Martino, and David Mellis, built a platform that combined different open technologies. They called it Arduino. The platform integrated software, hardware, microcontrollers, and electronics. All aspects of the platform were openly licensed: hardware designs and documentation with the Attribution-Share-Alike license (CC BY-SA), and software with the GNU General Public License.

Arduino boards are able to read inputs—light on a sensor, a finger on a button, or a Twitter message—and turn it into outputs—activating a motor, turning on an LED, publish-

ing something online. You send a set of instructions to the microcontroller on the board by using the Arduino programming language and Arduino software (based on a piece of open-source software called Processing, a programming tool used to make visual art).

"The reasons for making Arduino open source are complicated," Tom says. Partly it was about supporting flexibility. The open-source nature of Arduino empowers users to modify it and create a lot of different variations, adding on top of what the founders build. David says this "ended up strengthening the platform far beyond what we had even thought of building."

For Tom another factor was the impending closure of the Ivrea design school. He'd seen other organizations close their doors and all their work and research just disappear. Open-sourcing ensured that Arduino would outlive the Ivrea closure. Persistence is one

thing Tom really likes about open source. If key people leave, or a company shuts down, an open-source product lives on. In Tom's view, "Open sourcing makes it easier to trust a product."

With the school closing, David and some of the other Arduino founders started a consulting firm and multidisciplinary design studio they called Tinker, in London. Tinker designed products and services that bridged the digital and the physical, and they taught people how to use new technologies in creative ways. Revenue from Tinker was invested in sustaining and enhancing Arduino.

For Tom, part of Arduino's success is because the founders made themselves the first customer of their product. They made products they themselves personally wanted. It was a matter of "I need this thing," not "If we make this, we'll make a lot of money." Tom notes that being your own first customer makes you more confident and convincing at selling your product.

Arduino's business model has evolved over time—and Tom says *model* is a grandiose term for it. Originally, they just wanted to make a few boards and get them out into the world. They started out with two hundred boards, sold them, and made a little profit. They used that to make another thousand, which generated enough revenue to make five thousand. In the early days, they simply tried to generate enough funding to keep the venture going day to day. When they hit the ten thousand mark, they started to think about Arduino as a company. By then it was clear you can open-source the design but still manufacture the physical product. As long as it's a quality product and sold at a reasonable price, people will buy it.

Arduino now has a worldwide community of makers—students, hobbyists, artists, programmers, and professionals. Arduino pro-

vides a wiki called Playground (a wiki is where all users can edit and add pages, contributing to and benefiting from collective research). People share code, circuit diagrams, tutorials, DIY instructions, and tips and tricks, and show off their projects. In addition, there's a multi-language discussion forum where users can get help using Arduino, discuss topics like robotics, and make suggestions for new Arduino product designs. As of January 2017, 324,928 members had made 2,989,489 posts on 379,044 topics. The worldwide community of makers has contributed an incredible amount of accessible knowledge helpful to novices and experts alike.

Transitioning Arduino from a project to a company was a big step. Other businesses who made boards were charging a lot of money for them. Arduino wanted to make theirs available at a low price to people across a wide range of industries. As with any business, pricing was key. They wanted prices that would get lots of customers but were also high enough to sustain the business.

For a business, getting to the end of the year and not being in the red is a success. Arduino may have an open-licensing strategy, but they are still a business, and all the things needed to successfully run one still apply. David says, "If you do those other things well, sharing things in an open-source way can only help you."

While openly licensing the designs, documentation, and software ensures longevity, it does have risks. There's a possibility that others will create knockoffs, clones, and copies. The CC BY-SA license means anyone can produce copies of their boards, redesign them, and even sell boards that copy the design. They don't have to pay a license fee to Arduino or even ask permission. However, if they republish the design of the board, they have to give attribution to Arduino. If they change the design, they must release the new design using the same Creative Commons license to ensure that the new version is equally free and open.

Tom and David say that a lot of people have built companies off of Arduino, with dozens of

Arduino derivatives out there. But in contrast to closed business models that can wring money out of the system over many years because there is no competition, Arduino founders saw competition as keeping them honest, and aimed for an environment of collaboration. A benefit of open over closed is the many new ideas and designs others have contributed back to the Arduino ecosystem, ideas and designs that Arduino and the Arduino community use and incorporate into new products.

Over time, the range of Arduino products has diversified, changing and adapting to new needs and challenges. In addition to simple entry level boards, new products have been added ranging from enhanced boards that provide advanced functionality and faster performance, to boards for creating Internet of Things applications, wearables, and 3-D printing. The full range of official Arduino products includes boards, modules (a smaller form-factor of classic boards), shields (elements that can be plugged onto a board to give it extra features), and kits.¹

THE OPEN-SOURCE NATURE OF ARDUINO EMPOWERS USERS TO MODIFY IT AND CREATE A LOT OF DIFFERENT VARIATIONS, STRENGTHENING THE PLATFORM FAR BEYOND WHAT THE FOUNDERS THOUGHT OF BUILDING.

Arduino's focus is on high-quality boards, well-designed support materials, and the

building of community; this focus is one of the keys to their success. And being open lets you build a real community. David says Arduino's community is a big strength and something that really does matter—in his words, “It’s good business.” When they started, the Arduino team had almost entirely no idea how to build a community. They started by conducting numerous workshops, working directly with people using the platform to make sure the hardware and software worked the way it was meant to work and solved people’s problems. The community grew organically from there.

A key decision for Arduino was trademarking the name. The founders needed a way to guarantee to people that they were buying a quality product from a company committed to open-source values and knowledge sharing. Trademarking the Arduino name and logo expresses that guarantee and helps customers easily identify their products, and the products sanctioned by them. If others want to sell boards using the Arduino name and logo, they have to pay a small fee to Arduino. This allows Arduino to scale up manufacturing and distribution while at the same time ensuring the Arduino brand isn’t hurt by low-quality copies.

Current official manufacturers are Smart Projects in Italy, SparkFun in the United States, and Dog Hunter in Taiwan/China. These are the only manufacturers that are allowed to use the Arduino logo on their boards. Trademarking their brand provided the founders with a way to protect Arduino, build it out further, and fund software and tutorial development. The trademark-licensing fee for the brand became Arduino’s revenue-generating model.

How far to open things up wasn’t always something the founders perfectly agreed on. David, who was always one to advocate for opening things up more, had some fears about protecting the Arduino name, thinking people would be mad if they policed their brand. There was some early backlash with

a project called Freeduino, but overall, trademarking and branding has been a critical tool for Arduino.

David encourages people and businesses to start by sharing everything as a default strategy, and then think about whether there is anything that really needs to be protected and why. There are lots of good reasons to not open up certain elements. This strategy of sharing everything is certainly the complete opposite of how today's world operates, where nothing is shared. Tom suggests a business formalize which elements are based on open sharing and which are closed. An Arduino blog post from 2013 entitled "Send In the Clones," by one of the founders Massimo Banzi, does a great job of explaining the full complexities of how trademarking their brand has played out, distinguishing between official boards and those that are clones, derivatives, compatibles, and counterfeits.²

For David, an exciting aspect of Arduino is the way lots of people can use it to adapt

technology in many different ways. Technology is always making more things possible but doesn't always focus on making it easy to use and adapt. This is where Arduino steps in. Arduino's goal is "making things that help other people make things."

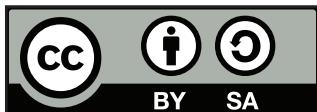
Arduino has been hugely successful in making technology and electronics reach a larger audience. For Tom, Arduino has been about "the democratization of technology." Tom sees Arduino's open-source strategy as helping the world get over the idea that technology has to be protected. Tom says, "Technology is a literacy everyone should learn."

Ultimately, for Arduino, going open has been good business—good for product development, good for distribution, good for pricing, and good for manufacturing.

Web links

- 1 www.arduino.cc/en/Main/Products
- 2 blog.arduino.cc/2013/07/10/send-in-the-clones/

ÁRTICA



Ártica provides online courses and consulting services focused on how to use digital technology to share knowledge and enable collaboration in arts and culture. Founded in 2011 in Uruguay.

www.articaonline.com

Revenue model: charging for custom services

Interview date: March 9, 2016

Interviewees: Mariana Fossatti and Jorge Gemetto, cofounders

Profile written by Sarah Hinchliff Pearson

The story of Mariana Fossatti and Jorge Gemetto's business, Ártica, is the ultimate example of DIY. Not only are they successful entrepreneurs, the niche in which their small business operates is essentially one they built themselves.

Their dream jobs didn't exist, so they created them.

In 2011, Mariana was a sociologist working for an international organization to develop research and online education about rural-development issues. Jorge was a psychologist, also working in online education. Both were bloggers and heavy users of social media, and both had a passion for arts and culture. They decided to take their skills in digital technology and online learning and apply them to a topic area they loved. They launched Ártica, an online business that provides education and consulting for people and institutions creating artistic and cultural projects on the Internet.

Ártica feels like a uniquely twenty-first century business. The small company has a global online presence with no physical offices. Jorge and Mariana live in Uruguay, and the other two full-time employees, who Jorge and Mariana have never actually met in person, live in Spain. They started by creating a MOOC (massive open online course) about remix culture and collaboration in the arts, which gave them a direct way to reach an international audience, attracting students from across Latin America and Spain. In other words, it is the classic Internet story of being able to directly tap into an audience without relying upon gatekeepers or intermediaries.

Ártica offers personalized education and consulting services, and helps clients implement projects. All of these services are customized. They call it an "artisan" process because of the time and effort it takes to adapt their work for the particular needs of students

IN THE EDUCATIONAL AND CULTURAL BUSINESS, IT IS MORE IMPORTANT TO PAY ATTENTION TO PEOPLE AND PROCESS, RATHER THAN CONTENT OR SPECIFIC FORMATS OR MATERIALS.

and clients. "Each student or client is paying for a specific solution to his or her problems and questions," Mariana said. Rather than sell access to their content, they provide it for free and charge for the personalized services.

When they started, they offered a smaller number of courses designed to attract large audiences. "Over the years, we realized that online communities are more specific than we thought," Mariana said. Ártica now provides more options for classes and has lower enrollment in each course. This means they can provide more attention to individual students and offer classes on more specialized topics.

Online courses are their biggest revenue stream, but they also do more than a dozen consulting projects each year, ranging from digitization to event planning to marketing campaigns. Some are significant in scope, particularly when they work with cultural institutions, and some are smaller projects commissioned by individual artists.

Ártica also seeks out public and private funding for specific projects. Sometimes, even if they are unsuccessful in subsidizing a project like a new course or e-book, they will go ahead because they believe in it. They take the stance that every new project leads them to something new, every new resource they create opens new doors.

Ártica relies heavily on their free Creative Commons-licensed content to attract new

students and clients. Everything they create—online education, blog posts, videos—is published under an Attribution-ShareAlike license (CC BY-SA). "We use a ShareAlike license because we want to give the greatest freedom to our students and readers, and we also want that freedom to be viral," Jorge said. For them, giving others the right to reuse and remix their content is a fundamental value. "How can you offer an online educational service without giving permission to download, make and keep copies, or print the educational resources?" Jorge said. "If we want to do the best for our students—those who trust in us to the point that they are willing to pay online without face-to-face contact—we have to offer them a fair and ethical agreement."

They also believe sharing their ideas and expertise openly helps them build their reputation and visibility. People often share and cite their work. A few years ago, a publisher even picked up one of their e-books and distributed printed copies. Ártica views reuse of their work as a way to open up new opportunities for their business.

This belief that openness creates new opportunities reflects another belief—in serendipity. When describing their process for creating content, they spoke of all of the spontaneous and organic ways they find inspiration. "Sometimes, the collaborative process starts with a conversation between us, or with friends from other projects," Jorge said. "That can be the first step for a new blog post or another simple piece of content, which can evolve to a more complex product in the future, like a course or a book."

Rather than planning their work in advance, they let their creative process be dynamic. "This doesn't mean that we don't need to work hard in order to get good professional results, but the design process is more flexible," Jorge said. They share early and often, and they adjust based on what they learn, always exploring and testing new ideas and ways of operating. In many ways, for them, the process is just as important as the final product.

People and relationships are also just as important, sometimes more. "In the educational and cultural business, it is more important to pay attention to people and process, rather than content or specific formats or materials," Mariana said. "Materials and content are fluid. The important thing is the relationships."

Ártica believes in the power of the network. They seek to make connections with people and institutions across the globe so they can learn from them and share their knowledge.

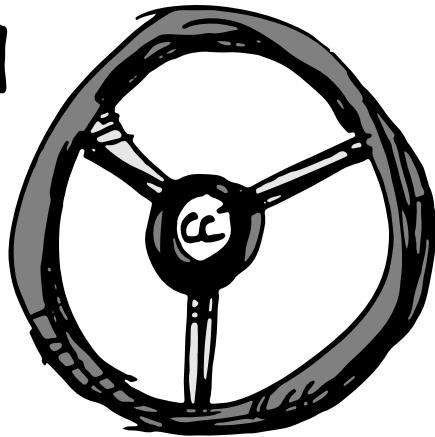
At the core of everything Ártica does is a set of values. "Good content is not enough," Jorge said. "We also think that it is very important to take a stand for some things in the cultural sector." Mariana and Jorge are activists. They defend free culture (the movement promoting the freedom to modify and distribute creative work) and work to demonstrate the intersection between free culture and other social-justice movements. Their efforts to involve people in their work and enable artists and cultural institutions to better use technology are all tied closely to their belief system. Ultimately, what drives their work is a mission to democratize art and culture.

Of course, Ártica also has to make enough money to cover its expenses. Human resources are, by far, their biggest expense. They tap a network of collaborators on a case-by-case basis and hire contractors for specific projects. Whenever possible, they draw from artistic and cultural resources in the commons, and they rely on free software. Their operation is small, efficient, and sustainable, and because of that, it is a success.

"There are lots of people offering online courses," Jorge said. "But it is easy to differentiate us. We have an approach that is very specific and personal." Ártica's model is rooted in the personal at every level. For Mariana and Jorge, success means doing what brings them personal meaning and purpose, and doing it sustainably and collaboratively.

In their work with younger artists, Mariana and Jorge try to emphasize that this model of success is just as valuable as the picture of success we get from the media. "If they seek only the traditional type of success, they will get frustrated," Mariana said. "We try to show them another image of what it looks like."

DRIVEN
SOCIAL



BY
GOOD

BLENDER INSTITUTE



The Blender Institute is an animation studio that creates 3-D films using Blender software. Founded in 2006 in the Netherlands.

www.blender.org

Revenue model: crowdfunding (subscription-based), charging for physical copies, selling merchandise

Interview date: March 8, 2016

Interviewee: Francesco Siddi, production coordinator

Profile written by Sarah Hinchliff Pearson

For Ton Roosendaal, the creator of Blender software and its related entities, sharing is practical. Making their 3-D content creation software available under a free software license has been integral to its development and popularity. Using that software to make movies that were licensed with Creative Commons pushed that development even further. Sharing enables people to participate and to interact with and build upon the technology and content they create in a way that benefits Blender and its community in concrete ways.

Each open-movie project Blender runs produces a host of openly licensed outputs, not just the final film itself but all of the source material as well. The creative process also enhances the development of the Blender software

because the technical team responds directly to the needs of the film production team, creating tools and features that make their lives easier. And, of course, each project involves a long, rewarding process for the creative and technical community working together.

Rather than just talking about the theoretical benefits of sharing and free culture, Ton is very much about *doing* and *making* free culture. Blender's production coordinator Francesco Siddi told us, "Ton believes if you don't make content using your tools, then you're not doing anything."

Blender's history begins in the late 1990s, when Ton created the Blender software. Originally, the software was an in-house resource for his animation studio based in the Netherlands. Investors became interested in the software, so he began marketing the software to the public, offering a free version in addition to a paid version. Sales were disappointing, and his investors gave up on the endeavor in the early 2000s. He made a deal with investors—if he could raise enough money, he could then make the Blender software available under the GNU General Public License.

This was long before Kickstarter and other online crowdfunding sites existed, but Ton ran his own version of a crowdfunding campaign and quickly raised the money he needed. The Blender software became freely available for anyone to use. Simply applying the General Public License to the software, however, was not enough to create a thriving community around it. Francesco told us, "Software of this complexity relies on people and their vision of how people work together. Ton is a fantastic community builder and manager, and he put a lot of work into fostering a community of developers so that the project could live."

Like any successful free and open-source software project, Blender developed quickly because the community could make fixes and improvements. "Software should be free and open to hack," Francesco said. "Otherwise, everyone is doing the same thing in the dark for ten years." Ton set up the Blender Foundation to oversee and steward the software development and maintenance.

After a few years, Ton began looking for new ways to push development of the software. He came up with the idea of creating CC-licensed films using the Blender software. Ton put a call online for all interested and skilled artists. Francesco said the idea was to get the best artists available, put them in a building together with the best developers, and have them work together. They would not only produce high-quality openly licensed content, they would improve the Blender software in the process.

They turned to crowdfunding to subsidize the costs of the project. They had about twenty people working full-time for six to ten months, so the costs were significant. Francesco said that when their crowdfunding campaign succeeded, people were astounded. "The idea that making money was possible by producing CC-licensed material was mind-blowing to people," he said. "They were like, 'I have to see it to believe it.'"

The first film, which was released in 2006, was an experiment. It was so successful that Ton decided to set up the Blender Institute, an entity dedicated to hosting open-movie projects. The Blender Institute's next project was an even bigger success. The film, *Big Buck Bunny*, went viral, and its animated characters were picked up by marketers.

Francesco said that, over time, the Blender Institute projects have gotten bigger and more prominent. That means the filmmaking process has become more complex, combining technical experts and artists who focus on storytelling. Francesco says the process is almost on an industrial scale because of the number of moving parts. This requires a lot of specialized assistance, but the Blender Institute has no problem finding the talent it needs to help on projects. "Blender hardly does any recruiting for film projects because the talent emerges naturally," Francesco said. "So many people want to work with us, and we can't always hire them because of budget constraints."

Blender has had a lot of success raising money from its community over the years. In many ways, the pitch has gotten easier to make. Not only is crowdfunding simply more familiar to the public, but people know and trust Blender to deliver, and Ton has developed a reputation as an effective community leader and visionary for their work. "There is a whole community who sees and understands the benefit of these projects," Francesco said.

While these benefits of each open-movie project make a compelling pitch for crowd-

TON BELIEVES IF YOU DON'T MAKE CONTENT USING YOUR TOOLS, THEN YOU'RE NOT DOING ANYTHING.

funding campaigns, Francesco told us the Blender Institute has found some limitations in the standard crowdfunding model where you propose a specific project and ask for funding. "Once a project is over, everyone goes home," he said. "It is great fun, but then it ends. That is a problem."

To make their work more sustainable, they needed a way to receive ongoing support rather than on a project-by-project basis. Their solution is Blender Cloud, a subscription-style crowdfunding model akin to the online crowdfunding platform, Patreon. For about ten euros each month, subscribers get access to download everything the Blender Institute produces—software, art, training, and more. All of the assets are available under an Attribution license (CC BY) or placed in the public domain (CC0), but they are initially made available only to subscribers. Blender Cloud enables subscribers to follow Blender's movie projects as they develop, sharing detailed information and content used in the creative process. Blender Cloud also has extensive training materials and libraries of characters and other assets used in various projects.

The continuous financial support provided by Blender Cloud subsidizes five to six full-time employees at the Blender Institute. Francesco says their goal is to grow their subscriber base. "This is our freedom," he told us, "and for artists, freedom is everything."

Blender Cloud is the primary revenue stream of the Blender Institute. The Blender Foundation is funded primarily by donations, and that money goes toward software development and maintenance. The revenue streams of the Institute and Foundation are deliberately kept separate. Blender also has other reve-

nue streams, such as the Blender Store, where people can purchase DVDs, T-shirts, and other Blender products.

Ton has worked on projects relating to his Blender software for nearly twenty years. Throughout most of that time, he has been committed to making the software and the content produced with the software free and open. Selling a license has never been part of the business model.

Since 2006, he has been making films available along with all of their source material. He says he has hardly ever seen people stepping into Blender's shoes and trying to make money off of their content. Ton believes this is because the true value of what they do is in the creative and production process. "Even when you share everything, all your original sources, it still takes a lot of talent, skills, time, and budget to reproduce what you did," Ton said.

For Ton and Blender, it all comes back to *doing*.

CARDS AGAINST HUMANITY



Cards Against Humanity is a private, for-profit company that makes a popular party game by the same name. Founded in 2011 in the U.S.

www.cardsagainsthumanity.com

Revenue model: charging for physical copies

Interview date: February 3, 2016
Interviewee: Max Temkin, cofounder

Profile written by Sarah Hinchliff Pearson

If you ask cofounder Max Temkin, there is nothing particularly interesting about the Cards Against Humanity business model. "We make a product. We sell it for money. Then we spend less money than we make," Max said.

He is right. Cards Against Humanity is a simple party game, modeled after the game Apples to Apples. To play, one player asks a question or fill-in-the-blank statement from a black card, and the other players submit their funniest white card in response. The catch is that all of the cards are filled with crude, gruesome, and otherwise awful things. For the right kind of people ("horrible people," according to Cards Against Humanity advertising), this makes for a hilarious and fun game.

The revenue model is simple. Physical copies of the game are sold for a profit. And it works. At the time of this writing, Cards Against

Humanity is the number-one best-selling item out of all toys and games on Amazon. There are official expansion packs available, and several official themed packs and international editions as well.

But Cards Against Humanity is also available for free. Anyone can download a digital version of the game on the Cards Against Humanity website. More than one million people have downloaded the game since the company began tracking the numbers.

The game is available under an Attribution-NonCommercial-ShareAlike license (CC BY-NC-SA). That means, in addition to copying the game, anyone can create new versions of the game as long as they make it available under the same noncommercial terms. The ability to adapt the game is like an entire new game unto itself.

All together, these factors—the crass tone of the game and company, the free download, the openness to fans remixing the game—give the game a massive cult following.

Their success is not the result of a grand plan. Instead, Cards Against Humanity was the last in a long line of games and comedy projects that Max Temkin and his friends put together for their own amusement. As Max tells the story, they made the game so they could play it themselves on New Year's Eve because they were too nerdy to be invited to other parties. The game was a hit, so they decided to put it up online as a free PDF. People started asking if they could pay to have the game printed for them, and eventually they decided to run a Kickstarter to fund the printing. They set their Kickstarter goal at \$4,000—and raised \$15,000. The game was officially released in May 2011.

The game caught on quickly, and it has only grown more popular over time. Max says the eight founders never had a meeting where they decided to make it an ongoing business. "It kind of just happened," he said.

But this tale of a "happy accident" belies marketing genius. Just like the game, the Cards Against Humanity brand is irreverent and memorable. It is hard to forget a company that calls the FAQ on their website "Your dumb questions."

Like most quality satire, however, there is more to the joke than vulgarity and shock value. The company's marketing efforts around Black Friday illustrate this particularly well. For those outside the United States, Black Friday is the term for the day after the Thanksgiving holiday, the biggest shopping day of the year. It is an incredibly important day for Cards Against Humanity, like it is for all U.S. retailers. Max said they struggled with what to do on Black Friday because they didn't want to support what he called the "orgy of consumerism" the day has become, particularly since it follows a day that is about being grateful for what you

have. In 2013, after deliberating, they decided to have an Everything Costs \$5 More sale.

"We sweated it out the night before Black Friday, wondering if our fans were going to hate us for it," he said. "But it made us laugh so we went with it. People totally caught the joke."

This sort of bold transparency delights the media, but more importantly, it engages their fans. "One of the most surprising things you can do in capitalism is just be honest with people," Max said. "It shocks people that there is transparency about what you are doing."

Max also likened it to a grand improv scene. "If we do something a little subversive and unexpected, the public wants to be a part of the joke." One year they did a Give Cards Against Humanity \$5 event, where people literally paid them five dollars for no reason. Their fans wanted to make the joke funnier by making it successful. They made \$70,000 in a single day.

This remarkable trust they have in their customers is what inspired their decision to apply a Creative Commons license to the game. Trusting your customers to reuse and remix your work requires a leap of faith. Cards Against Humanity obviously isn't afraid of doing the unexpected, but there are lines even they do not want to cross. Before applying the license, Max said they worried that some fans would adapt the game to include all of the jokes they intentionally never made because they crossed that line. "It happened, and the world didn't end," Max said. "If that is the worst cost of using CC, I'd pay that a hundred times over because there are so many benefits."

Any successful product inspires its biggest fans to create remixes of it, but unsanctioned adaptations are more likely to fly under the radar. The Creative Commons license gives fans of Cards Against Humanity the freedom to run with the game and copy, adapt, and promote their creations openly. Today there are thousands of fan expansions of the game.

Max said, "CC was a no-brainer for us because it gets the most people involved. Making the game free *and* available under a CC license led to the unbelievable situation where we are one of the best-marketed games in the world, and we have never spent a dime on marketing."

Of course, there are limits to what the company allows its customers to do with the game. They chose the Attribution-NonCommercial-ShareAlike license because it restricts people from using the game to make money. It also requires that adaptations of the game be made available under the same licensing terms if they are shared publicly. Cards Against Humanity also polices its brand. "We feel like we're the only ones who can use our brand and our game and make money off of it," Max said. About 99.9 percent of the time, they just send an email to those making commercial use of the game, and that is the end of it. There have only been a handful of instances where they had to get a lawyer involved.

Just as there is more than meets the eye to the Cards Against Humanity business model, the same can be said of the game itself. To be playable, every white card has to work syntactically with enough black cards. The eight creators invest an incredible amount of work into creating new cards for the game. "We have daylong arguments about commas," Max said. "The slacker tone of the cards gives people the impression that it is easy to write them, but it is actually a lot of work and quibbling."

That means cocreation with their fans really doesn't work. The company has a submission mechanism on their website, and they get thousands of suggestions, but it is very rare that a submitted card is adopted. Instead, the eight initial creators remain the primary authors of expansion decks and other new products released by the company. Interestingly, the creativity of their customer base is really only an asset to the company once their original work is created and published when people make their own adaptations of the game.

CC WAS A NO-BRAINER FOR US BECAUSE IT GETS THE MOST PEOPLE INVOLVED. MAKING THE GAME FREE AND AVAILABLE UNDER A CC LICENSE LED TO THE UNBELIEVABLE SITUATION WHERE WE ARE ONE OF THE BEST-MARKETED GAMES IN THE WORLD, AND WE HAVE NEVER SPENT A DIME ON MARKETING.

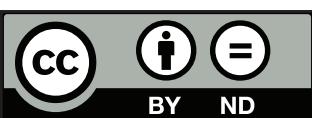
For all of their success, the creators of Cards Against Humanity are only partially motivated by money. Max says they have always been interested in the Walt Disney philosophy of financial success. "We don't make jokes and games to make money—we make money so we can make more jokes and games," he said.

In fact, the company has given more than \$4 million to various charities and causes. "Cards is not our life plan," Max said. "We all have other interests and hobbies. We are passionate about other things going on in our lives. A lot of the activism we have done comes out of us taking things from the rest of our lives and channeling some of the excitement from the game into it."

Seeing money as fuel rather than the ultimate goal is what has enabled them to embrace Creative Commons licensing without reservation. CC licensing ended up being a savvy marketing move for the company, but nonetheless, giving up exclusive control of your work necessarily means giving up some opportunities to extract more money from customers.

"It's not right for everyone to release everything under CC licensing," Max said. "If your only goal is to make a lot of money, then CC is not best strategy. *This* kind of business model, though, speaks to your values, and who you are and why you're making things."

THE CONVERSATION



The Conversation is an independent source of news, sourced from the academic and research community and delivered direct to the public over the Internet. Founded in 2011 in Australia.

theconversation.com

Revenue model: charging content creators (universities pay membership fees to have their faculties serve as writers), grant funding

Interview date: February 4, 2016
Interviewee: Andrew Jaspan, founder

Profile written by Paul Stacey

Andrew Jaspan spent years as an editor of major newspapers including the *Observer* in London, the *Sunday Herald* in Glasgow, and the *Age* in Melbourne, Australia. He experienced first-hand the decline of newspapers, including the collapse of revenues, layoffs, and the constant pressure to reduce costs. After he left the *Age* in 2005, his concern for the future journalism didn't go away. Andrew made a commitment to come up with an alternative model.

Around the time he left his job as editor of the *Melbourne Age*, Andrew wondered where citizens would get news grounded in fact and evidence rather than opinion or ideology. He believed there was still an appetite for journalism

with depth and substance but was concerned about the increasing focus on the sensational and sexy.

While at the *Age*, he'd become friends with a vice-chancellor of a university in Melbourne who encouraged him to talk to smart people across campus—an astrophysicist, a Nobel laureate, earth scientists, economists... These were the kind of smart people he wished were more involved in informing the world about what is going on and correcting the errors that appear in media. However, they were reluctant to engage with mass media. Often, journalists didn't understand what they said, or unilaterally chose what aspect of a story to tell, putting out a version that these people felt was wrong or mischaracterized. Newspapers want to attract a mass audience. Scholars want to com-

municate serious news, findings, and insights. It's not a perfect match.

Universities are massive repositories of knowledge, research, wisdom, and expertise. But a lot of that stays behind a wall of their own making—there are the walled garden and ivory tower metaphors, and in more literal terms, the paywall. Broadly speaking, universities are part of society but disconnected from it. They are an enormous public resource but not that good at presenting their expertise to the wider public.

Andrew believed he could help connect academics back into the public arena, and maybe help society find solutions to big problems. He thought about pairing professional editors with university and research experts, working one-on-one to refine everything from story structure to headline, captions, and quotes. The editors could help turn something that is academic into something understandable and readable. And this would be a key difference from traditional journalism—the subject matter expert would get a chance to check the article and give final approval before it is published. Compare this with reporters just picking and choosing the quotes and writing whatever they want.

The people he spoke to liked this idea, and Andrew embarked on raising money and support with the help of the Commonwealth Scientific and Industrial Research Organisation (CSIRO), the University of Melbourne, Monash University, the University of Technology Sydney, and the University of Western Australia. These founding partners saw the value of an independent information channel that would also showcase the talent and knowledge of the university and research sector. With their help, in 2011, the Conversation, was launched as an independent news site in Australia. Everything published in the Conversation is openly licensed with Creative Commons.

The Conversation is founded on the belief that underpinning a functioning democracy is

access to independent, high-quality, informative journalism. The Conversation's aim is for people to have a better understanding of current affairs and complex issues—and hopefully a better quality of public discourse. The Conversation sees itself as a source of trusted information dedicated to the public good. Their core mission is simple: to provide readers with a reliable source of evidence-based information.

Andrew worked hard to reinvent a methodology for creating reliable, credible content. He introduced strict new working practices, a charter, and codes of conduct.¹ These include fully disclosing who every author is (with their relevant expertise); who is funding their research; and if there are any potential or real conflicts of interest. Also important is where

ACCESS TO INFORMATION IS AN ISSUE OF EQUALITY—EVERYONE SHOULD HAVE ACCESS, LIKE ACCESS TO CLEAN WATER.

the content originates, and even though it comes from the university and research community, it still needs to be fully disclosed.

The Conversation does not sit behind a paywall. Andrew believes access to information is an issue of equality—everyone should have access, like access to clean water. The Conversation is committed to an open and free Internet. Everyone should have free access to their content, and be able to share it or republish it.

Creative Commons help with these goals; articles are published with the Attribution-NoDerivs license (CC BY-ND). They're freely available for others to republish elsewhere as long as attribution is given and the content is not edited. Over five years, more than twenty-two thousand sites have republished their content. The Conversation website gets about 2.9 million unique views per month,

but through republication they have *thirty-five* million readers. This couldn't have been done without the Creative Commons license, and in Andrew's view, Creative Commons is central to everything the Conversation does.

When readers come across the Conversation, they seem to like what they find and recommend it to their friends, peers, and networks. Readership has grown primarily through word of mouth. While they don't have sales and marketing, they do promote their work through social media (including Twitter and Facebook), and by being an accredited supplier to Google News.

It's usual for the founders of any company to ask themselves what kind of company it should be. It quickly became clear to the founders of the Conversation that they wanted to create a public good rather than make money off of information. Most media companies are working to aggregate as many eyeballs as possible and sell ads. The Conversation founders didn't want this model. It takes no advertising and is a not-for-profit venture.

There are now different editions of the Conversation for Africa, the United Kingdom, France, and the United States, in addition to the one for Australia. All five editions have their own editorial mastheads, advisory boards, and content. The Conversation's global virtual newsroom has roughly ninety staff working with thirty-five thousand academics from over sixteen hundred universities around the world. The Conversation would like to be working with university scholars from even more parts of the world.

Additionally, each edition has its own set of founding partners, strategic partners, and funders. They've received funding from foundations, corporates, institutions, and individual donations, but the Conversation is shifting toward paid memberships by universities and research institutions to sustain operations. This would safeguard the current service and help improve coverage and features.

When professors from member universities write an article, there is some branding of the university associated with the article. On the Conversation website, paying university members are listed as "members and funders." Early participants may be designated as "founding members," with seats on the editorial advisory board.

Academics are not paid for their contributions, but they get free editing from a professional (four to five hours per piece, on average). They also get access to a large audience. Every author and member university has access to a special analytics dashboard where they can check the reach of an article. The metrics include what people are tweeting, the comments, countries the readership represents, where the article is being republished, and the number of readers per article.

The Conversation plans to expand the dashboard to show not just reach but impact. This tracks activities, behaviors, and events that occurred as a result of publication, including things like a scholar being asked to go on a show to discuss their piece, give a talk at a conference, collaborate, submit a journal paper, and consult a company on a topic.

These reach and impact metrics show the benefits of membership. With the Conversation, universities can engage with the public and show why they're of value.

With its tagline, "Academic Rigor, Journalistic Flair," the Conversation represents a new form of journalism that contributes to a more informed citizenry and improved democracy around the world. Its open business model and use of Creative Commons show how it's possible to generate both a public good and operational revenue at the same time.

Web link

- 1 theconversation.com/us/charter

CORY DOCTOROW



Cory Doctorow is a science fiction writer, activist, blogger, and journalist. Based in the U.S.

raphound.com and boingboing.net

Revenue model: charging for physical copies (book sales), pay-what-you-want, selling translation rights to books

Interview date: January 12, 2016

Profile written by Sarah Hinchliff Pearson

Cory Doctorow hates the term “business model,” and he is adamant that he is not a brand. “To me, branding is the idea that you can take a thing that has certain qualities, remove the qualities, and go on selling it,” he said. “I’m not out there trying to figure out how to be a brand. I’m doing this thing that animates me to work crazy insane hours because it’s the most important thing I know how to do.”

Cory calls himself an entrepreneur. He likes to say his success came from making stuff people happened to like and then getting out of the way of them sharing it.

He is a science fiction writer, activist, blogger, and journalist. Beginning with his first novel, *Down and Out in the Magic Kingdom*, in 2003, his work has been published under a Creative Commons license. Cory is coeditor of the popular CC-licensed site *Boing Boing*, where he

writes about technology, politics, and intellectual property. He has also written several nonfiction books, including the most recent *Information Doesn’t Want to Be Free*, about the ways in which creators can make a living in the Internet age.

Cory primarily makes money by selling physical books, but he also takes on paid speaking gigs and is experimenting with pay-what-you-want models for his work.

While Cory’s extensive body of fiction work has a large following, he is just as well known for his activism. He is an outspoken opponent of restrictive copyright and digital-rights-management (DRM) technology used to lock up content because he thinks both undermine creators and the public interest. He is currently a special adviser at the Electronic Frontier Foundation, where he is involved in a lawsuit

challenging the U.S. law that protects DRM. Cory says his political work doesn't directly make him money, but if he gave it up, he thinks he would lose credibility and, more importantly, lose the drive that propels him to create. "My political work is a different expression of the same artistic-political urge," he said. "I have this suspicion that if I gave up the things that didn't make me money, the genuineness would leach out of what I do, and the quality that causes people to like what I do would be gone."

Cory has been financially successful, but money is not his primary motivation. At the start of his book *Information Doesn't Want to Be Free*, he stresses how important it is not to become an artist if your goal is to get rich. "Entering the arts because you want to get rich is like buying lottery tickets because you want to get rich," he wrote. "It might work, but it almost certainly won't. Though, of course, someone always wins the lottery." He acknowledges that he is one of the lucky few to "make it," but he says he would be writing no matter what. "I am compelled to write," he wrote. "Long before I wrote to keep myself fed and sheltered, I was writing to keep myself sane."

Just as money is not his primary motivation to create, money is not his primary motivation to share. For Cory, sharing his work with Creative Commons is a moral imperative. "It felt morally right," he said of his decision to adopt Creative Commons licenses. "I felt like I wasn't contributing to the culture of surveillance and censorship that has been created to try to stop copying." In other words, using CC licenses symbolizes his worldview.

He also feels like there is a solid commercial basis for licensing his work with Creative Commons. While he acknowledges he hasn't been able to do a controlled experiment to compare the commercial benefits of licensing with CC against reserving all rights, he thinks he has sold more books using a CC license than he would have without it. Cory says his goal is to

convince people they should pay him for his work. "I started by not calling them thieves," he said.

Cory started using CC licenses soon after they were first created. At the time his first novel came out, he says the science fiction genre was overrun with people scanning and downloading books without permission. When he and his publisher took a closer look at who was doing that sort of thing online, they realized it looked a lot like book promotion. "I knew there was a relationship between having enthusiastic readers and having a successful career as a writer," he said. "At the time, it took eighty hours to OCR a book, which is a big effort. I decided to spare them the time and energy, and give them the book for free in a format destined to spread."

Cory admits the stakes were pretty low for him when he first adopted Creative Commons licenses. He only had to sell two thousand copies of his book to break even. People often said he was only able to use CC licenses successfully at that time because he was just starting out. Now they say he can only do it because he is an established author.

The bottom line, Cory says, is that no one has found a way to prevent people from copying the stuff they like. Rather than fighting the tide, Cory makes his work intrinsically shareable. "Getting the hell out of the way for people who want to share their love of you with other people sounds obvious, but it's remarkable how many people don't do it," he said.

Making his work available under Creative Commons licenses enables him to view his biggest fans as his ambassadors. "Being open to fan activity makes you part of the conversation about what fans do with your work and how they interact with it," he said. Cory's own website routinely highlights cool things his audience has done with his work. Unlike corporations like Disney that tend to have a hands-off relationship with their fan activity, he has a symbiotic relationship with his audience. "En-

gaging with your audience can't guarantee you success," he said. "And Disney is an example of being able to remain aloof and still being the most successful company in the creative industry in history. But I figure my likelihood of being Disney is pretty slim, so I should take all the help I can get."

His first book was published under the most restrictive Creative Commons license, Attribution-NonCommercial-NoDerivs (CC BY-NC-ND). It allows only verbatim copying for noncommercial purposes. His later work is published under the Attribution-NonCommercial-ShareAlike license (CC BY-NC-SA), which gives people the right to adapt his work for noncommercial purposes but only if they share it back under the same license terms. Before releasing his work under a CC license that allows adaptations, he always sells the right to translate the book to other languages to a commercial publisher first. He wants to reach new potential buyers in other parts of the world, and he thinks it is more difficult to get people to pay for translations if there are fan translations already available for free.

In his book *Information Doesn't Want to Be Free*, Cory likens his philosophy to thinking like a dandelion. Dandelions produce thousands of seeds each spring, and they are blown into the air going in every direction. The strategy is to maximize the number of blind chances the dandelion has for continuing its genetic line. Similarly, he says there are lots of people out there who may want to buy creative work or compensate authors for it in some other way.

GETTING THE HELL OUT OF THE WAY FOR PEOPLE WHO WANT TO SHARE THEIR LOVE OF YOU WITH OTHER PEOPLE SOUNDS OBVIOUS, BUT IT'S REMARKABLE HOW MANY PEOPLE DON'T DO IT.

"The more places your work can find itself, the greater the likelihood that it will find one of those would-be customers in some unsuspected crack in the metaphorical pavement," he wrote. "The copies that others make of my work cost me nothing, and present the possibility that I'll get something."

Applying a CC license to his work increases the chances it will be shared more widely around the Web. He avoids DRM—and openly opposes the practice—for similar reasons. DRM has the effect of tying a work to a particular platform. This digital lock, in turn, strips the authors of control over their own work and hands that control over to the platform. He calls it Cory's First Law: "Anytime someone puts a lock on something that belongs to you and won't give you the key, that lock isn't there for your benefit."

Cory operates under the premise that artists benefit when there are more, rather than fewer, places where people can access their work. The Internet has opened up those avenues, but DRM is designed to limit them. "On the one hand, we can credibly make our work available to a widely dispersed audience," he said. "On the other hand, the intermediaries we historically sold to are making it harder to go around them." Cory continually looks for ways to reach his audience without relying upon major platforms that will try to take control over his work.

Cory says his e-book sales have been lower than those of his competitors, and he attributes some of that to the CC license making the work available for free. But he believes people are willing to pay for content they like, even when it is available for free, as long as it is easy to do. He was extremely successful using Humble Bundle, a platform that allows people to pay what they want for DRM-free versions of a bundle of a particular creator's work. He is planning to try his own pay-what-you-want experiment soon.

Fans are particularly willing to pay when they feel personally connected to the artist. Cory works hard to create that personal connection. One way he does this is by personally answering every single email he gets. "If you look at the history of artists, most die in penury," he said. "That reality means that for artists, we have to find ways to support ourselves when public tastes shift, when copyright stops producing. Future-proofing your artistic career in many ways means figuring out how to stay connected to those people who have been touched by your work."

Cory's realism about the difficulty of making a living in the arts does not reflect pessimism about the Internet age. Instead, he says the fact that it is hard to make a living as an artist is nothing new. What is new, he writes in his book, "is how many ways there are to make things, and to get them into other people's hands and minds."

It has never been easier to think like a dandelion.

FIGSHARE

Figshare is a for-profit company offering an online repository where researchers can preserve and share the output of their research, including figures, data sets, images, and videos. Founded in 2011 in the UK.

Interview date: January 28, 2016

Interviewee: Mark Hahnel, founder

Profile written by Paul Stacey

Figshare's mission is to change the face of academic publishing through improved dissemination, discoverability, and reusability of scholarly research. Figshare is a repository where users can make all the output of their research available—from posters and presentations to data sets and code—in a way that's easy to discover, cite, and share. Users can upload any file format, which can then be previewed in a Web browser. Research output is disseminated in a way that the current scholarly-publishing model does not allow.

Figshare founder Mark Hahnel often gets asked: How do you make money? How do we know you'll be here in five years? Can you, as a for-profit venture, be trusted? Answers have evolved over time.

Mark traces the origins of Figshare back to when he was a graduate student getting his PhD in stem cell biology. His research involved working with videos of stem cells in motion. However, when he went to publish his re-

figshare.com

Revenue model: platform providing paid services to creators

search, there was no way for him to also publish the videos, figures, graphs, and data sets. This was frustrating. Mark believed publishing his complete research would lead to more citations and be better for his career.

Mark does not consider himself an advanced software programmer. Fortunately, things like cloud-based computing and wikis had become mainstream, and he believed it ought to be possible to put all his research online and share it with anyone. So he began working on a solution.

There were two key needs: licenses to make the data citable, and persistent identifiers—URL links that always point back to the original object ensuring the research is citable for the long term.

Mark chose Digital Object Identifiers (DOIs) to meet the need for a persistent identifier. In the DOI system, an object's metadata is stored as a series of numbers in the DOI name. Referring to an object by its DOI is more stable than referring to it by its URL, because the location of an object (the web page or URL) can often

CHANGE THE FACE OF ACADEMIC PUBLISHING THROUGH IMPROVED DISSEMINATION, DISCOVERABILITY, AND REUSABILITY OF SCHOLARLY RESEARCH.

change. Mark partnered with DataCite for the provision of DOIs for research data.

As for licenses, Mark chose Creative Commons. The open-access and open-science communities were already using and recommending Creative Commons. Based on what was happening in those communities and Mark's dialogue with peers, he went with CC0 (in the public domain) for data sets and CC BY (Attribution) for figures, videos, and data sets.

So Mark began using DOIs and Creative Commons for his own research work. He had a science blog where he wrote about it and made all his data open. People started commenting on his blog that they wanted to do the same. So he opened it up for them to use, too.

People liked the interface and simple upload process. People started asking if they could also share theses, grant proposals, and code. Inclusion of code raised new licensing issues, as Creative Commons licenses are not used for software. To allow the sharing of software code, Mark chose the MIT license, but GNU and Apache licenses can also be used.

Mark sought investment to make this into a scalable product. After a few unsuccessful funding pitches, UK-based Digital Science expressed interest but insisted on a more viable business model. They made an initial investment, and together they came up with a freemium-like business model.

Under the freemium model, academics upload their research to Figshare for storage and sharing for free. Each research object is licensed with Creative Commons and receives a DOI link. The *premium* option charges researchers a fee for gigabytes of private storage space, and for private online space designed for a set number of research collaborators, which is ideal for larger teams and geographically dispersed research groups. Figshare sums up its value proposition to researchers as "You retain ownership. You license it. You get credit. We just make sure it persists."

In January 2012, Figshare was launched. (The *fig* in Figshare stands for *figures*.) Using investment funds, Mark made significant improvements to Figshare. For example, researchers could quickly preview their research files within a browser without having to download them first or require third-party software. Journals who were still largely publishing articles as static noninteractive PDFs became interested in having Figshare provide that functionality for them.

Figshare diversified its business model to include services for journals. Figshare began hosting large amounts of data for the journals' online articles. This additional data improved the quality of the articles. Outsourcing this service to Figshare freed publishers from having to develop this functionality as part of their own infrastructure. Figshare-hosted data also provides a link back to the article, generating additional click-through and readership—a benefit to both journal publishers and researchers. Figshare now provides research-data infrastructure for a wide variety of publishers including Wiley, Springer Nature, PLOS, and Taylor and Francis, to name a few, and has convinced them to use Creative Commons licenses for the data.

Governments allocate significant public funds to research. In parallel with the launch of Figshare, governments around the world began requesting the research they fund be open

and accessible. They mandated that researchers and academic institutions better manage and disseminate their research outputs. Institutions looking to comply with this new mandate became interested in Figshare. Figshare once again diversified its business model, adding services for institutions.

Figshare now offers a range of fee-based services to institutions, including their own minibranded Figshare space (called Figshare for Institutions) that securely hosts research data of institutions in the cloud. Services include not just hosting but data metrics, data dissemination, and user-group administration. Figshare's workflow, and the services they offer for institutions, take into account the needs of librarians and administrators, as well as of the researchers.

As with researchers and publishers, Figshare encouraged institutions to share their research with CC BY (Attribution) and their data with CC0 (into the public domain). Funders who require researchers and institutions to use open licensing believe in the social responsibilities and benefits of making research accessible to all. Publishing research in this open way has come to be called open access. But not all funders specify CC BY; some institutions want to offer their researchers a choice, including less permissive licenses like CC BY-NC (Attribution-NonCommercial), CC BY-SA (Attribution-ShareAlike), or CC BY-ND (Attribution-NoDerivs).

For Mark this created a conflict. On the one hand, the principles and benefits of open science are at the heart of Figshare, and Mark believes CC BY is the best license for this. On the other hand, institutions were saying they wouldn't use Figshare unless it offered a choice in licenses. He initially refused to offer anything beyond CC0 and CC BY, but after seeing an open-source CERN project offer all Creative Commons licenses without any negative repercussions, he decided to follow suit.

Mark is thinking of doing a Figshare study that tracks research dissemination according to Creative Commons license, and gathering metrics on views, citations, and downloads.

You could see which license generates the biggest impact. If the data showed that CC BY is more impactful, Mark believes more and more researchers and institutions will make it their license of choice.

Figshare has an Application Programming Interface (API) that makes it possible for data to be pulled from Figshare and used in other applications. As an example, Mark shared a Figshare data set showing the journal subscriptions that higher-education institutions in the United Kingdom paid to ten major publishers.¹ Figshare's API enables that data to be pulled into an app developed by a completely different researcher that converts the data into a visually interesting graph, which any viewer can alter by changing any of the variables.²

The free version of Figshare has built a community of academics, who through word of mouth and presentations have promoted and spread awareness of Figshare. To amplify and reward the community, Figshare established an Advisor program, providing those who promoted Figshare with hoodies and T-shirts, early access to new features, and travel expenses when they gave presentations outside of their area. These Advisors also helped Mark on what license to use for software code and whether to offer universities an option of using Creative Commons licenses.

Mark says his success is partly about being in the right place at the right time. He also believes that the diversification of Figshare's model over time has been key to success. Figshare now offers a comprehensive set of services to researchers, publishers, and institutions.³ If he had relied solely on revenue from premium subscriptions, he believes Figshare would have struggled. In Figshare's early days, their primary users were early-career and late-career academics. It has only been because funders mandated open licensing that Figshare is now being used by the mainstream.

Today Figshare has 26 million-plus page views, 7.5 million-plus downloads, 800,000-plus user uploads, 2 million-plus articles, 500,000-plus collections, and 5,000-plus projects. Sixty percent of their traffic comes from Google. A sister company called Altmetric tracks the use of Figshare by others, including Wikipedia and news sources.

Figshare uses the revenue it generates from the premium subscribers, journal publishers, and institutions to fund and expand what it can offer to researchers for free. Figshare has publicly stuck to its principles—keeping the free service free and requiring the use of CC BY and CC0 from the start—and from Mark's perspective, this is why people trust Figshare. Mark sees new competitors coming forward who are just in it for money. If Figshare was only in it for the money, they wouldn't care about offering a free version. Figshare's principles and advocacy for openness are a key differentiator. Going forward, Mark sees Figshare not only as supporting open access to research but also enabling people to collaborate and make new discoveries.

Web links

- 1 figshare.com/articles/Journal_subscription_costs_FOIs_to_UK_universities/1186832
- 2 retr0.shinyapps.io/journal_costs/?year=2014&inst=19,22,38,42,59,64,80,95,136
- 3 figshare.com/features

FIGURE.NZ



Figure.NZ is a nonprofit charity that makes an online data platform designed to make data reusable and easy to understand. Founded in 2012 in New Zealand.

figure.nz

Revenue model: platform providing paid services to creators, donations, sponsorships

Interview date: May 3, 2016

Interviewee: Lillian Grace, founder

Profile written by Paul Stacey

In the paper *Harnessing the Economic and Social Power of Data* presented at the New Zealand Data Futures Forum in 2014,¹ Figure.NZ founder Lillian Grace said there are thousands of valuable and relevant data sets freely available to us right now, but most people don't use them. She used to think this meant people didn't care about being informed, but she's come to see that she was wrong. Almost everyone wants to be informed about issues that matter—not only to them, but also to their families, their communities, their businesses, and their country. But there's a big difference between availability and accessibility of information. Data is spread across thousands of sites and is held within databases and spreadsheets that require both time and skill to engage with. To use data when making a decision, you have to know what specific question to ask, identify a source that has collected the data, and manipulate complex tools to extract and visualize the information within the data set. Lillian established Figure.NZ to make data

truly accessible to all, with a specific focus on New Zealand.

Lillian had the idea for Figure.NZ in February 2012 while working for the New Zealand Institute, a think tank concerned with improving economic prosperity, social well-being, environmental quality, and environmental productivity for New Zealand and New Zealanders. While giving talks to community and business groups, Lillian realized "every single issue we addressed would have been easier to deal with if more people understood the basic facts." But understanding the basic facts sometimes requires data and research that you often have to pay for.

Lillian began to imagine a website that lifted data up to a visual form that could be easily understood and freely accessed. Initially launched as Wiki New Zealand, the original idea was that people could contribute their

data and visuals via a wiki. However, few people had graphs that could be used and shared, and there were no standards or consistency around the data and the visuals. Realizing the wiki model wasn't working, Lillian brought the process of data aggregation, curation, and visual presentation in-house, and invested in the technology to help automate some of it. Wiki New Zealand became Figure.NZ, and efforts were reoriented toward providing services to those wanting to open their data and present it visually.

Here's how it works. Figure.NZ sources data from other organizations, including corporations, public repositories, government departments, and academics. Figure.NZ imports and extracts that data, and then validates and standardizes it—all with a strong eye on what will be best for users. They then make the data available in a series of standardized forms, both human- and machine-readable, with rich metadata about the sources, the licenses, and data types. Figure.NZ has a chart-designing tool that makes simple bar, line, and area graphs from any data source. The graphs are posted to the Figure.NZ website, and they can also be exported in a variety of formats for print or online use. Figure.NZ makes its data and graphs available using the Attribution (CC BY) license. This allows others to reuse, revise, remix, and redistribute Figure.NZ data and graphs as long as they give attribution to the original source and to Figure.NZ.

Lillian characterizes the initial decision to use Creative Commons as naively fortunate. It was first recommended to her by a colleague. Lillian spent time looking at what Creative Commons offered and thought it looked good, was clear, and made common sense. It was easy to use and easy for others to understand. Over time, she's come to realize just how fortunate and important that decision turned out to be. New Zealand's government has an open-access and licensing framework called NZGOAL, which provides guidance for agencies when they release copyrighted and noncopyrighted work and material.² It aims to standardize the licensing of works with government copyright

and how they can be reused, and it does this with Creative Commons licenses. As a result, 98 percent of all government-agency data is Creative Commons licensed, fitting in nicely with Figure.NZ's decision.

Lillian thinks current ideas of what a business is are relatively new, only a hundred years old or so. She's convinced that twenty years from now, we will see new and different models for business. Figure.NZ is set up as a nonprofit charity. It is purpose-driven but also strives to pay people well and thinks like a business. Lillian sees the charity-nonprofit status as an essential element for the mission and purpose of Figure.NZ. She believes Wikipedia would not work if it were for profit, and similarly, Figure.NZ's nonprofit status assures people who have data and people who want to use it that they can rely on Figure.NZ's motives. People see them as a trusted wrangler and source.

Although Figure.NZ is a social enterprise that openly licenses their data and graphs for everyone to use for free, they have taken care not to be perceived as a free service all around the table. Lillian believes hundreds of millions of dollars are spent by the government and organizations to collect data. However, very little money is spent on taking that data and making it accessible, understandable, and useful for decision making. Government uses some of the data for policy, but Lillian believes that it is underutilized and the potential value is much larger. Figure.NZ is focused on solving that problem. They believe a portion of money allocated to collecting data should go into making sure that data is useful and generates value. If the government wants citizens to understand why certain decisions are being made and to be more aware about what the government is doing, why not transform the data it collects into easily understood visuals? It could even become a way for a government or any organization to differentiate, market, and brand itself.

Figure.NZ spends a lot of time seeking to understand the motivations of data collectors and to identify the channels where it can provide value. Every part of their business model has been focused on who is going to get value from the data and visuals.

Figure.NZ has multiple lines of business. They provide commercial services to organizations that want their data publicly available and want to use Figure.NZ as their publishing platform. People who want to publish open data appreciate Figure.NZ's ability to do it faster, more easily, and better than they can. Customers are encouraged to help their users find, use, and make things from the data they make available on Figure.NZ's website. Customers control what is released and the license terms (although Figure.NZ encourages Creative Commons licensing). Figure.NZ also serves customers who want a specific collection of charts created—for example, for their website or annual report. Charging the organizations that want to make their data available enables Figure.NZ to provide their site free to all users, to truly democratize data.

Lillian notes that the current state of most data is terrible and often not well understood by the people who have it. This sometimes makes it difficult for customers and Figure.NZ to figure out what it would cost to import, standardize, and display that data in a useful way. To deal with this, Figure.NZ uses “high-trust contracts,” where customers allocate a certain budget to the task that Figure.NZ is then free to draw from, as long as Figure.NZ frequently reports on what they’ve produced so the customer can determine the value for money. This strategy has helped build trust and transparency about the level of effort associated with doing work that has never been done before.

A second line of business is what Figure.NZ calls *partners*. ASB Bank and Statistics New Zealand are partners who back Figure.NZ’s efforts. As one example, with their support Figure.NZ has been able to create Business Figures, a special way for businesses to find useful data without having to know what questions to ask.³

Figure.NZ also has patrons.⁴ Patrons donate to topic areas they care about, directly enabling Figure.NZ to get data together to flesh out those areas. Patrons do not direct what data is included or excluded.

Figure.NZ also accepts philanthropic donations, which are used to provide more content, extend technology, and improve services, or are targeted to fund a specific effort or provide in-kind support. As a charity, donations are tax deductible.

Figure.NZ has morphed and grown over time. With data aggregation, curation, and visualizing services all in-house, Figure.NZ has developed a deep expertise in taking random styles of data, standardizing it, and making it useful. Lillian realized that Figure.NZ could easily become a warehouse of seventy people doing data. But for Lillian, growth isn’t always good. In her view, bigger often means less effective. Lillian set artificial constraints on growth, forcing the organization to think differently and be more efficient. Rather than in-house growth, they are growing and building *external* relationships.

IN THE WORLD WE LIVE IN NOW, THE BEST FUTURE IS THE ONE WHERE EVERYONE CAN MAKE WELL-INFORMED DECISIONS.

Figure.NZ’s website displays visuals and data associated with a wide range of categories including crime, economy, education, employment, energy, environment, health, information and communications technology, industry, tourism, and many others. A search function helps users find tables and graphs. Figure.NZ does not provide analysis or interpretation of the data or visuals. Their goal is to

teach people how to think, not think for them. Figure.NZ wants to create intuitive experiences, not user manuals.

Figure.NZ believes data and visuals should be useful. They provide their customers with a data collection template and teach them why it's important and how to use it. They've begun putting more emphasis on tracking what users of their website want. They also get requests from social media and through email for them to share data for a specific topic—for example, can you share data for water quality? If they have the data, they respond quickly; if they don't, they try and identify the organizations that would have that data and forge a relationship so they can be included on Figure.NZ's site. Overall, Figure.NZ is seeking to provide a place for people to be curious about, access, and interpret data on topics they are interested in.

Lillian has a deep and profound vision for Figure.NZ that goes well beyond simply providing open-data services. She says things are different now. "We used to live in a world where it was really hard to share information widely. And in that world, the best future was created by having a few great leaders who essentially had access to the information and made decisions on behalf of others, whether it was on behalf of a country or companies.

"But now we live in a world where it's really easy to share information widely and also to communicate widely. In the world we live in now, the best future is the one where everyone can make well-informed decisions.

"The use of numbers and data as a way of making well-informed decisions is one of the areas where there is the biggest gaps. We don't really use numbers as a part of our thinking and part of our understanding yet.

"Part of the reason is the way data is spread across hundreds of sites. In addition, for the most part, deep thinking based on data is constrained to experts because most people don't have data literacy. There once was a time

when many citizens in society couldn't read or write. However, as a society, we've now come to believe that reading and writing skills should be something all citizens have. We haven't yet adopted a similar belief around numbers and data literacy. We largely still believe that only a few specially trained people can analyze and think with numbers.

"Figure.NZ may be the first organization to assert that *everyone* can use numbers in their thinking, and it's built a technological platform along with trust and a network of relationships to make that possible. What you can see on Figure.NZ are tens of thousands of graphs, maps, and data.

"Figure.NZ sees this as a new kind of alphabet that can help people analyze what they see around them. A way to be thoughtful and informed about society. A means of engaging in conversation and shaping decision making that transcends personal experience. The long-term value and impact is almost impossible to measure, but the goal is to help citizens gain understanding and work together in more informed ways to shape the future."

Lillian sees Figure.NZ's model as having global potential. But for now, their focus is completely on making Figure.NZ work in New Zealand and to get the "network effect"—users dramatically increasing value for themselves and for others through use of their service. Creative Commons is core to making the network effect possible.

Web links

- 1 www.nzdatafutures.org.nz/sites/default/files/NZDFF_harness-the-power.pdf
- 2 www.ict.govt.nz/guidance-and-resources/open-government/new-zealand-government-open-access-and-licensing-nzgoal-framework/
- 3 figure.nz/business/
- 4 figure.nz/patrons/

KNOWLEDGE UNLATCHED



Knowledge Unlatched is a not-for-profit community interest company that brings libraries together to pool funds to publish open-access books. Founded in 2012 in the UK.

knowledgeunlatched.org

Revenue model: crowdfunding (specialized)

Interview date: February 26, 2016
Interviewee: Frances Pinter, founder

Profile written by Paul Stacey

The serial entrepreneur Dr. Frances Pinter has been at the forefront of innovation in the publishing industry for nearly forty years. She founded the UK-based Knowledge Unlatched with a mission to enable open access to scholarly books. For Frances, the current scholarly-book-publishing system is not working for anyone, and especially not for monographs in the humanities and social sciences. Knowledge Unlatched is committed to changing this and has been working with libraries to create a sustainable alternative model for publishing scholarly books, sharing the cost of making monographs (released under a Creative Commons license) and savings costs over the long term. Since its launch, Knowledge Unlatched has received several awards, including the IFLA/Brill Open Access award in 2014 and a Curtin University

Commercial Innovation Award for Innovation in Education in 2015.

Dr. Pinter has been in academic publishing most of her career. About ten years ago, she became acquainted with the Creative Commons founder Lawrence Lessig and got interested in Creative Commons as a tool for both protecting content online and distributing it free to users.

Not long after, she ran a project in Africa convincing publishers in Uganda and South Africa to put some of their content online for free using a Creative Commons license and to see what happened to print sales. Sales went up, not down.

In 2008, Bloomsbury Academic, a new imprint of Bloomsbury Publishing in the United Kingdom, appointed her its founding publisher in London. As part of the launch, Frances convinced Bloomsbury to differentiate themselves by putting out monographs for free online under a Creative Commons license (BY-NC or BY-NC-ND, i.e., Attribution-NonCommercial or Attribution-NonCommercial-NoDerivs). This was seen as risky, as the biggest cost for publishers is getting a book to the stage where it can be printed. If everyone read the online book for free, there would be no print-book sales at all, and the costs associated with getting the book to print would be lost. Surprisingly, Bloomsbury found that sales of the print versions of these books were 10 to 20 percent higher than normal. Frances found it intriguing that the Creative Commons-licensed free online book acts as a marketing vehicle for the print format.

Frances began to look at customer interest in the three forms of the book: 1) the Creative Commons-licensed free online book in PDF form, 2) the printed book, and 3) a digital version of the book on an aggregator platform with enhanced features. She thought of this as the “ice cream model”: the free PDF was vanilla ice cream, the printed book was an ice cream cone, and the enhanced e-book was an ice cream sundae.

After a while, Frances had an epiphany—what if there was a way to get libraries to underwrite the costs of making these books up until they’re ready to be printed, in other words, cover the fixed costs of getting to the first digital copy? Then you could either bring down the cost of the printed book, or do a whole bunch of interesting things with the printed book and e-book—the ice cream cone or sundae part of the model.

This idea is similar to the article-processing charge some open-access journals charge researchers to cover publishing costs. Frances began to imagine a coalition of libraries paying for the prepress costs—a “book-processing charge”—and providing everyone in the world

with an open-access version of the books released under a Creative Commons license.

This idea really took hold in her mind. She didn’t really have a name for it but began talking about it and making presentations to see if there was interest. The more she talked about it, the more people agreed it had appeal. She offered a bottle of champagne to anyone who could come up with a good name for the idea. Her husband came up with Knowledge Unlatched, and after two years of generating interest, she decided to move forward and launch a community interest company (a UK term for not-for-profit social enterprises) in 2012.

She describes the business model in a paper called *Knowledge Unlatched: Toward an Open and Networked Future for Academic Publishing*:

- 1 Publishers offer titles for sale reflecting origination costs only via Knowledge Unlatched.
- 2 Individual libraries select titles either as individual titles or as collections (as they do from library suppliers now).
- 3 Their selections are sent to Knowledge Unlatched specifying the titles to be purchased at the stated price(s).
- 4 The price, called a Title Fee (set by publishers and negotiated by Knowledge Unlatched), is paid to publishers to cover the fixed costs of publishing each of the titles that were selected by a minimum number of libraries to cover the Title Fee.
- 5 Publishers make the selected titles available Open Access (on a Creative Commons or similar open license) and are then paid the Title Fee which is the total collected from the libraries.

- 6 Publishers make print copies, e-Pub, and other digital versions of selected titles available to member libraries at a discount that reflects their contribution to the Title Fee and incentivizes membership.¹

The first round of this model resulted in a collection of twenty-eight current titles from thirteen recognized scholarly publishers being unlatched. The target was to have two hundred libraries participate. The cost of the package per library was capped at \$1,680, which was an average price of sixty dollars per book, but in the end they had nearly three hundred libraries sharing the costs, and the price per book came in at just under forty-three dollars.

The open-access, Creative Commons versions of these twenty-eight books are still available online.⁴ Most books have been licensed with CC BY-NC or CC BY-NC-ND. Authors are the copyright holder, not the publisher, and negotiate choice of license as part of the publishing agreement. Frances has found that most authors want to retain control over the commercial and remix use of their work. Publishers list the book in their catalogs, and the noncommercial restriction in the Creative Commons license ensures authors continue to get royalties on sales of physical copies.

There are three cost variables to consider for each round: the overall cost incurred by the publishers, total cost for each library to acquire all the books, and the individual price per book. The fee publishers charge for each title is a fixed charge, and Knowledge Unlatched calculates the total amount for all the books being unlatched at a time. The cost of an order for each library is capped at a maximum based on a minimum number of libraries participating. If the number of participating libraries exceeds the minimum, then the cost of the order and the price per book go down for each library.

The second round, recently completed, unlatched seventy-eight books from twenty-six publishers. For this round, Frances was experimenting with the size and shape of the of-

ferings. Books were being bundled into eight small packages separated by subject (including Anthropology, History, Literature, Media and Communications, and Politics), of around ten books per package. Three hundred libraries around the world have to commit to at least six of the eight packages to enable unlatching. The average cost per book was just under fifty dollars. The unlatching process took roughly ten months. It started with a call to publishers for titles, followed by having a library task force select the titles, getting authors' permissions, getting the libraries to pledge, billing the libraries, and finally, unlatching.

The longest part of the whole process is getting libraries to pledge and commit funds. It takes about five months, as library buy-in has to fit within acquisition cycles, budget cycles, and library-committee meetings.

Knowledge Unlatched informs and recruits libraries through social media, mailing lists, listservs, and library associations. Of the three hundred libraries that participated in the first round, 80 percent are also participating in the second round, and there are an additional eighty new libraries taking part. Knowledge Unlatched is also working not just with individual libraries but also library consortia, which has been getting even more libraries involved.

Knowledge Unlatched is scaling up, offering 150 new titles in the second half of 2016. It will also offer backlist titles, and in 2017 will start to make journals open access too.

Knowledge Unlatched deliberately chose monographs as the initial type of book to unlatch. Monographs are foundational and important, but also problematic to keep going in the standard closed publishing model.

The cost for the publisher to get to a first digital copy of a monograph is \$5,000 to \$50,000. A good one costs in the \$10,000 to \$15,000 range. Monographs typically don't sell a lot of

copies. A publisher who in the past sold three thousand copies now typically sells only three hundred. That makes unlatching monographs a low risk for publishers. For the first round, it took five months to get thirteen publishers. For the second round, it took one month to get twenty-six.

Authors don't generally make a lot of royalties from monographs. Royalties range from zero dollars to 5 to 10 percent of receipts. The value to the author is the awareness it brings to them; when their book is being read, it increases their reputation. Open access through unlatching generates many more downloads and therefore awareness. (On the Knowledge Unlatched website, you can find interviews with the twenty-eight round-one authors describing their experience and the benefits of taking part.)⁵

Library budgets are constantly being squeezed, partly due to the inflation of journal subscriptions. But even without budget constraints, academic libraries are moving away from buying physical copies. An academic library catalog entry is typically a URL to wherever the book is hosted. Or if they have enough electronic storage space, they may download the digital file into their digital repository. Only secondarily do they consider getting a print book, and if they do, they buy it separately from the digital version.

Knowledge Unlatched offers libraries a compelling economic argument. Many of the participating libraries would have bought a copy of the monograph anyway, but instead of paying \$95 for a print copy or \$150 for a digital multiple-use copy, they pay \$50 to unlatch. It costs them less, and it opens the book to not just the participating libraries, but to the world.

Not only do the economics make sense, but there is very strong alignment with library mandates. The participating libraries pay less than they would have in the closed model, and the open-access book is available to all libraries. While this means nonparticipating libraries could be seen as free riders, in the library world, wealthy libraries are used to paying more than poor libraries and accept that part

of their money should be spent to support open access. "Free ride" is more like community responsibility. By the end of March 2016, the round-one books had been downloaded nearly eighty thousand times in 175 countries.

For publishers, authors, and librarians, the Knowledge Unlatched model for monographs is a win-win-win.

In the first round, Knowledge Unlatched's overheads were covered by grants. In the second round, they aim to demonstrate the model is sustainable. Libraries and publishers will each pay a 7.5 percent service charge that will go toward Knowledge Unlatched's running costs. With plans to scale up in future rounds, Frances figures they can fully recover costs when they are unlatching two hundred books at a time. Moving forward, Knowledge Unlatched is making investments in technology and processes. Future plans include unlatching journals and older books.

Frances believes that Knowledge Unlatched is tapping into new ways of valuing academic content. It's about considering how many people can find, access, and use your content without pay barriers. Knowledge Unlatched taps into the new possibilities and behaviors of the digital world. In the Knowledge Unlatched model, the content-creation process is exactly the same as it always has been, but the economics are different. For Frances, Knowledge Unlatched is connected to the past but moving into the future, an evolution rather than a revolution.

Web links

- 1 www.pinter.org.uk/pdfs/Toward_an_Open.pdf
- 2 www.oopen.org
- 3 www.hathitrust.org
- 4 collections.knowledgeunlatched.org/collection-availability-1/
- 5 www.knowledgeunlatched.org/featured-authors-section/

LUMEN LEARNING



Lumen Learning is a for-profit company helping educational institutions use open educational resources (OER). Founded in 2013 in the U.S.

lumenlearning.com

Revenue model: charging for custom services, grant funding

Interview date: December 21, 2015

Interviewees: David Wiley and Kim Thanos, cofounders

Profile written by Paul Stacey

Cofounded by open education visionary Dr. David Wiley and education-technology strategist Kim Thanos, Lumen Learning is dedicated to improving student success, bringing new ideas to pedagogy, and making education more affordable by facilitating adoption of open educational resources. In 2012, David and Kim partnered on a grant-funded project called the Kaleidoscope Open Course Initiative.¹ It involved a set of fully open general-education courses across eight colleges predominantly serving at-risk students, with goals to dramatically reduce textbook costs and collaborate to improve the courses to help students succeed. David and Kim exceeded those goals: the cost of the required textbooks, replaced with OER, decreased to zero dollars, and average student-success rates improved by 5 to 10 percent when compared with previous years.

After a second round of funding, a total of more than twenty-five institutions participated in and benefited from this project. It was career changing for David and Kim to see the impact this initiative had on low-income students. David and Kim sought further funding from the Bill and Melinda Gates Foundation, who asked them to define a plan to scale their work in a financially sustainable way. That is when they decided to create Lumen Learning.

David and Kim went back and forth on whether it should be a nonprofit or for-profit. A nonprofit would make it a more comfortable fit with the education sector but meant they'd be constantly fund-raising and seeking grants from philanthropies. Also, grants usually require money to be used in certain ways for specific deliverables. If you learn things along the way that change how you think the grant

money should be used, there often isn't a lot of flexibility to do so.

But as a for-profit, they'd have to convince educational institutions to pay for what Lumen had to offer. On the positive side, they'd have more control over what to do with the revenue and investment money; they could make decisions to invest the funds or use them differently based on the situation and shifting opportunities. In the end, they chose the for-profit status, with its different model for and approach to sustainability.

Right from the start, David and Kim positioned Lumen Learning as a way to help institutions engage in open educational resources, or OER. OER are teaching, learning, and research materials, in all different media, that reside in the public domain or are released under an open license that permits free use and repurposing by others.

Originally, Lumen did custom contracts for each institution. This was complicated and challenging to manage. However, through that process patterns emerged which allowed them to generalize a set of approaches and offerings. Today they don't customize as much as they used to, and instead they tend to work with customers who can use their off-the-shelf options. Lumen finds that institutions and faculty are generally very good at seeing the value Lumen brings and are willing to pay for it. Serving disadvantaged learner populations has led Lumen to be very pragmatic; they describe what they offer in quantitative terms—with facts and figures—and in a way that is very student-focused. Lumen Learning helps colleges and universities—

- replace expensive textbooks in high-enrollment courses with OER;
- provide enrolled students day one access to Lumen's fully customizable OER course materials through the institution's learning-management system;

- measure improvements in student success with metrics like passing rates, persistence, and course completion; and
- collaborate with faculty to make ongoing improvements to OER based on student success research.

Lumen has developed a suite of open, Creative Commons-licensed courseware in more than sixty-five subjects. All courses are freely and publicly available right off their website. They can be copied and used by others as long as they provide attribution to Lumen Learning following the terms of the Creative Commons license.

Then there are three types of bundled services that cost money. One option, which Lumen calls Candela courseware, offers integration with the institution's learning-management system, technical and pedagogical support, and tracking of effectiveness. Candela courseware costs institutions ten dollars per enrolled student.

A second option is Waymaker, which offers the services of Candela but adds personalized learning technologies, such as study plans, automated messages, and assessments, and helps instructors find and support the students who need it most. Waymaker courses cost twenty-five dollars per enrolled student.

The third and emerging line of business for Lumen is providing guidance and support for institutions and state systems that are pursuing the development of complete OER degrees. Often called Z-Degrees, these programs eliminate textbook costs for students in all courses that make up the degree (both required and elective) by replacing commercial textbooks and other expensive resources with OER.

Lumen generates revenue by charging for their value-added tools and services on top of their free courses, just as solar-power companies provide the tools and services that help people use a free resource—sunlight. And Lumen's business model focuses on getting the institutions to pay, not the students. With projects they did prior to Lumen, David and Kim

learned that students who have access to all course materials from day one have greater success. If students had to pay, Lumen would have to restrict access to those who paid. Right from the start, their stance was that they would not put their content behind a paywall. Lumen invests zero dollars in technologies and processes for restricting access—no digital rights management, no time bombs. While this has been a challenge from a business-model perspective, from an open-access perspective, it has generated immense goodwill in the community.

In most cases, development of their courses is funded by the institution Lumen has a contract with. When creating new courses, Lumen typically works with the faculty who are teaching the new course. They're often part of the institution paying Lumen, but sometimes Lumen has to expand the team and contract faculty from other institutions. First, the faculty identifies all of the course's learning outcomes. Lumen then searches for, aggregates, and curates the best OER they can find that addresses those learning needs, which the faculty reviews.

Sometimes faculty like the existing OER but not the way it is presented. The open licensing of existing OER allows Lumen to pick and choose from images, videos, and other media to adapt and customize the course. Lumen creates new content as they discover gaps in existing OER. Test-bank items and feedback for students on their progress are areas where new content is frequently needed. Once a course is created, Lumen puts it on their platform with all the attributions and links to the original sources intact, and any of Lumen's new content is given an Attribution (CC BY) license.

Using only OER made them experience firsthand how complex it could be to mix differently licensed work together. A common strategy with OER is to place the Creative Commons license and attribution information in the

website's footer, which stays the same for all pages. This doesn't quite work, however, when mixing different OER together.

Remixing OER often results in multiple attributions on every page of every course—text from one place, images from another, and videos from yet another. Some are licensed as Attribution (CC BY), others as Attribution-ShareAlike (CC BY-SA). If this information is put within the text of the course, faculty members sometimes try to edit it and students find it a distraction. Lumen dealt with this challenge by capturing the license and attribution information as metadata, and getting it to show up at the end of each page.

Lumen's commitment to open licensing and helping low-income students has led to strong relationships with institutions, open-education enthusiasts, and grant funders. People in their network generously increase the visibility of Lumen through presentations, word of mouth, and referrals. Sometimes the number of general inquiries exceed Lumen's sales capacity.

To manage demand and ensure the success of projects, their strategy is to be proactive and focus on what's going on in higher education in different regions of the United States, watching out for things happening at the system level in a way that fits with what Lumen offers. A great example is the Virginia community college system, which is building out Z-Degrees. David and Kim say there are nine other U.S. states with similar system-level activity where Lumen is strategically focusing its efforts. Where there are projects that would require a lot of resources on Lumen's part, they prioritize the ones that would impact the largest number of students.

As a business, Lumen is committed to openness. There are two core nonnegotiables: Lumen's use of CC BY, the most permissive of the

Creative Commons licenses, for all the materials it creates; and day-one access for students. Having clear nonnegotiables allows them to then engage with the education community to solve for other challenges and work with institutions to identify new business models that achieve institution goals, while keeping Lumen healthy.

Openness also means that Lumen's OER must necessarily be nonexclusive and nonrivalrous. This represents several big challenges for the business model: Why should you invest in creating something that people will be reluctant to pay for? How do you ensure that the investment the diverse education community makes in OER is not exploited? Lumen thinks we all need to be clear about how we are benefiting from and contributing to the open community.

In the OER sector, there are examples of corporations, and even institutions, acting as free riders. Some simply take and use open resources without paying anything or contributing anything back. Others give back the minimum amount so they can save face. Sustainability will require those using open resources to give back an amount that seems fair or even give back something that is generous.

Lumen does track institutions accessing and using their free content. They proactively contact those institutions, with an estimate of how much their students are saving and encouraging them to switch to a paid model. Lumen explains the advantages of the paid model: a more interactive relationship with Lumen; integration with the institution's learning-management system; a guarantee of support for faculty and students; and future sustainability with funding supporting the evolution and improvement of the OER they are using.

Lumen works hard to be a good corporate citizen in the OER community. For David and Kim, a good corporate citizen gives more than they take, adds unique value, and is very transparent about what they are taking from community, what they are giving back, and what they are monetizing. Lumen believes these are the building blocks of a sustainable model

and strives for a correct balance of all these factors.

Licensing all the content they produce with CC BY is a key part of giving more value than they take. They've also worked hard at finding the right structure for their value-add and how to package it in a way that is understandable and repeatable.

As of the fall 2016 term, Lumen had eighty-six different open courses, working relationships with ninety-two institutions, and more than seventy-five thousand student enrollments. Lumen received early start-up funding from the Bill and Melinda Gates Foundation, the Hewlett Foundation, and the Shuttleworth Foundation. Since then, Lumen has also attracted investment funding. Over the last three years, Lumen has been roughly 60 percent grant funded, 20 percent revenue earned, and 20 percent funded with angel capital. Going forward, their strategy is to replace grant funding with revenue.

In creating Lumen Learning, David and Kim say they've landed on solutions they never imagined, and there is still a lot of learning taking place. For them, open business models are an emerging field where we are all learning through sharing. Their biggest recommendations for others wanting to pursue the open model are to make your commitment to open resources public, let people know where you stand, and don't back away from it. It really is about trust.

Web link

1 lumenlearning.com/innovative-projects/

JONATHAN MANN



Jonathan Mann is a singer and songwriter who is most well known as the "Song A Day" guy. Based in the U.S.

jonathanmann.net and
jonathanmann.bandcamp.com

Revenue model: charging for custom services, pay-what-you-want, crowdfunding (subscription-based), charging for in-person version (speaking engagements and musical performances)

Interview date: February 22, 2016

Profile written by Sarah Hinchliff Pearson

Jonathan Mann thinks of his business model as "hustling"—seizing nearly every opportunity he sees to make money. The bulk of his income comes from writing songs under commission for people and companies, but he has a wide variety of income sources. He has supporters on the crowdfunding site Patreon. He gets advertising revenue from YouTube and Bandcamp, where he posts all of his music. He gives paid speaking engagements about creativity and motivation. He has been hired by major conferences to write songs summarizing what speakers have said in the conference sessions.

His entrepreneurial spirit is coupled with a willingness to take action quickly. A perfect illustration of his ability to act fast happened in

2010, when he read that Apple was having a conference the following day to address a snafu related to the iPhone 4. He decided to write and post a song about the iPhone 4 that day, and the next day he got a call from the public relations people at Apple wanting to use and promote his video at the Apple conference. The song then went viral, and the experience landed him in *Time* magazine.

Jonathan's successful "hustling" is also about old-fashioned persistence. He is currently in his eighth straight year of writing one song each day. He holds the Guinness World Record for consecutive daily songwriting, and he is widely known as the "song-a-day guy."

He fell into this role by, naturally, seizing a random opportunity a friend alerted him to seven years ago—an event called Fun-A-Day, where people are supposed to create a piece of art every day for thirty-one days straight. He was in need of a new project, so he decided to give it a try by writing and posting a song each day. He added a video component to the songs because he knew people were more likely to watch video online than simply listening to audio files.

He had a really good time doing the thirty-one-day challenge, so he decided to see if he could continue it for one year. He never stopped. He has written and posted a new song literally every day, seven days a week, since he began the project in 2009. When he isn't writing songs that he is hired to write by clients, he writes songs about whatever is on his mind that day. His songs are catchy and mostly lighthearted, but they often contain at least an undercurrent of a deeper theme or meaning. Occasionally, they are extremely personal, like the song he cowrote with his exgirlfriend announcing their breakup. Rain or shine, in sickness or health, Jonathan posts and writes a song every day. If he is on a flight or otherwise incapable of getting Internet access in time to meet the deadline, he will prepare ahead and have someone else post the song for him.

Over time, the song-a-day gig became the basis of his livelihood. In the beginning, he made money one of two ways. The first was by entering a wide variety of contests and winning a handful. The second was by having the occasional song and video go some varying degree of viral, which would bring more eyeballs and mean that there were more people wanting him to write songs for them. Today he earns most of his money this way.

His website explains his gig as "taking any message, from the super simple to the totally complicated, and conveying that message through a heartfelt, fun and quirky song." He charges \$500 to create a produced song and \$300 for an acoustic song. He has been hired for product launches, weddings, conferences,

and even Kickstarter campaigns like the one that funded the production of this book.

Jonathan can't recall when exactly he first learned about Creative Commons, but he began applying CC licenses to his songs and videos as soon as he discovered the option. "CC seems like such a no-brainer," Jonathan said. "I don't understand how anything else would make sense. It seems like such an obvious thing that you would want your work to be able to be shared."

His songs are essentially marketing for his services, so obviously the further his songs spread, the better. Using CC licenses helps grease the wheels, letting people know that Jonathan allows and encourages them to copy, interact with, and remix his music. "If you let someone cover your song or remix it or use parts of it, that's how music is supposed to work," Jonathan said. "That is how music has worked since the beginning of time. Our me-me, mine-mine culture has undermined that."

There are some people who cover his songs fairly regularly, and he would never shut that down. But he acknowledges there is a lot more he could do to build community. "There is all of this conventional wisdom about how to build an audience online, and I generally think I don't do any of that," Jonathan said.

He does have a fan community he cultivates on Bandcamp, but it isn't his major focus. "I do have a core audience that has stuck around for a really long time, some even longer than I've been doing song-a-day," he said. "There is also a transitional aspect that drop in and get what they need and then move on." Focusing less on community building than other artists makes sense given Jonathan's primary income source of writing custom songs for clients.

Jonathan recognizes what comes naturally to him and leverages those skills. Through the practice of daily songwriting, he realized he has a gift for distilling complicated subjects into simple concepts and putting them to music. In his song "How to Choose a Master Password,"

IT SEEMS LIKE SUCH AN OBVIOUS THING THAT YOU WOULD WANT YOUR WORK TO BE ABLE TO BE SHARED.

Jonathan explained the process of creating a secure password in a silly, simple song. He was hired to write the song by a client who handed him a long technical blog post from which to draw the information. Like a good (and rare) journalist, he translated the technical concepts into something understandable.

When he is hired by a client to write a song, he first asks them to send a list of talking points and other information they want to include in the song. He puts all of that into a text file and starts moving things around, cutting and pasting until the message starts to come together. The first thing he tries to do is grok the core message and develop the chorus. Then he looks for connections or parts he can make rhyme. The entire process really does resemble good journalism, but of course the final product of his work is a song rather than news. "There is something about being challenged and forced to take information that doesn't seem like it should be sung about or doesn't seem like it lends itself to a song," he said. "I find that creative challenge really satisfying. I enjoy getting lost in that process."

Jonathan admits that in an ideal world, he would exclusively write the music he wanted to write, rather than what clients hire him to write. But his business model is about capitalizing on his strengths as a songwriter, and he has found a way to keep it interesting for himself.

Jonathan uses nearly every tool possible to make money from his art, but he does have lines he won't cross. He won't write songs about things he fundamentally does not believe in, and there are times he has turned down jobs on principle. He also won't stray

too much from his natural style. "My style is silly, so I can't really accommodate people who want something super serious," Jonathan said. "I do what I do very easily, and it's part of who I am." Jonathan hasn't gotten into writing commercials for the same reasons; he is best at using his own unique style rather than mimicking others.

Jonathan's song-a-day commitment exemplifies the power of habit and grit. Conventional wisdom about creative productivity, including advice in books like the best-seller *The Creative Habit* by Twyla Tharp, routinely emphasizes the importance of ritual and action. No amount of planning can replace the value of simple practice and just *doing*. Jonathan Mann's work is a living embodiment of these principles.

When he speaks about his work, he talks about how much the song-a-day process has changed him. Rather than seeing any given piece of work as precious and getting stuck on trying to make it perfect, he has become comfortable with just doing. If today's song is a bust, tomorrow's song might be better.

Jonathan seems to have this mentality about his career more generally. He is constantly experimenting with ways to make a living while sharing his work as widely as possible, seeing what sticks. While he has major accomplishments he is proud of, like being in the *Guinness World Records* or having his song used by Steve Jobs, he says he never truly feels successful.

"Success feels like it's over," he said. "To a certain extent, a creative person is not ever going to feel completely satisfied because then so much of what drives you would be gone."

NOUN PROJECT



The Noun Project is a for-profit company offering an online platform to display visual icons from a global network of designers. Founded in 2010 in the U.S.

thenounproject.com

Revenue model: charging a transaction fee, charging for custom services

Interview date: October 6, 2015
Interviewee: Edward Boatman, cofounder

Profile written by Paul Stacey

The Noun Project creates and shares visual language. There are millions who use Noun Project symbols to simplify communication across borders, languages, and cultures.

The original idea for the Noun Project came to cofounder Edward Boatman while he was a student in architecture design school. He'd always done a lot of sketches and started to draw what used to fascinate him as a child, like trains, sequoias, and bulldozers. He began thinking how great it would be if he had a simple image or small icon of every single object or concept on the planet.

When Edward went on to work at an architecture firm, he had to make a lot of presentation boards for clients. But finding high-quality sources for symbols and icons was difficult. He

couldn't find any website that could provide them. Perhaps his idea for creating a library of icons could actually help people in similar situations.

With his partner, Sofya Polyakov, he began collecting symbols for a website and writing a business plan. Inspiration came from the book *Professor and the Madman*, which chronicles the use of crowdsourcing to create the *Oxford English Dictionary* in 1870. Edward began to imagine crowdsourcing icons and symbols from volunteer designers around the world.

Then Edward got laid off during the recession, which turned out to be a huge catalyst. He decided to give his idea a go, and in 2010 Edward and Sofya launched the Noun Project with a Kickstarter campaign, back when

Kickstarter was in its infancy.¹ They thought it'd be a good way to introduce the global web community to their idea. Their goal was to raise \$1,500, but in twenty days they got over \$14,000. They realized their idea had the potential to be something much bigger.

They created a platform where symbols and icons could be uploaded, and Edward began recruiting talented designers to contribute their designs, a process he describes as a relatively easy sell. Lots of designers have old drawings just gathering "digital dust" on their hard drives. It's easy to convince them to finally share them with the world.

The Noun Project currently has about seven thousand designers from around the world. But not all submissions are accepted. The Noun Project's quality-review process means that only the best works become part of its collection. They make sure to provide encouraging, constructive feedback whenever they reject a piece of work, which maintains and builds the relationship they have with their global community of designers.

Creative Commons is an integral part of the Noun Project's business model; this decision was inspired by Chris Anderson's book *Free: The Future of Radical Price*, which introduced Edward to the idea that you could build a business model around free content.

Edward knew he wanted to offer a *free* visual language while still providing some protection and reward for its contributors. There is a tension between those two goals, but for Edward, Creative Commons licenses bring this idealism and business opportunity together elegantly. He chose the Attribution (CC BY) license, which means people can download the icons for free and modify them and even use them commercially. The requirement to give attribution to the original creator ensures that the creator can build a reputation and get global recognition for their work. And if they simply want to offer an icon that people can use without hav-

ing to give credit, they can use CC0 to put the work into the public domain.

Noun Project's business model and means of generating revenue have evolved significantly over time. Their initial plan was to sell T-shirts with the icons on it, which in retrospect Edward says was a horrible idea. They did get a lot of email from people saying they loved the icons but asking if they could pay a fee instead of giving attribution. Ad agencies (among others) wanted to keep marketing and presentation materials clean and free of attribution statements. For Edward, "That's when our lightbulb went off."

They asked their global network of designers whether they'd be open to receiving modest remuneration instead of attribution. Designers saw it as a win-win. The idea that you could offer your designs for free and have a global audience *and* maybe even make some money was pretty exciting for most designers.

The Noun Project first adopted a model whereby using an icon without giving attribution would cost \$1.99 per icon. The model's second iteration added a subscription component, where there would be a monthly fee to access a certain number of icons—ten, fifty, a hundred, or five hundred. However, users didn't like these hard-count options. They preferred to try out many similar icons to see which worked best before eventually choosing the one they wanted to use. So the Noun Project moved to an unlimited model, whereby users have unlimited access to the whole library for a flat monthly fee. This service is called NounPro and costs \$9.99 per month. Edward says this model is working well—good for customers, good for creators, and good for the platform.

Customers then began asking for an application-programming interface (API), which would allow Noun Project icons and symbols to be directly accessed from within other applications. Edward knew that the icons and symbols would be valuable in a lot of different

contexts and that they couldn't possibly know all of them in advance, so they built an API with a lot of flexibility. Knowing that most API applications would want to use the icons without giving attribution, the API was built with the aim of charging for its use. You can use what's called the "Playground API" for free to test how it integrates with your application, but full implementation will require you to purchase the API Pro version.

The Noun Project shares revenue with its international designers. For one-off purchases, the revenue is split 70 percent to the designer and 30 percent to Noun Project.

THE NOUN PROJECT'S SUCCESS LIES IN CREATING SERVICES AND CONTENT THAT ARE A STRATEGIC MIX OF FREE AND PAID WHILE STAYING TRUE TO THEIR MISSION—CREATING, SHARING, AND CELEBRATING THE WORLD'S VISUAL LANGUAGE.

The revenue from premium purchases (the subscription and API options) is split a little differently. At the end of each month, the total revenue from subscriptions is divided by Noun Project's total number of downloads, resulting in a rate per download—for example, it could be \$0.13 per download for that month. For each download, the revenue is split 40 percent to the designer and 60 percent to the Noun Project. (For API usage, it's per use instead of per download.) Noun Project's share is higher

this time as it's providing more service to the user.

The Noun Project tries to be completely transparent about their royalty structure.² They tend to over communicate with creators about it because building trust is the top priority.

For most creators, contributing to the Noun Project is not a full-time job but something they do on the side. Edward categorizes monthly earnings for creators into three broad categories: enough money to buy beer; enough to pay the bills; and most successful of all, enough to pay the rent.

Recently the Noun Project launched a new app called Lingo. Designers can use Lingo to organize not just their Noun Project icons and symbols but also their photos, illustrations, UX designs, et cetera. You simply drag any visual item directly into Lingo to save it. Lingo also works for teams so people can share visuals with each other and search across their combined collections. Lingo is free for personal use. A pro version for \$9.99 per month lets you add guests. A team version for \$49.95 per month allows up to twenty-five team members to collaborate, and to view, use, edit, and add new assets to each other's collections. And if you subscribe to NounPro, you can access Noun Project from within Lingo.

The Noun Project gives a ton of value away for free. A very large percentage of their roughly one million members have a free account, but there are still lots of paid accounts coming from digital designers, advertising and design agencies, educators, and others who need to communicate ideas visually.

For Edward, "creating, sharing, and celebrating the world's visual language" is the most important aspect of what they do; it's their stated mission. It differentiates them from others who offer graphics, icons, or clip art.

Noun Project creators agree. When surveyed on why they participate in the Noun Project, this is how designers rank their reasons: 1) to support the Noun Project mission, 2) to promote their own personal brand, and 3) to generate money. It's striking to see that money comes third, and mission, first. If you want to engage a global network of contributors, it's important to have a mission beyond making money.

In Edward's view, Creative Commons is central to their mission of sharing and social good. Using Creative Commons makes the Noun Project's mission genuine and has generated a lot of their initial traction and credibility. CC comes with a built-in community of users and fans.

Edward told us, "Don't underestimate the power of a passionate community around your product or your business. They are going to go to bat for you when you're getting ripped in the media. If you go down the road of choosing to work with Creative Commons, you're taking the first step to building a great community and tapping into a really awesome community that comes with it. But you need to continue to foster that community through other initiatives and continue to nurture it."

The Noun Project nurtures their creators' second motivation—promoting a personal brand—by connecting every icon and symbol to the creator's name and profile page; each profile features their full collection. Users can also search the icons by the creator's name.

The Noun Project also builds community through Iconathons—hackathons for icons.² In partnership with a sponsoring organization, the Noun Project comes up with a theme (e.g., sustainable energy, food bank, guerrilla gardening, human rights) and a list of icons that are needed, which designers are invited to create at the event. The results are vectorized, and added to the Noun Project using CC0 so they can be used by anyone for free.

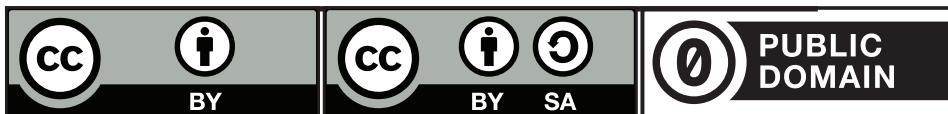
Providing a free version of their product that satisfies a lot of their customers' needs has actually enabled the Noun Project to build the paid version, using a service-oriented

model. The Noun Project's success lies in creating services and content that are a strategic mix of free and paid while staying true to their mission—creating, sharing, and celebrating the world's visual language. Integrating Creative Commons into their model has been key to that goal.

Web links

- 1 www.kickstarter.com/projects/tnp/building-a-free-collection-of-our-worlds-visual-sy/description
- 2 thenounproject.com/handbook/royalties/#getting_paid
- 3 thenounproject.com/iconathon/

OPEN DATA INSTITUTE



The Open Data Institute is an independent nonprofit that connects, equips, and inspires people around the world to innovate with data. Founded in 2012 in the UK.

theodi.org

Revenue model: grant and government funding, charging for custom services, donations

Interview date: November 11, 2015

Interviewee: Jeni Tennison, technical director

Profile written by Paul Stacey

Cofounded by Sir Tim Berners-Lee and Sir Nigel Shadbolt in 2012, the London-based Open Data Institute (ODI) offers data-related training, events, consulting services, and research. For ODI, Creative Commons licenses are central to making their own business model and their customers' open. CC BY (Attribution), CC BY-SA (Attribution-ShareAlike), and CC0 (placed in the public domain) all play a critical role in ODI's mission to help people around the world innovate with data.

Data underpins planning and decision making across all aspects of society. Weather data helps farmers know when to plant their crops, flight time data from airplane companies helps us plan our travel, data on local housing informs city planning. When this data

is not only accurate and timely, but open and accessible, it opens up new possibilities. Open data can be a resource businesses use to build new products and services. It can help governments measure progress, improve efficiency, and target investments. It can help citizens improve their lives by better understanding what is happening around them.

The Open Data Institute's 2012-17 business plan starts out by describing its vision to establish itself as a world-leading center and to research and be innovative with the opportunities created by the UK government's open data policy. (The government was an early pioneer in open policy and open-data initiatives.) It goes on to say that the ODI wants to—

- demonstrate the commercial value of open government data and how open-data policies affect this;
- develop the economic benefits case and business models for open data;
- help UK businesses use open data; and
- show how open data can improve public services.¹

ODI is very explicit about how it wants to make *open* business models, and defining what this means. Jeni Tennison, ODI's technical director, puts it this way: "There is a whole ecosystem of *open*—open-source software, open government, open-access research—and a whole ecosystem of *data*. ODI's work cuts across both, with an emphasis on where they overlap—with *open data*." ODI's particular focus is to show open data's potential for revenue.

As an independent nonprofit, ODI secured £10 million over five years from the UK government via Innovate UK, an agency that promotes innovation in science and technology. For this funding, ODI has to secure matching funds from other sources, some of which were met through a \$4.75-million investment from the Omidyar Network.

Jeni started out as a developer and technical architect for data.gov.uk, the UK government's pioneering open-data initiative. She helped make data sets from government departments available as open data. She joined ODI in 2012 when it was just starting up, as one of six people. It now has a staff of about sixty.

ODI strives to have half its annual budget come from the core UK government and Omidyar grants, and the other half from project-based research and commercial work. In Jeni's view, having this balance of revenue sources establishes some stability, but also keeps them motivated to go out and generate

these matching funds in response to market needs.

On the commercial side, ODI generates funding through memberships, training, and advisory services.

You can join the ODI as an individual or commercial member. Individual membership is pay-what-you-can, with options ranging from £1 to £100. Members receive a newsletter and related communications and a discount on ODI training courses and the annual summit, and they can display an ODI-supporter badge on their website. Commercial membership is divided into two tiers: small to medium size enterprises and nonprofits at £720 a year, and corporations and government organizations at £2,200 a year. Commercial members have greater opportunities to connect and collaborate, explore the benefits of open data, and unlock new business opportunities. (All members are listed on their website.)²

ODI provides standardized open data training courses in which anyone can enroll. The initial idea was to offer an intensive and academically oriented diploma in open data, but it quickly became clear there was no market for that. Instead, they offered a five-day-long public training course, which has subsequently been reduced to three days; now the most popular course is one day long. The fee, in addition to the time commitment, can be a barrier for participation. Jeni says, "Most of the people who would be able to pay don't know they need it. Most who know they need it can't pay." Public-sector organizations sometimes give vouchers to their employees so they can attend as a form of professional development.

ODI customizes training for clients as well, for which there is more demand. Custom training usually emerges through an established relationship with an organization. The training program is based on a definition of open-data knowledge as applicable to the organization and on the skills needed by their high-level executives, management, and technical staff.

The training tends to generate high interest and commitment.

Education about open data is also a part of ODI's annual summit event, where curated presentations and speakers showcase the work of ODI and its members across the entire ecosystem. Tickets to the summit are available to the public, and hundreds of people and organizations attend and participate. In 2014, there were four thematic tracks and over 750 attendees.

In addition to memberships and training, ODI provides advisory services to help with technical-data support, technology development, change management, policies, and other areas. ODI has advised large commercial organizations, small businesses, and international governments; the focus at the moment is on government, but ODI is working to shift more toward commercial organizations.

On the commercial side, the following value propositions seem to resonate:

- Data-driven insights. Businesses need data from outside their business to get more insight. Businesses can generate value and more effectively pursue their own goals if they open up their own data too. Big data is a hot topic.
- Open innovation. Many large-scale enterprises are aware they don't innovate very well. One way they can innovate is to open up their data. ODI encourages them to do so even if it exposes problems and challenges. The key is to invite other people to help while still maintaining organizational autonomy.
- Corporate social responsibility. While this resonates with businesses, ODI cautions against having it be the sole reason for making data open. If a business is just thinking about open data as a way to be transparent and accountable, they can miss out on efficiencies and opportunities.

IT IS PERFECTLY POSSIBLE TO GENERATE SUSTAINABLE REVENUE STREAMS THAT DO NOT RELY ON RESTRICTIVE LICENSING OF CONTENT, DATA, OR CODE.

During their early years, ODI wanted to focus solely on the United Kingdom. But in their first year, large delegations of government visitors from over fifty countries wanted to learn more about the UK government's open-data practices and how ODI saw that translating into economic value. They were contracted as a service provider to international governments, which prompted a need to set up international ODI "nodes."

Nodes are franchises of the ODI at a regional or city level. Hosted by existing (for-profit or not-for-profit) organizations, they operate locally but are part of the global network. Each ODI node adopts the charter, a set of guiding principles and rules under which ODI operates. They develop and deliver training, connect people and businesses through membership and events, and communicate open-data stories from their part of the world. There are twenty-seven different nodes across nineteen countries. ODI nodes are charged a small fee to be part of the network and to use the brand.

ODI also runs programs to help start-ups in the UK and across Europe develop a sustainable business around open data, offering mentoring, advice, training, and even office space.³

A big part of ODI's business model revolves around community building. Memberships, training, summits, consulting services, nodes, and start-up programs create an ever-growing network of open-data users and leaders. (In fact, ODI even operates something called an Open Data Leaders Network.) For ODI, community is key to success. They devote significant time and effort to build it, not just online but through face-to-face events.

ODI has created an online tool that organizations can use to assess the legal, practical, technical, and social aspects of their open data. If it is of high quality, the organization can earn ODI's Open Data Certificate, a globally recognized mark that signals that their open data is useful, reliable, accessible, discoverable, and supported.⁴

Separate from commercial activities, the ODI generates funding through research grants. Research includes looking at evidence on the impact of open data, development of open-data tools and standards, and how to deploy open data at scale.

Creative Commons 4.0 licenses cover database rights and ODI recommends CC BY, CC BY-SA, and CC0 for data releases. ODI encourages publishers of data to use Creative Commons licenses rather than creating new "open licenses" of their own.

For ODI, *open* is at the heart of what they do. They also release any software code they produce under open-source-software licenses, and publications and reports under CC BY or CC BY-SA licenses. ODI's mission is to connect and equip people around the world so they can innovate with data. Disseminating stories, research, guidance, and code under an open license is essential for achieving that mission. It also demonstrates that it is perfectly possible to generate sustainable revenue streams that do not rely on restrictive licensing of content, data, or code. People pay to have ODI experts provide training to them, not for the content of the training; people pay for the advice ODI gives them, not for the methodologies they use. Producing open content, data, and source code helps establish credibility and creates leads for the paid services that they offer. According to Jeni, "The biggest lesson we have learned is that it is completely possible to be open, get customers, and make money."

To serve as evidence of a successful open business model and return on investment, ODI has a public dashboard of key performance in-

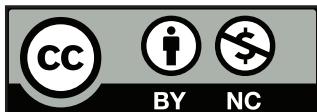
dicators. Here are a few metrics as of April 27, 2016:

- Total amount of cash investments unlocked in direct investments in ODI, competition funding, direct contracts, and partnerships, and income that ODI nodes and ODI start-ups have generated since joining the ODI program: £44.5 million
- Total number of active members and nodes across the globe: 1,350
- Total sales since ODI began: £7.44 million
- Total number of unique people reached since ODI began, in person and online: 2.2 million
- Total Open Data Certificates created: 151,000
- Total number of people trained by ODI and its nodes since ODI began: 5,0805

Web links

- 1 e642e8368e3bf8d5526e-464b4b70b4554c1a79566214d402739e.r6.cf3.rackcdn.com/odi-business-plan-may-release.pdf
- 2 directory.theodi.org/members
- 3 [theodi.org/odi-startup-programme; theodi.org/open-data-incubator-for-europe](https://theodi.org/odi-startup-programme; https://theodi.org/open-data-incubator-for-europe)
- 4 certificates.theodi.org
- 5 dashboards.theodi.org/company/all

OPENDESK



Opendesk is a for-profit company offering an online platform that connects furniture designers around the world with customers and local makers who bring the designs to life.
Founded in 2014 in the UK.

www.opendesk.cc

Revenue model: charging a transaction fee

Interview date: November 4, 2015

Interviewees: Nick Ierodiaconou and Joni Steiner, cofounders

Profile written by Paul Stacey

Opendesk is an online platform that connects furniture designers around the world not just with customers but also with local registered makers who bring the designs to life. Opendesk and the designer receive a portion of every sale that is made by a maker.

Cofounders Nick Ierodiaconou and Joni Steiner studied and worked as architects together. They also made goods. Their first client was Mint Digital, who had an interest in open licensing. Nick and Joni were exploring digital fabrication, and Mint's interest in open licensing got them to thinking how the open-source world may interact and apply to physical goods. They sought to design something for their client that was also reproducible. As they put it, they decided to "ship the recipe, but not the goods." They created the design using software, put it under an open license, and had it manufactured locally near the client. This was

the start of the idea for Opendesk. The idea for Wikihouse—another open project dedicated to accessible housing for all—started as discussions around the same table. The two projects ultimately went on separate paths, with Wikihouse becoming a nonprofit foundation and Opendesk a for-profit company.

When Nick and Joni set out to create Opendesk, there were a lot of questions about the viability of distributed manufacturing. No one was doing it in a way that was even close to realistic or competitive. The design community had the intent, but fulfilling this vision was still a long way away.

And now this sector is emerging, and Nick and Joni are highly interested in the commercialization aspects of it. As part of coming up

with a business model, they began investigating intellectual property and licensing options. It was a thorny space, especially for designs. Just what aspect of a design is copyrightable? What is patentable? How can allowing for digital sharing and distribution be balanced against the designer's desire to still hold ownership? In the end, they decided there was no need to reinvent the wheel and settled on using Creative Commons.

When designing the Opendesk system, they had two goals. They wanted anyone, anywhere in the world, to be able to download designs so that they could be made locally, and they wanted a viable model that benefited designers when their designs were sold. Coming up with a business model was going to be complex.

They gave a lot of thought to three angles—the potential for social sharing, allowing designers to choose their license, and the impact these choices would have on the business model.

In support of social sharing, Opendesk actively advocates for (but doesn't demand) open licensing. And Nick and Joni are agnostic about which Creative Commons license is used; it's up to the designer. They can be proprietary or choose from the full suite of Creative Commons licenses, deciding for themselves how open or closed they want to be.

For the most part, designers love the idea of sharing content. They understand that you get positive feedback when you're attributed, what Nick and Joni called "reputational glow." And Opendesk does an awesome job profiling the designers.¹

While designers are largely OK with personal sharing, there is a concern that someone will take the design and manufacture the furniture in bulk, with the designer not getting any benefits. So most Opendesk designers choose the Attribution-NonCommercial license (CC BY-NC).

Anyone can download a design and make it themselves, provided it's for noncommercial use—and there have been many, many downloads. Or users can buy the product

from Opendesk, or from a registered maker in Opendesk's network, for on-demand personal fabrication. The network of Opendesk makers currently is made up of those who do digital fabrication using a computer-controlled CNC (Computer Numeric Control) machining device that cuts shapes out of wooden sheets according to the specifications in the design file.

Makers benefit from being part of Opendesk's network. Making furniture for local customers is paid work, and Opendesk generates business for them. Joni said, "Finding a whole network and community of makers was pretty easy because we built a site where people could write in about their capabilities. Building the community by learning from the maker community is how we have moved forward." Opendesk now has relationships with hundreds of makers in countries all around the world.²

The makers are a critical part of the Opendesk business model. Their model builds off the makers' quotes. Here's how it's expressed on Opendesk's website:

When customers buy an Opendesk product directly from a registered maker, they pay:

- the manufacturing cost as set by the maker (this covers material and labour costs for the product to be manufactured and any extra assembly costs charged by the maker)
- a design fee for the designer (a design fee that is paid to the designer every time their design is used)
- a percentage fee to the Opendesk platform (this supports the infrastructure and ongoing development of the platform that helps us build out our marketplace)

- a percentage fee to the channel through which the sale is made (at the moment this is Opendesk, but in the future we aim to open this up to third-party sellers who can sell Opendesk products through their own channels—this covers sales and marketing fees for the relevant channel)
- a local delivery service charge (the delivery is typically charged by the maker, but in some cases may be paid to a third-party delivery partner)
- charges for any additional services the customer chooses, such as on-site assembly (additional services are discretionary—in many cases makers will be happy to quote for assembly on-site and designers may offer bespoke design options)
- local sales taxes (variable by customer and maker location)³

They then go into detail how makers' quotes are created:

When a customer wants to buy an Opendesk... they are provided with a transparent breakdown of fees including the manufacturing cost, design fee, Opendesk platform fee and channel fees. If a customer opts to buy by getting in touch directly with a registered local maker using a downloaded Opendesk file, the maker is responsible for ensuring the design fee, Opendesk platform fee and channel fees are included in any quote at the time of sale. Percentage fees are always based on the underlying manufacturing cost and are typically apportioned as follows:

- manufacturing cost: fabrication, finishing and any other costs as set by the maker (excluding any services like delivery or on-site assembly)
- design fee: 8 percent of the manufacturing cost

- platform fee: 12 percent of the manufacturing cost
- channel fee: 18 percent of the manufacturing cost
- sales tax: as applicable (depends on product and location)

Opendesk shares revenue with their community of designers. According to Nick and Joni, a typical designer fee is around 2.5 percent, so Opendesk's 8 percent is more generous, and providing a higher value to the designer.

The Opendesk website features stories of designers and makers. Denis Fuzii published the design for the Valovi Chair from his studio in São Paulo. His designs have been downloaded over five thousand times in ninety-five countries. I.J. CNC Services is Ian Jinks, a professional maker based in the United Kingdom. Opendesk now makes up a large proportion of his business.

To manage resources and remain effective, Opendesk has so far focused on a very narrow niche—primarily office furniture of a certain simple aesthetic, which uses only one type of material and one manufacturing technique. This allows them to be more strategic and more disruptive in the market, by getting things to market quickly with competitive prices. It also reflects their vision of creating reproducible and functional pieces.

On their website, Opendesk describes what they do as "open making": "Designers get a global distribution channel. Makers get profitable jobs and new customers. You get designer products without the designer price tag, a more social, eco-friendly alternative to mass-production and an affordable way to buy custom-made products."

Nick and Joni say that customers like the fact that the furniture has a known provenance. People really like that their furniture was designed by a certain international designer but was made by a maker in their local community; it's a great story to tell. It certainly sets apart Opendesk furniture from the usual mass-produced items from a store.

YOU GET DESIGNER PRODUCTS WITHOUT THE DESIGNER PRICE TAG, A MORE SOCIAL, ECO- FRIENDLY ALTERNATIVE TO MASSPRODUCTION, AND AN AFFORDABLE WAY TO BUY CUSTOM-MADE PRODUCTS.

Nick and Joni are taking a community-based approach to define and evolve Opendesk and the "open making" business model. They're engaging thought leaders and practitioners to define this new movement. They have a separate Open Making site, which includes a manifesto, a field guide, and an invitation to get involved in the Open Making community.⁴ People can submit ideas and discuss the principles and business practices they'd like to see used.

Nick and Joni talked a lot with us about intellectual property (IP) and commercialization. Many of their designers fear the idea that someone could take one of their design files and make and sell infinite number of pieces of furniture with it. As a consequence, most Opendesk designers choose the Attribution-NonCommercial license (CC BY-NC).

Opendesk established a set of principles for what their community considers commercial and noncommercial use. Their website states:

It is unambiguously commercial use when anyone:

- charges a fee or makes a profit when making an Opendesk
- sells (or bases a commercial service on) an Opendesk

It follows from this that noncommercial use is when you make an Opendesk yourself, with no intention to gain commercial advantage or monetary compensation. For example, these qualify as noncommercial:

- you are an individual with your own CNC machine, or access to a shared CNC machine, and will personally cut and make a few pieces of furniture yourself
- you are a student (or teacher) and you use the design files for educational purposes or training (and do not intend to sell the resulting pieces)
- you work for a charity and get furniture cut by volunteers, or by employees at a fab lab or maker space

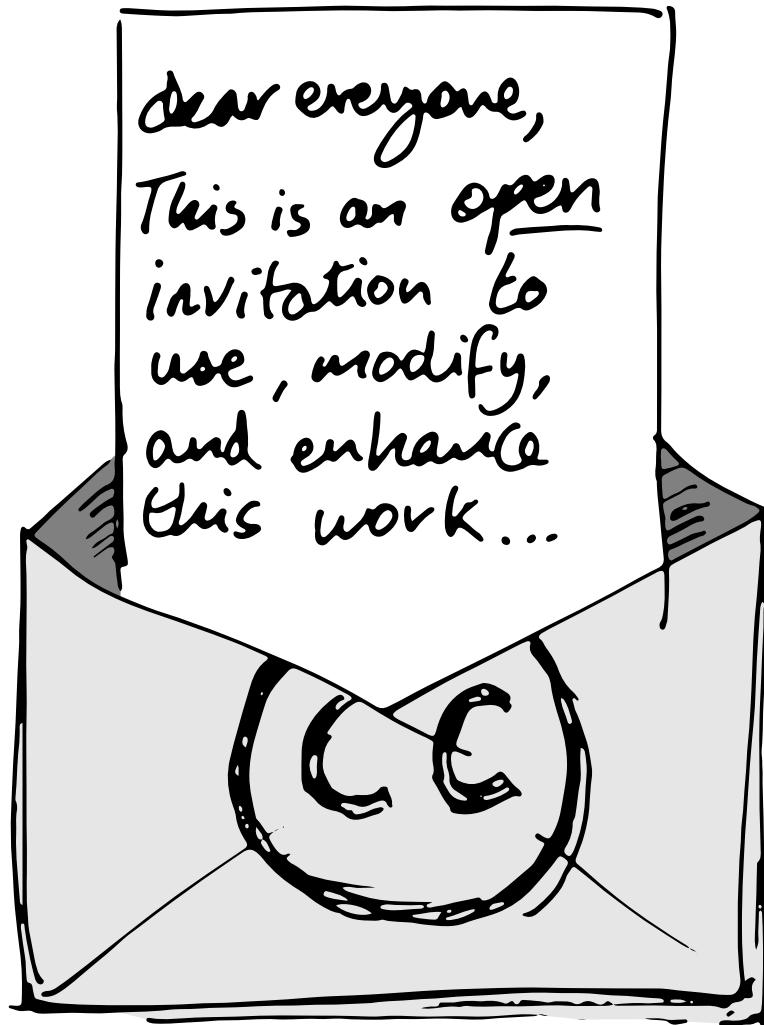
Whether or not people technically are doing things that implicate IP, Nick and Joni have found that people tend to comply with the wishes of creators out of a sense of fairness. They have found that behavioral economics can replace some of the thorny legal issues. In their business model, Nick and Joni are trying to suspend the focus on IP and build an open business model that works for all stakeholders—designers, channels, manufacturers, and customers. For them, the value Opendesk generates hangs off "open," not IP.

The mission of Opendesk is about relocalizing manufacturing, which changes the way we think about how goods are made. Commercialization is integral to their mission, and they've begun to focus on success metrics that track how many makers and designers are engaged through Opendesk in revenue-making work.

As a global platform for local making, Opendesk's business model has been built on honesty, transparency, and inclusivity. As Nick and Joni describe it, they put ideas out there that get traction and then have faith in people.

Web links

- 1 www.opendesk.cc/designers
- 2 www.opendesk.cc/open-making/makers/
- 3 www.opendesk.cc/open-making/join
- 4 openmaking.is



an
unusual
invitation

OPENSTAX



OpenStax is a nonprofit that provides free, openly licensed textbooks for high-enrollment introductory college courses and Advanced Placement courses. Founded in 2012 in the U.S.

www.openstaxcollege.org

Revenue model: grant funding, charging for custom services, charging for physical copies (textbook sales)

Interview date: December 16, 2015
Interviewee: David Harris, editor-in-chief

Profile written by Paul Stacey

OpenStax is an extension of a program called Connexions, which was started in 1999 by Dr. Richard Baraniuk, the Victor E. Cameron Professor of Electrical and Computer Engineering at Rice University in Houston, Texas. Frustrated by the limitations of traditional textbooks and courses, Dr. Baraniuk wanted to provide authors and learners a way to share and freely adapt educational materials such as courses, books, and reports. Today, Connexions (now called OpenStax CNX) is one of the world's best libraries of customizable educational materials, all licensed with Creative Commons and available to anyone, anywhere, anytime—for free.

In 2008, while in a senior leadership role at WebAssign and looking at ways to reduce the risk that came with relying on publishers, David began investigating open educational resources (OER) and discovered Con-

nexions. A year and a half later, Connexions received a grant to help grow the use of OER so that it could meet the needs of students who couldn't afford textbooks. David came on board to spearhead this effort. Connexions became OpenStax CNX; the program to create open textbooks became OpenStax College, now simply called OpenStax.

David brought with him a deep understanding of the best practices of publishing along with where publishers have inefficiencies. In David's view, peer review and high standards for quality are critically important if you want to scale easily. Books have to have logical scope and sequence, they have to exist as a whole and not in pieces, and they have to be easy to find. The working hypothesis for the launch of OpenStax was to professionally produce a turnkey textbook by investing effort up front, with the expectation that this would lead to

rapid growth through easy downstream adoptions by faculty and students.

In 2012, OpenStax College launched as a nonprofit with the aim of producing high-quality, peer-reviewed full-color textbooks that would be available for free for the twenty-five most heavily attended college courses in the nation. Today they are fast approaching that number. There is data that proves the success of their original hypothesis on how many students they could help and how much money they could help save.¹ Professionally produced content scales rapidly. All with no sales force!

OpenStax textbooks are all Attribution (CC BY) licensed, and each textbook is available as a PDF, an e-book, or web pages. Those who want a physical copy can buy one for an affordable price. Given the cost of education and student debt in North America, free or very low-cost textbooks are very appealing. OpenStax encourages students to talk to their professor and librarians about these textbooks and to advocate for their use.

Teachers are invited to try out a single chapter from one of the textbooks with students. If that goes well, they're encouraged to adopt the entire book. They can simply paste a URL into their course syllabus, for free and unlimited access. And with the CC BY license, teachers are free to delete chapters, make changes, and customize any book to fit their needs.

Any teacher can post corrections, suggest examples for difficult concepts, or volunteer as an editor or author. As many teachers also want supplemental material to accompany a textbook, OpenStax also provides slide presentations, test banks, answer keys, and so on.

Institutions can stand out by offering students a lower-cost education through the use of OpenStax textbooks; there's even a textbook-savings calculator they can use to see how much students would save. OpenStax keeps a running list of institutions that have adopted their textbooks.²

Unlike traditional publishers' monolithic approach of controlling intellectual property, distribution, and so many other aspects, OpenStax has adopted a model that embraces open licensing and relies on an extensive network of partners.

Up-front funding of a professionally produced all-color turnkey textbook is expensive. For this part of their model, OpenStax relies on philanthropy. They have initially been funded by the William and Flora Hewlett Foundation, the Laura and John Arnold Foundation, the Bill and Melinda Gates Foundation, the 20 Million Minds Foundation, the Maxfield Foundation, the Calvin K. Kazanjian Foundation, and Rice University. To develop additional titles and supporting technology is probably still going to require philanthropic investment.

However, ongoing operations will not rely on foundation grants but instead on funds received through an ecosystem of over forty partners, whereby a partner takes core content from OpenStax and adds features that it can create revenue from. For example, WebAssign, an online homework and assessment tool, takes the physics book and adds algorithmically generated physics problems, with problem-specific feedback, detailed solutions, and tutorial support. WebAssign resources are available to students for a fee.

Another example is Odigia, who has turned OpenStax books into interactive learning experiences and created additional tools to measure and promote student engagement. Odigia licenses its learning platform to institutions. Partners like Odigia and WebAssign give a percentage of the revenue they earn back to OpenStax, as mission-support fees. OpenStax has already published revisions of their titles, such as *Introduction to Sociology 2e*, using these funds.

In David's view, this approach lets the market operate at peak efficiency. OpenStax's partners don't have to worry about developing textbook content, freeing them up from those

development costs and letting them focus on what they do best. With OpenStax textbooks available at no cost, they can provide their services at a lower cost—not free, but still saving students money. OpenStax benefits not only by receiving mission-support fees but through free publicity and marketing. OpenStax doesn't have a sales force; partners are out there showcasing their materials.

OpenStax's cost of sales to acquire a single student is very, very low and is a fraction of what traditional players in the market face. This year, Tyton Partners is actually evaluating the costs of sales for an OER effort like OpenStax in comparison with incumbents. David looks forward to sharing these findings with the community.

MAKE IT POSSIBLE FOR EVERY STUDENT WHO WANTS ACCESS TO EDUCATION TO GET IT.

While OpenStax books are available online for free, many students still want a print copy. Through a partnership with a print and courier company, OpenStax offers a complete solution that scales. OpenStax sells tens of thousands of print books. The price of an OpenStax sociology textbook is about twenty-eight dollars, a fraction of what sociology textbooks usually cost. OpenStax keeps the prices low but does aim to earn a small margin on each book sold, which also contributes to ongoing operations.

Campus-based bookstores are part of the OpenStax solution. OpenStax collaborates with NACSCORP (the National Association of College Stores Corporation) to provide print versions of their textbooks in the stores. While the overall cost of the textbook is significantly less than a traditional textbook, bookstores can still make a profit on sales. Sometimes students take the savings they have from the lower-priced book and use it to buy other things in the bookstore. And OpenStax is trying to break

the expensive behavior of excessive returns by having a no-returns policy. This is working well, since the sell-through of their print titles is virtually a hundred percent.

David thinks of the OpenStax model as "OER 2.0." So what is OER 1.0? Historically in the OER field, many OER initiatives have been locally funded by institutions or government ministries. In David's view, this results in content that has high local value but is infrequently adopted nationally. It's therefore difficult to show payback over a time scale that is reasonable.

OER 2.0 is about OER intended to be used and adopted on a national level right from the start. This requires a bigger investment up front but pays off through wide geographic adoption. The OER 2.0 process for OpenStax involves two development models. The first is what David calls the acquisition model, where OpenStax purchases the rights from a publisher or author for an already published book and then extensively revises it. The OpenStax physics textbook, for example, was licensed from an author after the publisher released the rights back to the authors. The second model is to develop a book from scratch, a good example being their biology book.

The process is similar for both models. First they look at the scope and sequence of existing textbooks. They ask questions like what does the customer need? Where are students having challenges? Then they identify potential authors and put them through a rigorous evaluation—only one in ten authors make it through. OpenStax selects a team of authors who come together to develop a template for a chapter and collectively write the first draft (or revise it, in the acquisitions model). (OpenStax doesn't do books with just a single author as David says it risks the project going longer than scheduled.) The draft is peer-reviewed with no less than three reviewers per chapter. A second draft is generated, with artists producing illustrations and visuals to go along with the text. The book is then copyedited to

ensure grammatical correctness and a singular voice. Finally, it goes into production and through a final proofread. The whole process is very time-consuming.

All the people involved in this process are paid. OpenStax does not rely on volunteers. Writers, reviewers, illustrators, and editors are all paid an up-front fee—OpenStax does not use a royalty model. A best-selling author might make more money under the traditional publishing model, but that is only maybe 5 percent of all authors. From David's perspective, 95 percent of all authors do better under the OER 2.0 model, as there is no risk to them and they earn all the money up front.

David thinks of the Attribution license (CC BY) as the "innovation license." It's core to the mission of OpenStax, letting people use their textbooks in innovative ways without having to ask for permission. It frees up the whole market and has been central to OpenStax being able to bring on partners. OpenStax sees a lot of customization of their materials. By enabling frictionless remixing, CC BY gives teachers control and academic freedom.

Using CC BY is also a good example of using strategies that traditional publishers can't. Traditional publishers rely on copyright to prevent others from making copies and heavily invest in digital rights management to ensure their books aren't shared. By using CC BY, OpenStax avoids having to deal with digital rights management and its costs. OpenStax books can be copied and shared over and over again. CC BY changes the rules of engagement and takes advantage of traditional market inefficiencies.

As of September 16, 2016, OpenStax has achieved some impressive results. From the *OpenStax at a Glance* fact sheet from their recent press kit:

- Books published: 23
- Students who have used OpenStax: 1.6 million

- Money saved for students: \$155 million
- Money saved for students in the 2016/17 academic year: \$77 million
- Schools that have used OpenStax: 2,668 (This number reflects all institutions using at least one OpenStax textbook. Out of 2,668 schools, 517 are two-year colleges, 835 four-year colleges and universities, and 344 colleges and universities outside the U.S.)

While OpenStax has to date been focused on the United States, there is overseas adoption especially in the science, technology, engineering, and math (STEM) fields. Large scale adoption in the United States is seen as a necessary precursor to international interest.

OpenStax has primarily focused on introductory-level college courses where there is high enrollment, but they are starting to think about *verticals*—a broad offering for a specific group or need. David thinks it would be terrific if OpenStax could provide access to free textbooks through the entire curriculum of a nursing degree, for example.

Finally, for OpenStax success is not just about the adoption of their textbooks and student savings. There is a human aspect to the work that is hard to quantify but incredibly important. They get emails from students saying how OpenStax saved them from making difficult choices like buying food or a textbook. OpenStax would also like to assess the impact their books have on learning efficiency, persistence, and completion. By building an open business model based on Creative Commons, OpenStax is making it possible for every student who wants access to education to get it.

Web links

- 1 news.rice.edu/files/2016/01/0119-OPENSTAX-2016Infographic-lg-1tahxiu.jpg
- 2 openstax.org/adopters

AMANDA PALMER



Amanda Palmer is a musician, artist, and writer. Based in the U.S.

amandapalmer.net

Revenue model: crowdfunding (subscription-based), pay-what-you-want, charging for physical copies (book and album sales), charging for in-person version (performances), selling merchandise

Interview date: December 15, 2015

Profile written by Sarah Hinchliff Pearson

Since the beginning of her career, Amanda Palmer has been on what she calls a “journey with no roadmap,” continually experimenting to find new ways to sustain her creative work.¹ In her best-selling book, *The Art of Asking*, Amanda articulates exactly what she has been and continues to strive for—“the ideal sweet spot . . . in which the artist can share freely and directly feel the reverberations of their artistic gifts to the community, and make a living doing that.”

While she seems to have successfully found that sweet spot for herself, Amanda is the first to acknowledge there is no silver bullet. She thinks the digital age is both an exciting and frustrating time for creators. “On the one hand, we have this beautiful shareability,” Amanda said. “On the other, you’ve got a bunch of con-

fused artists wondering how to make money to buy food so we can make more art.”

Amanda began her artistic career as a street performer. She would dress up in an antique wedding gown, paint her face white, stand on a stack of milk crates, and hand out flowers to strangers as part of a silent dramatic performance. She collected money in a hat. Most people walked by her without stopping, but an essential few stopped to watch and drop some money into her hat to show their appreciation. Rather than dwelling on the majority of people who ignored her, she felt thankful for those who stopped. “All I needed was . . . some people,” she wrote in her book. “Enough people.

Enough to make it worth coming back the next day, enough people to help me make rent and put food on the table. Enough so I could keep making art."

Amanda has come a long way from her street-performing days, but her career remains dominated by that same sentiment—finding ways to reach “her crowd” and feeling gratitude when she does. With her band the Dresden Dolls, Amanda tried the traditional path of signing with a record label. It didn’t take for a variety of reasons, but one of them was that the label had absolutely no interest in Amanda’s view of success. They wanted hits, but making music for the masses was never what Amanda and the Dresden Dolls set out to do.

After leaving the record label in 2008, she began experimenting with different ways to make a living. She released music directly to the public without involving a middle man, releasing digital files on a “pay what you want” basis and selling CDs and vinyl. She also made money from live performances and merchandise sales. Eventually, in 2012 she decided to try her hand at the sort of crowdfunding we know so well today. Her Kickstarter project started with a goal of \$100,000, and she made \$1.2 million. It remains one of the most successful Kickstarter projects of all time.

Today, Amanda has switched gears away from crowdfunding for specific projects to instead getting consistent financial support from her fan base on Patreon, a crowdfunding site that allows artists to get recurring donations from fans. More than eight thousand people have signed up to support her so she can create music, art, and any other creative “thing” that she is inspired to make. The recurring pledges are made on a “per thing” basis. All of the content she makes is made freely available under an Attribution-NonCommercial-ShareAlike license (CC BY-NC-SA).

Making her music and art available under Creative Commons licensing undoubtedly limits her options for how she makes a living. But sharing her work has been part of her model since the beginning of her career, even before

she discovered Creative Commons. Amanda says the Dresden Dolls used to get ten emails per week from fans asking if they could use their music for different projects. They said yes to all of the requests, as long as it wasn’t for a completely for-profit venture. At the time, they used a short-form agreement written by Amanda herself. “I made everyone sign that contract so at least I wouldn’t be leaving the band vulnerable to someone later going on and putting our music in a Camel cigarette ad,” Amanda said. Once she discovered Creative Commons, adopting the licenses was an easy decision because it gave them a more formal, standardized way of doing what they had been doing all along. The NonCommercial licenses were a natural fit.

Amanda embraces the way her fans share and build upon her music. In *The Art of Asking*, she wrote that some of her fans’ unofficial videos using her music surpass the official videos in number of views on YouTube. Rather than seeing this sort of thing as competition, Amanda celebrates it. “We got into this because we wanted to share the joy of music,” she said.

This is symbolic of how nearly everything she does in her career is motivated by a desire to connect with her fans. At the start of her career, she and the band would throw concerts at house parties. As the gatherings grew, the line between fans and friends was completely blurred. “Not only did most our early fans know where I lived and where we practiced, but most of them had also been in my kitchen,” Amanda wrote in *The Art of Asking*.

Even though her fan base is now huge and global, she continues to seek this sort of human connection with her fans. She seeks out face-to-face contact with her fans every chance she can get. Her hugely successful Kickstarter featured fifty concerts at house parties for backers. She spends hours in the signing line after shows. It helps that Amanda has the kind of dynamic, engaging personality that instantly draws people to her, but a big component of

her ability to connect with people is her willingness to listen. "Listening fast and caring immediately is a skill unto itself," Amanda wrote.

Another part of the connection fans feel with Amanda is how much they know about her life. Rather than trying to craft a public persona or image, she essentially lives her life as an open book. She has written openly about incredibly personal events in her life, and she isn't afraid to be vulnerable. Having that kind of trust in her fans—the trust it takes to be truly honest—begets trust from her fans in return. When she meets fans for the first time after a show, they can legitimately feel like they know her.

"With social media, we're so concerned with the picture looking palatable and consumable that we forget that being human and showing the flaws and exposing the vulnerability actually create a deeper connection than just looking fantastic," Amanda said. "Everything in our culture is telling us otherwise. But my experience has shown me that the risk of making yourself vulnerable is almost always worth it."

Not only does she disclose intimate details of her life to them, she sleeps on their couches, listens to their stories, cries with them. In short, she treats her fans like friends in nearly every possible way, even when they are complete strangers. This mentality—that fans are friends—is completely intertwined with Amanda's success as an artist. It is also intertwined with her use of Creative Commons licenses. Because that is what you do with your friends—you share.

After years of investing time and energy into building trust with her fans, she has a strong enough relationship with them to ask for support—through pay-what-you-want donations, Kickstarter, Patreon, or even asking them to lend a hand at a concert. As Amanda explains it, crowdfunding (which is really what all of these different things are) is about asking for support from people who know and trust you.

IT SOUNDS SO CORNY, BUT MY EXPERIENCE IN FORTY YEARS ON THIS PLANET HAS POINTED ME TO AN OBVIOUS TRUTH—THAT CONNECTION WITH HUMAN BEINGS FEELS SO MUCH BETTER AND MORE FULFILLING THAN APPROACHING ART THROUGH A CAPITALIST LENS. THERE IS NO MORE SATISFYING END GOAL THAN HAVING SOMEONE TELL YOU THAT WHAT YOU DO IS GENUINELY OF VALUE TO THEM.

People who feel personally invested in your success.

"When you openly, radically trust people, they not only take care of you, they become your allies, your family," she wrote. There really is a feeling of solidarity within her core fan base. From the beginning, Amanda and her band encouraged people to dress up for their shows. They consciously cultivated a feeling of belonging to their "weird little family."

This sort of intimacy with fans is not possible or even desirable for every creator. "I don't take for granted that I happen to be the type of person who loves cavorting with strangers," Amanda said. "I recognize that it's not necessarily everyone's idea of a good time. Everyone does it differently. Replicating what I have done won't work for others if it isn't joyful to them. It's about finding a way to channel energy in a way that is joyful to you."

Yet while Amanda joyfully interacts with her fans and involves them in her work as much as possible, she does keep one job primarily to herself—writing the music. She loves the creativity with which her fans use and adapt her work, but she intentionally does not involve them at the first stage of creating her artistic work. And, of course, the songs and music are what initially draw people to Amanda Palmer. It is only once she has connected to people through her music that she can then begin to build ties with them on a more personal level, both in person and online. In her book, Amanda describes it as casting a net. It starts with the art and then the bond strengthens with human connection.

For Amanda, the entire point of being an artist is to establish and maintain this connection. “It sounds so corny,” she said, “but my experience in forty years on this planet has pointed me to an obvious truth—that connection with human beings feels so much better and more fulfilling than approaching art through a capitalist lens. There is no more satisfying end goal than having someone tell you that what you do is genuinely of value to them.”

As she explains it, when a fan gives her a ten-dollar bill, usually what they are saying is that the money symbolizes some deeper value the music provided them. For Amanda, art is not just a product; it’s a relationship. Viewed from this lens, what Amanda does today is not that different from what she did as a young street performer. She shares her music and other artistic gifts. She shares herself. And then rather than forcing people to help her, she lets them.

Web link

- 1 <http://www.forbes.com/sites/zackomalleygreenburg/2015/04/16/amanda-palmer-uncut-the-kickstarter-queen-on-spotify-patreon-and-taylor-swift/#44e20ce46d67>

PLOS (PUBLIC LIBRARY OF SCIENCE)



PLOS (Public Library of Science) is a nonprofit that publishes a library of academic journals and other scientific literature. Founded in 2000 in the U.S.

plos.org

Revenue model: charging content creators an author processing charge to be featured in the journal

Interview date: March 7, 2016
Interviewee: Louise Page, publisher

Profile written by Paul Stacey

The Public Library of Science (PLOS) began in 2000 when three leading scientists—Harold E. Varmus, Patrick O. Brown, and Michael Eisen—started an online petition. They were calling for scientists to stop submitting papers to journals that didn't make the full text of their papers freely available immediately or within six months. Although tens of thousands signed the petition, most did not follow through. In August 2001, Patrick and Michael announced that they would start their own nonprofit publishing operation to do just what the petition

promised. With start-up grant support from the Gordon and Betty Moore Foundation, PLOS was launched to provide new open-access journals for biomedicine, with research articles being released under Attribution (CC BY) licenses.

Traditionally, academic publishing begins with an author submitting a manuscript to a publisher. After in-house technical and ethical considerations, the article is then peer-reviewed to determine if the quality of the work is acceptable for publishing. Once accepted,

the publisher takes the article through the process of copyediting, typesetting, and eventual publishing in a print or online publication. Traditional journal publishers recover costs and earn profit by charging a subscription fee to libraries or an access fee to users wanting to read the journal or article.

For Louise Page, the current publisher of PLOS, this traditional model results in inequity. Access is restricted to those who can pay. Most research is funded through government-appointed agencies, that is, with public funds. It's unjust that the public who funded the research would be required to pay again to access the results. Not everyone can afford the ever-escalating subscription fees publishers charge, especially when library budgets are being reduced. Restricting access to the results of scientific research slows the dissemination of this research and advancement of the field. It was time for a new model.

That new model became known as open access. That is, free and open availability on the Internet. Open-access research articles are not behind a paywall and do not require a log-in. A key benefit of open access is that it allows people to freely use, copy, and distribute the articles, as they are primarily published under an Attribution (CC BY) license (which only requires the user to provide appropriate attribution). And more importantly, policy makers, clinicians, entrepreneurs, educators, and students around the world have free and timely access to the latest research immediately on publication.

However, open access requires rethinking the business model of research publication. Rather than charge a subscription fee to access the journal, PLOS decided to turn the model on its head and charge a *publication* fee, known as an article-processing charge. This up-front fee, generally paid by the funder of the research or the author's institution, covers the expenses such as editorial oversight, peer-review management, journal production, online hosting,

and support for discovery. Fees are per article and are billed upon acceptance for publishing. There are no additional charges based on word length, figures, or other elements.

Calculating the article-processing charge involves taking all the costs associated with publishing the journal and determining a cost per article that collectively recovers costs. For PLOS's journals in biology, medicine, genetics, computational biology, neglected tropical diseases, and pathogens, the article-processing charge ranges from \$2,250 to \$2,900. Article-publication charges for *PLOS ONE*, a journal started in 2006, are just under \$1,500.

PLOS believes that lack of funds should not be a barrier to publication. Since its inception, PLOS has provided fee support for individuals and institutions to help authors who can't afford the article-processing charges.

Louise identifies marketing as one area of big difference between PLOS and traditional journal publishers. Traditional journals have to invest heavily in staff, buildings, and infrastructure to market their journal and convince customers to subscribe. Restricting access to subscribers means that tools for managing access control are necessary. They spend millions of dollars on access-control systems, staff to manage them, and sales staff. With PLOS's open-access publishing, there's no need for these massive expenses; the articles are free, open, and accessible to all upon publication. Additionally, traditional publishers tend to spend more on marketing to libraries, who ultimately pay the subscription fees. PLOS provides a better service for authors by promoting their research directly to the research community and giving the authors exposure. And this encourages other authors to submit their work for publication.

For Louise, PLOS would not exist without the Attribution license (CC BY). This makes it very clear what rights are associated with the content and provides a safe way for researchers to make their work available while ensuring

they get recognition (appropriate attribution). For PLOS, all of this aligns with how they think research content should be published and disseminated.

PLOS also has a broad open-data policy. To get their research paper published, PLOS authors must also make their *data* available in a public repository and provide a data-availability statement.

Business-operation costs associated with the open-access model still largely follow the existing publishing model. PLOS journals are online only, but the editorial, peer-review, production, typesetting, and publishing stages are all the same as for a traditional publisher. The editorial teams must be top notch. PLOS has to function as well as or better than other premier journals, as researchers have a choice about where to publish.

Researchers are influenced by journal rankings, which reflect the place of a journal within its field, the relative difficulty of being published in that journal, and the prestige associated with it. PLOS journals rank high, even though they are relatively new.

The promotion and tenure of researchers are partially based how many times other researchers cite their articles. Louise says when researchers want to discover and read the work of others in their field, they go to an online aggregator or search engine, and not typically to a particular journal. The CC BY licensing of PLOS research articles ensures easy access for readers and generates more discovery and citations for authors.

Louise believes that open access has been a huge success, progressing from a movement led by a small cadre of researchers to something that is now widespread and used in some form by every journal publisher. PLOS has had a big impact. In 2012 to 2014, they published more open-access articles than BioMed Central, the original open-access publisher, or anyone else.

PLOS further disrupted the traditional journal-publishing model by pioneering the concept of a megajournal. The *PLOS ONE* megajournal, launched in 2006, is an open-access

peer-reviewed academic journal that is much larger than a traditional journal, publishing thousands of articles per year and benefiting from economies of scale. *PLOS ONE* has a broad scope, covering science and medicine as well as social sciences and the humanities. The review and editorial process is less subjective. Articles are accepted for publication based on whether they are technically sound rather than perceived importance or relevance. This is very important in the current debate about the integrity and reproducibility of research because negative or null results can then be published as well, which are generally rejected by traditional journals. *PLOS ONE*, like all the PLOS journals, is online only with no print version. PLOS passes on the financial savings accrued through economies of scale to researchers and the public by lowering the article-processing charges, which are below that of other journals. *PLOS ONE* is the biggest journal in the world and has really set the bar for publishing academic journal articles on a large scale. Other publishers see the value of the *PLOS ONE* model and are now offering their own multidisciplinary forums for publishing all sound science.

Louise outlined some other aspects of the research-journal business model PLOS is experimenting with, describing each as a kind of slider that could be adjusted to change current practice.

One slider is time to publication. Time to publication may shorten as journals get better at providing quicker decisions to authors. However, there is always a trade-off with scale, as the bigger the volume of articles, the more time the approval process inevitably takes.

Peer review is another part of the process that could change. It's possible to redefine what peer review actually is, when to review, and what constitutes the final article for publication. Louise talked about the potential to shift to an open-review process, placing the emphasis on transparency rather than dou-

ble-blind reviews. Louise thinks we're moving into a direction where it's actually beneficial for an author to know who is reviewing their paper and for the reviewer to know their review will be public. An open-review process can also ensure everyone gets credit; right now, credit is limited to the publisher and author.

Louise says research with negative outcomes is almost as important as positive results. If journals published more research with negative outcomes, we'd learn from what didn't work. It could also reduce how much the research wheel gets reinvented around the world.

Another adjustable practice is the sharing of articles at early preprint stages. Publication of research in a peer-reviewed journal can take a long time because articles must undergo extensive peer review. The need to quickly circulate current results within a scientific community has led to a practice of distributing pre-print documents that have not yet undergone peer review. Preprints broaden the peer-review process, allowing authors to receive early feedback from a wide group of peers, which can help revise and prepare the article for submission. Offsetting the advantages of preprints are author concerns over ensuring their primacy of being first to come up with findings based on their research. Other researchers may see findings the preprint author has not yet thought of. However, preprints help researchers get their discoveries out early and establish precedence. A big challenge is that researchers don't have a lot of time to comment on preprints.

What constitutes a journal article could also change. The idea of a research article as printed, bound, and in a library stack is outdated. Digital and online open up new possibilities, such as a living document evolving over time, inclusion of audio and video, and interactivity, like discussion and recommendations. Even the size of what gets published could change. With these changes the current form factor for what constitutes a research article would undergo transformation.

As journals scale up, and new journals are introduced, more and more information is be-

ing pushed out to readers, making the experience feel like drinking from a fire hose. To help mitigate this, PLOS aggregates and curates content from PLOS journals and their network of blogs.¹ It also offers something called Article-Level Metrics, which helps users assess research most relevant to the field itself, based on indicators like usage, citations, social bookmarking and dissemination activity, media and blog coverage, discussions, and ratings.² Louise believes that the journal model could evolve to provide a more friendly and interactive user experience, including a way for readers to communicate with authors.

The big picture for PLOS going forward is to combine and adjust these experimental practices in ways that continue to improve accessibility and dissemination of research, while ensuring its integrity and reliability. The ways they interlink are complex. The process of change and adjustment is not linear. PLOS sees itself as a very flexible publisher interested in exploring all the permutations research-publishing can take, with authors and readers who are open to experimentation.

For PLOS, success is not about revenue. Success is about proving that scientific research can be communicated rapidly and economically at scale, for the benefit of researchers and society. The CC BY license makes it possible for PLOS to publish in a way that is unfettered, open, and fast, while ensuring that the authors get credit for their work. More than two million scientists, scholars, and clinicians visit PLOS every month, with more than 135,000 quality articles to peruse for free.

Ultimately, for PLOS, its authors, and its readers, success is about making research discoverable, available, and reproducible for the advancement of science.

Web links

- 1 collections.plos.org
- 2 plos.org/article-level-metrics

RIJKSMUSEUM



PUBLIC
DOMAIN

The Rijksmuseum is a Dutch national museum dedicated to art and history. Founded in 1800 in the Netherlands

www.rijksmuseum.nl

Revenue model: grants and government funding, charging for in-person version (museum admission), selling merchandise

Interview date: December 11, 2015

Interviewee: Lizzy Jongma, the data manager of the collections information department

Profile written by Paul Stacey

The Rijksmuseum, a national museum in the Netherlands dedicated to art and history, has been housed in its current building since 1885. The monumental building enjoyed more than 125 years of intensive use before needing a thorough overhaul. In 2003, the museum was closed for renovations. Asbestos was found in the roof, and although the museum was scheduled to be closed for only three to four years, renovations ended up taking ten years. During this time, the collection was moved to a different part of Amsterdam, which created a physical distance with the curators. Out of necessity, they started digitally photographing the collection and creating metadata (information about each object to put into a database). With the renovations going on for so long, the museum became largely forgotten by the public. Out of these circumstances emerged a new and more open model for the museum.

By the time Lizzy Jongma joined the Rijksmuseum in 2011 as a data manager, staff were fed up with the situation the museum was in. They also realized that even with the new and larger space, it still wouldn't be able to show very much of the whole collection—eight thousand of over one million works representing just 1 percent. Staff began exploring ways to express themselves, to have something to show for all of the work they had been doing. The Rijksmuseum is primarily funded by Dutch taxpayers, so was there a way for the museum provide benefit to the public while it was closed? They began thinking about sharing Rijksmuseum's collection using information technology. And they put up a card-catalog like database of the entire collection online.

It was effective but a bit boring. It was just data. A hackathon they were invited to got them to start talking about events like that as having potential. They liked the idea of inviting people to do cool stuff with their collection.

What about giving online access to digital representations of the one hundred most important pieces in the Rijksmuseum collection? That eventually led to why not put *the whole collection* online?

Then, Lizzy says, Europeana came along. Europeana is Europe's digital library, museum, and archive for cultural heritage.¹ As an online portal to museum collections all across Europe, Europeana had become an important online platform. In October 2010 Creative Commons released CC0 and its public-domain mark as tools people could use to identify works as free of known copyright. Europeana was the first major adopter, using CC0 to release metadata about their collection and the public domain mark for millions of digital works in their collection. Lizzy says the Rijksmuseum initially found this change in business practice a bit scary, but at the same time it stimulated even more discussion on whether the Rijksmuseum should follow suit.

They realized that they don't "own" the collection and couldn't realistically monitor and enforce compliance with the restrictive licensing terms they currently had in place. For example, many copies and versions of Vermeer's *Milkmaid* (part of their collection) were already online, many of them of very poor quality. They could spend time and money policing its use, but it would probably be futile and wouldn't make people stop using their images online. They ended up thinking it's an utter waste of time to hunt down people who use the Rijksmuseum collection. And anyway, restricting access meant the people they were frustrating the most were schoolkids.

In 2011 the Rijksmuseum began making their digital photos of works known to be free of copyright available online, using Creative Commons CC0 to place works in the public domain. A medium-resolution image was offered for free, but a high-resolution version cost forty euros. People started paying, but Lizzy says getting the money was frequently a

nightmare, especially from overseas customers. The administrative costs often offset revenue, and income above costs was relatively low. In addition, having to pay for an image of a work in the public domain from a collection owned by the Dutch government (i.e., paid for by the public) was contentious and frustrating for some. Lizzy says they had lots of fierce debates about what to do.

In 2013 the Rijksmuseum changed its business model. They Creative Commons licensed their highest-quality images and released them online for free. Digitization still cost money, however; they decided to define discrete digitization projects and find sponsors willing to fund each project. This turned out to be a successful strategy, generating high interest from sponsors and lower administrative effort for the Rijksmuseum. They started out making 150,000 high-quality images of their collection available, with the goal to eventually have the entire collection online.

Releasing these high-quality images for free reduced the number of poor-quality images that were proliferating. The high-quality image of Vermeer's *Milkmaid*, for example, is downloaded two to three thousand times a month. On the Internet, images from a source like the Rijksmuseum are more trusted, and releasing them with a Creative Commons CC0 means they can easily be found in other platforms. For example, Rijksmuseum images are now used in thousands of Wikipedia articles, receiving ten to eleven million views per month. This extends Rijksmuseum's reach far beyond the scope of its website. Sharing these images online creates what Lizzy calls the "Mona Lisa effect," where a work of art becomes so famous that people want to see it in real life by visiting the actual museum.

Every museum tends to be driven by the number of physical visitors. The Rijksmuseum is primarily publicly funded, receiving roughly 70 percent of its operating budget from the government. But like many museums, it must generate the rest of the funding through other means. The admission fee has long been a way

to generate revenue generation, including for the Rijksmuseum.

As museums create a digital presence for themselves and put up digital representations of their collection online, there's frequently a worry that it will lead to a drop in actual physical visits. For the Rijksmuseum, this has not turned out to be the case. Lizzy told us the Rijksmuseum used to get about one million visitors a year before closing and now gets more than two million a year. Making the collection available online has generated publicity and acts as a form of marketing. The Creative Commons mark encourages reuse as well. When the image is found on protest leaflets, milk cartons, and children's toys, people also see what museum the image comes from and this increases the museum's visibility.

In 2011 the Rijksmuseum received €1 million from the Dutch lottery to create a new web presence that would be different from any other museum's. In addition to redesigning their main website to be mobile friendly and responsive to devices like the iPad, the Rijksmuseum also created the Rijksstudio, where users and artists could use and do various things with the Rijksmuseum collection.²

The Rijksstudio gives users access to over two hundred thousand high-quality digital representations of masterworks from the collection. Users can zoom in to any work and even clip small parts of images they like. Rijksstudio is a bit like Pinterest. You can "like" works and compile your personal favorites, and you can share them with friends or download them free of charge. All the images in the Rijksstudio are copyright and royalty free, and users are encouraged to use them as they like, for private or even commercial purposes.

Users have created over 276,000 Rijksstudios, generating their own themed virtual exhibitions on a wide variety of topics ranging from tapestries to ugly babies and birds. Sets of images have also been created for ed-

RIJKSMUSEUM IMAGES ARE NOW USED IN THOUSANDS OF WIKIPEDIA ARTICLES, RECEIVING TEN TO ELEVEN MILLION VIEWS PER MONTH EXTENDING REACH FAR BEYOND THE SCOPE OF THEIR OWN WEBSITE.

ucational purposes including use for school exams.

Some contemporary artists who have works in the Rijksmuseum collection contacted them to ask why their works were not included in the Rijksstudio. The answer was that contemporary artists' works are still bound by copyright. The Rijksmuseum does encourage contemporary artists to use a Creative Commons license for their works, usually a CC BY-SA license (Attribution-ShareAlike), or a CC BY-NC (Attribution-NonCommercial) if they want to preclude commercial use. That way, their works can be made available to the public, but within limits the artists have specified.

The Rijksmuseum believes that art stimulates entrepreneurial activity. The line between creative and commercial can be blurry. As Lizzy says, even Rembrandt was commercial, making his livelihood from selling his paintings. The Rijksmuseum encourages entrepreneurial commercial use of the images in Rijksstudio. They've even partnered with the DIY marketplace Etsy to inspire people to sell their creations. One great example you can find on Etsy is a kimono designed by Angie Johnson, who used an image of an elaborate cabinet along with an oil painting by Jan Asselijn called *The Threatened Swan*.³

In 2013 the Rijksmuseum organized their first high-profile design competition, known as the Rijksstudio Award.⁴ With the call to action Make Your Own Masterpiece, the competition invites the public to use Rijksstudio images to

make new creative designs. A jury of renowned designers and curators selects ten finalists and three winners. The final award comes with a prize of €10,000. The second edition in 2015 attracted a staggering 892 top-class entries. Some award winners end up with their work sold through the Rijksmuseum store, such as the 2014 entry featuring makeup based on a specific color scheme of a work of art.⁵ The Rijksmuseum has been thrilled with the results. Entries range from the fun to the weird to the inspirational. The third international edition of the Rijksstudio Award started in September 2016.

For the next iteration of the Rijksstudio, the Rijksmuseum is considering an upload tool, for people to upload their own works of art, and enhanced social elements so users can interact with each other more.

Going with a more open business model generated lots of publicity for the Rijksmuseum. They were one of the first museums to open up their collection (that is, give free access) with high-quality images. This strategy, along with the many improvements to the Rijksmuseum's website, dramatically increased visits to their website from thirty-five thousand visits per month to three hundred thousand.

The Rijksmuseum has been experimenting with other ways to invite the public to look at and interact with their collection. On an international day celebrating animals, they ran a successful bird-themed event. The museum put together a showing of two thousand works that featured birds and invited bird-watchers to identify the birds depicted. Lizzy notes that while museum curators know a lot about the works in their collections, they may not know about certain details in the paintings such as bird species. Over eight hundred different birds were identified, including a specific species of crane bird that was unknown to the scientific community at the time of the painting.

For the Rijksmuseum, adopting an open business model was scary. They came up with many worst-case scenarios, imagining all kinds of awful things people might do with the museum's works. But Lizzy says those fears did not come true because "ninety-nine percent of people have respect for great art." Many museums think they can make a lot of money by selling things related to their collection. But in Lizzy's experience, museums are usually bad at selling things, and sometimes efforts to generate a small amount of money block something much bigger—the real value that the collection has. For Lizzy, clinging to small amounts of revenue is being penny-wise but pound-foolish. For the Rijksmuseum, a key lesson has been to never lose sight of its vision for the collection. Allowing access to and use of their collection has generated great promotional value—far more than the previous practice of charging fees for access and use. Lizzy sums up their experience: "Give away; get something in return. Generosity makes people happy to join you and help out."

Web links

- 1 www.europeana.eu/portal/en
- 2 www.rijksmuseum.nl/en/rijksstudio
- 3 www.etsy.com/ca/listing/175696771/fringe-kimono-silk-kimono-kimono-robe
- 4 www.rijksmuseum.nl/en/rijksstudio-award; the 2014 award: www.rijksmuseum.nl/en/rijksstudio-award-2014;
[the 2015 award: www.rijksmuseum.nl/en/rijksstudio-award-2015](http://www.rijksmuseum.nl/en/rijksstudio-award-2015)
- 5 www.rijksmuseum.nl/nl/rijksstudio/142328--nominees-rijksstudio-award/creatives/ba595afe-452d-46bd-9c8c-48dcbdd7f0a4

SHAREABLE



Shareable is an online magazine about sharing. Founded in 2009 in the U.S.

www.shareable.net

Revenue model: grant funding, crowdfunding (project-based), donations, sponsorships

Interview date: February 24, 2016

Interviewee: Neal Gorenflo, cofounder and executive editor

Profile written by Sarah Hinchliff Pearson

In 2013, Shareable faced an impasse. The non-profit online publication had helped start a sharing movement four years prior, but over time, they watched one part of the movement stray from its ideals. As giants like Uber and Airbnb gained ground, attention began to center on the “sharing economy” we know now—profit-driven, transactional, and loaded with venture-capital money. Leaders of corporate start-ups in this domain invited Shareable to advocate for them. The magazine faced a choice: ride the wave or stand on principle.

As an organization, Shareable decided to draw a line in the sand. In 2013, the cofounder and executive editor Neal Gorenflo wrote an opinion piece in the *PandoDaily* that charted Shareable’s new critical stance on the Silicon Valley version of the sharing economy, while contrasting it with aspects of the real sharing economy like open-source software, partici-

patory budgeting (where citizens decide how a public budget is spent), cooperatives, and more. He wrote, “It’s not so much that collaborative consumption is dead, it’s more that it risks dying as it gets absorbed by the ‘Borg.’”

Neal said their public critique of the corporate sharing economy defined what Shareable was and is. He does not think the magazine would still be around had they chosen differently. “We would have gotten another type of audience, but it would have spelled the end of us,” he said. “We are a small, mission-driven organization. We would never have been able to weather the criticism that Airbnb and Uber are getting now.”

Interestingly, impassioned supporters are only a small sliver of Shareable’s total audience. Most are casual readers who come across a Shareable story because it happens to align with a project or interest they have.

But choosing principles over the possibility of riding the coattails of the major corporate players in the sharing space saved Shareable's credibility. Although they became detached from the corporate sharing economy, the online magazine became the voice of the "real sharing economy" and continued to grow their audience.

Shareable is a magazine, but the content they publish is a means to furthering their role as a leader and catalyst of a movement. Shareable became a leader in the movement in 2009. "At that time, there was a sharing movement bubbling beneath the surface, but no one was connecting the dots," Neal said. "We decided to step into that space and take on that role." The small team behind the nonprofit publication truly believed sharing could be central to solving some of the major problems human beings face—resource inequality, social isolation, and global warming.

They have worked hard to find ways to tell stories that show different metrics for success. "We wanted to change the notion of what constitutes the good life," Neal said. While they started out with a very broad focus on sharing generally, today they emphasize stories about the physical commons like "sharing cities" (i.e., urban areas managed in a sustainable, cooperative way), as well as digital platforms that are run democratically. They particularly focus on how-to content that help their readers make changes in their own lives and communities.

More than half of Shareable's stories are written by paid journalists that are contracted by the magazine. "Particularly in content areas that are a priority for us, we really want to go deep and control the quality," Neal said. The rest of the content is either contributed by guest writers, often for free, or written by other publications from their network of content publishers. Shareable is a member of the Post Growth Alliance, which facilitates the sharing of content and audiences among a large and growing group of mostly nonprofits. Each or-

ganization gets a chance to present stories to the group, and the organizations can use and promote each other's stories. Much of the content created by the network is licensed with Creative Commons.

All of Shareable's original content is published under the Attribution license (CC BY), meaning it can be used for any purpose as long as credit is given to Shareable. Creative Commons licensing is aligned with Shareable's vision, mission, and identity. That alone explains the organization's embrace of the licenses for their content, but Neal also believes CC licensing helps them increase their reach. "By using CC licensing," he said, "we realized we could reach far more people through a formal and informal network of republishers or affiliates. That has definitely been the case. It's hard for us to measure the reach of other media properties, but most of the outlets who republish our work have much bigger audiences than we do."

In addition to their regular news and commentary online, Shareable has also experimented with book publishing. In 2012, they worked with a traditional publisher to release *Share or Die: Voices of the Get Lost Generation in an Age of Crisis*. The CC-licensed book was available in print form for purchase or online for free. To this day, the book—along with their CC-licensed guide *Policies for Shareable Cities*—are two of the biggest generators of traffic on their website.

In 2016, Shareable self-published a book of curated Shareable stories called *How to: Share, Save Money and Have Fun*. The book was available for sale, but a PDF version of the book was available for free. Shareable plans to offer the book in upcoming fund-raising campaigns.

This recent book is one of many fund-raising experiments Shareable has conducted in recent years. Currently, Shareable is primarily funded by grants from foundations, but they are actively moving toward a more diversified model. They have organizational sponsors and are working to expand their base of individual donors. Ideally, they will eventually be a hundred percent funded by their audience. Neal

believes being fully community-supported will better represent their vision of the world.

For Shareable, success is very much about their impact on the world. This is true for Neal, but also for everyone who works for Shareable. "We attract passionate people," Neal said. At times, that means employees work so hard they burn out. Neal tries to stress to the Shareable team that another part of success is having fun and taking care of yourself while you do something you love. "A central part of human beings is that we long to be on a great adventure with people we love," he said. "We are a species who look over the horizon and imagine and create new worlds, but we also seek the comfort of hearth and home."

In 2013, Shareable ran its first crowdfunding campaign to launch their Sharing Cities Network. Neal said at first they were on pace to fail spectacularly. They called in their advisers in a panic and asked for help. The advice they received was simple—"Sit your ass in a chair and start making calls." That's exactly what they did, and they ended up reaching their \$50,000 goal. Neal said the campaign helped them reach new people, but the vast majority of backers were people in their existing base.

For Neal, this symbolized how so much of success comes down to relationships. Over time, Shareable has invested time and energy into the relationships they have forged with their readers and supporters. They have also invested resources into building relationships *between* their readers and supporters.

Shareable began hosting events in 2010. These events were designed to bring the sharing community together. But over time they realized they could reach far more people if they helped their readers to host their own events. "If we wanted to go big on a conference, there was a huge risk and huge staffing needs, plus only a fraction of our community could travel to the event," Neal said. Enabling others to create their own events around the globe allowed them to scale up their work more effectively

and reach far more people. Shareable has catalyzed three hundred different events reaching over twenty thousand people since implementing this strategy three years ago. Going forward, Shareable is focusing the network on creating and distributing content meant to spur local action. For instance, Shareable will publish a new CC-licensed book in 2017 filled with ideas for their network to implement.

Neal says Shareable stumbled upon this strategy, but it seems to perfectly encapsulate just how the commons is supposed to work. Rather than a one-size-fits-all approach, Shareable puts the tools out there for people take the ideas and adapt them to their own communities.

SIYAVULA



Siyavula is a for-profit educational-technology company that creates textbooks and integrated learning experiences. Founded in 2012 in South Africa.

www.siyavula.com

Revenue model: charging for custom services, sponsorships

Interview date: April 5, 2016
Interviewee: Mark Horner, CEO

Profile written by Paul Stacey

Openness is a key principle for Siyavula. They believe that every learner and teacher should have access to high-quality educational resources, as this forms the basis for long-term growth and development. Siyavula has been a pioneer in creating high-quality open textbooks on mathematics and science subjects for grades 4 to 12 in South Africa.

In terms of creating an open business model that involves Creative Commons, Siyavula—and its founder, Mark Horner—have been around the block a few times. Siyavula has significantly shifted directions and strategies to survive and prosper. Mark says it's been very organic.

It all started in 2002, when Mark and several other colleagues at the University of Cape Town in South Africa founded the Free High School Science Texts project. Most students in South Africa high schools didn't have access to high-quality, comprehensive science and math

textbooks, so Mark and his colleagues set out to write them and make them freely available.

As physicists, Mark and his colleagues were advocates of open-source software. To make the books open and free, they adopted the Free Software Foundation's GNU Free Documentation License.¹ They chose LaTeX, a typesetting program used to publish scientific documents, to author the books. Over a period of five years, the Free High School Science Texts project produced math and physical-science textbooks for grades 10 to 12.

In 2007, the Shuttleworth Foundation offered funding support to make the textbooks available for trial use at more schools. Surveys before and after the textbooks were adopted showed there were no substantial criticisms of the textbooks' pedagogical content. This pleased both the authors and Shuttleworth; Mark remains incredibly proud of this accomplishment.

But the development of new textbooks froze at this stage. Mark shifted his focus to rural schools, which didn't have textbooks at all, and looked into the printing and distribution options. A few sponsors came on board but not enough to meet the need.

In 2007, Shuttleworth and the Open Society Institute convened a group of open-education activists for a small but lively meeting in Cape Town. One result was the Cape Town Open Education Declaration, a statement of principles, strategies, and commitment to help the open-education movement grow.² Shuttleworth also invited Mark to run a project writing open content for all subjects for K-12 in English. That project became Siyavula.

They wrote six original textbooks. A small publishing company offered Shuttleworth the option to buy out the publisher's existing K-9 content for every subject in South African schools in both English and Afrikaans. A deal was struck, and all the acquired content was licensed with Creative Commons, significantly expanding the collection beyond the six original books.

Mark wanted to build out the remaining curricula collaboratively through communities of practice—that is, with fellow educators and writers. Although sharing is fundamental to teaching, there can be a few challenges when you create educational resources collectively. One concern is legal. It is standard practice in education to copy diagrams and snippets of text, but of course this doesn't always comply with copyright law. Another concern is transparency. Sharing what you've authored means everyone can see it and opens you up to criticism. To alleviate these concerns, Mark adopted a team-based approach to authoring and insisted the curricula be based entirely on resources with Creative Commons licenses, thereby ensuring they were safe to share and free from legal repercussions.

Not only did Mark want the resources to be shareable, he wanted all teachers to be able to

remix and edit the content. Mark and his team had to come up with an open editable format and provide tools for editing. They ended up putting all the books they'd acquired and authored on a platform called Connexions.³ Siyavula trained many teachers to use Connexions, but it proved to be too complex and the textbooks were rarely edited.

Then the Shuttleworth Foundation decided to completely restructure its work as a foundation into a fellowship model (for reasons completely unrelated to Siyavula). As part of that transition in 2009–10, Mark inherited Siyavula as an independent entity and took ownership over it as a Shuttleworth fellow.

Mark and his team experimented with several different strategies. They tried creating an authoring and hosting platform called Full Marks so that teachers could share assessment items. They tried creating a service called Open Press, where teachers could ask for open educational resources to be aggregated into a package and printed for them. These services never really panned out.

Then the South African government approached Siyavula with an interest in printing out the original six Free High School Science Texts (math and physical-science textbooks for grades 10 to 12) for all high school students in South Africa. Although at this point Siyavula was a bit discouraged by open educational resources, they saw this as a big opportunity.

They began to conceive of the six books as having massive marketing potential for Siyavula. Printing Siyavula books for every kid in South Africa would give their brand huge exposure and could drive vast amounts of traffic to their website. In addition to print books, Si

USING SIYAVULA BOOKS

GENERATED HUGE SAVINGS FOR THE GOVERNMENT.

yavula could also make the books available on their website, making it possible for learners to access them using any device—computer, tablet, or mobile phone.

Mark and his team began imagining what they could develop beyond what was in the textbooks as a service they charge for. One key thing you can't do well in a printed textbook is demonstrate solutions. Typically, a one-line answer is given at the end of the book but nothing on the process for arriving at that solution. Mark and his team developed practice items and detailed solutions, giving learners plenty of opportunity to test out what they've learned. Furthermore, an algorithm could adapt these practice items to the individual needs of each learner. They called this service Intelligent Practice and embedded links to it in the open textbooks.

The costs for using Intelligent Practice were set very low, making it accessible even to those with limited financial means. Siyavula was going for large volumes and wide-scale use rather than an expensive product targeting only the high end of the market.

The government distributed the books to 1.5 million students, but there was an unexpected wrinkle: the books were delivered late. Rather than wait, schools who could afford it provided students with a different textbook. The Siyavula books were eventually distributed, but with well-off schools mainly using a different book, the primary market for Siyavula's Intelligent Practice service inadvertently became low-income learners.

Siyavula's site did see a dramatic increase in traffic. They got five hundred thousand visitors per month to their math site and the same number to their science site. Two-fifths of the traffic was reading on a "feature phone" (a nonsmartphone with no apps). People on basic phones were reading math and science on a two-inch screen at all hours of the day. To Mark, it was quite amazing and spoke to a need they were servicing.

At first, the Intelligent Practice services could only be paid using a credit card. This proved problematic, especially for those in the

low-income demographic, as credit cards were not prevalent. Mark says Siyavula got a harsh business-model lesson early on. As he describes it, it's not just about product, but how you sell it, who the market is, what the price is, and what the barriers to entry are.

Mark describes this as the first version of Siyavula's business model: open textbooks serving as marketing material and driving traffic to your site, where you can offer a related service and convert some people into a paid customer.

For Mark a key decision for Siyavula's business was to focus on how they can add value on top of their basic service. They'll charge only if they are adding unique value. The actual content of the textbook isn't unique at all, so Siyavula sees no value in locking it down and charging for it. Mark contrasts this with traditional publishers who charge over and over again for the same content without adding value.

Version two of Siyavula's business model was a big, ambitious idea—scale up. They also decided to sell the Intelligent Practice service to schools directly. Schools can subscribe on a per-student, per-subject basis. A single subscription gives a learner access to a single subject, including practice content from every grade available for that subject. Lower subscription rates are provided when there are over two hundred students, and big schools have a price cap. A 40 percent discount is offered to schools where both the science and math departments subscribe.

Teachers get a dashboard that allows them to monitor the progress of an entire class or view an individual learner's results. They can see the questions that learners are working on, identify areas of difficulty, and be more strategic in their teaching. Students also have their own personalized dashboard, where they can view the sections they've practiced, how many points they've earned, and how their performance is improving.

Based on the success of this effort, Siyavula decided to substantially increase the production of open educational resources so they could provide the Intelligent Practice service for a wider range of books. Grades 10 to 12 math and science books were reworked each year, and new books created for grades 4 to 6 and later grades 7 to 9.

In partnership with, and sponsored by, the Sasol Inzalo Foundation, Siyavula produced a series of natural sciences and technology workbooks for grades 4 to 6 called Thunderbolt Kids that uses a fun comic-book style.⁴ It's a complete curriculum that also comes with teacher's guides and other resources.

Through this experience, Siyavula learned they could get sponsors to help fund openly licensed textbooks. It helped that Siyavula had by this time nailed the production model. It cost roughly \$150,000 to produce a book in two languages. Sponsors liked the social-benefit aspect of textbooks unlocked via a Creative Commons license. They also liked the exposure their brand got. For roughly \$150,000, their logo would be visible on books distributed to over one million students.

The Siyavula books that are reviewed, approved, and branded by the government are freely and openly available on Siyavula's website under an Attribution-NoDerivs license (CC BY-ND) —NoDerivs means that these books cannot be modified. Non-government-branded books are available under an Attribution license (CC BY), allowing others to modify and redistribute the books.

Although the South African government paid to print and distribute hard copies of the books to schoolkids, Siyavula itself received no funding from the government. Siyavula initially tried to convince the government to provide them with five rand per book (about US35¢). With those funds, Mark says that Siyavula could have run its entire operation, built a community-based model for producing more books, and provide Intelligent Practice for free to every child in the country. But after a lengthy negotiation, the government said no.

Using Siyavula books generated huge savings for the government. Providing students with a traditionally published grade 12 science or math textbook costs around 250 rand per book (about US\$18). Providing the Siyavula version cost around 36 rand (about \$2.60), a savings of over 200 rand per book. But none of those savings were passed on to Siyavula. In retrospect, Mark thinks this may have turned out in their favor as it allowed them to remain independent from the government.

Just as Siyavula was planning to scale up the production of open textbooks even more, the South African government changed its textbook policy. To save costs, the government declared there would be only *one* authorized textbook for each grade and each subject. There was no guarantee that Siyavula's would be chosen. This scared away potential sponsors.

Rather than producing more textbooks, Siyavula focused on improving its Intelligent Practice technology for its existing books. Mark calls this version three of Siyavula's business model—focusing on the technology that provides the revenue-generating service and generating more users of this service. Version three got a significant boost in 2014 with an investment by the Omidyar Network (the philanthropic venture started by eBay founder Pierre Omidyar and his spouse), and continues to be the model Siyavula uses today.

Mark says sales are way up, and they are really nailing Intelligent Practice. Schools continue to use their open textbooks. The government-announced policy that there would be only one textbook per subject turned out to be highly contentious and is in limbo.

Siyavula is exploring a range of enhancements to their business model. These include charging a small amount for assessment services provided over the phone, diversifying their market to all English-speaking countries in Africa, and setting up a consortium that

makes Intelligent Practice free to all kids by selling the nonpersonal data Intelligent Practice collects.

Siyavula is a for-profit business but one with a social mission. Their shareholders' agreement lists lots of requirements around openness for Siyavula, including stipulations that content always be put under an open license and that they can't charge for something that people volunteered to do for them. They believe each individual should have access to the resources and support they need to achieve the education they deserve. Having educational resources openly licensed with Creative Commons means they can fulfill their social mission, on top of which they can build revenue-generating services to sustain the ongoing operation of Siyavula. In terms of open business models, Mark and Siyavula may have been around the block a few times, but both he and the company are stronger for it.

Web links

- 1 www.gnu.org/licenses/fdl
- 2 www.capetowndeclaration.org
- 3 cnx.org
- 4 www.siyavula.com/products-primary-school.html

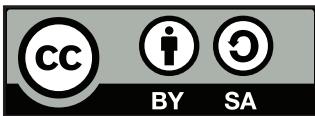
BUSINESS

2010.

AND THIS TIME IT'S

Personal

SPARKFUN



SparkFun is an online electronics retailer specializing in open hardware. Founded in 2003 in the U.S.

www.sparkfun.com

Revenue model: charging for physical copies (electronics sales)

Interview date: February 29, 2016
Interviewee: Nathan Seidle, founder

Profile written by Sarah Hinchliff Pearson

SparkFun founder and former CEO Nathan Seidle has a picture of himself holding up a clone of a SparkFun product in an electronics market in China, with a huge grin on his face. He was traveling in China when he came across their LilyPad wearable technology being made by someone else. His reaction was glee.

"Being copied is the greatest earmark of flattery and success," Nathan said. "I thought it was so cool that they were selling to a market we were never going to get access to otherwise. It was evidence of our impact on the world."

This worldview runs through everything SparkFun does. SparkFun is an electronics manufacturer. The company sells its products directly to the public online, and it bundles them with educational tools to sell to schools and teachers. SparkFun applies Creative Commons licenses to all of its schematics, images, tutorial content, and curricula, so anyone can

make their products on their own. Being copied is part of the design.

Nathan believes open licensing is good for the world. "It touches on our natural human instinct to share," he said. But he also strongly believes it makes SparkFun better at what they do. They encourage copying, and their products are copied at a very fast rate, often within ten to twelve weeks of release. This forces the company to compete on something other than product design, or what most commonly consider their intellectual property.

"We compete on business principles," Nathan said. "Claiming your territory with intellectual property allows you to get comfy and rest on your laurels. It gives you a safety net. We took away that safety net."

The result is an intense company-wide focus on product development and improvement. "Our products are so much better than they were five years ago," Nathan said. "We

used to just sell products. Now it's a product plus a video, a seventeen-page hookup guide, and example firmware on three different platforms to get you up and running faster. We have gotten better because we had to in order to compete. As painful as it is for us, it's better for the customers."

SparkFun parts are available on eBay for lower prices. But people come directly to SparkFun because SparkFun makes their lives easier. The example code works; there is a service number to call; they ship replacement parts the day they get a service call. They invest heavily in service and support. "I don't believe businesses should be competing with IP [intellectual property] barriers," Nathan said. "*This is the stuff they should be competing on.*"

SparkFun's company history began in Nathan's college dorm room. He spent a lot of time experimenting with and building electronics, and he realized there was a void in the market. "If you wanted to place an order for something," he said, "you first had to search far and wide to find it, and then you had to call or fax someone." In 2003, during his third year of college, he registered sparkfun.com and started re-selling products out of his bedroom. After he graduated, he started making and selling his own products.

Once he started designing his own products, he began putting the software and schematics online to help with technical support. After doing some research on licensing options, he chose Creative Commons licenses because he was drawn to the "human-readable deeds" that explain the licensing terms in simple terms. SparkFun still uses CC licenses for all of the schematics and firmware for the products they create.

The company has grown from a solo project to a corporation with 140 employees. In 2015, SparkFun earned \$33 million in revenue. Selling components and widgets to hobbyists, professionals, and artists remains a major part of SparkFun's business. They sell their own prod-

BEING COPIED IS THE GREATEST EARMARK OF FLATTERY AND SUCCESS.

ucts, but they also partner with Arduino (also profiled in this book) by manufacturing boards for resale using Arduino's brand.

SparkFun also has an educational department dedicated to creating a hands-on curriculum to teach students about electronics using prototyping parts. Because SparkFun has always been dedicated to enabling others to re-create and fix their products on their own, the more recent focus on introducing young people to technology is a natural extension of their core business.

"We have the burden and opportunity to educate the next generation of technical citizens," Nathan said. "Our goal is to affect the lives of three hundred and fifty thousand high school students by 2020."

The Creative Commons license underlying all of SparkFun's products is central to this mission. The license not only signals a willingness to share, but it also expresses a desire for others to get in and tinker with their products, both to learn and to make their products better. SparkFun uses the Attribution-ShareAlike license (CC BY-SA), which is a "copyleft" license that allows people to do anything with the content as long as they provide credit and make any adaptations available under the same licensing terms.

From the beginning, Nathan has tried to create a work environment at SparkFun that he himself would want to work in. The result is what appears to be a pretty fun workplace. The U.S. company is based in Boulder, Colorado. They have an eighty-thousand-square-foot facility (approximately seventy-four-hundred square meters), where they design and manufacture their products. They offer public tours of the

space several times a week, and they open their doors to the public for a competition once a year.

The public event, called the Autonomous Vehicle Competition, brings in a thousand to two thousand customers and other technology enthusiasts from around the area to race their own self-created bots against each other, participate in training workshops, and socialize. From a business perspective, Nathan says it's a terrible idea. But they don't hold the event for business reasons. "The reason we do it is because I get to travel and have interactions with our customers all the time, but most of our employees don't," he said. "This event gives our employees the opportunity to get face-to-face contact with our customers." The event infuses their work with a human element, which makes it more meaningful.

Nathan has worked hard to imbue a deeper meaning into the work SparkFun does. The company is, of course, focused on being fiscally responsible, but they are ultimately driven by something other than money. "Profit is not the goal; it is the outcome of a well-executed plan," Nathan said. "We focus on having a bigger impact on the world." Nathan believes they get some of the brightest and most amazing employees because they aren't singularly focused on the bottom line.

The company is committed to transparency and shares all of its financials with its employees. They also generally strive to avoid being another soulless corporation. They actively try to reveal the humans behind the company, and they work to ensure people coming to their site don't find only unchanging content.

SparkFun's customer base is largely made up of industrious electronics enthusiasts. They have customers who are regularly involved in the company's customer support, independently responding to questions in forums and product-comment sections. Customers also bring product ideas to the company. SparkFun regularly sifts through suggestions from custom-

ers and tries to build on them where they can. "From the beginning, we have been listening to the community," Nathan said. "Customers would identify a pain point, and we would design something to address it."

However, this sort of customer engagement does not always translate to people actively contributing to SparkFun's projects. The company has a public repository of software code for each of its devices online. On a particularly active project, there will only be about two dozen people contributing significant improvements. The vast majority of projects are relatively untouched by the public. "There is a theory that if you open-source it, they will come," Nathan said. "That's not really true."

Rather than focusing on cocreation with their customers, SparkFun instead focuses on enabling people to copy, tinker, and improve products on their own. They heavily invest in tutorials and other material designed to help people understand how the products work so they can fix and improve things independently. "What gives me joy is when people take open-source layouts and then build their own circuit boards from our designs," Nathan said.

Obviously, opening up the design of their products is a necessary step if their goal is to empower the public. Nathan also firmly believes it makes them more money because it requires them to focus on how to provide maximum value. Rather than designing a new product and protecting it in order to extract as much money as possible from it, they release the keys necessary for others to build it themselves and then spend company time and resources on innovation and service. From a short-term perspective, SparkFun may lose a few dollars when others copy their products. But in the long run, it makes them a more nimble, innovative business. In other words, it makes them the kind of company they set out to be.

TEACHAIDS



TeachAIDS is a nonprofit that creates educational materials designed to teach people around the world about HIV and AIDS. Founded in 2005 in the U.S.

teachaids.org

Revenue model: sponsorships

Interview date: March 24, 2016

Interviewees: Piya Sorcar, the CEO, and Shuman Ghosemajumder, the chair

Profile written by Sarah Hinchliff Pearson

TeachAIDS is an unconventional media company with a conventional revenue model. Like most media companies, they are subsidized by advertising. Corporations pay to have their logos appear on the educational materials TeachAIDS distributes.

But unlike most media companies, TeachAIDS is a nonprofit organization with a purely social mission. TeachAIDS is dedicated to educating the global population about HIV and AIDS, particularly in parts of the world where education efforts have been historically unsuccessful. Their educational content is conveyed through interactive software, using methods based on the latest research about how people learn. TeachAIDS serves content in more than eighty countries around the world. In each instance, the content is translated to the local language and adjusted to conform to local norms and customs. All content is free and

made available under a Creative Commons license.

TeachAIDS is a labor of love for founder and CEO Piya Sorcar, who earns a salary of one dollar per year from the nonprofit. The project grew out of research she was doing while pursuing her doctorate at Stanford University. She was reading reports about India, noting it would be the next hot zone of people living with HIV. Despite international and national entities pouring in hundreds of millions of dollars on HIV-prevention efforts, the reports showed knowledge levels were still low. People were unaware of whether the virus could be transmitted through coughing and sneezing, for instance. Supported by an interdisciplinary team of experts at Stanford, Piya conducted similar studies, which corroborated

the previous research. They found that the primary cause of the limited understanding was that HIV, and issues relating to it, were often considered too taboo to discuss comprehensively. The other major problem was that most of the education on this topic was being taught through television advertising, billboards, and other mass-media campaigns, which meant people were only receiving bits and pieces of information.

In late 2005, Piya and her team used research-based design to create new educational materials and worked with local partners in India to help distribute them. As soon as the animated software was posted online, Piya's team started receiving requests from individuals and governments who were interested in bringing this model to more countries. "We realized fairly quickly that educating large populations about a topic that was considered taboo would be challenging. We began by identifying optimal local partners and worked toward creating an effective, culturally appropriate education," Piya said.

Very shortly after the initial release, Piya's team decided to spin the endeavor into an independent nonprofit out of Stanford University. They also decided to use Creative Commons licenses on the materials.

Given their educational mission, TeachAIDS had an obvious interest in seeing the materials as widely shared as possible. But they also needed to preserve the integrity of the medical information in the content. They chose the Attribution-NonCommercial-NoDerivs license (CC BY-NC-ND), which essentially gives the public the right to distribute only verbatim copies of the content, and for noncommercial purposes. "We wanted attribution for TeachAIDS, and we couldn't stand by derivatives without vetting them," the cofounder and chair Shuman Ghosemajumder said. "It was almost a no-brainer to go with a CC license because it was a plug-and-play solution to this exact problem. It has allowed us to scale our materials safely and quickly worldwide while preserving our content and protecting us at the same time."

Choosing a license that does not allow adaptation of the content was an outgrowth of the careful precision with which TeachAIDS crafts their content. The organization invests heavily in research and testing to determine the best method of conveying the information. "Creating high-quality content is what matters most to us," Piya said. "Research drives everything we do."

One important finding was that people accept the message best when it comes from familiar voices they trust and admire. To achieve this, TeachAIDS researches cultural icons that would best resonate with their target audiences and recruits them to donate their likenesses and voices for use in the animated software. The celebrities involved vary for each localized version of the materials.

Localization is probably the single-most important aspect of the way TeachAIDS creates its content. While each regional version builds from the same core scientific materials, they pour a lot of resources into customizing the content for a particular population. Because they use a CC license that does not allow the public to adapt the content, TeachAIDS retains careful control over the localization process. The content is translated into the local language, but there are also changes in substance and format to reflect cultural differences. This process results in minor changes, like choosing different idioms based on the local language, and significant changes, like creating gendered versions for places where people are more likely to accept information from someone of the same gender.

The localization process relies heavily on volunteers. Their volunteer base is deeply committed to the cause, and the organization has had better luck controlling the quality of the materials when they tap volunteers instead of using paid translators. For quality control, TeachAIDS has three separate volunteer teams translate the materials from English to the local language and customize the content based on local customs and norms. Those three versions are then analyzed and combined into a single master translation. TeachAIDS has ad-

ditional teams of volunteers then translate that version back into English to see how well it lines up with the original materials. They repeat this process until they reach a translated version that meets their standards. For the Tibetan version, they went through this cycle eleven times.

TeachAIDS employs full-time employees, contractors, and volunteers, all in different capacities and organizational configurations. They are careful to use people from diverse backgrounds to create the materials, including teachers, students, and doctors, as well as individuals experienced in working in the NGO space. This diversity and breadth of knowledge help ensure their materials resonate with people from all walks of life. Additionally, TeachAIDS works closely with film writers and directors to help keep the concepts entertaining and easy to understand. The inclusive, but highly controlled, creative process is undertaken entirely by people who are specifically brought on to help with a particular project, rather than ongoing staff. The final product they create is designed to require zero training for people to implement in practice. "In our research, we found we can't depend on people passing on the information correctly, even if they have the best of intentions," Piya said. "We need materials where you can push play and they will work."

Piya's team was able to produce all of these versions over several years with a head count that never exceeded eight full-time employees. The organization is able to reduce costs by relying heavily on volunteers and in-kind donations. Nevertheless, the nonprofit needed a sustainable revenue model to subsidize content creation and physical distribution of the materials. Charging even a low price was simply not an option. "Educators from various nonprofits around the world were just creating their own materials using whatever they could find for free online," Shuman said. "The only

way to persuade them to use our highly effective model was to make it completely free."

Like many content creators offering their work for free, they settled on advertising as a funding model. But they were extremely careful not to let the advertising compromise their credibility or undermine the heavy investment they put into creating quality content. Sponsors of the content have no ability to influence the substance of the content, and they cannot even create advertising content. Sponsors only get the right to have their logo appear before and after the educational content. All of the content remains branded as TeachAIDS.

TeachAIDS is careful not to seek funding to cover the costs of a specific project. Instead, sponsorships are structured as unrestricted donations to the nonprofit. This gives the nonprofit more stability, but even more importantly, it enables them to subsidize projects being localized for an area with no sponsors. "If we just created versions based on where we could get sponsorships, we would only have materials for wealthier countries," Shuman said.

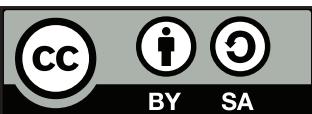
As of 2016, TeachAIDS has dozens of sponsors. "When we go into a new country, various companies hear about us and reach out to us," Piya said. "We don't have to do much to find or attract them." They believe the sponsorships are easy to sell because they offer so much value to sponsors. TeachAIDS sponsorships give corporations the chance to reach new eyeballs with their brand, but at a much lower cost than other advertising channels. The audience for TeachAIDS content also tends to skew young, which is often a desirable demographic for brands. Unlike traditional advertising, the content is not time-sensitive, so an investment in a sponsorship can benefit a brand for many years to come.

Importantly, the value to corporate sponsors goes beyond commercial considerations. As a nonprofit with a clearly articulated social mission, corporate sponsorships are donations to a cause. "This is something companies can be proud of internally," Shuman said. Some companies have even built public-

ity campaigns around the fact that they have sponsored these initiatives.

The core mission of TeachAIDS—ensuring global access to life-saving education—is at the root of everything the organization does. It underpins the work; it motivates the funders. The CC license on the materials they create furthers that mission, allowing them to safely and quickly scale their materials worldwide. “The Creative Commons license has been a game changer for TeachAIDS,” Piya said.

TRIBE OF NOISE



Tribe of Noise is a for-profit online music platform serving the film, TV, video, gaming, and in-store-media industries. Founded in 2008 in the Netherlands.

www.tribeofnoise.com

Revenue model: charging a transaction fee

Interview date: January 26, 2016

Interviewee: Hessel van Oorschot, cofounder

Profile written by Paul Stacey

In the early 2000s, Hessel van Oorschot was an entrepreneur running a business where he coached other midsize entrepreneurs how to create an online business. He also coauthored a number of workbooks for small- to medium-size enterprises to use to optimize their business for the Web. Through this early work, Hessel became familiar with the principles of open licensing, including the use of open-source software and Creative Commons.

In 2005, Hessel and Sandra Brandenburg launched a niche video-production initiative. Almost immediately, they ran into issues around finding and licensing music tracks. All they could find was standard, cold stock-music. They thought of looking up websites where you could license music directly from the mu-

sician without going through record labels or agents. But in 2005, the ability to directly license music from a rights holder was not readily available.

They hired two lawyers to investigate further, and while they uncovered five or six examples, Hessel found the business models lacking. The lawyers expressed interest in being their legal team should they decide to pursue this as an entrepreneurial opportunity. Hessel says, "When lawyers are interested in a venture like this, you might have something special." So after some more research, in early 2008, Hessel and Sandra decided to build a platform.

Building a platform posed a real chicken-and-egg problem. The platform had to build an online community of music-rights holders and, at the same time, provide the community with information and ideas about how the new economy works. Community willingness to try new music business models requires a trust relationship.

In July 2008, Tribe of Noise opened its virtual doors with a couple hundred musicians willing to use the CC BY-SA license (Attribution-Share-Alike) for a limited part of their repertoire. The two entrepreneurs wanted to take the pain away for media makers who wanted to license music and solve the problems the two had personally experienced finding this music.

As they were growing the community, Hessel got a phone call from a company that made in-store music playlists asking if they had enough music licensed with Creative Commons that they could use. Stores need quality, good-listening music but not necessarily hits, a bit like a radio show without the DJ. This opened a new opportunity for Tribe of Noise. They started their In-store Music Service, using music (licensed with CC BY-SA) uploaded by the Tribe of Noise community of musicians.¹

In most countries, artists, authors, and musicians join a collecting society that manages the licensing and helps collect the royalties. Copyright collecting societies in the European Union usually hold monopolies in their respective national markets. In addition, they require their members to transfer exclusive administration rights to them of all of their works. This complicates the picture for Tribe of Noise, who wants to represent artists, or at least a portion of their repertoire. Hessel and his legal team reached out to collecting societies, starting with those in the Netherlands. What would be the best legal way forward that would respect the wishes of composers and musicians who'd be interested in trying out new models like the In-store Music Service? Collecting societies at first were hesitant and said no, but Tribe of Noise persisted

arguing that they primarily work with unknown artists and provide them exposure in parts of the world where they don't get airtime normally and a source of revenue—and this convinced them that it was OK. However, Hessel says, "We are still fighting for a good cause every single day."

Instead of building a large sales force, Tribe of Noise partnered with big organizations who have lots of clients and can act as a kind of Tribe of Noise reseller. The largest telecom network in the Netherlands, for example, sells Tribe's In-store Music Service subscriptions to their business clients, which include fashion retailers and fitness centers. They have a similar deal with the leading trade association representing hotels and restaurants in the country. Hessel hopes to "copy and paste" this service into other countries where collecting societies understand what you can do with Creative Commons. Outside of the Netherlands, early adoptions have happened in Scandinavia, Belgium, and the U.S.

Tribe of Noise doesn't pay the musicians up front; they get paid when their music ends up in Tribe of Noise's in-store music channels. The musicians' share is 42.5 percent. It's not uncommon in a traditional model for the artist to get only 5 to 10 percent, so a share of over 40 percent is a significantly better deal. Here's how they give an example on their website:

A few of your songs [licensed with CC BY-SA], for example five in total, are selected for a bespoke in-store music channel broadcasting at a large retailer with 1,000 stores nationwide. In this case the overall playlist contains 350 songs so the musician's share is $5/350 = 1.43\%$. The license fee agreed with this retailer is US\$12 per month per play-out. So if 42.5% is shared with the Tribe musicians in this playlist and your share is 1.43%, you end up with $US\$12 * 1000 \text{ stores} * 0.425 * 0.0143 = US\73 per month .²

Tribe of Noise has another model that does not involve Creative Commons. In a survey with

members, most said they liked the exposure using Creative Commons gets them and the way it lets them reach out to others to share and remix. However, they had a bit of a mental struggle with Creative Commons licenses being perpetual. A lot of musicians have the mind-set that one day one of their songs may become an overnight hit. If that happened the CC BY-SA license would preclude them getting rich off the sale of that song.

Hessel's legal team took this feedback and created a second model and separate area of the platform called Tribe of Noise Pro. Songs uploaded to Tribe of Noise Pro aren't Creative Commons licensed; Tribe of Noise has instead created a "nonexclusive exploitation" contract, similar to a Creative Commons license but allowing musicians to opt out whenever they want. When you opt out, Tribe of Noise agrees to take your music off the Tribe of Noise platform within one to two months. This lets the musician reuse their song for a better deal.

Tribe of Noise Pro is primarily geared toward media makers who are looking for music. If they buy a license from this catalog, they don't have to state the name of the creator; they just license the song for a specific amount. This is a big plus for media makers. And musicians can pull their repertoire at any time. Hessel sees this as a more direct and clean deal.

Lots of Tribe of Noise musicians upload songs to both Tribe of Noise Pro and the community area of Tribe of Noise. There aren't that many artists who upload only to Tribe of Noise Pro, which has a smaller repertoire of music than the community area.

Hessel sees the two as complementary. Both are needed for the model to work. With a whole generation of musicians interested in the sharing economy, the community area of Tribe of Noise is where they can build trust, create exposure, and generate money. And after that, musicians may become more interested in exploring other models like Tribe of Noise Pro.

Every musician who joins Tribe of Noise gets their own home page and free unlimited Web space to upload as much of their own music

WITH A WHOLE GENERATION OF MUSICIANS INTERESTED IN THE SHARING ECONOMY, THE COMMUNITY AREA OF TRIBE OF NOISE IS WHERE THEY CAN BUILD TRUST, CREATE EXPOSURE, AND GENERATE MONEY.

as they like. Tribe of Noise is also a social network; fellow musicians and professionals can vote for, comment on, and like your music. Community managers interact with and support members, and music supervisors pick and choose from the uploaded songs for in-store play or to promote them to media producers. Members really like having people working for the platform who truly engage with them.

Another way Tribe of Noise creates community and interest is with contests, which are organized in partnership with Tribe of Noise clients. The client specifies what they want, and any member can submit a song. Contests usually involve prizes, exposure, and money. In addition to building member engagement, contests help members learn how to work with clients: listening to them, understanding what they want, and creating a song to meet that need.

Tribe of Noise now has twenty-seven thousand members from 192 countries, and many are exploring do-it-yourself models for generating revenue. Some came from music labels and publishers, having gone through the traditional way of music licensing and now seeing if this new model makes sense for them. Others are young musicians, who grew up with a DIY mentality and see little reason to sign with a third party or hand over some of the control. Still a small but growing group of Tribe members are pursuing a hybrid model by licensing some of their songs under CC BY-SA and

opting in others with collecting societies like ASCAP or BMI.

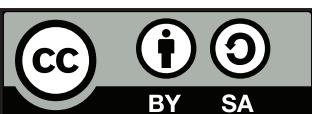
It's not uncommon for performance-rights organizations, record labels, or music publishers to sign contracts with musicians based on exclusivity. Such an arrangement prevents those musicians from uploading their music to Tribe of Noise. In the United States, you can have a collecting society handle only some of your tracks, whereas in many countries in Europe, a collecting society prefers to represent your entire repertoire (although the European Commission is making some changes). Tribe of Noise deals with this issue all the time and gives you a warning whenever you upload a song. If collecting societies are willing to be open and flexible and do the most they can for their members, then they can consider organizations like Tribe of Noise as a nice add-on, generating more exposure and revenue for the musicians they represent. So far, Tribe of Noise has been able to make all this work without litigation.

For Hessel the key to Tribe of Noise's success is trust. The fact that Creative Commons licenses work the same way all over the world and have been translated into all languages really helps build that trust. Tribe of Noise believes in creating a model where they work together with musicians. They can only do that if they have a live and kicking community, with people who think that the Tribe of Noise team has their best interests in mind. Creative Commons makes it possible to create a new business model for music, a model that's based on trust.

Web links

- 1 www.instoremusicservice.com
- 2 [www.tribeofnoise.com
/info_instoremusic.php](http://www.tribeofnoise.com/info_instoremusic.php)

WIKIMEDIA FOUNDATION



The Wikimedia Foundation is the nonprofit organization that hosts Wikipedia and its sister projects. Founded in 2003 in the U.S.

wikimediafoundation.org

Revenue model: donations

Interview date: December 18, 2015

Interviewees: Luis Villa, former Chief Officer of Community Engagement, and Stephen LaPorte, legal counsel

Profile written by Sarah Hinchliff Pearson

Nearly every person with an online presence knows Wikipedia.

In many ways, it is *the* preeminent open project: The online encyclopedia is created entirely by volunteers. Anyone in the world can edit the articles. All of the content is available for free to anyone online. All of the content is released under a Creative Commons license that enables people to reuse and adapt it for any purpose.

As of December 2016, there were more than forty-two million articles in the 295 language editions of the online encyclopedia, according to—what else?—the Wikipedia article about Wikipedia.

The Wikimedia Foundation is a U.S.-based nonprofit organization that owns the Wikipedia

domain name and hosts the site, along with many other related sites like Wikidata and Wikimedia Commons. The foundation employs about two hundred and eighty people, who all work to support the projects it hosts. But the true heart of Wikipedia and its sister projects is its community. The numbers of people in the community are variable, but about seventy-five thousand volunteers edit and improve Wikipedia articles every month. Volunteers are organized in a variety of ways across the globe, including formal Wikimedia chapters (mostly national), groups focused on a particular theme, user groups, and many thousands who are not connected to a particular organization.

As Wikimedia legal counsel Stephen LaPorte told us, "There is a common saying that Wiki-

pedia works in practice but not in theory.” While it undoubtedly has its challenges and flaws, Wikipedia and its sister projects are a striking testament to the power of human collaboration.

Because of its extraordinary breadth and scope, it does feel a bit like a unicorn. Indeed, there is nothing else like Wikipedia. Still, much of what makes the projects successful—community, transparency, a strong mission, trust—are consistent with what it takes to be successfully **Made with Creative Commons** more generally. With Wikipedia, everything just happens at an unprecedented scale.

The story of Wikipedia has been told many times. For our purposes, it is enough to know the experiment started in 2001 at a small scale, inspired by the crazy notion that perhaps a truly open, collaborative project could create something meaningful. At this point, Wikipedia is so ubiquitous and ingrained in our digital lives that the fact of its existence seems less remarkable. But outside of software, Wikipedia is perhaps the single most stunning example of successful community cocreation. Every day, seven thousand new articles are created on Wikipedia, and nearly fifteen thousand edits are made every hour.

The nature of the content the community creates is ideal for asynchronous cocreation. “An encyclopedia is something where incremental community improvement really works,” Luis Villa, former Chief Officer of Community Engagement, told us. The rules and processes that govern cocreation on Wikipedia and its sister projects are all community-driven and vary by language edition. There are entire books written on the intricacies of their systems, but generally speaking, there are very few exceptions to the rule that anyone can edit any article, even without an account on their system. The extensive peer-review process includes elaborate systems to resolve disputes, methods for managing particularly controver-

sial subject areas, talk pages explaining decisions, and much, much more.

The Wikimedia Foundation’s decision to leave governance of the projects to the community is very deliberate. “We look at the things that the community can do well, and we want to let them do those things,” Stephen told us. Instead, the foundation focuses its time and resources on what the community cannot do as effectively, like the software engineering that supports the technical infrastructure of the sites. In 2015-16, about half of the foundation’s budget went to direct support for the Wikimedia sites.

Some of that is directed at servers and general IT support, but the foundation also invests a significant amount on architecture designed to help the site function as effectively as possible. “There is a constantly evolving system to keep the balance in place to avoid Wikipedia becoming the world’s biggest graffiti wall,” Luis said. Depending on how you measure it, somewhere between 90 to 98 percent of edits to Wikipedia are positive. Some portion of that success is attributable to the tools Wikimedia has in place to try to incentivize good actors. “The secret to having any healthy community is bringing back the right people,” Luis said. “Vandals tend to get bored and go away. That is partially our model working, and partially just human nature.” Most of the time, people want to do the right thing.

Wikipedia not only relies on good behavior within its community and on its sites, but also by everyone else once the content leaves Wikipedia. All of the text of Wikipedia is available under an Attribution-ShareAlike license (CC BY-SA), which means it can be used for any purpose and modified so long as credit is given and anything new is shared back with the public under the same license. In theory, that means anyone can copy the content and start a new Wikipedia. But as Stephen explained, “Being open has only made Wikipedia bigger and stronger. The desire to protect is not always what is best for everyone.”

Of course, the primary reason no one has successfully co-opted Wikipedia is that copycat

efforts do not have the Wikipedia community to sustain what they do. Wikipedia is not simply a source of up-to-the-minute content on every given topic—it is also a global patchwork of humans working together in a million different ways, in a million different capacities, for a million different reasons. While many have tried to guess what makes Wikipedia work as well it does, the fact is there is no single explanation. “In a movement as large as ours, there is an incredible diversity of motivations,” Stephen said. For example, there is one editor of the English Wikipedia edition who has corrected a single grammatical error in articles more than forty-eight thousand times.¹

Only a fraction of Wikipedia users are also editors. But editing is not the only way to contribute to Wikipedia. “Some donate text, some donate images, some donate financially,” Stephen told us. “They are all contributors.”

But the vast majority of us who use Wikipedia are not contributors; we are passive readers. The Wikimedia Foundation survives primarily on individual donations, with about \$15 as the average. Because Wikipedia is one of the ten most popular websites in terms of total page views, donations from a small portion of that audience can translate into a lot of money. In the 2015-16 fiscal year, they received more than \$77 million from more than five million donors.

The foundation has a fund-raising team that works year-round to raise money, but the bulk of their revenue comes in during the December campaign in Australia, Canada, Ireland, New Zealand, the United Kingdom, and the United States. They engage in extensive user testing and research to maximize the reach of their fund-raising campaigns. Their basic fund-raising message is simple: We provide our readers and the world immense value, so give back. Every little bit helps. With enough eyeballs, they are right.

freely share in the sum of all knowledge. They work to realize this vision by empowering people around the globe to create educational content made freely available under an open license or in the public domain. Stephen and Luis said the mission, which is rooted in the same philosophy behind Creative Commons, drives everything the foundation does.

The philosophy behind the endeavor also enables the foundation to be financially sustainable. It instills trust in their readership, which is critical for a revenue strategy that relies on reader donations. It also instills trust in their community.

Any given edit on Wikipedia could be motivated by nearly an infinite number of reasons. But the social mission of the project is what binds the global community together. “Wikipedia is an example of how a mission can motivate an entire movement,” Stephen told us.

Of course, what results from that movement is one of the Internet’s great public resources. “The Internet has a lot of businesses and stores, but it is missing the digital equivalent of parks and open public spaces,” Stephen said. “Wikipedia has found a way to be that open public space.”

Web link

- 1 gimletmedia.com/episode/14-the-art-of-making-and-fixing-mistakes/

The vision of the Wikimedia Foundation is a world in which every single human being can

BIBLIOGRAPHY

- Alperovitz, Gar. *What Then Must We Do? Straight Talk about the Next American Revolution; Democratizing Wealth and Building a Community-Sustaining Economy from the Ground Up*. White River Junction, VT: Chelsea Green, 2013.
- Anderson, Chris. *Free: How Today's Smartest Businesses Profit by Giving Something for Nothing*, reprint with new preface. New York: Hyperion, 2010.
- . *Makers: The New Industrial Revolution*. New York: Signal, 2012.
- Ariely, Dan. *Predictably Irrational: The Hidden Forces That Shape Our Decisions*. Rev. ed. New York: Harper Perennial, 2010.
- Bacon, Jono. *The Art of Community*. 2nd ed. Sebastopol, CA: O'Reilly Media, 2012.
- Benkler, Yochai. *The Wealth of Networks: How Social Production Transforms Markets and Freedom*. New Haven: Yale University Press, 2006. www.benkler.org/Benkler_Wealth_Of_Networks.pdf (licensed under CC BY-NC-SA).
- Benyayer, Louis-David, ed. *Open Models: Business Models of the Open Economy*. Cachan, France: Without Model, 2016. www.slideshare.net/WithoutModel/open-models-book-64463892 (licensed under CC BY-SA).
- Bollier, David. *Commoning as a Transformative Social Paradigm*. Paper commissioned by the Next Systems Project. Washington, DC: Democracy Collaborative, 2016. thenext-system.org/commoning-as-a-transformative-social-paradigm/.
- . *Think Like a Commoner: A Short Introduction to the Life of the Commons*. Gabriola Island, BC: New Society, 2014.
- Bollier, David, and Pat Conaty. *Democratic Money and Capital for the Commons: Strategies for Transforming Neoliberal Finance through Commons-Based Alternatives*. A report on a Commons Strategies Group Workshop in cooperation with the Heinrich Böll Foundation, Berlin, Germany, 2015. bollier.org/democratic-money-and-capital-commons-report-pdf. For more information, see bollier.org/blog/democratic-money-and-capital-commons.
- Bollier, David, and Silke Helfrich, eds. *The Wealth of the Commons: A World Beyond Market and State*. Amherst, MA: Levellers Press, 2012.
- Botsman, Rachel, and Roo Rogers. *What's Mine Is Yours: The Rise of Collaborative Consumption*. New York: Harper Business, 2010.
- Boyle, James. *The Public Domain: Enclosing the Commons of the Mind*. New Haven: Yale University Press, 2008. www.thepublicdomain.org/download/ (licensed under CC BY-NC-SA).
- Capra, Fritjof, and Ugo Mattei. *The Ecology of Law: Toward a Legal System in Tune with Nature and Community*. Oakland, CA: Berrett-Koehler, 2015.
- Chesbrough, Henry. *Open Business Models: How to Thrive in the New Innovation Landscape*. Boston: Harvard Business School Press, 2006.
- . *Open Innovation: The New Imperative for Creating and Profiting from Technology*. Boston: Harvard Business Review Press, 2006.
- City of Bologna. *Regulation on Collaboration between Citizens and the City for the Care and Regeneration of Urban Commons*. Translated by LabGov (LABoratory for the

- GOvernance of Commons). Bologna, Italy: City of Bologna, 2014). www.labgov.it/wp-content/uploads/sites/9/Bologna-Regulation-on-collaboration-between-citizens-and-the-city-for-the-cure-and-regeneration-of-urban-commons1.pdf.
- Cole, Daniel H. "Learning from Lin: Lessons and Cautions from the Natural Commons for the Knowledge Commons." Chap. 2 in Frischmann, Madison, and Strandburg, *Governing Knowledge Commons*.
- Creative Commons. *2015 State of the Commons*. Mountain View, CA: Creative Commons, 2015. stateof.creativecommons.org/2015/.
- Doctorow, Cory. *Information Doesn't Want to Be Free: Laws for the Internet Age*. San Francisco: McSweeney's, 2014.
- Eckhardt, Giana, and Fleura Bardhi. "The Sharing Economy Isn't about Sharing at All." *Harvard Business Review*, January 28, 2015. hbr.org/2015/01/the-sharing-economy-isnt-about-sharing-at-all.
- Elliott, Patricia W., and Daryl H. Hepting, eds. (2015). *Free Knowledge: Confronting the Commodification of Human Discovery*. Regina, SK: University of Regina Press, 2015. uofrpress.ca/publications/Free-Knowledge (licensed under CC BY-NC-ND).
- Eyal, Nir. *Hooked: How to Build Habit-Forming Products*. With Ryan Hoover. New York: Portfolio, 2014.
- Farley, Joshua, and Ida Kubiszewski. "The Economics of Information in a Post-Carbon Economy." Chap. 11 in Elliott and Hepting, *Free Knowledge*.
- Foster, William Landes, Peter Kim, and Barbara Christiansen. "Ten Nonprofit Funding Models." *Stanford Social Innovation Review*, Spring 2009. ssir.org/articles/entry/ten_nonprofit_funding_models.
- Frischmann, Brett M. *Infrastructure: The Social Value of Shared Resources*. New York: Oxford University Press, 2012.
- Frischmann, Brett M., Michael J. Madison, and Katherine J. Strandburg, eds. *Governing Knowledge Commons*. New York: Oxford University Press, 2014.
- Frischmann, Brett M., Michael J. Madison, and Katherine J. Strandburg. "Governing Knowledge Commons." Chap. 1 in Frischmann, Madison, and Strandburg, *Governing Knowledge Commons*.
- Gansky, Lisa. *The Mesh: Why the Future of Business Is Sharing*. Reprint with new epilogue. New York: Portfolio, 2012.
- Grant, Adam. *Give and Take: Why Helping Others Drives Our Success*. New York: Viking, 2013.
- Haiven, Max. *Crises of Imagination, Crises of Power: Capitalism, Creativity and the Commons*. New York: Zed Books, 2014.
- Harris, Malcom, ed. *Share or Die: Voices of the Get Lost Generation in the Age of Crisis*. With Neal Gorenflo. Gabriola Island, BC: New Society, 2012.
- Hermida, Alfred. *Tell Everyone: Why We Share and Why It Matters*. Toronto: Doubleday Canada, 2014.
- Hyde, Lewis. *Common as Air: Revolution, Art, and Ownership*. New York: Farrar, Straus and Giroux, 2010.
- . *The Gift: Creativity and the Artist in the Modern World*. 2nd Vintage Books edition. New York: Vintage Books, 2007.
- Kelley, Tom, and David Kelley. *Creative Confidence: Unleashing the Potential within Us All*. New York: Crown, 2013.
- Kelly, Marjorie. *Owning Our Future: The Emerging Ownership Revolution; Journeys to a Generative Economy*. San Francisco: Berrett-Koehler, 2012.
- Kleon, Austin. *Show Your Work: 10 Ways to Share Your Creativity and Get Discovered*. New York: Workman, 2014.
- . *Steal Like an Artist: 10 Things Nobody Told You about Being Creative*. New York: Workman, 2012.
- Kramer, Bryan. *Shareology: How Sharing Is Powering the Human Economy*. New York: Morgan James, 2016.
- Lee, David. "Inside Medium: An Attempt to Bring Civility to the Internet." BBC News, March 3, 2016. www.bbc.com/news/technology-35709680
- Lessig, Lawrence. *Remix: Making Art and Com-*

- merce Thrive in the Hybrid Economy.* New York: Penguin Press, 2008.
- Menzies, Heather. *Reclaiming the Commons for the Common Good: A Memoir and Manifesto.* Gabriola Island, BC: New Society, 2014.
- Mason, Paul. *Postcapitalism: A Guide to Our Future.* New York: Farrar, Straus and Giroux, 2015.
- New York Times Customer Insight Group. *The Psychology of Sharing: Why Do People Share Online?* New York: New York Times Customer Insight Group, 2011. www.iab.net/media/file/POSWhitePaper.pdf.
- Osterwalder, Alex, and Yves Pigneur. *Business Model Generation.* Hoboken, NJ: John Wiley and Sons, 2010. A preview of the book is available at strategyzer.com/books/business-model-generation.
- Osterwalder, Alex, Yves Pigneur, Greg Bernarda, and Adam Smith. *Value Proposition Design.* Hoboken, NJ: John Wiley and Sons, 2014. A preview of the book is available at strategyzer.com/books/value-proposition-design.
- Palmer, Amanda. *The Art of Asking: Or How I Learned to Stop Worrying and Let People Help.* New York: Grand Central, 2014.
- Pekel, Joris. *Democratising the Rijksmuseum: Why Did the Rijksmuseum Make Available Their Highest Quality Material without Restrictions, and What Are the Results?* The Hague, Netherlands: Europeana Foundation, 2014. pro.europeana.eu/publication/democratising-the-rijksmuseum (licensed under CC BY-SA).
- Ramos, José Maria, ed. *The City as Commons: A Policy Reader.* Melbourne, Australia: Commons Transition Coalition, 2016. www.academia.edu/27143172/The_City_as_Commons_a_Policy_Reader (licensed under CC BY-NC-ND).
- Raymond, Eric S. *The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary.* Rev. ed. Sebastopol, CA: O'Reilly Media, 2001. See esp. "The Magic Cauldron." www.catb.org/esr/writings/cathedral-bazaar/.
- Ries, Eric. *The Lean Startup: How Today's Entrepreneurs Use Continuous Innovation to Create Radically Successful Businesses.* New York: Crown Business, 2011.
- Rifkin, Jeremy. *The Zero Marginal Cost Society: The Internet of Things, the Collaborative Commons, and the Eclipse of Capitalism.* New York: Palgrave Macmillan, 2014.
- Rowe, Jonathan. *Our Common Wealth.* San Francisco: Berrett-Koehler, 2013.
- Rushkoff, Douglas. *Throwing Rocks at the Google Bus: How Growth Became the Enemy of Prosperity.* New York: Portfolio, 2016.
- Sandel, Michael J. *What Money Can't Buy: The Moral Limits of Markets.* New York: Farrar, Straus and Giroux, 2012.
- Shirky, Clay. *Cognitive Surplus: How Technology Makes Consumers into Collaborators.* London, England: Penguin Books, 2010.
- Slee, Tom. *What's Yours Is Mine: Against the Sharing Economy.* New York: OR Books, 2015.
- Stephany, Alex. *The Business of Sharing: Making in the New Sharing Economy.* New York: Palgrave Macmillan, 2015.
- Stepper, John. *Working Out Loud: For a Better Career and Life.* New York: Ikigai Press, 2015.
- Sull, Donald, and Kathleen M. Eisenhardt. *Simple Rules: How to Thrive in a Complex World.* Boston: Houghton Mifflin Harcourt, 2015.
- Sundararajan, Arun. *The Sharing Economy: The End of Employment and the Rise of Crowd-Based Capitalism.* Cambridge, MA: MIT Press, 2016.
- Surowiecki, James. *The Wisdom of Crowds.* New York: Anchor Books, 2005.
- Tapscott, Don, and Alex Tapscott. *Blockchain Revolution: How the Technology Behind Bitcoin Is Changing Money, Business, and the World.* Toronto: Portfolio, 2016.
- Tharp, Twyla. *The Creative Habit: Learn It and Use It for Life.* With Mark Reiter. New York: Simon and Schuster, 2006.
- Tkacz, Nathaniel. *Wikipedia and the Politics of Openness.* Chicago: University of Chicago Press, 2015.
- Van Abel, Bass, Lucas Evers, Roel Klaassen, and Peter Troxler, eds. *Open Design Now:*

Why Design Cannot Remain Exclusive. Amsterdam: BIS Publishers, with Creative Commons Netherlands; Premseala, the Netherlands Institute for Design and Fashion; and the Waag Society, 2011. opendesignnow.org (licensed under CC BY-NC-SA).

Van den Hoff, Ronald. *Mastering the Global Transition on Our Way to Society 3.0*. Utrecht, the Netherlands: Society 3.0 Foundation, 2014. society30.com/get-the-book/ (licensed under CC BY-NC-ND).

Von Hippel, Eric. *Democratizing Innovation*. London: MIT Press, 2005. [web.mit.edu/evhippel/www/democ1.htm](http://evhippel/www/democ1.htm) (licensed under CC BY-NC-ND).

Whitehurst, Jim. *The Open Organization: Igniting Passion and Performance*. Boston: Harvard Business Review Press, 2015.

ACKNOWLEDGMENTS

We extend special thanks to Creative Commons CEO Ryan Merkley, the Creative Commons Board, and all of our Creative Commons colleagues for enthusiastically supporting our work. Special gratitude to the William and Flora Hewlett Foundation for the initial seed funding that got us started on this project.

Huge appreciation to all the **Made with Creative Commons** interviewees for sharing their stories with us. You make the commons come alive. Thanks for the inspiration.

We interviewed more than the twenty-four organizations profiled in this book. We extend special thanks to Gooru, OERu, Sage Bionetworks, and Medium for sharing their stories with us. While not featured as case studies in this book, you all are equally interesting, and we encourage our readers to visit your sites and explore your work.

This book was made possible by the generous support of 1,687 Kickstarter backers listed below. We especially acknowledge our many Kickstarter co-editors who read early drafts of our work and provided invaluable feedback. Heartfelt thanks to all of you.

Co-editor Kickstarter backers (alphabetically by first name): Abraham Taherivand, Alan Graham, Alfredo Louro, Anatoly Volynets, Aurora Thornton, Austin Tolentino, Ben Sheridan, Benedikt Foit, Benjamin Costantini, Bernd Nurnberger, Bernhard Seefeld, Bethanye Blount, Bradford Benn, Bryan Mock, Carmen Garcia Wiedenhoeft, Carolyn Hinchliff, Casey Milford, Cat Cooper, Chip McIntosh, Chris Thorne, Chris Weber, Chutika Udomsinn, Claire Wardle, Claudia Cristiani, Cody Allard, Colleen Cressman, Craig Thomler, Creative Commons Uruguay, Curt McNamara, Dan Parson, Daniel Domin-

guez, Daniel Morado, Darius Irvin, Dave Taillefer, David Lewis, David Mikula, David Varnes, David Wiley, Deborah Nas, Diderik van Wingerden, Dirk Kiefer, Dom Lane, Domi Enders, Douglas Van Houweling, Dylan Field, Einar Joergensen, Elad Wieder, Elie Calhoun, Erika Reid, Evtim Papushev, Fauxton Software, Felix Maximiliano Obes, Ferdies Food Lab, Gatien de Broucker, Gaurav Kapil, Gavin Romig-Koch, George Baier IV, George De Bruin, Gianpaolo Rando, Glenn Otis Brown, Govindarajan Umapanthan, Graham Bird, Graham Freeman, Hamish MacEwan, Harry Kaczka, Humble Daisy, Ian Capstick, Iris Brest, James Cloos, Jamie Stevens, Jamil Khatib, Jane Finette, Jason Blasso, Jason E. Barkeloo, Jay M Williams, Jean-Philippe Turcotte, Jeanette Frey, Jeff De Cagna, Jérôme Mizeret, Jessica Dickinson Goodman, Jessy Kate Schingler, Jim O'Flaherty, Jim Pellegrini, Jiří Marek, Jo Allum, Joachim von Goetz, Johan Adda, John Benfield, John Bevan, Jonas Öberg, Jonathan Lin, JP Rangaswami, Juan Carlos Belair, Justin Christian, Justin Szlasa, Kate Chapman, Kate Stewart, Kellie Higginbottom, Kendra Byrne, Kevin Coates, Kristina Popova, Kristoffer Steen, Kyle Simpson, Laurie Racine, Leonardo Bueno Postacchini, Leticia Britos Cavagnaro, Livia Leskovec, Louis-David Benyayer, Maik Schmalstich, Mairi Thomson, Marcia Hofmann, Maria Liberman, Marino Hernandez, Mario R. Hemsley, MD, Mark Cohen, Mark Mullen, Mary Ellen Davis, Mathias Bavay, Matt Black, Matt Hall, Max van Balgooy, Médéric Droz-dit-Busset, Melissa Aho, Menachem Goldstein, Michael Harries, Michael Lewis, Michael Weiss, Miha Batic, Mike Stop Continues, Mike Stringer, Mustafa K Calik, MD, Neal Stimler, Niall McDonagh, Niall Twohig, Nicholas Norfolk, Nick Coghlan, Nicole Hick-

man, Nikki Thompson, Norrie Mailer, Omar Kaminski, OpenBuilds, Papp István Péter, Pat Sticks, Patricia Brennan, Paul and Iris Brest, Paul Elosegui, Penny Pearson, Peter Mengelers, Playground Inc., Pomax, Rafaela Kunz, Ravid Jhangiani, Rayna Stamboliyska, Rob Berkley, Rob Bertholf, Robert Jones, Robert Thompson, Ronald van den Hoff, Rusi Popov, Ryan Merkley, S. Searle, Salomon Riedo, Samuel A. Rebelsky, Samuel Tait, Sarah McGovern, Scott Gillespie, Seb Schmoller, Sharon Clapp, Sheona Thomson, Siena Oristaglio, Simon Law, Solomon Simon, Stefano Guidotti, Subhendu Ghosh, Susan Chun, Suzie Wiley, Sylvain Carle, Theresa Bernardo, Thomas Hartman, Thomas Kent, Timothée Planté, Timothy Hinchliff, Traci Long DeForge, Trevor Hogue, Tumuult, Vickie Goode, Vikas Shah, Virginia Kopelman, Wayne Mackintosh, William Peter Nash, Winie Evers, Wolfgang Renninger, Xavier Antoviaque, Yancy Strickler

All other Kickstarter backers (alphabetically by first name): A. Lee, Aaron C. Rathbun, Aaron Stubbs, Aaron Suggs, Abdul Razak Manaf, Abraham Taherivand, Adam Croom, Adam Finer, Adam Hansen, Adam Morris, Adam Procter, Adam Quirk, Adam Rory Porter, Adam Simmons, Adam Tinworth, Adam Zimmerman, Adrian Ho, Adrian Smith, Adriane Ruzak, Adriano Loconte, Al Sweigart, Alain Imbaud, Alan Graham, Alan M. Ford, Alan Swithenbank, Alan Vonlanthen, Albert O'Connor, Alec Foster, Alejandro Suarez Cebrian, Aleks Degtyarev, Alex Blood, Alex C. Ion, Alex Ross Shaw, Alexander Bartl, Alexander Brown, Alexander Brunner, Alexander Eliesen, Alexander Hawson, Alexander Klar, Alexander Neumann, Alexander Plaum, Alexander Wendland, Alexandre Rafalovitch, Alexey Volkow, Alexi Wheeler, Alexis Sevault, Alfredo Louro, Ali Sternburg, Alicia Gibb & Lunchbox Electronics, Alison Link, Alison Pentecost, Alistair Boettiger, Alistair Walder, Alix Bernier, Allan Callaghan, Allen Ridell, Allison Breland Crotwell, Allison Jane Smith, Álvaro Justen, Amanda Palmer, Amanda Wetherhold, Amit Bagree, Amit Tikare, Amos Blanton, Amy Sept, Anatoly Volynets, Anders

Ericsson, Andi Popp, André Bose Do Amaral, Andre Dickson, André Koot, André Ricardo, Andre van Rooyen, Andre Wallace, Andrea Bagnacani, Andrea Pepe, Andrea Pigato, Andreas Jagelund, Andres Gomez Casanova, Andrew A. Farke, Andrew Berhow, Andrew Hearse, Andrew Matangi, Andrew R. McHugh, Andrew Tam, Andrew Turvey, Andrew Walsh, Andrew Wilson, Andrey Novoseltsev, Andy McGhee, Andy Reeve, Andy Woods, Angela Brett, Angeliki Kapoglou, Angus Keenan, Anne-Marie Scott, Antero Garcia, Antoine Authier, Antoine Michard, Anton Kurkin, Anton Porsche, Antònia Folguera, António Ornelas, Antonis Triantafylakis, aois21 publishing, April Johnson, Aria F. Chernik, Ariane Allan, Ariel Katz, Arithmomaniac, Arnaud Tessier, Arnim Sommer, Ashima Bawa, Ashley Elsdon, Athanassios Diacakis, Aurora Thornton, Aurore Chavet Henry, Austin Hartzheim, Austin Tolentino, Avner Shanan, Axel Pettersson, Axel Stieglbauer, Ay Okpokam, Barb Bartkowiak, Barbara Lindsey, Barry Dayton, Bastian Hougaard, Ben Chad, Ben Doherty, Ben Hansen, Ben Nuttall, Ben Rosenthal, Ben Sheridan, Benedikt Foit, Benita Tsao, Benjamin Costantini, Benjamin Daemon, Benjamin Keele, Benjamin Pflanz, Berglind Ósk Bergsdóttir, Bernardo Miguel Antunes, Bernd Nurnberger, Bernhard Seefeld, Beth Gis, Beth Tillinghast, Bethanye Blount, Bill Bonwitt, Bill Browne, Bill Keaggy, Bill Maiden, Bill Rafferty, Bill Scanlon, Bill Shields, Bill Slankard, BJ Becker, Bjorn Freeman-Benson, Bjørn Otto Wallen, BK Bitner, Bo Ilsøe Hansen, Bo Sprotte Kofod, Bob Doran, Bob Recny, Bob Stuart, Bonnie Chiu, Boris Mindzak, Boriss Lariushin, Borjan Tchakaloff, Brad Kik, Braden Hassett, Bradford Benn, Bradley Keyes, Bradley L'Herrou, Brady Forrest, Brandon McGaha, Branka Tokic, Brant Anderson, Brenda Sullivan, Brendan O'Brien, Brendan Schlagel, Brett Abbott, Brett Gaylor, Brian Dysart, Brian Lampl, Brian Lipscomb, Brian S. Weis, Brian Schrader, Brian Walsh, Brian Walsh, Brooke Dukes, Brooke Schreier Ganz, Bruce Lerner, Bruce Wilson, Bruno Boutot, Bruno Girin, Bryan Mock, Bryant Durrell, Bryce Barbato, Buzz Technology Limited, Byung-Geun Jeon, C. Glen Williams, C. L. Couch,

Cable Green, Callum Gare, Cameron Callahan, Cameron Colby Thomson, Cameron Mulder, Camille Bissuel / Nylnook, Candace Robertson, Carl Morris, Carl Perry, Carl Rigney, Carles Mateu, Carlos Correa Loyola, Carlos Solis, Carmen Garcia Wiedenhoeft, Carol Long, Carol marquardsen, Caroline Calomme, Caroline Mailiou, Carolyn Hinchliff, Carolyn Rude, Carrie Cousins, Carrie Watkins, Casey Hunt, Casey Milford, Casey Powell Shorthouse, Cat Cooper, Cecilie Maria, Cedric Howe, Cefn Hoile, @ShrimpingIt, Celia Muller, Ces Keller, Chad Anderson, Charles Butler, Charles Carstensen, Charles Chi Thoi Le, Charles Kobbe, Charles S. Tritt, Charles Stanhope, Charlotte Ong-Wisenier, Chealsye Bowley, Chelle Destefano, Chempang Chou, Cheryl Corte, Cheryl Todd, Chip Dickerson, Chip McIntosh, Chris Bannister, Chris Betcher, Chris Coleman, Chris Conway, Chris Foote (Spike), Chris Hurst, Chris Mitchell, Chris Muscat Azzopardi, Chris Niewiarowski, Chris Opperwall, Chris Stieha, Chris Thorne, Chris Weber, Chris Woolfrey, Chris Zabriskie, Christi Reid, Christian Holzberger, Christian Schubert, Christian Sheehy, Christian Thibault, Christian Villum, Christian Wachter, Christina Bennett, Christine Henry, Christine Rico, Christopher Burrows, Christopher Chan, Christopher Clay, Christopher Harris, Christopher Opiah, Christopher Swenson, Christos Keramitis, Chuck Roslof, Chutika Udomsinn, Claire Wardle, Clare Forrest, Claudia Cristiani, Claudio Gallo, Claudio Ruiz, Clayton Dewey, Clement Delort, Cliff Church, Clint Lalonde, Clint O'Connor, Cody Allard, Cody Taylor, Colin Ayer, Colin Campbell, Colin Dean, Colin Mutchler, Colleen Cressman, Comfy Nomad, Connie Roberts, Connor Bär, Connor Merkley, Constantin Graf, Corbett Messa, Cory Chapman, Cosmic Wombat Games, Craig Engler, Craig Heath, Craig Maloney, Craig Thomler, Creative Commons Uruguay, Crina Kienle, Cristiano Gozzini, Curt McNamara, D C Petty, D. Moonfire, D. Rohhyn, D. Schulz, Dacian Herbei, Dagmar M. Meyer, Dan Mcalister, Dan Mohr, Dan Parson, Dana Freeman, Dana Ospina, Dani Leviss, Daniel Bustamante, Daniel Demmel, Daniel Dominguez, Daniel Dultz, Daniel Gallant, Dan-

iel Kossmann, Daniel Kruse, Daniel Morado, Daniel Morgan, Daniel Pimley, Daniel Sabo, Daniel Sobey, Daniel Stein, Daniel Wildt, Daniele Prati, Danielle Moss, Danny Mendoza, Dario Taraborelli, Darius Irvin, Darius Whelan, Darla Anderson, Dasha Brezinova, Dave Ainscough, Dave Bull, Dave Crosby, Dave Eagle, Dave Moskovitz, Dave Neeteson, Dave Taillefer, Dave Witzel, David Bailey, David Cheung, David Eriksson, David Gallagher, David H. Bronke, David Hartley, David Hellam, David Hood, David Hunter, David Jlaietta, David Lewis, David Mason, David Mcconville, David Mikula, David Nelson, David Orban, David Parry, David Spira, David T. Kindler, David Varnes, David Wiley, David Wormley, Deborah Nas, Denis Jean, dennis straub, Dennis Whittle, Denver Gingerich, Derek Slater, Devon Cooke, Diana Pasek-Atkinson, Diane Johnston Graves, Diane K. Kovacs, Diane Trout, Diderik van Wingerden, Diego Cuevas, Diego De La Cruz, Dimitrie Grigorescu, Dina Marie Rodriguez, Dinah Fabela, Dirk Haun, Dirk Kiefer, Dirk Loop, DJ Fusion - FuseBox Radio Broadcast, Dom jurkewitz, Dom Lane, Domi Enders, Domingo Gallardo, Dominic de Haas, Dominique Karadjian, Dongpo Deng, Donovan Knight, Door de Flines, Doug Fitzpatrick, Doug Hoover, Douglas Craver, Douglas Van Camp, Douglas Van Houweling, Dr. Braddlee, Drew Spencer, Duncan Sample, Durand D'souza, Dylan Field, E C Humphries, Eamon Caddigan, Earleen Smith, Eden Sarid, Eden Spodek, Eduardo Belinchon, Eduardo Castro, Edwin Vandam, Einar Joergensen, Ejnar Brendsdal, Elad Wieder, Elar Haljas, Elena Valhalla, Eli Doran, Elias Bouchi, Elie Calhoun, Elizabeth Holloway, Ellen Buecher, Ellen Kaye-Cheveldayoff, Elli Verhulst, Elroy Fernandes, Emery Hurst Mikel, Emily Catedral, Enrique Mandujano R., Eric Astor, Eric Axelrod, Eric Celeste, Eric Finkenbiner, Eric Hellman, Eric Steuer, Erica Fletcher, Erik Hedman, Erik Lindholm Bundgaard, Erika Reid, Erin Hawley, Erin McKean of Wordnik, Ernest Risner, Erwan Bousse, Erwin Bell, Ethan Celery, Étienne Gilli, Eugeen Sablin, Evan Tangman, Evonne Okafor, Evtim Papushev, Fabien Cambi, Fabio Natali, Fauxton Software, Felix Deierlein, Felix Gebau-

er, Felix Maximiliano Obes, Felix Schmidt, Felix Zephyr Hsiao, Ferdies Food Lab, Fernand Deschambault, Filipe Rodrigues, Filippo Toso, Fiona MacAlister, fiona.mac.uk, Floor Scheffer, Florent Darrault, Florian Hähnel, Florian Schneider, Floyd Wilde, Foxtrot Games, Francis Clarke, Francisco Rivas-Portillo, Francois Dechery, Francois Grey, Francois Gros, Francois Pelletier, Fred Benenson, Frédéric Abella, Frédéric Schütz, Fredrik Ekelund, Fumi Yamazaki, Gabor Sooki-Toth, Gabriel Staples, Gabriel Véjar Valenzuela, Gal Buki, Gareth Jordan, Garrett Heath, Gary Anson, Gary Forster, Gatién de Broucker, Gaurav Kapil, Gauthier de Valensart, Gavin Gray, Gavin Romig-Koch, Geoff Wood, Geoffrey Lehr, George Baier IV, George De Bruin, George Lawie, George Strakhov, Gerard Gorman, Geronimo de la Lama, Gianpaolo Rando, Gil Stendig, Gino Cingolani Trucco, Giovanna Sala, Glen Moffat, Glenn D. Jones, Glenn Otis Brown, Global Lives Project, Gorm Lai, Govindarajan Umakanthan, Graham Bird, Graham Freeman, Graham Heath, Graham Jones, Graham Smith-Gordon, Graham Vowles, Greg Brodsky, Greg Malone, Grégoire Detrez, Gregory Chevalley, Gregory Flynn, Grit Matthias, Gui Louback, Guillaume Rischard, Gustavo Vaz de Carvalho Gonçalves, Gustin Johnson, Gwen Franck, Gwilym Lucas, Haggen So, Håkon T Sønderland, Hamid Larbi, Hamish MacEwan, Hannes Leo, Hans Bickhofe, Hans de Raad, Hans Vd Horst, Harold van Ingen, Harold Watson, Harry Chapman, Harry Kaczka, Harry Torque, Hayden Glass, Hayley Rosenblum, Heather Leson, Helen Crisp, Helen Michaud, Helen Qubain, Helle Rekdal Schønemann, Henrique Flach Latorre Moreno, Henry Finn, Henry Kaiser, Henry Lahore, Henry Steingeser, Hermann Paar, Hillary Miller, Hiroroni Kuriaki, Holly Dykes, Holly Lyne, Hubert Gertis, Hugh Geenen, Humble Daisy, Hüppe Keith, Iain Davidson, Ian Capstick, Ian Johnson, Ian Upton, Icaro Ferracini, Igor Lesko, Imran Haider, Inma de la Torre, Iris Brest, Irwin Madriaga, Isaac Sandaljian, Isaiah Tanenbaum, Ivan F. Villanueva B., J P Cleverdon, Jaakko Tamula Jr, Jacek Darken Gołębowski, Jack Hart, Jacky Hood, Jacob Dante Leffler, Jaime Perla,

Jaime Woo, Jake Campbell, Jake Loeterman, Jakes Rawlinson, James Allenspach, James Chesky, James Cloos, James Docherty, James Ellars, James K Wood, James Tyler, Jamie Finlay, Jamie Stevens, Jamil Khatib, Jan E Ellison, Jan Gondol, Jan Sepp, Jan Zuppinger, Jane Finette, Jane Lofton, Jane Mason, Jane Park, Janos Kovacs, Jasmina Bricic, Jason Blasso, Jason Chu, Jason Cole, Jason E. Barkeloo, Jason Hibbets, Jason Owen, Jason Sigal, Jay M Williams, Jazzy Bear Brown, JC Lara, Jean-Baptiste Carré, Jean-Philippe Dufraigne, Jean-Philippe Turcotte, Jean-Yves Hemlin, Jeanette Frey, Jeff Atwood, Jeff De Cagna, Jeff Donoghue, Jeff Edwards, Jeff Hilnbrand, Jeff Lowe, Jeff Rasalla, Jeff Ski Kinsey, Jeff Smith, Jeffrey L Tucker, Jeffrey Meyer, Jen Garcia, Jens Erat, Jeppe Bager Skjerning, Jeremy Dudet, Jeremy Russell, Jeremy Sabo, Jeremy Zauder, Jerko Grubisic, Jerome Glacken, Jérôme Mizeret, Jessica Dickinson Goodman, Jessica Litman, Jessica Mackay, Jessy Kate Schingler, Jesús Longás Gamarra, Jesus Marin, Jim Matt, Jim Meloy, Jim O'Flaherty, Jim Pellegrini, Jim Tittsler, Jimmy Alenius, Jiří Marek, Jo Allum, Joachim Brandon LeBlanc, Joachim Pileborg, Joachim von Goetz, Joakim Bang Larsen, Joan Rieu, Joanna Penn, João Almeida, Jochen Muetsch, Jodi Sandfort, Joe Cardillo, Joe Carpita, Joe Moross, Joerg Fricke, Johan Adda, Johan Meeusen, Johannes Förstner, Johannes Visintini, John Benfield, John Bevan, John C Patterson, John Crumrine, John Dimatos, John Feyler, John Huntsman, John Manoogian III, John Muller, John Ober, John Paul Blodgett, John Pearce, John Shale, John Sharp, John Simpson, John Sumser, John Weeks, John Wilbanks, John Worland, Johnny Mayall, Jollean Matsen, Jon Alberdi, Jon Andersen, Jon Cohrs, Jon Gotlin, Jon Schull, Jon Selmer Friberg, Jon Smith, Jonas Öberg, Jonas Weitzmann, Jonathan Campbell, Jonathan Deamer, Jonathan Holst, Jonathan Lin, Jonathan Schmid, Jonathan Yao, Jordon Kalilich, Jörg Schwarz, Jose Antonio Gallego Vázquez, Joseph Mcarthur, Joseph Noll, Joseph Sullivan, Joseph Tucker, Josh Bernhard, Josh Tong, Joshua Tobkin, JP Rangaswami, Juan Carlos Belair, Juan Irmig, Juan Pablo Carbajal, Juan Pablo Marin Diaz, Judith Newman, Judy

Tuan, Jukka Hellén, Julia Benson-Slaughter, Julia Devonshire, Julian Fietkau, Julie Harboe, Julien Brossoit, Julien Leroy, Juliet Chen, Julio Terra, Julius Mikkelä, Justin Christian, Justin Grimes, Justin Jones, Justin Szlasa, Justin Walsh, JustinChung.com, K. J. Przybylski, Kaloyan Raev, Kamil Śliwowski, Kaniska Padhi, Kara Malenfant, Kara Monroe, Karen Pe, Karl Jahn, Karl Jonsson, Karl Nelson, Kasia Zygmuntowicz, Kat Lim, Kate Chapman, Kate Stewart, Kathleen Beck, Kathleen Hanrahan, Kathryn Abuzzahab, Kathryn Deiss, Kathryn Rose, Kathy Payne, Katie Lynn Daniels, Katie Meek, Katie Teague, Katrina Hennessy, Katriona Main, Kavan Antani, Keith Adams, Keith Berndtson, MD, Keith Luebke, Kellie Higginbottom, Ken Friis Larsen, Ken Haase, Ken Torbeck, Kendel Ratley, Kendra Byrne, Kerry Hicks, Kevin Brown, Kevin Coates, Kevin Flynn, Kevin Ruman, Kevin Shannon, Kevin Taylor, Kevin Tostado, Kewhyun Kelly-Yuoh, Kiane l'Azin, Kianosh Pourian, Kiran Kadekoppa, Kit Walsh, Klaus Mickus, Konrad Rennert, Kris Kasianovitz, Kristian Lundquist, Kristin Buxton, Kristina Popova, Kristofer Bratt, Kristoffer Steen, Kumar McMillan, Kurt Whittemore, Kyle Pinches, Kyle Simpson, L Eaton, Lalo Martins, Lane Rasberry, Larry Garfield, Larry Singer, Lars Josephsen, Lars Klaeboe, Laura Anne Brown, Laura Billings, Laura Ferejohn, Lauren Pedersen, Laurence Gonsalves, Laurent Muchacho, Laurie Racine, Laurie Reynolds, Lawrence M. Schoen, Leandro Pangilinan, Leigh Verlandson, Lenka Gondolova, Leonardo Bueno Postacchini, Leonardo menegola, Lesley Mitchell, Leslie Krumholz, Leticia Britos Cavagnaro, Levi Bostian, Leyla Acaroglu, Liisa Ummelas, Lilly Kashmir Marques, Lior Mazliah, Lisa Bjerke, Lisa Brewster, Lisa Canning, Lisa Cronin, Lisa Di Valentino, Lisandro Gaertner, Livia Leskovec, Lynn Worldlaw, Liz Berg, Liz White, Logan Cox, Loki Carbis, Lora Lynn, Lorna Prescott, Lou Yu-fan, Louie Amphlett, Louis-David Benyayer, Louise Denman, Luca Corsato, Luca Lesinigo, Luca Palli, Luca Pianigiani, Luca S.G. de Marinis, Lucas Lopez, Lukas Mathis, Luke Chamberlin, Luke Chesser, Luke Woodbury, Lulu Tang, Lydia Pintscher, M Alexander Jurkat, Maarten

Sander, Macie J Klosowski, Magnus Adamsson, Magnus Killingberg, Mahmoud Abu-Wardeh, Maik Schmalstich, Maiken Håvarstein, Maira Sutton, Mairi Thomson, Mandy Wultsch, Manickavasakam Rajasekar, Marc Bogonovich, Marc Harpster, Marc Martí, Marc Olivier Bastien, Marc Stober, Marc-André Martin, Marcel de Leeuw, Marcel Hill, Marcia Hofmann, Marcin Olender, Marco Massarotto, Marco Montanari, Marco Morales, Marcos Medionegro, Marcus Bitzl, Marcus Norrgren, Margaret Gary, Mari Moreshead, Maria Liberman, Marielle Hsu, Marino Hernandez, Mario Lurig, Mario R. Hemsley, MD, Marissa Demers, Mark Chandler, Mark Cohen, Mark De Solla Price, Mark Gabby, Mark Gray, Mark Koudritsky, Mark Kupfer, Mark Lednor, Mark McGuire, Mark Moleda, Mark Mullen, Mark Murphy, Mark Perot, Mark Reeder, Mark Spickett, Mark Vincent Adams, Mark Waks, Mark Zuccarell II, Markus Deimann, Markus Jaritz, Markus Luethi, Marshal Miller, Marshall Warner, Martijn Arets, Martin Beaudoin, Martin Decky, Martin DeMello, Martin Humpolec, Martin Mayr, Martin Peck, Martin Sanchez, Martino Loco, Martti Remmelgas, Martyn Eggleton, Martyn Lewis, Mary Ellen Davis, Mary Heacock, Mary Hess, Mary Mi, Masahiro Takagi, Mason Du, Massimo V.A. Manzari, Mathias Bavay, Mathias Nicolajsen Kjærgaard, Matias Kruk, Matija Nalis, Matt Alcock, Matt Black, Matt Broach, Matt Hall, Matt Haughey, Matt Lee, Matt Plec, Matt Skoss, Matt Thompson, Matt Vance, Matt Wagstaff, Matteo Cocco, Matthew Bendert, Matthew Bergholt, Matthew Darlison, Matthew Epler, Matthew Hawken, Matthew Heimbecker, Matthew Orstad, Matthew Peterworth, Matthew Sheehy, Matthew Tucker, Adaptive Handy Apps, LLC, Matthias Axell, Max Green, Max Kossatz, Max Iupo, Max Temkin, Max van Balgooy, Médéric Droz-dit-Busset, Megan Ingle, Megan Wacha, Meghan Finlayson, Melissa Aho, Melissa Sterry, Melle Funambuline, Menachem Goldstein, Micah Bridges, Michael Alberto, Michael Anderson, Michael Andersson Skane, Michael C. Stewart, Michael Carroll, Michael Cavette, Michael Crees, Michael David Johas Teener, Michael Dennis Moore, Michael Freundt Karlsen,

Michael Harries, Michael Hawel, Michael Lewis, Michael May, Michael Murphy, Michael Murvine, Michael Perkins, Michael Sauers, Michael St.Onge, Michael Stanford, Michael Stanley, Michael Underwood, Michael Weiss, Michael Wright, Michael-Andreas Kuttner, Michaela Voigt, Michal Rosenn, Michał Szymański, Michel Gallez, Michell Zappa, Michelle Heeyeon You, Miha Batic, Mik Ishmael, Mikael Andersson, Mike Chelen, Mike Habicher, Mike Maloney, Mike Masnick, Mike McDaniel, Mike Pouraryan, Mike Sheldon, Mike Stop Continues, Mike Stringer, Mike Wittenstein, Mikkel Ovesen, Mikołaj Podlaszewski, Millie Gonzalez, Mindi Lovell, Mindy Lin, Mirko "Macro" Fichtner, Mitch Featherston, Mitchell Adams, Molika Oum, Molly Shaffer Van Houweling, Monica Mora, Morgan Loomis, Moritz Schubert, Mrs. Paganini, Mushin Schilling, Mustafa K Callik, MD, Myk Pilgrim, Myra Harmer, Nadine Forget-Dubois, Nagle Industries, LLC, Nah Wee Yang, Natalie Brown, Natalie Freed, Nathan D Howell, Nathan Massey, Nathan Miller, Neal Gorenflo, Neal McBurnett, Neal Stimler, Neil Wilson, Nele Wollert, Neuchee Chang, Niall McDonagh, Niall Twohig, Nic McPhee, Nicholas Bentley, Nicholas Koran, Nicholas Norfolk, Nicholas Potter, Nick Bell, Nick Coghlan, Nick Isaacs, Nick M. Daly, Nick Vance, Nickolay Vedenikov, Nicky Weaver-Weinberg, Nico Prin, Nicolas Weidinger, Nicole Hickman, Niek Theunissen, Nigel Robertson, Nikki Thompson, Nikko Marie, Nikola Chernev, Nils Laveson, Noah Blumenson-Cook, Noah Fang, Noah Kardos-Fein, Noah Meyerhans, Noel Hanigan, Noel Hart, Norrie Mailer, O.P. Gobée, Ohad Mayblum, Olivia Wilson, Olivier De Doncker, Olivier Schulbaum, Olle Ahnve, Omar Kaminski, Omar Willey, OpenBuilds, Ove Ødegård, Øystein Kjærnet, Pablo López Soriano, Pablo Vasquez, Pacific Design, Paige Mackay, Papp István Péter, Paris Marx, Parker Higgins, Pasquale Borriello, Pat Allan, Pat Hawks, Pat Ludwig, Pat Sticks, Patricia Brennan, Patricia Rosnel, Patricia Wolf, Patrick Berry, Patrick Beseda, Patrick Hurley, Patrick M. Lozeau, Patrick McCabe, Patrick Nafarrete, Patrick Tanguay, Patrick von Hauff, Patrik Kernstock, Patti

J Ryan, Paul A Golder, Paul and Iris Brest, Paul Bailey, Paul Bryan, Paul Bunkham, Paul Elosegui, Paul Hibbitts, Paul Jacobson, Paul Keller, Paul Rowe, Paul Timpson, Paul Walker, Pavel Dostál, Peeter Sällström Randsalu, Peggy Frith, Pen-Yuan Hsing, Penny Pearson, Per Åström, Perry Jetter, Péter Fankhauser, Peter Hirtle, Peter Humphries, Peter Jenkins, Peter Langmar, Peter le Roux, Peter Marinari, Peter Mengelers, Peter O'Brien, Peter Pinch, Peter S. Crosby, Peter Wells, Petr Fristedt, Petr Viktorin, Petronella Jeurissen, Phil Flickinger, Philip Chung, Philip Pangrac, Philip R. Skaggs Jr., Philip Young, Philippa Lorne Channer, Philippe Vandenbroeck, Pierluigi Luisi, Pierre Suter, Pieter-Jan Pauwels, Playground Inc., Pomax, Popenoe, Pouhiou Noenaute, Prilutskiy Kirill, Print3Dreams Ltd., Quentin Coispeau, R. Smith, Race DiLoreto, Rachel Mercer, Rafael Scapin, Rafaela Kunz, Rain Doggerel, Raine Lourie, Rajiv Jhangiani, Ralph Chapoteau, Randall Kirby, Randy Brians, Raphaël Alexandre, Raphaël Schröder, Rasmus Jensen, Rayn Drahps, Rayna Stamboliyska, Rebecca Godar, Rebecca Lendl, Rebecca Weir, Regina Tschud, Remi Dino, Ric Herrero, Rich McCue, Richard "TalkToMeGuy" Olson, Richard Best, Richard Blumberg, Richard Fannon, Richard Heying, Richard Karnesky, Richard Kelly, Richard Littauer, Richard Sobey, Richard White, Richard Winchell, Rik ToeWater, Rita Lewis, Rita Wood, Riyadh Al Balushi, Rob Balder, Rob Berkley, Rob Bertholf, Rob Emanuele, Rob McAuliffe, Rob McLaughan, Rob Tillie, Rob Utter, Rob Vincent, Robert Gaffney, Robert Jones, Robert Kelly, Robert Lawlis, Robert McDonald, Robert Orzanna, Robert Paterson Hunter, Robert R. Daniel Jr., Robert Ryan-Silva, Robert Thompson, Robert Wagoner, Roberto Selvaggio, Robin DeRosa, Robin Rist Kildal, Rodrigo Castilhos, Roger Bacon, Roger Saner, Roger So, Roger Solé, Roger Tregear, Roland Tanglao, Rolf and Mari von Walhausen, Rolf Egstad, Rolf Schaller, Ron Zuijlen, Ronald Bissell, Ronald van den Hoff, Ronda Snow, Rory Landon Aronson, Ross Findlay, Ross Pruden, Ross Williams, Rowan Skewes, Roy Ivy III, Ruben Flores, Rupert Hitzenberger, Rusi Popov, Russ Antonucci, Russ Spillin, Russell Brand,

Rute Correia, Ruth Ann Carpenter, Ruth White, Ryan Mentock, Ryan Merkley, Ryan Price, Ryan Sasaki, Ryan Singer, Ryan Voisin, Ryan Weir, Searle, Salem Bin Kenaid, Salomon Riedo, Sam Hokin, Sam Twidale, Samantha Levin, Samantha-Jayne Chapman, Samarth Agarwal, Sami Al-AbdRabbuh, Samuel A. Rebelsky, Samuel Goëta, Samuel Hauser, Samuel Landete, Samuel Oliveira Cersosimo, Samuel Tait, Sandra Fauconnier, Sandra Markus, Sandy Bjar, Sandy ONEil, Sang-Phil Ju, Sanjay Basu, Santiago Garcia, Sara Armstrong, Sara Lucca, Sara Rodriguez Marin, Sarah Brand, Sarah Cove, Sarah Curran, Sarah Gold, Sarah McGovern, Sarah Smith, Sarinee Achavanuntakul, Sasha Moss, Sasha VanHoven, Saul Gasca, Scott Abbott, Scott Akerman, Scott Beattie, Scott Bruinooge, Scott Conroy, Scott Gillespie, Scott Williams, Sean Anderson, Sean Johnson, Sean Lim, Sean Wickett, Seb Schmoller, Sebastiaan Bekker, Sebastiaan ter Burg, Sebastian Makowiecki, Sebastian Meyer, Sebastian Schweizer, Sebastian Sigloch, Sebastien Huchet, Seokwon Yang, Sergey Chernyshev, Sergey Storchay, Sergio Cardoso, Seth Dreibitko, Seth Gover, Seth Lepore, Shannon Turner, Sharon Clapp, Shauna Redmond, Shawn Gaston, Shawn Martin, Shay Knohl, Shelby Hatfield, Sheldon (Vila) Widuch, Sheona Thomson, Si Jie, Sicco van Sas, Siena Oristaglio, Simon Glover, Simon John King, Simon Klose, Simon Law, Simon Linder, Simon Moffitt, Solomon Kahn, Solomon Simon, Soujanna Sarkar, Stanislav Trifonov, Stefan Dumont, Stefan Jansson, Stefan Langer, Stefan Lindblad, Stefano Guidotti, Stefano Luzardi, Stephan Meißl, Stéphane Wojewoda, Stephanie Pereira, Stephen Gates, Stephen Murphey, Stephen Pearce, Stephen Rose, Stephen Suen, Stephen Walli, Stevan Matheson, Steve Battle, Steve Fischers, Steve Fitzhugh, Steve Guengerich, Steve Ingram, Steve Kroy, Steve Midgley, Steve Rhine, Steven Kasprzyk, Steven Knudsen, Steven Melvin, Stig-Jørund B. Ö. Arnesen, Stuart Drewer, Stuart Maxwell, Stuart Reich, Subhendu Ghosh, Sujal Shah, Sune Bøegh, Susan Chun, Susan R Grossman, Suzie Wiley, Sven Fielitz, Swan/Starts, Sylvain Carle, Sylvain Chery, Sylvia Green, Sylvia van Brug-

gen, Szabolcs Berecz, T. L. Mason, Tanbir Baeg, Tanya Hart, Tara Tiger Brown, Tara Westover, Tarmo Toikkanen, Tasha Turner Lennhoff, Thagat Varma, Ted Timmons, Tej Dhawan, Teresa Gonczy, Terry Hook, Theis Madsen, Theo M. Scholl, Theresa Bernardo, Thibault Badenas, Thomas Bacig, Thomas Boehnlein, Thomas Bøvith, Thomas Chang, Thomas Hartman, Thomas Kent, Thomas Morgan, Thomas Philipp-Edmonds, Thomas Thrush, Thomas Werkmeister, Tieg Zaharia, Tieu Thuy Nguyen, Tim Chambers, Tim Cook, Tim Evers, Tim Nichols, Tim Stahmer, Timothée Planté, Timothy Arfsten, Timothy Hinckliff, Timothy Vollmer, Tina Coffman, Tisza Gergő, Tobias Schonwetter, Todd Brown, Todd Pousley, Todd Sattersten, Tom Bamford, Tom Caswell, Tom Goren, Tom Kent, Tom MacWright, Tom Maillioux, Tom Merkli, Tom Merritt, Tom Myers, Tom Olijhoek, Tom Rubin, Tommaso De Benetti, Tommy Dahlen, Tony Ciak, Tony Nwachukwu, Torsten Skomp, Tracey Depellegrin, Tracey Henton, Tracey James, Traci Long DeForge, Trent Yarwood, Trevor Hogue, Trey Blalock, Trey Hunner, Tryggvi Björgvinsson, Tumuult, Tushar Roy, Tyler Occhiogrosso, Udo Blenkhorn, Uri Sivan, Vanja Bobas, Vantharith Oum, Vaughan Jenkins, Veethika Mishra, Vic King, Vickie Goode, Victor DePina, Victor Grigas, Victoria Klassen, Victorien Elvinger, VIGA Manufacture, Vikas Shah, Vinayak S. Kaujalgi, Vincent O'Leary, Violette Paquet, Virginia Gentilini, Virginia Kopelman, Vitor Menezes, Vivian Marthell, Wayne Mackintosh, Wendy Keenan, Werner Wiethge, Wesley Derbyshire, Widar Hellwig, Willa Köerner, William Bettridge-Radford, William Jefferson, William Marshall, William Peter Nash, William Ray, William Robins, Willow Rosenberg, Winie Evers, Wolfgang Renninger, Xavier Anto-viaque, Xavier Hugonet, Xavier Moisant, Xueqi Li, Yancey Strickler, Yann Heurtaux, Yasmine Hajjar, Yu-Hsian Sun, Yves Deruisseau, Zach Chandler, Zak Zebrowski, Zane Amiralis and Joshua de Haan, ZeMarmot Open Movie

Think Python

How to Think Like a Computer Scientist

2nd Edition, Version 2.4.0

Think Python

How to Think Like a Computer Scientist

2nd Edition, Version 2.4.0

Allen Downey

Green Tea Press

Needham, Massachusetts

Copyright © 2015 Allen Downey.

Green Tea Press
9 Washburn Ave
Needham MA 02492

Permission is granted to copy, distribute, and/or modify this document under the terms of the Creative Commons Attribution-NonCommercial 3.0 Unported License, which is available at <http://creativecommons.org/licenses/by-nc/3.0/>.

The original form of this book is \LaTeX source code. Compiling this \LaTeX source has the effect of generating a device-independent representation of a textbook, which can be converted to other formats and printed.

The \LaTeX source for this book is available from <http://www.thinkpython2.com>

Preface

The strange history of this book

In January 1999 I was preparing to teach an introductory programming class in Java. I had taught it three times and I was getting frustrated. The failure rate in the class was too high and, even for students who succeeded, the overall level of achievement was too low.

One of the problems I saw was the books. They were too big, with too much unnecessary detail about Java, and not enough high-level guidance about how to program. And they all suffered from the trap door effect: they would start out easy, proceed gradually, and then somewhere around Chapter 5 the bottom would fall out. The students would get too much new material, too fast, and I would spend the rest of the semester picking up the pieces.

Two weeks before the first day of classes, I decided to write my own book. My goals were:

- Keep it short. It is better for students to read 10 pages than not read 50 pages.
- Be careful with vocabulary. I tried to minimize jargon and define each term at first use.
- Build gradually. To avoid trap doors, I took the most difficult topics and split them into a series of small steps.
- Focus on programming, not the programming language. I included the minimum useful subset of Java and left out the rest.

I needed a title, so on a whim I chose *How to Think Like a Computer Scientist*.

My first version was rough, but it worked. Students did the reading, and they understood enough that I could spend class time on the hard topics, the interesting topics and (most important) letting the students practice.

I released the book under the GNU Free Documentation License, which allows users to copy, modify, and distribute the book.

What happened next is the cool part. Jeff Elkner, a high school teacher in Virginia, adopted my book and translated it into Python. He sent me a copy of his translation, and I had the unusual experience of learning Python by reading my own book. As Green Tea Press, I published the first Python version in 2001.

In 2003 I started teaching at Olin College and I got to teach Python for the first time. The contrast with Java was striking. Students struggled less, learned more, worked on more interesting projects, and generally had a lot more fun.

Since then I've continued to develop the book, correcting errors, improving some of the examples and adding material, especially exercises.

The result is this book, now with the less grandiose title *Think Python*. Some of the changes are:

- I added a section about debugging at the end of each chapter. These sections present general techniques for finding and avoiding bugs, and warnings about Python pitfalls.
- I added more exercises, ranging from short tests of understanding to a few substantial projects. Most exercises include a link to my solution.
- I added a series of case studies—longer examples with exercises, solutions, and discussion.
- I expanded the discussion of program development plans and basic design patterns.
- I added appendices about debugging and analysis of algorithms.

The second edition of *Think Python* has these new features:

- The book and all supporting code have been updated to Python 3.
- I added a few sections, and more details on the web, to help beginners get started running Python in a browser, so you don't have to deal with installing Python until you want to.
- For Chapter 4.1 I switched from my own turtle graphics package, called `Swampy`, to a more standard Python module, `turtle`, which is easier to install and more powerful.
- I added a new chapter called “The Goodies”, which introduces some additional Python features that are not strictly necessary, but sometimes handy.

I hope you enjoy working with this book, and that it helps you learn to program and think like a computer scientist, at least a little bit.

Allen B. Downey

Olin College

Acknowledgments

Many thanks to Jeff Elkner, who translated my Java book into Python, which got this project started and introduced me to what has turned out to be my favorite language.

Thanks also to Chris Meyers, who contributed several sections to *How to Think Like a Computer Scientist*.

Thanks to the Free Software Foundation for developing the GNU Free Documentation License, which helped make my collaboration with Jeff and Chris possible, and Creative Commons for the license I am using now.

Thanks to the editors at Lulu who worked on *How to Think Like a Computer Scientist*.

Thanks to the editors at O'Reilly Media who worked on *Think Python*.

Thanks to all the students who worked with earlier versions of this book and all the contributors (listed below) who sent in corrections and suggestions.

Contributor List

More than 100 sharp-eyed and thoughtful readers have sent in suggestions and corrections over the past few years. Their contributions, and enthusiasm for this project, have been a huge help.

If you have a suggestion or correction, please send email to feedback@thinkpython.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!

- Lloyd Hugh Allen sent in a correction to Section 8.4.
- Yvon Boulianne sent in a correction of a semantic error in Chapter 5.
- Fred Bremmer submitted a correction in Section 2.1.
- Jonah Cohen wrote the Perl scripts to convert the LaTeX source for this book into beautiful HTML.
- Michael Conlon sent in a grammar correction in Chapter 2 and an improvement in style in Chapter 1, and he initiated discussion on the technical aspects of interpreters.
- Benoît Girard sent in a correction to a humorous mistake in Section 5.6.
- Courtney Gleason and Katherine Smith wrote `horsebet.py`, which was used as a case study in an earlier version of the book. Their program can now be found on the website.
- Lee Harr submitted more corrections than we have room to list here, and indeed he should be listed as one of the principal editors of the text.
- James Kaylin is a student using the text. He has submitted numerous corrections.
- David Kershaw fixed the broken `catTwice` function in Section 3.10.
- Eddie Lam has sent in numerous corrections to Chapters 1, 2, and 3. He also fixed the Makefile so that it creates an index the first time it is run and helped us set up a versioning scheme.
- Man-Yong Lee sent in a correction to the example code in Section 2.4.
- David Mayo pointed out that the word "unconsciously" in Chapter 1 needed to be changed to "subconsciously".
- Chris McAloon sent in several corrections to Sections 3.9 and 3.10.
- Matthew J. Moelter has been a long-time contributor who sent in numerous corrections and suggestions to the book.

- Simon Dicon Montford reported a missing function definition and several typos in Chapter 3. He also found errors in the `increment` function in Chapter 13.
- John Ouzts corrected the definition of “return value” in Chapter 3.
- Kevin Parks sent in valuable comments and suggestions as to how to improve the distribution of the book.
- David Pool sent in a typo in the glossary of Chapter 1, as well as kind words of encouragement.
- Michael Schmitt sent in a correction to the chapter on files and exceptions.
- Robin Shaw pointed out an error in Section 13.1, where the `printTime` function was used in an example without being defined.
- Paul Sleigh found an error in Chapter 7 and a bug in Jonah Cohen’s Perl script that generates HTML from LaTeX.
- Craig T. Snydal is testing the text in a course at Drew University. He has contributed several valuable suggestions and corrections.
- Ian Thomas and his students are using the text in a programming course. They are the first ones to test the chapters in the latter half of the book, and they have made numerous corrections and suggestions.
- Keith Verheyden sent in a correction in Chapter 3.
- Peter Winstanley let us know about a longstanding error in our Latin in Chapter 3.
- Chris Wrobel made corrections to the code in the chapter on file I/O and exceptions.
- Moshe Zadka has made invaluable contributions to this project. In addition to writing the first draft of the chapter on Dictionaries, he provided continual guidance in the early stages of the book.
- Christoph Zworschke sent several corrections and pedagogic suggestions, and explained the difference between *gleich* and *selbe*.
- James Mayer sent us a whole slew of spelling and typographical errors, including two in the contributor list.
- Hayden McAfee caught a potentially confusing inconsistency between two examples.
- Angel Arnal is part of an international team of translators working on the Spanish version of the text. He has also found several errors in the English version.
- Tauhidul Hoque and Lex Berezhny created the illustrations in Chapter 1 and improved many of the other illustrations.
- Dr. Michele Alzetta caught an error in Chapter 8 and sent some interesting pedagogic comments and suggestions about Fibonacci and Old Maid.
- Andy Mitchell caught a typo in Chapter 1 and a broken example in Chapter 2.
- Kalin Harvey suggested a clarification in Chapter 7 and caught some typos.
- Christopher P. Smith caught several typos and helped us update the book for Python 2.2.
- David Hutchins caught a typo in the Foreword.
- Gregor Lingl is teaching Python at a high school in Vienna, Austria. He is working on a German translation of the book, and he caught a couple of bad errors in Chapter 5.

- Julie Peters caught a typo in the Preface.
- Florin Oprina sent in an improvement in `makeTime`, a correction in `printTime`, and a nice typo.
- D. J. Webre suggested a clarification in Chapter 3.
- Ken found a fistful of errors in Chapters 8, 9 and 11.
- Ivo Wever caught a typo in Chapter 5 and suggested a clarification in Chapter 3.
- Curtis Yanko suggested a clarification in Chapter 2.
- Ben Logan sent in a number of typos and problems with translating the book into HTML.
- Jason Armstrong saw the missing word in Chapter 2.
- Louis Cordier noticed a spot in Chapter 16 where the code didn't match the text.
- Brian Cain suggested several clarifications in Chapters 2 and 3.
- Rob Black sent in a passel of corrections, including some changes for Python 2.2.
- Jean-Philippe Rey at École Centrale Paris sent a number of patches, including some updates for Python 2.2 and other thoughtful improvements.
- Jason Mader at George Washington University made a number of useful suggestions and corrections.
- Jan Gundtofte-Bruun reminded us that “a error” is an error.
- Abel David and Alexis Dinno reminded us that the plural of “matrix” is “matrices”, not “matrices”. This error was in the book for years, but two readers with the same initials reported it on the same day. Weird.
- Charles Thayer encouraged us to get rid of the semi-colons we had put at the ends of some statements and to clean up our use of “argument” and “parameter”.
- Roger Sperberg pointed out a twisted piece of logic in Chapter 3.
- Sam Bull pointed out a confusing paragraph in Chapter 2.
- Andrew Cheung pointed out two instances of “use before def”.
- C. Corey Capel spotted the missing word in the Third Theorem of Debugging and a typo in Chapter 4.
- Alessandra helped clear up some Turtle confusion.
- Wim Champagne found a brain-o in a dictionary example.
- Douglas Wright pointed out a problem with floor division in `arc`.
- Jared Spindor found some jetsam at the end of a sentence.
- Lin Peiheng sent a number of very helpful suggestions.
- Ray Hagtvedt sent in two errors and a not-quite-error.
- Torsten Hübsch pointed out an inconsistency in `Swampy`.
- Inga Petuhhov corrected an example in Chapter 14.
- Arne Babenhauserheide sent several helpful corrections.

- Mark E. Casida is good at spotting repeated words.
- Scott Tyler filled in a that was missing. And then sent in a heap of corrections.
- Gordon Shephard sent in several corrections, all in separate emails.
- Andrew Turner spotted an error in Chapter 8.
- Adam Hobart fixed a problem with floor division in `arc`.
- Daryl Hammond and Sarah Zimmerman pointed out that I served up `math.pi` too early. And Zim spotted a typo.
- George Sass found a bug in a Debugging section.
- Brian Bingham suggested Exercise 11.5.
- Leah Engelbert-Fenton pointed out that I used `tuple` as a variable name, contrary to my own advice. And then found a bunch of typos and a “use before def”.
- Joe Funke spotted a typo.
- Chao-chao Chen found an inconsistency in the Fibonacci example.
- Jeff Paine knows the difference between space and spam.
- Lubos Pintes sent in a typo.
- Gregg Lind and Abigail Heithoff suggested Exercise 14.3.
- Max Hailperin has sent in a number of corrections and suggestions. Max is one of the authors of the extraordinary *Concrete Abstractions*, which you might want to read when you are done with this book.
- Chotipat Pornavalai found an error in an error message.
- Stanislaw Antol sent a list of very helpful suggestions.
- Eric Pashman sent a number of corrections for Chapters 4–11.
- Miguel Azevedo found some typos.
- Jianhua Liu sent in a long list of corrections.
- Nick King found a missing word.
- Martin Zuther sent a long list of suggestions.
- Adam Zimmerman found an inconsistency in my instance of an “instance” and several other errors.
- Ratnakar Tiwari suggested a footnote explaining degenerate triangles.
- Anurag Goel suggested another solution for `is_abecedarian` and sent some additional corrections. And he knows how to spell Jane Austen.
- Kelli Kratzer spotted one of the typos.
- Mark Griffiths pointed out a confusing example in Chapter 3.
- Roydan Ongie found an error in my Newton’s method.
- Patryk Wolowiec helped me with a problem in the HTML version.

-
- Mark Chonofsky told me about a new keyword in Python 3.
 - Russell Coleman helped me with my geometry.
 - Nam Nguyen found a typo and pointed out that I used the Decorator pattern but didn't mention it by name.
 - Stéphane Morin sent in several corrections and suggestions.
 - Paul Stoop corrected a typo in `uses_only`.
 - Eric Bronner pointed out a confusion in the discussion of the order of operations.
 - Alexandros Gezerlis set a new standard for the number and quality of suggestions he submitted. We are deeply grateful!
 - Gray Thomas knows his right from his left.
 - Giovanni Escobar Sosa sent a long list of corrections and suggestions.
 - Daniel Neilson corrected an error about the order of operations.
 - Will McGinnis pointed out that `polyline` was defined differently in two places.
 - Frank Hecker pointed out an exercise that was under-specified, and some broken links.
 - Animesh B helped me clean up a confusing example.
 - Martin Caspersen found two round-off errors.
 - Gregor Ulm sent several corrections and suggestions.
 - Dimitrios Tsirigkas suggested I clarify an exercise.
 - Carlos Tafur sent a page of corrections and suggestions.
 - Martin Nordsletten found a bug in an exercise solution.
 - Sven Hoexter pointed out that a variable named `input` shadows a build-in function.
 - Stephen Gregory pointed out the problem with `cmp` in Python 3.
 - Ishwar Bhat corrected my statement of Fermat's last theorem.
 - Andrea Zanella translated the book into Italian, and sent a number of corrections along the way.
 - Many, many thanks to Melissa Lewis and Luciano Ramalho for excellent comments and suggestions on the second edition.
 - Thanks to Harry Percival from PythonAnywhere for his help getting people started running Python in a browser.
 - Xavier Van Aubel made several useful corrections in the second edition.
 - William Murray corrected my definition of floor division.
 - Per Starbäck brought me up to date on universal newlines in Python 3.
 - Laurent Rosenfeld and Mihaela Rotaru translated this book into French. Along the way, they sent many corrections and suggestions.

In addition, people who spotted typos or made corrections include Czeslaw Czapla, Dale Wilson, Francesco Carlo Cimini, Richard Fursa, Brian McGhie, Lokesh Kumar Makani, Matthew Shultz, Viet Le, Victor Simeone, Lars O.D. Christensen, Swarup Sahoo, Alix Etienne, Kuang He, Wei Huang, Karen Barber, and Eric Ransom.

Contents

Preface	v
1 The way of the program	1
1.1 What is a program?	1
1.2 Running Python	2
1.3 The first program	3
1.4 Arithmetic operators	3
1.5 Values and types	4
1.6 Formal and natural languages	4
1.7 Debugging	6
1.8 Glossary	6
1.9 Exercises	7
2 Variables, expressions and statements	9
2.1 Assignment statements	9
2.2 Variable names	9
2.3 Expressions and statements	10
2.4 Script mode	11
2.5 Order of operations	11
2.6 String operations	12
2.7 Comments	12
2.8 Debugging	13
2.9 Glossary	14
2.10 Exercises	14

3 Functions	17
3.1 Function calls	17
3.2 Math functions	18
3.3 Composition	19
3.4 Adding new functions	19
3.5 Definitions and uses	20
3.6 Flow of execution	21
3.7 Parameters and arguments	21
3.8 Variables and parameters are local	22
3.9 Stack diagrams	23
3.10 Fruitful functions and void functions	24
3.11 Why functions?	24
3.12 Debugging	25
3.13 Glossary	25
3.14 Exercises	26
4 Case study: interface design	29
4.1 The turtle module	29
4.2 Simple repetition	30
4.3 Exercises	31
4.4 Encapsulation	32
4.5 Generalization	32
4.6 Interface design	33
4.7 Refactoring	34
4.8 A development plan	35
4.9 docstring	35
4.10 Debugging	36
4.11 Glossary	36
4.12 Exercises	37

5	Conditionals and recursion	39
5.1	Floor division and modulus	39
5.2	Boolean expressions	40
5.3	Logical operators	40
5.4	Conditional execution	41
5.5	Alternative execution	41
5.6	Chained conditionals	41
5.7	Nested conditionals	42
5.8	Recursion	43
5.9	Stack diagrams for recursive functions	44
5.10	Infinite recursion	44
5.11	Keyboard input	45
5.12	Debugging	46
5.13	Glossary	47
5.14	Exercises	47
6	Fruitful functions	51
6.1	Return values	51
6.2	Incremental development	52
6.3	Composition	54
6.4	Boolean functions	54
6.5	More recursion	55
6.6	Leap of faith	57
6.7	One more example	57
6.8	Checking types	58
6.9	Debugging	59
6.10	Glossary	60
6.11	Exercises	60

7 Iteration	63
7.1 Reassignment	63
7.2 Updating variables	64
7.3 The <code>while</code> statement	64
7.4 <code>break</code>	66
7.5 Square roots	66
7.6 Algorithms	67
7.7 Debugging	68
7.8 Glossary	68
7.9 Exercises	69
8 Strings	71
8.1 A string is a sequence	71
8.2 <code>len</code>	72
8.3 Traversal with a <code>for</code> loop	72
8.4 String slices	73
8.5 Strings are immutable	74
8.6 Searching	74
8.7 Looping and counting	75
8.8 String methods	75
8.9 The <code>in</code> operator	76
8.10 String comparison	77
8.11 Debugging	77
8.12 Glossary	79
8.13 Exercises	79
9 Case study: word play	83
9.1 Reading word lists	83
9.2 Exercises	84
9.3 Search	85
9.4 Looping with indices	86
9.5 Debugging	87
9.6 Glossary	87
9.7 Exercises	88

10 Lists	89
10.1 A list is a sequence	89
10.2 Lists are mutable	90
10.3 Traversing a list	91
10.4 List operations	91
10.5 List slices	91
10.6 List methods	92
10.7 Map, filter and reduce	93
10.8 Deleting elements	94
10.9 Lists and strings	94
10.10 Objects and values	95
10.11 Aliasing	96
10.12 List arguments	97
10.13 Debugging	98
10.14 Glossary	100
10.15 Exercises	100
11 Dictionaries	103
11.1 A dictionary is a mapping	103
11.2 Dictionary as a collection of counters	104
11.3 Looping and dictionaries	106
11.4 Reverse lookup	106
11.5 Dictionaries and lists	107
11.6 Memos	109
11.7 Global variables	110
11.8 Debugging	111
11.9 Glossary	112
11.10 Exercises	113

12 Tuples	115
12.1 Tuples are immutable	115
12.2 Tuple assignment	116
12.3 Tuples as return values	117
12.4 Variable-length argument tuples	118
12.5 Lists and tuples	118
12.6 Dictionaries and tuples	120
12.7 Sequences of sequences	121
12.8 Debugging	122
12.9 Glossary	122
12.10 Exercises	123
13 Case study: data structure selection	125
13.1 Word frequency analysis	125
13.2 Random numbers	126
13.3 Word histogram	127
13.4 Most common words	128
13.5 Optional parameters	129
13.6 Dictionary subtraction	129
13.7 Random words	130
13.8 Markov analysis	130
13.9 Data structures	132
13.10 Debugging	133
13.11 Glossary	134
13.12 Exercises	134
14 Files	137
14.1 Persistence	137
14.2 Reading and writing	137
14.3 Format operator	138
14.4 Filenames and paths	139
14.5 Catching exceptions	140

14.6	Databases	141
14.7	Pickling	142
14.8	Pipes	142
14.9	Writing modules	143
14.10	Debugging	144
14.11	Glossary	145
14.12	Exercises	145
15	Classes and objects	147
15.1	Programmer-defined types	147
15.2	Attributes	148
15.3	Rectangles	149
15.4	Instances as return values	150
15.5	Objects are mutable	151
15.6	Copying	151
15.7	Debugging	152
15.8	Glossary	153
15.9	Exercises	154
16	Classes and functions	155
16.1	Time	155
16.2	Pure functions	156
16.3	Modifiers	157
16.4	Prototyping versus planning	158
16.5	Debugging	159
16.6	Glossary	160
16.7	Exercises	160
17	Classes and methods	161
17.1	Object-oriented features	161
17.2	Printing objects	162
17.3	Another example	163

17.4	A more complicated example	164
17.5	The <code>init</code> method	164
17.6	The <code>__str__</code> method	165
17.7	Operator overloading	165
17.8	Type-based dispatch	166
17.9	Polymorphism	167
17.10	Debugging	168
17.11	Interface and implementation	169
17.12	Glossary	169
17.13	Exercises	170
18	Inheritance	171
18.1	Card objects	171
18.2	Class attributes	172
18.3	Comparing cards	173
18.4	Decks	174
18.5	Printing the deck	174
18.6	Add, remove, shuffle and sort	175
18.7	Inheritance	176
18.8	Class diagrams	177
18.9	Debugging	178
18.10	Data encapsulation	179
18.11	Glossary	180
18.12	Exercises	181
19	The Goodies	183
19.1	Conditional expressions	183
19.2	List comprehensions	184
19.3	Generator expressions	185
19.4	<code>any</code> and <code>all</code>	185
19.5	Sets	186
19.6	Counters	187

19.7	defaultdict	188
19.8	Named tuples	189
19.9	Gathering keyword args	190
19.10	Glossary	191
19.11	Exercises	192
A	Debugging	193
A.1	Syntax errors	193
A.2	Runtime errors	195
A.3	Semantic errors	198
B	Analysis of Algorithms	201
B.1	Order of growth	202
B.2	Analysis of basic Python operations	204
B.3	Analysis of search algorithms	205
B.4	Hashtables	206
B.5	Glossary	209

Chapter 1

The way of the program

The goal of this book is to teach you to think like a computer scientist. This way of thinking combines some of the best features of mathematics, engineering, and natural science. Like mathematicians, computer scientists use formal languages to denote ideas (specifically computations). Like engineers, they design things, assembling components into systems and evaluating tradeoffs among alternatives. Like scientists, they observe the behavior of complex systems, form hypotheses, and test predictions.

The single most important skill for a computer scientist is **problem solving**. Problem solving means the ability to formulate problems, think creatively about solutions, and express a solution clearly and accurately. As it turns out, the process of learning to program is an excellent opportunity to practice problem-solving skills. That's why this chapter is called, "The way of the program".

On one level, you will be learning to program, a useful skill by itself. On another level, you will use programming as a means to an end. As we go along, that end will become clearer.

1.1 What is a program?

A **program** is a sequence of instructions that specifies how to perform a computation. The computation might be something mathematical, such as solving a system of equations or finding the roots of a polynomial, but it can also be a symbolic computation, such as searching and replacing text in a document or something graphical, like processing an image or playing a video.

The details look different in different languages, but a few basic instructions appear in just about every language:

input: Get data from the keyboard, a file, the network, or some other device.

output: Display data on the screen, save it in a file, send it over the network, etc.

math: Perform basic mathematical operations like addition and multiplication.

conditional execution: Check for certain conditions and run the appropriate code.

repetition: Perform some action repeatedly, usually with some variation.

Believe it or not, that's pretty much all there is to it. Every program you've ever used, no matter how complicated, is made up of instructions that look pretty much like these. So you can think of programming as the process of breaking a large, complex task into smaller and smaller subtasks until the subtasks are simple enough to be performed with one of these basic instructions.

1.2 Running Python

One of the challenges of getting started with Python is that you might have to install Python and related software on your computer. If you are familiar with your operating system, and especially if you are comfortable with the command-line interface, you will have no trouble installing Python. But for beginners, it can be painful to learn about system administration and programming at the same time.

To avoid that problem, I recommend that you start out running Python in a browser. Later, when you are comfortable with Python, I'll make suggestions for installing Python on your computer.

There are a number of web pages you can use to run Python. If you already have a favorite, go ahead and use it. Otherwise I recommend PythonAnywhere. I provide detailed instructions for getting started at <http://tinyurl.com/thinkpython2e>.

There are two versions of Python, called Python 2 and Python 3. They are very similar, so if you learn one, it is easy to switch to the other. In fact, there are only a few differences you will encounter as a beginner. This book is written for Python 3, but I include some notes about Python 2.

The Python **interpreter** is a program that reads and executes Python code. Depending on your environment, you might start the interpreter by clicking on an icon, or by typing `python` on a command line. When it starts, you should see output like this:

```
Python 3.4.0 (default, Jun 19 2015, 14:20:21)
[GCC 4.8.2] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

The first three lines contain information about the interpreter and the operating system it's running on, so it might be different for you. But you should check that the version number, which is 3.4.0 in this example, begins with 3, which indicates that you are running Python 3. If it begins with 2, you are running (you guessed it) Python 2.

The last line is a **prompt** that indicates that the interpreter is ready for you to enter code. If you type a line of code and hit Enter, the interpreter displays the result:

```
>>> 1 + 1
2
```

Now you're ready to get started. From here on, I assume that you know how to start the Python interpreter and run code.

1.3 The first program

Traditionally, the first program you write in a new language is called “Hello, World!” because all it does is display the words “Hello, World!”. In Python, it looks like this:

```
>>> print('Hello, World!')
```

This is an example of a **print statement**, although it doesn’t actually print anything on paper. It displays a result on the screen. In this case, the result is the words

Hello, World!

The quotation marks in the program mark the beginning and end of the text to be displayed; they don’t appear in the result.

The parentheses indicate that `print` is a function. We’ll get to functions in Chapter 3.

In Python 2, the `print` statement is slightly different; it is not a function, so it doesn’t use parentheses.

```
>>> print 'Hello, World!'
```

This distinction will make more sense soon, but that’s enough to get started.

1.4 Arithmetic operators

After “Hello, World”, the next step is arithmetic. Python provides **operators**, which are special symbols that represent computations like addition and multiplication.

The operators `+`, `-`, and `*` perform addition, subtraction, and multiplication, as in the following examples:

```
>>> 40 + 2  
42  
>>> 43 - 1  
42  
>>> 6 * 7  
42
```

The operator `/` performs division:

```
>>> 84 / 2  
42.0
```

You might wonder why the result is `42.0` instead of `42`. I’ll explain in the next section.

Finally, the operator `**` performs exponentiation; that is, it raises a number to a power:

```
>>> 6**2 + 6  
42
```

In some other languages, `^` is used for exponentiation, but in Python it is a bitwise operator called XOR. If you are not familiar with bitwise operators, the result will surprise you:

```
>>> 6 ^ 2  
4
```

I won’t cover bitwise operators in this book, but you can read about them at <http://wiki.python.org/moin/BitwiseOperators>.

1.5 Values and types

A **value** is one of the basic things a program works with, like a letter or a number. Some values we have seen so far are `2`, `42.0`, and `'Hello, World!'`.

These values belong to different **types**: `2` is an **integer**, `42.0` is a **floating-point number**, and `'Hello, World!'` is a **string**, so-called because the letters it contains are strung together.

If you are not sure what type a value has, the interpreter can tell you:

```
>>> type(2)
<class 'int'>
>>> type(42.0)
<class 'float'>
>>> type('Hello, World!')
<class 'str'>
```

In these results, the word “class” is used in the sense of a category; a type is a category of values.

Not surprisingly, integers belong to the type `int`, strings belong to `str` and floating-point numbers belong to `float`.

What about values like `'2'` and `'42.0'`? They look like numbers, but they are in quotation marks like strings.

```
>>> type('2')
<class 'str'>
>>> type('42.0')
<class 'str'>
```

They’re strings.

When you type a large integer, you might be tempted to use commas between groups of digits, as in `1,000,000`. This is not a legal *integer* in Python, but it is legal:

```
>>> 1,000,000
(1, 0, 0)
```

That’s not what we expected at all! Python interprets `1,000,000` as a comma-separated sequence of integers. We’ll learn more about this kind of sequence later.

1.6 Formal and natural languages

Natural languages are the languages people speak, such as English, Spanish, and French. They were not designed by people (although people try to impose some order on them); they evolved naturally.

Formal languages are languages that are designed by people for specific applications. For example, the notation that mathematicians use is a formal language that is particularly good at denoting relationships among numbers and symbols. Chemists use a formal language to represent the chemical structure of molecules. And most importantly:

Programming languages are formal languages that have been designed to express computations.

Formal languages tend to have strict **syntax** rules that govern the structure of statements. For example, in mathematics the statement $3 + 3 = 6$ has correct syntax, but $3+ = 3\$6$ does not. In chemistry H_2O is a syntactically correct formula, but $_2Zz$ is not.

Syntax rules come in two flavors, pertaining to **tokens** and structure. Tokens are the basic elements of the language, such as words, numbers, and chemical elements. One of the problems with $3+ = 3\$6$ is that $\$$ is not a legal token in mathematics (at least as far as I know). Similarly, $_2Zz$ is not legal because there is no element with the abbreviation Zz .

The second type of syntax rule pertains to the way tokens are combined. The equation $3 + /3$ is illegal because even though $+$ and $/$ are legal tokens, you can't have one right after the other. Similarly, in a chemical formula the subscript comes after the element name, not before.

This is @ well-structured Engl\$h sentence with invalid t*kens in it. This sentence all valid tokens has, but invalid structure with.

When you read a sentence in English or a statement in a formal language, you have to figure out the structure (although in a natural language you do this subconsciously). This process is called **parsing**.

Although formal and natural languages have many features in common—tokens, structure, and syntax—there are some differences:

ambiguity: Natural languages are full of ambiguity, which people deal with by using contextual clues and other information. Formal languages are designed to be nearly or completely unambiguous, which means that any statement has exactly one meaning, regardless of context.

redundancy: In order to make up for ambiguity and reduce misunderstandings, natural languages employ lots of redundancy. As a result, they are often verbose. Formal languages are less redundant and more concise.

literalness: Natural languages are full of idiom and metaphor. If I say, "The penny dropped", there is probably no penny and nothing dropping (this idiom means that someone understood something after a period of confusion). Formal languages mean exactly what they say.

Because we all grow up speaking natural languages, it is sometimes hard to adjust to formal languages. The difference between formal and natural language is like the difference between poetry and prose, but more so:

Poetry: Words are used for their sounds as well as for their meaning, and the whole poem together creates an effect or emotional response. Ambiguity is not only common but often deliberate.

Prose: The literal meaning of words is more important, and the structure contributes more meaning. Prose is more amenable to analysis than poetry but still often ambiguous.

Programs: The meaning of a computer program is unambiguous and literal, and can be understood entirely by analysis of the tokens and structure.

Formal languages are more dense than natural languages, so it takes longer to read them. Also, the structure is important, so it is not always best to read from top to bottom, left to right. Instead, learn to parse the program in your head, identifying the tokens and interpreting the structure. Finally, the details matter. Small errors in spelling and punctuation, which you can get away with in natural languages, can make a big difference in a formal language.

1.7 Debugging

Programmers make mistakes. For whimsical reasons, programming errors are called **bugs** and the process of tracking them down is called **debugging**.

Programming, and especially debugging, sometimes brings out strong emotions. If you are struggling with a difficult bug, you might feel angry, despondent, or embarrassed.

There is evidence that people naturally respond to computers as if they were people. When they work well, we think of them as teammates, and when they are obstinate or rude, we respond to them the same way we respond to rude, obstinate people (Reeves and Nass, *The Media Equation: How People Treat Computers, Television, and New Media Like Real People and Places*).

Preparing for these reactions might help you deal with them. One approach is to think of the computer as an employee with certain strengths, like speed and precision, and particular weaknesses, like lack of empathy and inability to grasp the big picture.

Your job is to be a good manager: find ways to take advantage of the strengths and mitigate the weaknesses. And find ways to use your emotions to engage with the problem, without letting your reactions interfere with your ability to work effectively.

Learning to debug can be frustrating, but it is a valuable skill that is useful for many activities beyond programming. At the end of each chapter there is a section, like this one, with my suggestions for debugging. I hope they help!

1.8 Glossary

problem solving: The process of formulating a problem, finding a solution, and expressing it.

high-level language: A programming language like Python that is designed to be easy for humans to read and write.

low-level language: A programming language that is designed to be easy for a computer to run; also called “machine language” or “assembly language”.

portability: A property of a program that can run on more than one kind of computer.

interpreter: A program that reads another program and executes it

prompt: Characters displayed by the interpreter to indicate that it is ready to take input from the user.

program: A set of instructions that specifies a computation.

print statement: An instruction that causes the Python interpreter to display a value on the screen.

operator: A special symbol that represents a simple computation like addition, multiplication, or string concatenation.

value: One of the basic units of data, like a number or string, that a program manipulates.

type: A category of values. The types we have seen so far are integers (type `int`), floating-point numbers (type `float`), and strings (type `str`).

integer: A type that represents whole numbers.

floating-point: A type that represents numbers with fractional parts.

string: A type that represents sequences of characters.

natural language: Any one of the languages that people speak that evolved naturally.

formal language: Any one of the languages that people have designed for specific purposes, such as representing mathematical ideas or computer programs; all programming languages are formal languages.

token: One of the basic elements of the syntactic structure of a program, analogous to a word in a natural language.

syntax: The rules that govern the structure of a program.

parse: To examine a program and analyze the syntactic structure.

bug: An error in a program.

debugging: The process of finding and correcting bugs.

1.9 Exercises

Exercise 1.1. *It is a good idea to read this book in front of a computer so you can try out the examples as you go.*

Whenever you are experimenting with a new feature, you should try to make mistakes. For example, in the “Hello, world!” program, what happens if you leave out one of the quotation marks? What if you leave out both? What if you spell print wrong?

This kind of experiment helps you remember what you read; it also helps when you are programming, because you get to know what the error messages mean. It is better to make mistakes now and on purpose than later and accidentally.

1. In a `print` statement, what happens if you leave out one of the parentheses, or both?
2. If you are trying to print a string, what happens if you leave out one of the quotation marks, or both?
3. You can use a minus sign to make a negative number like `-2`. What happens if you put a plus sign before a number? What about `2++2`?

4. In math notation, leading zeros are ok, as in 09. What happens if you try this in Python? What about 011?

5. What happens if you have two values with no operator between them?

Exercise 1.2. Start the Python interpreter and use it as a calculator.

1. How many seconds are there in 42 minutes 42 seconds?

2. How many miles are there in 10 kilometers? Hint: there are 1.61 kilometers in a mile.

3. If you run a 10 kilometer race in 42 minutes 42 seconds, what is your average pace (time per mile in minutes and seconds)? What is your average speed in miles per hour?

Chapter 2

Variables, expressions and statements

One of the most powerful features of a programming language is the ability to manipulate **variables**. A variable is a name that refers to a value.

2.1 Assignment statements

An **assignment statement** creates a new variable and gives it a value:

```
>>> message = 'And now for something completely different'  
>>> n = 17  
>>> pi = 3.1415926535897932
```

This example makes three assignments. The first assigns a string to a new variable named `message`; the second gives the integer 17 to `n`; the third assigns the (approximate) value of π to `pi`.

A common way to represent variables on paper is to write the name with an arrow pointing to its value. This kind of figure is called a **state diagram** because it shows what state each of the variables is in (think of it as the variable's state of mind). Figure 2.1 shows the result of the previous example.

2.2 Variable names

Programmers generally choose names for their variables that are meaningful—they document what the variable is used for.

```
message —> 'And now for something completely different'  
n —> 17  
pi —> 3.1415926535897932
```

Figure 2.1: State diagram.

Variable names can be as long as you like. They can contain both letters and numbers, but they can't begin with a number. It is legal to use uppercase letters, but it is conventional to use only lower case for variables names.

The underscore character, `_`, can appear in a name. It is often used in names with multiple words, such as `your_name` or `airspeed_of_unladen_swallow`.

If you give a variable an illegal name, you get a syntax error:

```
>>> 76trombones = 'big parade'
SyntaxError: invalid syntax
>>> more@ = 1000000
SyntaxError: invalid syntax
>>> class = 'Advanced Theoretical Zymurgy'
SyntaxError: invalid syntax
```

`76trombones` is illegal because it begins with a number. `more@` is illegal because it contains an illegal character, `@`. But what's wrong with `class`?

It turns out that `class` is one of Python's **keywords**. The interpreter uses keywords to recognize the structure of the program, and they cannot be used as variable names.

Python 3 has these keywords:

False	class	finally	is	return
None	continue	for	lambda	try
True	def	from	nonlocal	while
and	del	global	not	with
as	elif	if	or	yield
assert	else	import	pass	
break	except	in	raise	

You don't have to memorize this list. In most development environments, keywords are displayed in a different color; if you try to use one as a variable name, you'll know.

2.3 Expressions and statements

An **expression** is a combination of values, variables, and operators. A value all by itself is considered an expression, and so is a variable, so the following are all legal expressions:

```
>>> 42
42
>>> n
17
>>> n + 25
42
```

When you type an expression at the prompt, the interpreter **evaluates** it, which means that it finds the value of the expression. In this example, `n` has the value 17 and `n + 25` has the value 42.

A **statement** is a unit of code that has an effect, like creating a variable or displaying a value.

```
>>> n = 17
>>> print(n)
```

The first line is an assignment statement that gives a value to `n`. The second line is a print statement that displays the value of `n`.

When you type a statement, the interpreter **executes** it, which means that it does whatever the statement says. In general, statements don't have values.

2.4 Script mode

So far we have run Python in **interactive mode**, which means that you interact directly with the interpreter. Interactive mode is a good way to get started, but if you are working with more than a few lines of code, it can be clumsy.

The alternative is to save code in a file called a **script** and then run the interpreter in **script mode** to execute the script. By convention, Python scripts have names that end with `.py`.

If you know how to create and run a script on your computer, you are ready to go. Otherwise I recommend using PythonAnywhere again. I have posted instructions for running in script mode at <http://tinyurl.com/thinkpython2e>.

Because Python provides both modes, you can test bits of code in interactive mode before you put them in a script. But there are differences between interactive mode and script mode that can be confusing.

For example, if you are using Python as a calculator, you might type

```
>>> miles = 26.2  
>>> miles * 1.61  
42.182
```

The first line assigns a value to `miles`, but it has no visible effect. The second line is an expression, so the interpreter evaluates it and displays the result. It turns out that a marathon is about 42 kilometers.

But if you type the same code into a script and run it, you get no output at all. In script mode an expression, all by itself, has no visible effect. Python evaluates the expression, but it doesn't display the result. To display the result, you need a `print` statement like this:

```
miles = 26.2  
print(miles * 1.61)
```

This behavior can be confusing at first. To check your understanding, type the following statements in the Python interpreter and see what they do:

```
5  
x = 5  
x + 1
```

Now put the same statements in a script and run it. What is the output? Modify the script by transforming each expression into a `print` statement and then run it again.

2.5 Order of operations

When an expression contains more than one operator, the order of evaluation depends on the **order of operations**. For mathematical operators, Python follows mathematical convention. The acronym **PEMDAS** is a useful way to remember the rules:

- Parentheses have the highest precedence and can be used to force an expression to evaluate in the order you want. Since expressions in parentheses are evaluated first, $2 * (3-1)$ is 4, and $(1+1)**(5-2)$ is 8. You can also use parentheses to make an expression easier to read, as in $(\text{minute} * 100) / 60$, even if it doesn't change the result.
- Exponentiation has the next highest precedence, so $1 + 2**3$ is 9, not 27, and $2 * 3**2$ is 18, not 36.
- Multiplication and Division have higher precedence than Addition and Subtraction. So $2*3-1$ is 5, not 4, and $6+4/2$ is 8, not 5.
- Operators with the same precedence are evaluated from left to right (except exponentiation). So in the expression $\text{degrees} / 2 * \pi$, the division happens first and the result is multiplied by π . To divide by 2π , you can use parentheses or write $\text{degrees} / 2 / \pi$.

I don't work very hard to remember the precedence of operators. If I can't tell by looking at the expression, I use parentheses to make it obvious.

2.6 String operations

In general, you can't perform mathematical operations on strings, even if the strings look like numbers, so the following are illegal:

```
'chinese' - 'food'      'eggs' / 'easy'      'third' * 'a charm'
```

But there are two exceptions, `+` and `*`.

The `+` operator performs **string concatenation**, which means it joins the strings by linking them end-to-end. For example:

```
>>> first = 'throat'
>>> second = 'warbler'
>>> first + second
throatwarbler
```

The `*` operator also works on strings; it performs repetition. For example, `'Spam'*3` is `'SpamSpamSpam'`. If one of the values is a string, the other has to be an integer.

This use of `+` and `*` makes sense by analogy with addition and multiplication. Just as $4*3$ is equivalent to $4+4+4$, we expect `'Spam'*3` to be the same as `'Spam'+'Spam'+'Spam'`, and it is. On the other hand, there is a significant way in which string concatenation and repetition are different from integer addition and multiplication. Can you think of a property that addition has that string concatenation does not?

2.7 Comments

As programs get bigger and more complicated, they get more difficult to read. Formal languages are dense, and it is often difficult to look at a piece of code and figure out what it is doing, or why.

For this reason, it is a good idea to add notes to your programs to explain in natural language what the program is doing. These notes are called **comments**, and they start with the # symbol:

```
# compute the percentage of the hour that has elapsed  
percentage = (minute * 100) / 60
```

In this case, the comment appears on a line by itself. You can also put comments at the end of a line:

```
percentage = (minute * 100) / 60      # percentage of an hour
```

Everything from the # to the end of the line is ignored—it has no effect on the execution of the program.

Comments are most useful when they document non-obvious features of the code. It is reasonable to assume that the reader can figure out *what* the code does; it is more useful to explain *why*.

This comment is redundant with the code and useless:

```
v = 5      # assign 5 to v
```

This comment contains useful information that is not in the code:

```
v = 5      # velocity in meters/second.
```

Good variable names can reduce the need for comments, but long names can make complex expressions hard to read, so there is a tradeoff.

2.8 Debugging

Three kinds of errors can occur in a program: syntax errors, runtime errors, and semantic errors. It is useful to distinguish between them in order to track them down more quickly.

Syntax error: “Syntax” refers to the structure of a program and the rules about that structure. For example, parentheses have to come in matching pairs, so `(1 + 2)` is legal, but `8)` is a **syntax error**.

If there is a syntax error anywhere in your program, Python displays an error message and quits, and you will not be able to run the program. During the first few weeks of your programming career, you might spend a lot of time tracking down syntax errors. As you gain experience, you will make fewer errors and find them faster.

Runtime error: The second type of error is a runtime error, so called because the error does not appear until after the program has started running. These errors are also called **exceptions** because they usually indicate that something exceptional (and bad) has happened.

Runtime errors are rare in the simple programs you will see in the first few chapters, so it might be a while before you encounter one.

Semantic error: The third type of error is “semantic”, which means related to meaning. If there is a semantic error in your program, it will run without generating error messages, but it will not do the right thing. It will do something else. Specifically, it will do what you told it to do.

Identifying semantic errors can be tricky because it requires you to work backward by looking at the output of the program and trying to figure out what it is doing.

2.9 Glossary

variable: A name that refers to a value.

assignment: A statement that assigns a value to a variable.

state diagram: A graphical representation of a set of variables and the values they refer to.

keyword: A reserved word that is used to parse a program; you cannot use keywords like `if`, `def`, and `while` as variable names.

operand: One of the values on which an operator operates.

expression: A combination of variables, operators, and values that represents a single result.

evaluate: To simplify an expression by performing the operations in order to yield a single value.

statement: A section of code that represents a command or action. So far, the statements we have seen are assignments and print statements.

execute: To run a statement and do what it says.

interactive mode: A way of using the Python interpreter by typing code at the prompt.

script mode: A way of using the Python interpreter to read code from a script and run it.

script: A program stored in a file.

order of operations: Rules governing the order in which expressions involving multiple operators and operands are evaluated.

concatenate: To join two operands end-to-end.

comment: Information in a program that is meant for other programmers (or anyone reading the source code) and has no effect on the execution of the program.

syntax error: An error in a program that makes it impossible to parse (and therefore impossible to interpret).

exception: An error that is detected while the program is running.

semantics: The meaning of a program.

semantic error: An error in a program that makes it do something other than what the programmer intended.

2.10 Exercises

Exercise 2.1. *Repeating my advice from the previous chapter, whenever you learn a new feature, you should try it out in interactive mode and make errors on purpose to see what goes wrong.*

- We've seen that `n = 42` is legal. What about `42 = n`?

- How about $x = y = 1$?
- In some languages every statement ends with a semi-colon, ;. What happens if you put a semi-colon at the end of a Python statement?
- What if you put a period at the end of a statement?
- In math notation you can multiply x and y like this: xy . What happens if you try that in Python?

Exercise 2.2. Practice using the Python interpreter as a calculator:

1. The volume of a sphere with radius r is $\frac{4}{3}\pi r^3$. What is the volume of a sphere with radius 5?
2. Suppose the cover price of a book is \$24.95, but bookstores get a 40% discount. Shipping costs \$3 for the first copy and 75 cents for each additional copy. What is the total wholesale cost for 60 copies?
3. If I leave my house at 6:52 am and run 1 mile at an easy pace (8:15 per mile), then 3 miles at tempo (7:12 per mile) and 1 mile at easy pace again, what time do I get home for breakfast?

Chapter 3

Functions

In the context of programming, a **function** is a named sequence of statements that performs a computation. When you define a function, you specify the name and the sequence of statements. Later, you can “call” the function by name.

3.1 Function calls

We have already seen one example of a **function call**:

```
>>> type(42)
<class 'int'>
```

The name of the function is `type`. The expression in parentheses is called the **argument** of the function. The result, for this function, is the type of the argument.

It is common to say that a function “takes” an argument and “returns” a result. The result is also called the **return value**.

Python provides functions that convert values from one type to another. The `int` function takes any value and converts it to an integer, if it can, or complains otherwise:

```
>>> int('32')
32
>>> int('Hello')
ValueError: invalid literal for int(): Hello
```

`int` can convert floating-point values to integers, but it doesn’t round off; it chops off the fraction part:

```
>>> int(3.99999)
```

```
3
```

```
>>> int(-2.3)
```

```
-2
```

`float` converts integers and strings to floating-point numbers:

```
>>> float(32)
```

```
32.0
```

```
>>> float('3.14159')
```

```
3.14159
```

Finally, `str` converts its argument to a string:

```
>>> str(32)
'32'
>>> str(3.14159)
'3.14159'
```

3.2 Math functions

Python has a `math` module that provides most of the familiar mathematical functions. A **module** is a file that contains a collection of related functions.

Before we can use the functions in a module, we have to import it with an **import statement**:

```
>>> import math
```

This statement creates a **module object** named `math`. If you display the module object, you get some information about it:

```
>>> math
<module 'math' (built-in)>
```

The module object contains the functions and variables defined in the module. To access one of the functions, you have to specify the name of the module and the name of the function, separated by a dot (also known as a period). This format is called **dot notation**.

```
>>> ratio = signal_power / noise_power
>>> decibels = 10 * math.log10(ratio)

>>> radians = 0.7
>>> height = math.sin(radians)
```

The first example uses `math.log10` to compute a signal-to-noise ratio in decibels (assuming that `signal_power` and `noise_power` are defined). The `math` module also provides `log`, which computes logarithms base e.

The second example finds the sine of `radians`. The variable name `radians` is a hint that `sin` and the other trigonometric functions (`cos`, `tan`, etc.) take arguments in radians. To convert from degrees to radians, divide by 180 and multiply by π :

```
>>> degrees = 45
>>> radians = degrees / 180.0 * math.pi
>>> math.sin(radians)
0.707106781187
```

The expression `math.pi` gets the variable `pi` from the `math` module. Its value is a floating-point approximation of π , accurate to about 15 digits.

If you know trigonometry, you can check the previous result by comparing it to the square root of two, divided by two:

```
>>> math.sqrt(2) / 2.0
0.707106781187
```

3.3 Composition

So far, we have looked at the elements of a program—variables, expressions, and statements—in isolation, without talking about how to combine them.

One of the most useful features of programming languages is their ability to take small building blocks and **compose** them. For example, the argument of a function can be any kind of expression, including arithmetic operators:

```
x = math.sin(degrees / 360.0 * 2 * math.pi)
```

And even function calls:

```
x = math.exp(math.log(x+1))
```

Almost anywhere you can put a value, you can put an arbitrary expression, with one exception: the left side of an assignment statement has to be a variable name. Any other expression on the left side is a syntax error (we will see exceptions to this rule later).

```
>>> minutes = hours * 60          # right
>>> hours * 60 = minutes        # wrong!
SyntaxError: can't assign to operator
```

3.4 Adding new functions

So far, we have only been using the functions that come with Python, but it is also possible to add new functions. A **function definition** specifies the name of a new function and the sequence of statements that run when the function is called.

Here is an example:

```
def print_lyrics():
    print("I'm a lumberjack, and I'm okay.")
    print("I sleep all night and I work all day.")
```

`def` is a keyword that indicates that this is a function definition. The name of the function is `print_lyrics`. The rules for function names are the same as for variable names: letters, numbers and underscore are legal, but the first character can't be a number. You can't use a keyword as the name of a function, and you should avoid having a variable and a function with the same name.

The empty parentheses after the name indicate that this function doesn't take any arguments.

The first line of the function definition is called the **header**; the rest is called the **body**. The header has to end with a colon and the body has to be indented. By convention, indentation is always four spaces. The body can contain any number of statements.

The strings in the `print` statements are enclosed in double quotes. Single quotes and double quotes do the same thing; most people use single quotes except in cases like this where a single quote (which is also an apostrophe) appears in the string.

All quotation marks (single and double) must be “straight quotes”, usually located next to Enter on the keyboard. “Curly quotes”, like the ones in this sentence, are not legal in Python.

If you type a function definition in interactive mode, the interpreter prints dots (...) to let you know that the definition isn't complete:

```
>>> def print_lyrics():
...     print("I'm a lumberjack, and I'm okay.")
...     print("I sleep all night and I work all day.")
...
```

To end the function, you have to enter an empty line.

Defining a function creates a **function object**, which has type `function`:

```
>>> print(print_lyrics)
<function print_lyrics at 0xb7e99e9c>
>>> type(print_lyrics)
<class 'function'>
```

The syntax for calling the new function is the same as for built-in functions:

```
>>> print_lyrics()
I'm a lumberjack, and I'm okay.
I sleep all night and I work all day.
```

Once you have defined a function, you can use it inside another function. For example, to repeat the previous refrain, we could write a function called `repeat_lyrics`:

```
def repeat_lyrics():
    print_lyrics()
    print_lyrics()
```

And then call `repeat_lyrics`:

```
>>> repeat_lyrics()
I'm a lumberjack, and I'm okay.
I sleep all night and I work all day.
I'm a lumberjack, and I'm okay.
I sleep all night and I work all day.
```

But that's not really how the song goes.

3.5 Definitions and uses

Pulling together the code fragments from the previous section, the whole program looks like this:

```
def print_lyrics():
    print("I'm a lumberjack, and I'm okay.")
    print("I sleep all night and I work all day.")

def repeat_lyrics():
    print_lyrics()
    print_lyrics()

repeat_lyrics()
```

This program contains two function definitions: `print_lyrics` and `repeat_lyrics`. Function definitions get executed just like other statements, but the effect is to create function objects. The statements inside the function do not run until the function is called, and the function definition generates no output.

As you might expect, you have to create a function before you can run it. In other words, the function definition has to run before the function gets called.

As an exercise, move the last line of this program to the top, so the function call appears before the definitions. Run the program and see what error message you get.

Now move the function call back to the bottom and move the definition of `print_lyrics` after the definition of `repeat_lyrics`. What happens when you run this program?

3.6 Flow of execution

To ensure that a function is defined before its first use, you have to know the order statements run in, which is called the **flow of execution**.

Execution always begins at the first statement of the program. Statements are run one at a time, in order from top to bottom.

Function definitions do not alter the flow of execution of the program, but remember that statements inside the function don't run until the function is called.

A function call is like a detour in the flow of execution. Instead of going to the next statement, the flow jumps to the body of the function, runs the statements there, and then comes back to pick up where it left off.

That sounds simple enough, until you remember that one function can call another. While in the middle of one function, the program might have to run the statements in another function. Then, while running that new function, the program might have to run yet another function!

Fortunately, Python is good at keeping track of where it is, so each time a function completes, the program picks up where it left off in the function that called it. When it gets to the end of the program, it terminates.

In summary, when you read a program, you don't always want to read from top to bottom. Sometimes it makes more sense if you follow the flow of execution.

3.7 Parameters and arguments

Some of the functions we have seen require arguments. For example, when you call `math.sin` you pass a number as an argument. Some functions take more than one argument: `math.pow` takes two, the base and the exponent.

Inside the function, the arguments are assigned to variables called **parameters**. Here is a definition for a function that takes an argument:

```
def print_twice(bruce):
    print(bruce)
    print(bruce)
```

This function assigns the argument to a parameter named `bruce`. When the function is called, it prints the value of the parameter (whatever it is) twice.

This function works with any value that can be printed.

```
>>> print_twice('Spam')
Spam
Spam
>>> print_twice(42)
42
42
>>> print_twice(math.pi)
3.14159265359
3.14159265359
```

The same rules of composition that apply to built-in functions also apply to programmer-defined functions, so we can use any kind of expression as an argument for `print_twice`:

```
>>> print_twice('Spam '*4)
Spam Spam Spam Spam
Spam Spam Spam Spam
>>> print_twice(math.cos(math.pi))
-1.0
-1.0
```

The argument is evaluated before the function is called, so in the examples the expressions '`'Spam '`*4' and `math.cos(math.pi)` are only evaluated once.

You can also use a variable as an argument:

```
>>> michael = 'Eric, the half a bee.'
>>> print_twice(michael)
Eric, the half a bee.
Eric, the half a bee.
```

The name of the variable we pass as an argument (`michael`) has nothing to do with the name of the parameter (`bruce`). It doesn't matter what the value was called back home (in the caller); here in `print_twice`, we call everybody `bruce`.

3.8 Variables and parameters are local

When you create a variable inside a function, it is **local**, which means that it only exists inside the function. For example:

```
def cat_twice(part1, part2):
    cat = part1 + part2
    print_twice(cat)
```

This function takes two arguments, concatenates them, and prints the result twice. Here is an example that uses it:

```
>>> line1 = 'Bing tiddle '
>>> line2 = 'tiddle bang.'
>>> cat_twice(line1, line2)
Bing tiddle tiddle bang.
Bing tiddle tiddle bang.
```

When `cat_twice` terminates, the variable `cat` is destroyed. If we try to print it, we get an exception:

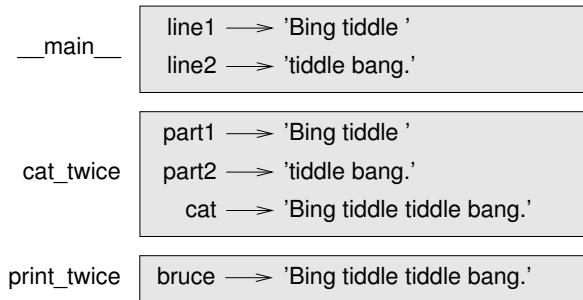


Figure 3.1: Stack diagram.

```

>>> print(cat)
NameError: name 'cat' is not defined
Parameters are also local. For example, outside print_twice, there is no such thing as
bruce.

```

Parameters are also local. For example, outside `print_twice`, there is no such thing as `bruce`.

3.9 Stack diagrams

To keep track of which variables can be used where, it is sometimes useful to draw a **stack diagram**. Like state diagrams, stack diagrams show the value of each variable, but they also show the function each variable belongs to.

Each function is represented by a **frame**. A frame is a box with the name of a function beside it and the parameters and variables of the function inside it. The stack diagram for the previous example is shown in Figure 3.1.

The frames are arranged in a stack that indicates which function called which, and so on. In this example, `print_twice` was called by `cat_twice`, and `cat_twice` was called by `__main__`, which is a special name for the topmost frame. When you create a variable outside of any function, it belongs to `__main__`.

Each parameter refers to the same value as its corresponding argument. So, `part1` has the same value as `line1`, `part2` has the same value as `line2`, and `bruce` has the same value as `cat`.

If an error occurs during a function call, Python prints the name of the function, the name of the function that called it, and the name of the function that called *that*, all the way back to `__main__`.

For example, if you try to access `cat` from within `print_twice`, you get a `NameError`:

```

Traceback (innermost last):
  File "test.py", line 13, in __main__
    cat_twice(line1, line2)
  File "test.py", line 5, in cat_twice
    print_twice(cat)
  File "test.py", line 9, in print_twice
    print(cat)
NameError: name 'cat' is not defined

```

This list of functions is called a **traceback**. It tells you what program file the error occurred in, and what line, and what functions were executing at the time. It also shows the line of code that caused the error.

The order of the functions in the traceback is the same as the order of the frames in the stack diagram. The function that is currently running is at the bottom.

3.10 Fruitful functions and void functions

Some of the functions we have used, such as the math functions, return results; for lack of a better name, I call them **fruitful functions**. Other functions, like `print_twice`, perform an action but don't return a value. They are called **void functions**.

When you call a fruitful function, you almost always want to do something with the result; for example, you might assign it to a variable or use it as part of an expression:

```
x = math.cos(radians)
golden = (math.sqrt(5) + 1) / 2
```

When you call a function in interactive mode, Python displays the result:

```
>>> math.sqrt(5)
2.2360679774997898
```

But in a script, if you call a fruitful function all by itself, the return value is lost forever!

```
math.sqrt(5)
```

This script computes the square root of 5, but since it doesn't store or display the result, it is not very useful.

Void functions might display something on the screen or have some other effect, but they don't have a return value. If you assign the result to a variable, you get a special value called `None`.

```
>>> result = print_twice('Bing')
Bing
Bing
>>> print(result)
None
```

The value `None` is not the same as the string '`None`'. It is a special value that has its own type:

```
>>> type(None)
<class 'NoneType'>
```

The functions we have written so far are all void. We will start writing fruitful functions in a few chapters.

3.11 Why functions?

It may not be clear why it is worth the trouble to divide a program into functions. There are several reasons:

- Creating a new function gives you an opportunity to name a group of statements, which makes your program easier to read and debug.
- Functions can make a program smaller by eliminating repetitive code. Later, if you make a change, you only have to make it in one place.
- Dividing a long program into functions allows you to debug the parts one at a time and then assemble them into a working whole.
- Well-designed functions are often useful for many programs. Once you write and debug one, you can reuse it.

3.12 Debugging

One of the most important skills you will acquire is debugging. Although it can be frustrating, debugging is one of the most intellectually rich, challenging, and interesting parts of programming.

In some ways debugging is like detective work. You are confronted with clues and you have to infer the processes and events that led to the results you see.

Debugging is also like an experimental science. Once you have an idea about what is going wrong, you modify your program and try again. If your hypothesis was correct, you can predict the result of the modification, and you take a step closer to a working program. If your hypothesis was wrong, you have to come up with a new one. As Sherlock Holmes pointed out, “When you have eliminated the impossible, whatever remains, however improbable, must be the truth.” (A. Conan Doyle, *The Sign of Four*)

For some people, programming and debugging are the same thing. That is, programming is the process of gradually debugging a program until it does what you want. The idea is that you should start with a working program and make small modifications, debugging them as you go.

For example, Linux is an operating system that contains millions of lines of code, but it started out as a simple program Linus Torvalds used to explore the Intel 80386 chip. According to Larry Greenfield, “One of Linus’s earlier projects was a program that would switch between printing AAAA and BBBB. This later evolved to Linux.” (*The Linux Users’ Guide* Beta Version 1).

3.13 Glossary

function: A named sequence of statements that performs some useful operation. Functions may or may not take arguments and may or may not produce a result.

function definition: A statement that creates a new function, specifying its name, parameters, and the statements it contains.

function object: A value created by a function definition. The name of the function is a variable that refers to a function object.

header: The first line of a function definition.

body: The sequence of statements inside a function definition.

parameter: A name used inside a function to refer to the value passed as an argument.

function call: A statement that runs a function. It consists of the function name followed by an argument list in parentheses.

argument: A value provided to a function when the function is called. This value is assigned to the corresponding parameter in the function.

local variable: A variable defined inside a function. A local variable can only be used inside its function.

return value: The result of a function. If a function call is used as an expression, the return value is the value of the expression.

fruitful function: A function that returns a value.

void function: A function that always returns None.

None: A special value returned by void functions.

module: A file that contains a collection of related functions and other definitions.

import statement: A statement that reads a module file and creates a module object.

module object: A value created by an import statement that provides access to the values defined in a module.

dot notation: The syntax for calling a function in another module by specifying the module name followed by a dot (period) and the function name.

composition: Using an expression as part of a larger expression, or a statement as part of a larger statement.

flow of execution: The order statements run in.

stack diagram: A graphical representation of a stack of functions, their variables, and the values they refer to.

frame: A box in a stack diagram that represents a function call. It contains the local variables and parameters of the function.

traceback: A list of the functions that are executing, printed when an exception occurs.

3.14 Exercises

Exercise 3.1. Write a function named `right_justify` that takes a string named `s` as a parameter and prints the string with enough leading spaces so that the last letter of the string is in column 70 of the display.

```
>>> right_justify('monty')
                           monty
```

Hint: Use string concatenation and repetition. Also, Python provides a built-in function called `len` that returns the length of a string, so the value of `len('monty')` is 5.

Exercise 3.2. A function object is a value you can assign to a variable or pass as an argument. For example, `do_twice` is a function that takes a function object as an argument and calls it twice:

```
def do_twice(f):
    f()
    f()
```

Here's an example that uses `do_twice` to call a function named `print_spam` twice.

```
def print_spam():
    print('spam')

do_twice(print_spam)
```

1. Type this example into a script and test it.
2. Modify `do_twice` so that it takes two arguments, a function object and a value, and calls the function twice, passing the value as an argument.
3. Copy the definition of `print_twice` from earlier in this chapter to your script.
4. Use the modified version of `do_twice` to call `print_twice` twice, passing '`spam`' as an argument.
5. Define a new function called `do_four` that takes a function object and a value and calls the function four times, passing the value as a parameter. There should be only two statements in the body of this function, not four.

Solution: http://thinkpython2.com/code/do_four.py.

Exercise 3.3. Note: This exercise should be done using only the statements and other features we have learned so far.

1. Write a function that draws a grid like the following:

```
+ - - - - + - - - - +
|           |           |
|           |           |
|           |           |
|           |           |
|           |           |
+ - - - - + - - - - +
|           |           |
|           |           |
|           |           |
|           |           |
|           |           |
+ - - - - + - - - - +
```

Hint: to print more than one value on a line, you can print a comma-separated sequence of values:

```
print('+', '-')
```

By default, `print` advances to the next line, but you can override that behavior and put a space at the end, like this:

```
print('+', end=' ')
print('-')
```

The output of these statements is '+ -' on the same line. The output from the next print statement would begin on the next line.

2. Write a function that draws a similar grid with four rows and four columns.

Solution: <http://thinkpython2.com/code/grid.py>. Credit: This exercise is based on an exercise in Oualline, Practical C Programming, Third Edition, O'Reilly Media, 1997.

Chapter 4

Case study: interface design

This chapter presents a case study that demonstrates a process for designing functions that work together.

It introduces the `turtle` module, which allows you to create images using turtle graphics. The `turtle` module is included in most Python installations, but if you are running Python using PythonAnywhere, you won't be able to run the turtle examples (at least you couldn't when I wrote this).

If you have already installed Python on your computer, you should be able to run the examples. Otherwise, now is a good time to install. I have posted instructions at <http://tinyurl.com/thinkpython2e>.

Code examples from this chapter are available from <http://thinkpython2.com/code/polygon.py>.

4.1 The turtle module

To check whether you have the `turtle` module, open the Python interpreter and type

```
>>> import turtle  
>>> bob = turtle.Turtle()
```

When you run this code, it should create a new window with small arrow that represents the turtle. Close the window.

Create a file named `mypolygon.py` and type in the following code:

```
import turtle  
bob = turtle.Turtle()  
print(bob)  
turtle.mainloop()
```

The `turtle` module (with a lowercase 't') provides a function called `Turtle` (with an uppercase 'T') that creates a `Turtle` object, which we assign to a variable named `bob`. Printing `bob` displays something like:

```
<turtle.Turtle object at 0xb7bfbf4c>
```

This means that `bob` refers to an object with type `Turtle` as defined in module `turtle`.

`mainloop` tells the window to wait for the user to do something, although in this case there's not much for the user to do except close the window.

Once you create a `Turtle`, you can call a **method** to move it around the window. A method is similar to a function, but it uses slightly different syntax. For example, to move the turtle forward:

```
bob.fd(100)
```

The method, `fd`, is associated with the `turtle` object we're calling `bob`. Calling a method is like making a request: you are asking `bob` to move forward.

The argument of `fd` is a distance in pixels, so the actual size depends on your display.

Other methods you can call on a `Turtle` are `bk` to move backward, `lt` for left turn, and `rt` right turn. The argument for `lt` and `rt` is an angle in degrees.

Also, each `Turtle` is holding a pen, which is either down or up; if the pen is down, the `Turtle` leaves a trail when it moves. The methods `pu` and `pd` stand for "pen up" and "pen down".

To draw a right angle, add these lines to the program (after creating `bob` and before calling `mainloop`):

```
bob.fd(100)
bob.lt(90)
bob.fd(100)
```

When you run this program, you should see `bob` move east and then north, leaving two line segments behind.

Now modify the program to draw a square. Don't go on until you've got it working!

4.2 Simple repetition

Chances are you wrote something like this:

```
bob.fd(100)
bob.lt(90)

bob.fd(100)
bob.lt(90)

bob.fd(100)
bob.lt(90)

bob.fd(100)
```

We can do the same thing more concisely with a `for` statement. Add this example to `mypolygon.py` and run it again:

```
for i in range(4):
    print('Hello!')
```

You should see something like this:

```
Hello!  
Hello!  
Hello!  
Hello!
```

This is the simplest use of the `for` statement; we will see more later. But that should be enough to let you rewrite your square-drawing program. Don't go on until you do.

Here is a `for` statement that draws a square:

```
for i in range(4):  
    bob.fd(100)  
    bob.lt(90)
```

The syntax of a `for` statement is similar to a function definition. It has a header that ends with a colon and an indented body. The body can contain any number of statements.

A `for` statement is also called a **loop** because the flow of execution runs through the body and then loops back to the top. In this case, it runs the body four times.

This version is actually a little different from the previous square-drawing code because it makes another turn after drawing the last side of the square. The extra turn takes more time, but it simplifies the code if we do the same thing every time through the loop. This version also has the effect of leaving the turtle back in the starting position, facing in the starting direction.

4.3 Exercises

The following is a series of exercises using the `turtle` module. They are meant to be fun, but they have a point, too. While you are working on them, think about what the point is.

The following sections have solutions to the exercises, so don't look until you have finished (or at least tried).

1. Write a function called `square` that takes a parameter named `t`, which is a turtle. It should use the turtle to draw a square.

Write a function call that passes `bob` as an argument to `square`, and then run the program again.

2. Add another parameter, named `length`, to `square`. Modify the body so length of the sides is `length`, and then modify the function call to provide a second argument. Run the program again. Test your program with a range of values for `length`.

3. Make a copy of `square` and change the name to `polygon`. Add another parameter named `n` and modify the body so it draws an `n`-sided regular polygon. Hint: The exterior angles of an `n`-sided regular polygon are $360/n$ degrees.

4. Write a function called `circle` that takes a turtle, `t`, and radius, `r`, as parameters and that draws an approximate circle by calling `polygon` with an appropriate length and number of sides. Test your function with a range of values of `r`.

Hint: figure out the circumference of the circle and make sure that `length * n = circumference`.

5. Make a more general version of `circle` called `arc` that takes an additional parameter `angle`, which determines what fraction of a circle to draw. `angle` is in units of degrees, so when `angle=360`, `arc` should draw a complete circle.

4.4 Encapsulation

The first exercise asks you to put your square-drawing code into a function definition and then call the function, passing the turtle as a parameter. Here is a solution:

```
def square(t):
    for i in range(4):
        t.fd(100)
        t.lt(90)

square(bob)
```

The innermost statements, `fd` and `lt` are indented twice to show that they are inside the `for` loop, which is inside the function definition. The next line, `square(bob)`, is flush with the left margin, which indicates the end of both the `for` loop and the function definition.

Inside the function, `t` refers to the same turtle `bob`, so `t.lt(90)` has the same effect as `bob.lt(90)`. In that case, why not call the parameter `bob`? The idea is that `t` can be any turtle, not just `bob`, so you could create a second turtle and pass it as an argument to `square`:

```
alice = turtle.Turtle()
square(alice)
```

Wrapping a piece of code up in a function is called **encapsulation**. One of the benefits of encapsulation is that it attaches a name to the code, which serves as a kind of documentation. Another advantage is that if you re-use the code, it is more concise to call a function twice than to copy and paste the body!

4.5 Generalization

The next step is to add a `length` parameter to `square`. Here is a solution:

```
def square(t, length):
    for i in range(4):
        t.fd(length)
        t.lt(90)

square(bob, 100)
```

Adding a parameter to a function is called **generalization** because it makes the function more general: in the previous version, the square is always the same size; in this version it can be any size.

The next step is also a generalization. Instead of drawing squares, `polygon` draws regular polygons with any number of sides. Here is a solution:

```
def polygon(t, n, length):
    angle = 360 / n
    for i in range(n):
        t.fd(length)
        t.lt(angle)

polygon(bob, 7, 70)
```

This example draws a 7-sided polygon with side length 70.

If you are using Python 2, the value of `angle` might be off because of integer division. A simple solution is to compute `angle = 360.0 / n`. Because the numerator is a floating-point number, the result is floating point.

When a function has more than a few numeric arguments, it is easy to forget what they are, or what order they should be in. In that case it is often a good idea to include the names of the parameters in the argument list:

```
polygon(bob, n=7, length=70)
```

These are called **keyword arguments** because they include the parameter names as “keywords” (not to be confused with Python keywords like `while` and `def`).

This syntax makes the program more readable. It is also a reminder about how arguments and parameters work: when you call a function, the arguments are assigned to the parameters.

4.6 Interface design

The next step is to write `circle`, which takes a radius, `r`, as a parameter. Here is a simple solution that uses `polygon` to draw a 50-sided polygon:

```
import math

def circle(t, r):
    circumference = 2 * math.pi * r
    n = 50
    length = circumference / n
    polygon(t, n, length)
```

The first line computes the circumference of a circle with radius `r` using the formula $2\pi r$. Since we use `math.pi`, we have to import `math`. By convention, `import` statements are usually at the beginning of the script.

`n` is the number of line segments in our approximation of a circle, so `length` is the length of each segment. Thus, `polygon` draws a 50-sided polygon that approximates a circle with radius `r`.

One limitation of this solution is that `n` is a constant, which means that for very big circles, the line segments are too long, and for small circles, we waste time drawing very small segments. One solution would be to generalize the function by taking `n` as a parameter. This would give the user (whoever calls `circle`) more control, but the interface would be less clean.

The **interface** of a function is a summary of how it is used: what are the parameters? What does the function do? And what is the return value? An interface is “clean” if it allows the caller to do what they want without dealing with unnecessary details.

In this example, `r` belongs in the interface because it specifies the circle to be drawn. `n` is less appropriate because it pertains to the details of *how* the circle should be rendered.

Rather than clutter up the interface, it is better to choose an appropriate value of `n` depending on `circumference`:

```
def circle(t, r):
    circumference = 2 * math.pi * r
    n = int(circumference / 3) + 3
    length = circumference / n
    polygon(t, n, length)
```

Now the number of segments is an integer near `circumference/3`, so the length of each segment is approximately 3, which is small enough that the circles look good, but big enough to be efficient, and acceptable for any size circle.

Adding 3 to `n` guarantees that the polygon has at least 3 sides.

4.7 Refactoring

When I wrote `circle`, I was able to re-use `polygon` because a many-sided polygon is a good approximation of a circle. But `arc` is not as cooperative; we can't use `polygon` or `circle` to draw an arc.

One alternative is to start with a copy of `polygon` and transform it into `arc`. The result might look like this:

```
def arc(t, r, angle):
    arc_length = 2 * math.pi * r * angle / 360
    n = int(arc_length / 3) + 1
    step_length = arc_length / n
    step_angle = angle / n

    for i in range(n):
        t.fd(step_length)
        t.lt(step_angle)
```

The second half of this function looks like `polygon`, but we can't re-use `polygon` without changing the interface. We could generalize `polygon` to take an angle as a third argument, but then `polygon` would no longer be an appropriate name! Instead, let's call the more general function `polyline`:

```
def polyline(t, n, length, angle):
    for i in range(n):
        t.fd(length)
        t.lt(angle)
```

Now we can rewrite `polygon` and `arc` to use `polyline`:

```
def polygon(t, n, length):
    angle = 360.0 / n
    polyline(t, n, length, angle)

def arc(t, r, angle):
    arc_length = 2 * math.pi * r * angle / 360
    n = int(arc_length / 3) + 1
    step_length = arc_length / n
    step_angle = float(angle) / n
    polyline(t, n, step_length, step_angle)
```

Finally, we can rewrite `circle` to use `arc`:

```
def circle(t, r):
    arc(t, r, 360)
```

This process—rearranging a program to improve interfaces and facilitate code re-use—is called **refactoring**. In this case, we noticed that there was similar code in `arc` and `polygon`, so we “factored it out” into `polyline`.

If we had planned ahead, we might have written `polyline` first and avoided refactoring, but often you don’t know enough at the beginning of a project to design all the interfaces. Once you start coding, you understand the problem better. Sometimes refactoring is a sign that you have learned something.

4.8 A development plan

A **development plan** is a process for writing programs. The process we used in this case study is “encapsulation and generalization”. The steps of this process are:

1. Start by writing a small program with no function definitions.
2. Once you get the program working, identify a coherent piece of it, encapsulate the piece in a function and give it a name.
3. Generalize the function by adding appropriate parameters.
4. Repeat steps 1–3 until you have a set of working functions. Copy and paste working code to avoid retyping (and re-debugging).
5. Look for opportunities to improve the program by refactoring. For example, if you have similar code in several places, consider factoring it into an appropriately general function.

This process has some drawbacks—we will see alternatives later—but it can be useful if you don’t know ahead of time how to divide the program into functions. This approach lets you design as you go along.

4.9 docstring

A **docstring** is a string at the beginning of a function that explains the interface (“doc” is short for “documentation”). Here is an example:

```
def polyline(t, n, length, angle):
    """Draws n line segments with the given length and
    angle (in degrees) between them. t is a turtle.
    """
    for i in range(n):
        t.fd(length)
        t.lt(angle)
```

By convention, all docstrings are triple-quoted strings, also known as multiline strings because the triple quotes allow the string to span more than one line.

It is terse, but it contains the essential information someone would need to use this function. It explains concisely what the function does (without getting into the details of how it does it). It explains what effect each parameter has on the behavior of the function and what type each parameter should be (if it is not obvious).

Writing this kind of documentation is an important part of interface design. A well-designed interface should be simple to explain; if you have a hard time explaining one of your functions, maybe the interface could be improved.

4.10 Debugging

An interface is like a contract between a function and a caller. The caller agrees to provide certain parameters and the function agrees to do certain work.

For example, `polyline` requires four arguments: `t` has to be a Turtle; `n` has to be an integer; `length` should be a positive number; and `angle` has to be a number, which is understood to be in degrees.

These requirements are called **preconditions** because they are supposed to be true before the function starts executing. Conversely, conditions at the end of the function are **postconditions**. Postconditions include the intended effect of the function (like drawing line segments) and any side effects (like moving the Turtle or making other changes).

Preconditions are the responsibility of the caller. If the caller violates a (properly documented!) precondition and the function doesn't work correctly, the bug is in the caller, not the function.

If the preconditions are satisfied and the postconditions are not, the bug is in the function. If your pre- and postconditions are clear, they can help with debugging.

4.11 Glossary

method: A function that is associated with an object and called using dot notation.

loop: A part of a program that can run repeatedly.

encapsulation: The process of transforming a sequence of statements into a function definition.

generalization: The process of replacing something unnecessarily specific (like a number) with something appropriately general (like a variable or parameter).

keyword argument: An argument that includes the name of the parameter as a "keyword".

interface: A description of how to use a function, including the name and descriptions of the arguments and return value.

refactoring: The process of modifying a working program to improve function interfaces and other qualities of the code.

development plan: A process for writing programs.

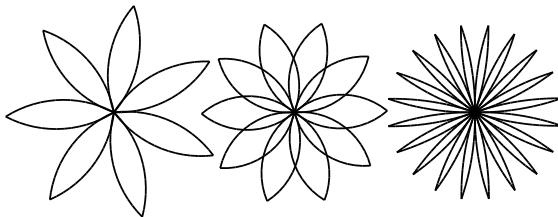


Figure 4.1: Turtle flowers.

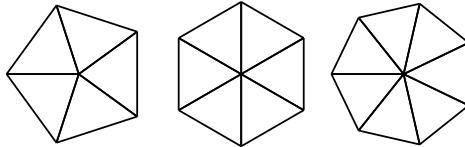


Figure 4.2: Turtle pies.

docstring: A string that appears at the top of a function definition to document the function's interface.

precondition: A requirement that should be satisfied by the caller before a function starts.

postcondition: A requirement that should be satisfied by the function before it ends.

4.12 Exercises

Exercise 4.1. Download the code in this chapter from <http://thinkpython2.com/code/polygon.py>.

1. Draw a stack diagram that shows the state of the program while executing `circle(bob, radius)`. You can do the arithmetic by hand or add print statements to the code.
2. The version of `arc` in Section 4.7 is not very accurate because the linear approximation of the circle is always outside the true circle. As a result, the Turtle ends up a few pixels away from the correct destination. My solution shows a way to reduce the effect of this error. Read the code and see if it makes sense to you. If you draw a diagram, you might see how it works.

Exercise 4.2. Write an appropriately general set of functions that can draw flowers as in Figure 4.1.

Solution: <http://thinkpython2.com/code/flower.py>, also requires <http://thinkpython2.com/code/polygon.py>.

Exercise 4.3. Write an appropriately general set of functions that can draw shapes as in Figure 4.2.

Solution: <http://thinkpython2.com/code/pie.py>.

Exercise 4.4. The letters of the alphabet can be constructed from a moderate number of basic elements, like vertical and horizontal lines and a few curves. Design an alphabet that can be drawn with a minimal number of basic elements and then write functions that draw the letters.

You should write one function for each letter, with names `draw_a`, `draw_b`, etc., and put your functions in a file named `letters.py`. You can download a “turtle typewriter” from <http://thinkpython2.com/code/typewriter.py> to help you test your code.

You can get a solution from <http://thinkpython2.com/code/letters.py>; it also requires <http://thinkpython2.com/code/polygon.py>.

Exercise 4.5. Read about spirals at <http://en.wikipedia.org/wiki/Spiral>; then write a program that draws an Archimedian spiral (or one of the other kinds). Solution: <http://thinkpython2.com/code/spiral.py>.

Chapter 5

Conditionals and recursion

The main topic of this chapter is the `if` statement, which executes different code depending on the state of the program. But first I want to introduce two new operators: floor division and modulus.

5.1 Floor division and modulus

The **floor division** operator, `//`, divides two numbers and rounds down to an integer. For example, suppose the run time of a movie is 105 minutes. You might want to know how long that is in hours. Conventional division returns a floating-point number:

```
>>> minutes = 105  
>>> minutes / 60  
1.75
```

But we don't normally write hours with decimal points. Floor division returns the integer number of hours, rounding down:

```
>>> minutes = 105  
>>> hours = minutes // 60  
>>> hours  
1
```

To get the remainder, you could subtract off one hour in minutes:

```
>>> remainder = minutes - hours * 60  
>>> remainder  
45
```

An alternative is to use the **modulus operator**, `%`, which divides two numbers and returns the remainder.

```
>>> remainder = minutes % 60  
>>> remainder  
45
```

The modulus operator is more useful than it seems. For example, you can check whether one number is divisible by another—if `x % y` is zero, then `x` is divisible by `y`.

Also, you can extract the right-most digit or digits from a number. For example, `x % 10` yields the right-most digit of `x` (in base 10). Similarly `x % 100` yields the last two digits.

If you are using Python 2, division works differently. The division operator, `/`, performs floor division if both operands are integers, and floating-point division if either operand is a `float`.

5.2 Boolean expressions

A **boolean expression** is an expression that is either true or false. The following examples use the operator `==`, which compares two operands and produces `True` if they are equal and `False` otherwise:

```
>>> 5 == 5
True
>>> 5 == 6
False
```

`True` and `False` are special values that belong to the type `bool`; they are not strings:

```
>>> type(True)
<class 'bool'>
>>> type(False)
<class 'bool'>
```

The `==` operator is one of the **relational operators**; the others are:

<code>x != y</code>	# <code>x</code> is not equal to <code>y</code>
<code>x > y</code>	# <code>x</code> is greater than <code>y</code>
<code>x < y</code>	# <code>x</code> is less than <code>y</code>
<code>x >= y</code>	# <code>x</code> is greater than or equal to <code>y</code>
<code>x <= y</code>	# <code>x</code> is less than or equal to <code>y</code>

Although these operations are probably familiar to you, the Python symbols are different from the mathematical symbols. A common error is to use a single equal sign (`=`) instead of a double equal sign (`==`). Remember that `=` is an assignment operator and `==` is a relational operator. There is no such thing as `=<` or `=>`.

5.3 Logical operators

There are three **logical operators**: `and`, `or`, and `not`. The semantics (meaning) of these operators is similar to their meaning in English. For example, `x > 0` and `x < 10` is true only if `x` is greater than 0 *and* less than 10.

`n%2 == 0 or n%3 == 0` is true if *either or both* of the conditions is true, that is, if the number is divisible by 2 *or* 3.

Finally, the `not` operator negates a boolean expression, so `not (x > y)` is true if `x > y` is false, that is, if `x` is less than or equal to `y`.

Strictly speaking, the operands of the logical operators should be boolean expressions, but Python is not very strict. Any nonzero number is interpreted as `True`:

```
>>> 42 and True  
True
```

This flexibility can be useful, but there are some subtleties to it that might be confusing. You might want to avoid it (unless you know what you are doing).

5.4 Conditional execution

In order to write useful programs, we almost always need the ability to check conditions and change the behavior of the program accordingly. **Conditional statements** give us this ability. The simplest form is the `if` statement:

```
if x > 0:  
    print('x is positive')
```

The boolean expression after `if` is called the **condition**. If it is true, the indented statement runs. If not, nothing happens.

`if` statements have the same structure as function definitions: a header followed by an indented body. Statements like this are called **compound statements**.

There is no limit on the number of statements that can appear in the body, but there has to be at least one. Occasionally, it is useful to have a body with no statements (usually as a place keeper for code you haven't written yet). In that case, you can use the `pass` statement, which does nothing.

```
if x < 0:  
    pass          # TODO: need to handle negative values!
```

5.5 Alternative execution

A second form of the `if` statement is "alternative execution", in which there are two possibilities and the condition determines which one runs. The syntax looks like this:

```
if x % 2 == 0:  
    print('x is even')  
else:  
    print('x is odd')
```

If the remainder when `x` is divided by 2 is 0, then we know that `x` is even, and the program displays an appropriate message. If the condition is false, the second set of statements runs. Since the condition must be true or false, exactly one of the alternatives will run. The alternatives are called **branches**, because they are branches in the flow of execution.

5.6 Chained conditionals

Sometimes there are more than two possibilities and we need more than two branches. One way to express a computation like that is a **chained conditional**:

```
if x < y:
    print('x is less than y')
elif x > y:
    print('x is greater than y')
else:
    print('x and y are equal')
```

`elif` is an abbreviation of “else if”. Again, exactly one branch will run. There is no limit on the number of `elif` statements. If there is an `else` clause, it has to be at the end, but there doesn’t have to be one.

```
if choice == 'a':
    draw_a()
elif choice == 'b':
    draw_b()
elif choice == 'c':
    draw_c()
```

Each condition is checked in order. If the first is false, the next is checked, and so on. If one of them is true, the corresponding branch runs and the statement ends. Even if more than one condition is true, only the first true branch runs.

5.7 Nested conditionals

One conditional can also be nested within another. We could have written the example in the previous section like this:

```
if x == y:
    print('x and y are equal')
else:
    if x < y:
        print('x is less than y')
    else:
        print('x is greater than y')
```

The outer conditional contains two branches. The first branch contains a simple statement. The second branch contains another `if` statement, which has two branches of its own. Those two branches are both simple statements, although they could have been conditional statements as well.

Although the indentation of the statements makes the structure apparent, **nested conditionals** become difficult to read very quickly. It is a good idea to avoid them when you can.

Logical operators often provide a way to simplify nested conditional statements. For example, we can rewrite the following code using a single conditional:

```
if 0 < x:
    if x < 10:
        print('x is a positive single-digit number.')
```

The `print` statement runs only if we make it past both conditionals, so we can get the same effect with the `and` operator:

```
if 0 < x and x < 10:
    print('x is a positive single-digit number.')
```

For this kind of condition, Python provides a more concise option:

```
if 0 < x < 10:  
    print('x is a positive single-digit number.')
```

5.8 Recursion

It is legal for one function to call another; it is also legal for a function to call itself. It may not be obvious why that is a good thing, but it turns out to be one of the most magical things a program can do. For example, look at the following function:

```
def countdown(n):  
    if n <= 0:  
        print('Blastoff!')  
    else:  
        print(n)  
        countdown(n-1)
```

If n is 0 or negative, it outputs the word, “Blastoff!” Otherwise, it outputs n and then calls a function named `countdown`—itself—passing $n-1$ as an argument.

What happens if we call this function like this?

```
>>> countdown(3)
```

The execution of `countdown` begins with $n=3$, and since n is greater than 0, it outputs the value 3, and then calls itself...

The execution of `countdown` begins with $n=2$, and since n is greater than 0, it outputs the value 2, and then calls itself...

The execution of `countdown` begins with $n=1$, and since n is greater than 0, it outputs the value 1, and then calls itself...

The execution of `countdown` begins with $n=0$, and since n is not greater than 0, it outputs the word, “Blastoff!” and then returns.

The `countdown` that got $n=1$ returns.

The `countdown` that got $n=2$ returns.

The `countdown` that got $n=3$ returns.

And then you’re back in `__main__`. So, the total output looks like this:

```
3  
2  
1  
Blastoff!
```

A function that calls itself is **recursive**; the process of executing it is called **recursion**.

As another example, we can write a function that prints a string n times.

```
def print_n(s, n):  
    if n <= 0:  
        return  
    print(s)  
    print_n(s, n-1)
```

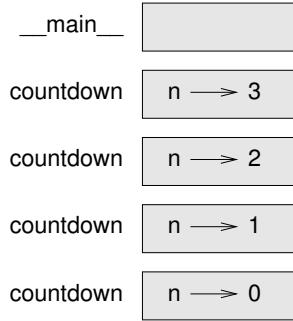


Figure 5.1: Stack diagram.

If $n \leq 0$ the **return statement** exits the function. The flow of execution immediately returns to the caller, and the remaining lines of the function don't run.

The rest of the function is similar to `countdown`: it displays `s` and then calls itself to display `s` $n - 1$ additional times. So the number of lines of output is $1 + (n - 1)$, which adds up to n .

For simple examples like this, it is probably easier to use a `for` loop. But we will see examples later that are hard to write with a `for` loop and easy to write with recursion, so it is good to start early.

5.9 Stack diagrams for recursive functions

In Section 3.9, we used a stack diagram to represent the state of a program during a function call. The same kind of diagram can help interpret a recursive function.

Every time a function gets called, Python creates a frame to contain the function's local variables and parameters. For a recursive function, there might be more than one frame on the stack at the same time.

Figure 5.1 shows a stack diagram for `countdown` called with $n = 3$.

As usual, the top of the stack is the frame for `__main__`. It is empty because we did not create any variables in `__main__` or pass any arguments to it.

The four `countdown` frames have different values for the parameter `n`. The bottom of the stack, where $n=0$, is called the **base case**. It does not make a recursive call, so there are no more frames.

As an exercise, draw a stack diagram for `print_n` called with `s = 'Hello'` and $n=2$. Then write a function called `do_n` that takes a function object and a number, `n`, as arguments, and that calls the given function n times.

5.10 Infinite recursion

If a recursion never reaches a base case, it goes on making recursive calls forever, and the program never terminates. This is known as **infinite recursion**, and it is generally not a good idea. Here is a minimal program with an infinite recursion:

```
def recurse():
    recurse()
```

In most programming environments, a program with infinite recursion does not really run forever. Python reports an error message when the maximum recursion depth is reached:

```
File "<stdin>", line 2, in recurse
File "<stdin>", line 2, in recurse
File "<stdin>", line 2, in recurse
.
.
.
File "<stdin>", line 2, in recurse
RuntimeError: Maximum recursion depth exceeded
```

This traceback is a little bigger than the one we saw in the previous chapter. When the error occurs, there are 1000 `recurse` frames on the stack!

If you encounter an infinite recursion by accident, review your function to confirm that there is a base case that does not make a recursive call. And if there is a base case, check whether you are guaranteed to reach it.

5.11 Keyboard input

The programs we have written so far accept no input from the user. They just do the same thing every time.

Python provides a built-in function called `input` that stops the program and waits for the user to type something. When the user presses Return or Enter, the program resumes and `input` returns what the user typed as a string. In Python 2, the same function is called `raw_input`.

```
>>> text = input()
What are you waiting for?
>>> text
'What are you waiting for?'
```

Before getting input from the user, it is a good idea to print a prompt telling the user what to type. `input` can take a prompt as an argument:

```
>>> name = input('What...is your name?\n')
What...is your name?
Arthur, King of the Britons!
>>> name
'Arthur, King of the Britons!'
```

The sequence `\n` at the end of the prompt represents a **newline**, which is a special character that causes a line break. That's why the user's input appears below the prompt.

If you expect the user to type an integer, you can try to convert the return value to `int`:

```
>>> prompt = 'What...is the airspeed velocity of an unladen swallow?\n'
>>> speed = input(prompt)
What...is the airspeed velocity of an unladen swallow?
42
>>> int(speed)
42
```

But if the user types something other than a string of digits, you get an error:

```
>>> speed = input(prompt)
What...is the airspeed velocity of an unladen swallow?
What do you mean, an African or a European swallow?
>>> int(speed)
ValueError: invalid literal for int() with base 10
```

We will see how to handle this kind of error later.

5.12 Debugging

When a syntax or runtime error occurs, the error message contains a lot of information, but it can be overwhelming. The most useful parts are usually:

- What kind of error it was, and
- Where it occurred.

Syntax errors are usually easy to find, but there are a few gotchas. Whitespace errors can be tricky because spaces and tabs are invisible and we are used to ignoring them.

```
>>> x = 5
>>> y = 6
File "<stdin>", line 1
    y = 6
^
IndentationError: unexpected indent
```

In this example, the problem is that the second line is indented by one space. But the error message points to `y`, which is misleading. In general, error messages indicate where the problem was discovered, but the actual error might be earlier in the code, sometimes on a previous line.

The same is true of runtime errors. Suppose you are trying to compute a signal-to-noise ratio in decibels. The formula is $SNR_{db} = 10 \log_{10}(P_{signal}/P_{noise})$. In Python, you might write something like this:

```
import math
signal_power = 9
noise_power = 10
ratio = signal_power // noise_power
decibels = 10 * math.log10(ratio)
print(decibels)
```

When you run this program, you get an exception:

```
Traceback (most recent call last):
  File "snr.py", line 5, in ?
    decibels = 10 * math.log10(ratio)
ValueError: math domain error
```

The error message indicates line 5, but there is nothing wrong with that line. To find the real error, it might be useful to print the value of `ratio`, which turns out to be 0. The problem is in line 4, which uses floor division instead of floating-point division.

You should take the time to read error messages carefully, but don't assume that everything they say is correct.

5.13 Glossary

floor division: An operator, denoted `//`, that divides two numbers and rounds down (toward negative infinity) to an integer.

modulus operator: An operator, denoted with a percent sign (`%`), that works on integers and returns the remainder when one number is divided by another.

boolean expression: An expression whose value is either `True` or `False`.

relational operator: One of the operators that compares its operands: `==`, `!=`, `>`, `<`, `>=`, and `<=`.

logical operator: One of the operators that combines boolean expressions: `and`, `or`, and `not`.

conditional statement: A statement that controls the flow of execution depending on some condition.

condition: The boolean expression in a conditional statement that determines which branch runs.

compound statement: A statement that consists of a header and a body. The header ends with a colon (`:`). The body is indented relative to the header.

branch: One of the alternative sequences of statements in a conditional statement.

chained conditional: A conditional statement with a series of alternative branches.

nested conditional: A conditional statement that appears in one of the branches of another conditional statement.

return statement: A statement that causes a function to end immediately and return to the caller.

recursion: The process of calling the function that is currently executing.

base case: A conditional branch in a recursive function that does not make a recursive call.

infinite recursion: A recursion that doesn't have a base case, or never reaches it. Eventually, an infinite recursion causes a runtime error.

5.14 Exercises

Exercise 5.1. *The time module provides a function, also named `time`, that returns the current Greenwich Mean Time in “the epoch”, which is an arbitrary time used as a reference point. On UNIX systems, the epoch is 1 January 1970.*

```
>>> import time  
>>> time.time()  
1437746094.5735958
```

Write a script that reads the current time and converts it to a time of day in hours, minutes, and seconds, plus the number of days since the epoch.

Exercise 5.2. Fermat's Last Theorem says that there are no positive integers a , b , and c such that

$$a^n + b^n = c^n$$

for any values of n greater than 2.

1. Write a function named `check_fermat` that takes four parameters—`a`, `b`, `c` and `n`—and checks to see if Fermat's theorem holds. If n is greater than 2 and

$$a^n + b^n = c^n$$

the program should print, "Holy smokes, Fermat was wrong!" Otherwise the program should print, "No, that doesn't work."

2. Write a function that prompts the user to input values for `a`, `b`, `c` and `n`, converts them to integers, and uses `check_fermat` to check whether they violate Fermat's theorem.

Exercise 5.3. If you are given three sticks, you may or may not be able to arrange them in a triangle. For example, if one of the sticks is 12 inches long and the other two are one inch long, you will not be able to get the short sticks to meet in the middle. For any three lengths, there is a simple test to see if it is possible to form a triangle:

If any of the three lengths is greater than the sum of the other two, then you cannot form a triangle. Otherwise, you can. (If the sum of two lengths equals the third, they form what is called a "degenerate" triangle.)

1. Write a function named `is_triangle` that takes three integers as arguments, and that prints either "Yes" or "No", depending on whether you can or cannot form a triangle from sticks with the given lengths.
2. Write a function that prompts the user to input three stick lengths, converts them to integers, and uses `is_triangle` to check whether sticks with the given lengths can form a triangle.

Exercise 5.4. What is the output of the following program? Draw a stack diagram that shows the state of the program when it prints the result.

```
def recurse(n, s):
    if n == 0:
        print(s)
    else:
        recurse(n-1, n+s)

recurse(3, 0)
```

1. What would happen if you called this function like this: `recurse(-1, 0)`?
2. Write a docstring that explains everything someone would need to know in order to use this function (and nothing else).

The following exercises use the `turtle` module, described in Chapter 4:

Exercise 5.5. Read the following function and see if you can figure out what it does (see the examples in Chapter 4). Then run it and see if you got it right.

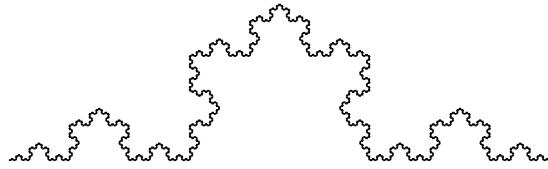


Figure 5.2: A Koch curve.

```
def draw(t, length, n):
    if n == 0:
        return
    angle = 50
    t.fd(length*n)
    t.lt(angle)
    draw(t, length, n-1)
    t.rt(2*angle)
    draw(t, length, n-1)
    t.lt(angle)
    t.bk(length*n)
```

Exercise 5.6. The Koch curve is a fractal that looks something like Figure 5.2. To draw a Koch curve with length x , all you have to do is

1. Draw a Koch curve with length $x/3$.
2. Turn left 60 degrees.
3. Draw a Koch curve with length $x/3$.
4. Turn right 120 degrees.
5. Draw a Koch curve with length $x/3$.
6. Turn left 60 degrees.
7. Draw a Koch curve with length $x/3$.

The exception is if x is less than 3: in that case, you can just draw a straight line with length x .

1. Write a function called `koch` that takes a turtle and a length as parameters, and that uses the turtle to draw a Koch curve with the given length.
2. Write a function called `snowflake` that draws three Koch curves to make the outline of a snowflake.

Solution: <http://thinkpython2.com/code/koch.py>.

3. The Koch curve can be generalized in several ways. See http://en.wikipedia.org/wiki/Koch_snowflake for examples and implement your favorite.

Chapter 6

Fruitful functions

Many of the Python functions we have used, such as the math functions, produce return values. But the functions we've written are all void: they have an effect, like printing a value or moving a turtle, but they don't have a return value. In this chapter you will learn to write fruitful functions.

6.1 Return values

Calling the function generates a return value, which we usually assign to a variable or use as part of an expression.

```
e = math.exp(1.0)
height = radius * math.sin(radians)
```

The functions we have written so far are void. Speaking casually, they have no return value; more precisely, their return value is `None`.

In this chapter, we are (finally) going to write fruitful functions. The first example is `area`, which returns the area of a circle with the given radius:

```
def area(radius):
    a = math.pi * radius**2
    return a
```

We have seen the `return` statement before, but in a fruitful function the `return` statement includes an expression. This statement means: “Return immediately from this function and use the following expression as a return value.” The expression can be arbitrarily complicated, so we could have written this function more concisely:

```
def area(radius):
    return math.pi * radius**2
```

On the other hand, **temporary variables** like `a` can make debugging easier.

Sometimes it is useful to have multiple `return` statements, one in each branch of a conditional:

```
def absolute_value(x):
    if x < 0:
        return -x
    else:
        return x
```

Since these `return` statements are in an alternative conditional, only one runs.

As soon as a `return` statement runs, the function terminates without executing any subsequent statements. Code that appears after a `return` statement, or any other place the flow of execution can never reach, is called **dead code**.

In a fruitful function, it is a good idea to ensure that every possible path through the program hits a `return` statement. For example:

```
def absolute_value(x):
    if x < 0:
        return -x
    if x > 0:
        return x
```

This function is incorrect because if `x` happens to be 0, neither condition is true, and the function ends without hitting a `return` statement. If the flow of execution gets to the end of a function, the return value is `None`, which is not the absolute value of 0.

```
>>> print(absolute_value(0))
None
```

By the way, Python provides a built-in function called `abs` that computes absolute values.

As an exercise, write a `compare` function that takes two values, `x` and `y`, and returns 1 if `x > y`, 0 if `x == y`, and -1 if `x < y`.

6.2 Incremental development

As you write larger functions, you might find yourself spending more time debugging.

To deal with increasingly complex programs, you might want to try a process called **incremental development**. The goal of incremental development is to avoid long debugging sessions by adding and testing only a small amount of code at a time.

As an example, suppose you want to find the distance between two points, given by the coordinates (x_1, y_1) and (x_2, y_2) . By the Pythagorean theorem, the distance is:

$$\text{distance} = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

The first step is to consider what a `distance` function should look like in Python. In other words, what are the inputs (parameters) and what is the output (return value)?

In this case, the inputs are two points, which you can represent using four numbers. The return value is the distance represented by a floating-point value.

Immediately you can write an outline of the function:

```
def distance(x1, y1, x2, y2):
    return 0.0
```

Obviously, this version doesn't compute distances; it always returns zero. But it is syntactically correct, and it runs, which means that you can test it before you make it more complicated.

To test the new function, call it with sample arguments:

```
>>> distance(1, 2, 4, 6)
0.0
```

I chose these values so that the horizontal distance is 3 and the vertical distance is 4; that way, the result is 5, the hypotenuse of a 3-4-5 right triangle. When testing a function, it is useful to know the right answer.

At this point we have confirmed that the function is syntactically correct, and we can start adding code to the body. A reasonable next step is to find the differences $x_2 - x_1$ and $y_2 - y_1$. The next version stores those values in temporary variables and prints them.

```
def distance(x1, y1, x2, y2):
    dx = x2 - x1
    dy = y2 - y1
    print('dx is', dx)
    print('dy is', dy)
    return 0.0
```

If the function is working, it should display `dx is 3` and `dy is 4`. If so, we know that the function is getting the right arguments and performing the first computation correctly. If not, there are only a few lines to check.

Next we compute the sum of squares of `dx` and `dy`:

```
def distance(x1, y1, x2, y2):
    dx = x2 - x1
    dy = y2 - y1
    dsquared = dx**2 + dy**2
    print('dsquared is: ', dsquared)
    return 0.0
```

Again, you would run the program at this stage and check the output (which should be 25). Finally, you can use `math.sqrt` to compute and return the result:

```
def distance(x1, y1, x2, y2):
    dx = x2 - x1
    dy = y2 - y1
    dsquared = dx**2 + dy**2
    result = math.sqrt(dsquared)
    return result
```

If that works correctly, you are done. Otherwise, you might want to print the value of `result` before the `return` statement.

The final version of the function doesn't display anything when it runs; it only returns a value. The `print` statements we wrote are useful for debugging, but once you get the function working, you should remove them. Code like that is called **scaffolding** because it is helpful for building the program but is not part of the final product.

When you start out, you should add only a line or two of code at a time. As you gain more experience, you might find yourself writing and debugging bigger chunks. Either way, incremental development can save you a lot of debugging time.

The key aspects of the process are:

1. Start with a working program and make small incremental changes. At any point, if there is an error, you should have a good idea where it is.
2. Use variables to hold intermediate values so you can display and check them.
3. Once the program is working, you might want to remove some of the scaffolding or consolidate multiple statements into compound expressions, but only if it does not make the program difficult to read.

As an exercise, use incremental development to write a function called `hypotenuse` that returns the length of the hypotenuse of a right triangle given the lengths of the other two legs as arguments. Record each stage of the development process as you go.

6.3 Composition

As you should expect by now, you can call one function from within another. As an example, we'll write a function that takes two points, the center of the circle and a point on the perimeter, and computes the area of the circle.

Assume that the center point is stored in the variables `xc` and `yc`, and the perimeter point is in `xp` and `yp`. The first step is to find the radius of the circle, which is the distance between the two points. We just wrote a function, `distance`, that does that:

```
radius = distance(xc, yc, xp, yp)
```

The next step is to find the area of a circle with that radius; we just wrote that, too:

```
result = area(radius)
```

Encapsulating these steps in a function, we get:

```
def circle_area(xc, yc, xp, yp):
    radius = distance(xc, yc, xp, yp)
    result = area(radius)
    return result
```

The temporary variables `radius` and `result` are useful for development and debugging, but once the program is working, we can make it more concise by composing the function calls:

```
def circle_area(xc, yc, xp, yp):
    return area(distance(xc, yc, xp, yp))
```

6.4 Boolean functions

Functions can return booleans, which is often convenient for hiding complicated tests inside functions. For example:

```
def is_divisible(x, y):
    if x % y == 0:
        return True
    else:
        return False
```

It is common to give boolean functions names that sound like yes/no questions; `is_divisible` returns either `True` or `False` to indicate whether `x` is divisible by `y`.

Here is an example:

```
>>> is_divisible(6, 4)
False
>>> is_divisible(6, 3)
True
```

The result of the `==` operator is a boolean, so we can write the function more concisely by returning it directly:

```
def is_divisible(x, y):
    return x % y == 0
```

Boolean functions are often used in conditional statements:

```
if is_divisible(x, y):
    print('x is divisible by y')
```

It might be tempting to write something like:

```
if is_divisible(x, y) == True:
    print('x is divisible by y')
```

But the extra comparison is unnecessary.

As an exercise, write a function `is_between(x, y, z)` that returns `True` if $x \leq y \leq z$ or `False` otherwise.

6.5 More recursion

We have only covered a small subset of Python, but you might be interested to know that this subset is a *complete* programming language, which means that anything that can be computed can be expressed in this language. Any program ever written could be rewritten using only the language features you have learned so far (actually, you would need a few commands to control devices like the mouse, disks, etc., but that's all).

Proving that claim is a nontrivial exercise first accomplished by Alan Turing, one of the first computer scientists (some would argue that he was a mathematician, but a lot of early computer scientists started as mathematicians). Accordingly, it is known as the Turing Thesis. For a more complete (and accurate) discussion of the Turing Thesis, I recommend Michael Sipser's book *Introduction to the Theory of Computation*.

To give you an idea of what you can do with the tools you have learned so far, we'll evaluate a few recursively defined mathematical functions. A recursive definition is similar to a circular definition, in the sense that the definition contains a reference to the thing being defined. A truly circular definition is not very useful:

vorpal: An adjective used to describe something that is vorpal.

If you saw that definition in the dictionary, you might be annoyed. On the other hand, if you looked up the definition of the factorial function, denoted with the symbol `!`, you might get something like this:

$$\begin{aligned} 0! &= 1 \\ n! &= n(n-1)! \end{aligned}$$

This definition says that the factorial of 0 is 1, and the factorial of any other value, n , is n multiplied by the factorial of $n - 1$.

So $3!$ is 3 times $2!$, which is 2 times $1!$, which is 1 times $0!$. Putting it all together, $3!$ equals 3 times 2 times 1 times 1, which is 6.

If you can write a recursive definition of something, you can write a Python program to evaluate it. The first step is to decide what the parameters should be. In this case it should be clear that `factorial` takes an integer:

```
def factorial(n):
```

If the argument happens to be 0, all we have to do is return 1:

```
def factorial(n):
    if n == 0:
        return 1
```

Otherwise, and this is the interesting part, we have to make a recursive call to find the factorial of $n - 1$ and then multiply it by n :

```
def factorial(n):
    if n == 0:
        return 1
    else:
        recurse = factorial(n-1)
        result = n * recurse
        return result
```

The flow of execution for this program is similar to the flow of `countdown` in Section 5.8. If we call `factorial` with the value 3:

Since 3 is not 0, we take the second branch and calculate the factorial of $n - 1$...

Since 2 is not 0, we take the second branch and calculate the factorial of $n - 1$...

Since 1 is not 0, we take the second branch and calculate the factorial of $n - 1$...

Since 0 equals 0, we take the first branch and return 1 without making any more recursive calls.

The return value, 1, is multiplied by n , which is 1, and the result is returned.

The return value, 1, is multiplied by n , which is 2, and the result is returned.

The return value (2) is multiplied by n , which is 3, and the result, 6, becomes the return value of the function call that started the whole process.

Figure 6.1 shows what the stack diagram looks like for this sequence of function calls.

The return values are shown being passed back up the stack. In each frame, the return value is the value of `result`, which is the product of `n` and `recurse`.

In the last frame, the local variables `recurse` and `result` do not exist, because the branch that creates them does not run.

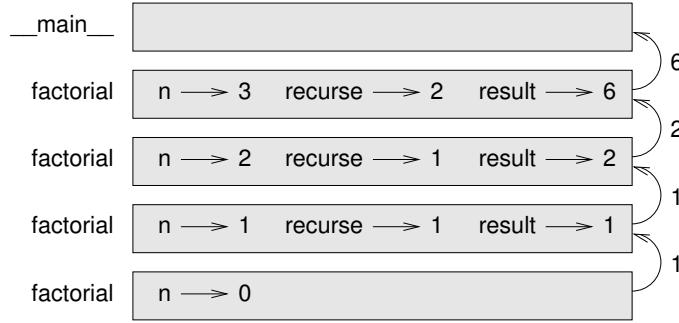


Figure 6.1: Stack diagram.

6.6 Leap of faith

Following the flow of execution is one way to read programs, but it can quickly become overwhelming. An alternative is what I call the “leap of faith”. When you come to a function call, instead of following the flow of execution, you *assume* that the function works correctly and returns the right result.

In fact, you are already practicing this leap of faith when you use built-in functions. When you call `math.cos` or `math.exp`, you don’t examine the bodies of those functions. You just assume that they work because the people who wrote the built-in functions were good programmers.

The same is true when you call one of your own functions. For example, in Section 6.4, we wrote a function called `is_divisible` that determines whether one number is divisible by another. Once we have convinced ourselves that this function is correct—by examining the code and testing—we can use the function without looking at the body again.

The same is true of recursive programs. When you get to the recursive call, instead of following the flow of execution, you should assume that the recursive call works (returns the correct result) and then ask yourself, “Assuming that I can find the factorial of $n - 1$, can I compute the factorial of n ? ” It is clear that you can, by multiplying by n .

Of course, it’s a bit strange to assume that the function works correctly when you haven’t finished writing it, but that’s why it’s called a leap of faith!

6.7 One more example

After `factorial`, the most common example of a recursively defined mathematical function is `fibonacci`, which has the following definition (see http://en.wikipedia.org/wiki/Fibonacci_number):

$$\begin{aligned} \text{fibonacci}(0) &= 0 \\ \text{fibonacci}(1) &= 1 \\ \text{fibonacci}(n) &= \text{fibonacci}(n - 1) + \text{fibonacci}(n - 2) \end{aligned}$$

Translated into Python, it looks like this:

```
def fibonacci(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fibonacci(n-1) + fibonacci(n-2)
```

If you try to follow the flow of execution here, even for fairly small values of n , your head explodes. But according to the leap of faith, if you assume that the two recursive calls work correctly, then it is clear that you get the right result by adding them together.

6.8 Checking types

What happens if we call `factorial` and give it 1.5 as an argument?

```
>>> factorial(1.5)
RuntimeError: Maximum recursion depth exceeded
```

It looks like an infinite recursion. How can that be? The function has a base case—when $n == 0$. But if n is not an integer, we can *miss* the base case and recurse forever.

In the first recursive call, the value of n is 0.5. In the next, it is -0.5. From there, it gets smaller (more negative), but it will never be 0.

We have two choices. We can try to generalize the `factorial` function to work with floating-point numbers, or we can make `factorial` check the type of its argument. The first option is called the gamma function and it's a little beyond the scope of this book. So we'll go for the second.

We can use the built-in function `isinstance` to verify the type of the argument. While we're at it, we can also make sure the argument is positive:

```
def factorial(n):
    if not isinstance(n, int):
        print('Factorial is only defined for integers.')
        return None
    elif n < 0:
        print('Factorial is not defined for negative integers.')
        return None
    elif n == 0:
        return 1
    else:
        return n * factorial(n-1)
```

The first base case handles nonintegers; the second handles negative integers. In both cases, the program prints an error message and returns `None` to indicate that something went wrong:

```
>>> print(factorial('fred'))
Factorial is only defined for integers.
None
>>> print(factorial(-2))
Factorial is not defined for negative integers.
None
```

If we get past both checks, we know that n is a non-negative integer, so we can prove that the recursion terminates.

This program demonstrates a pattern sometimes called a **guardian**. The first two conditionals act as guardians, protecting the code that follows from values that might cause an error. The guardians make it possible to prove the correctness of the code.

In Section 11.4 we will see a more flexible alternative to printing an error message: raising an exception.

6.9 Debugging

Breaking a large program into smaller functions creates natural checkpoints for debugging. If a function is not working, there are three possibilities to consider:

- There is something wrong with the arguments the function is getting; a precondition is violated.
- There is something wrong with the function; a postcondition is violated.
- There is something wrong with the return value or the way it is being used.

To rule out the first possibility, you can add a `print` statement at the beginning of the function and display the values of the parameters (and maybe their types). Or you can write code that checks the preconditions explicitly.

If the parameters look good, add a `print` statement before each `return` statement and display the return value. If possible, check the result by hand. Consider calling the function with values that make it easy to check the result (as in Section 6.2).

If the function seems to be working, look at the function call to make sure the return value is being used correctly (or used at all!).

Adding print statements at the beginning and end of a function can help make the flow of execution more visible. For example, here is a version of `factorial` with print statements:

```
def factorial(n):
    space = ' ' * (4 * n)
    print(space, 'factorial', n)
    if n == 0:
        print(space, 'returning 1')
        return 1
    else:
        recurse = factorial(n-1)
        result = n * recurse
        print(space, 'returning', result)
        return result
```

`space` is a string of space characters that controls the indentation of the output. Here is the result of `factorial(4)`:

```

        factorial 4
    factorial 3
    factorial 2
    factorial 1
factorial 0
returning 1
    returning 1
    returning 2
    returning 6
        returning 24

```

If you are confused about the flow of execution, this kind of output can be helpful. It takes some time to develop effective scaffolding, but a little bit of scaffolding can save a lot of debugging.

6.10 Glossary

temporary variable: A variable used to store an intermediate value in a complex calculation.

dead code: Part of a program that can never run, often because it appears after a `return` statement.

incremental development: A program development plan intended to avoid debugging by adding and testing only a small amount of code at a time.

scaffolding: Code that is used during program development but is not part of the final version.

guardian: A programming pattern that uses a conditional statement to check for and handle circumstances that might cause an error.

6.11 Exercises

Exercise 6.1. Draw a stack diagram for the following program. What does the program print?

```

def b(z):
    prod = a(z, z)
    print(z, prod)
    return prod

def a(x, y):
    x = x + 1
    return x * y

def c(x, y, z):
    total = x + y + z
    square = b(total)**2
    return square

```

```
x = 1
y = x + 1
print(c(x, y+3, x+y))
```

Exercise 6.2. The Ackermann function, $A(m, n)$, is defined:

$$A(m, n) = \begin{cases} n + 1 & \text{if } m = 0 \\ A(m - 1, 1) & \text{if } m > 0 \text{ and } n = 0 \\ A(m - 1, A(m, n - 1)) & \text{if } m > 0 \text{ and } n > 0. \end{cases}$$

See http://en.wikipedia.org/wiki/Ackermann_function. Write a function named `ack` that evaluates the Ackermann function. Use your function to evaluate `ack(3, 4)`, which should be 125. What happens for larger values of `m` and `n`? Solution: <http://thinkpython2.com/code/ackermann.py>.

Exercise 6.3. A palindrome is a word that is spelled the same backward and forward, like “noon” and “divider”. Recursively, a word is a palindrome if the first and last letters are the same and the middle is a palindrome.

The following are functions that take a string argument and return the first, last, and middle letters:

```
def first(word):
    return word[0]

def last(word):
    return word[-1]

def middle(word):
    return word[1:-1]
```

We'll see how they work in Chapter 8.

1. Type these functions into a file named `palindrome.py` and test them out. What happens if you call `middle` with a string with two letters? One letter? What about the empty string, which is written '' and contains no letters?
2. Write a function called `is_palindrome` that takes a string argument and returns `True` if it is a palindrome and `False` otherwise. Remember that you can use the built-in function `len` to check the length of a string.

Solution: http://thinkpython2.com/code/palindrome_soln.py.

Exercise 6.4. A number, a , is a power of b if it is divisible by b and a/b is a power of b . Write a function called `is_power` that takes parameters `a` and `b` and returns `True` if `a` is a power of `b`. Note: you will have to think about the base case.

Exercise 6.5. The greatest common divisor (GCD) of a and b is the largest number that divides both of them with no remainder.

One way to find the GCD of two numbers is based on the observation that if r is the remainder when a is divided by b , then $\gcd(a, b) = \gcd(b, r)$. As a base case, we can use $\gcd(a, 0) = a$.

Write a function called `gcd` that takes parameters `a` and `b` and returns their greatest common divisor.

Credit: This exercise is based on an example from Abelson and Sussman's Structure and Interpretation of Computer Programs.

Chapter 7

Iteration

This chapter is about iteration, which is the ability to run a block of statements repeatedly. We saw a kind of iteration, using recursion, in Section 5.8. We saw another kind, using a `for` loop, in Section 4.2. In this chapter we'll see yet another kind, using a `while` statement. But first I want to say a little more about variable assignment.

7.1 Reassignment

As you may have discovered, it is legal to make more than one assignment to the same variable. A new assignment makes an existing variable refer to a new value (and stop referring to the old value).

```
>>> x = 5
>>> x
5
>>> x = 7
>>> x
7
```

The first time we display `x`, its value is 5; the second time, its value is 7.

Figure 7.1 shows what **reassignment** looks like in a state diagram.

At this point I want to address a common source of confusion. Because Python uses the equal sign (`=`) for assignment, it is tempting to interpret a statement like `a = b` as a mathematical proposition of equality; that is, the claim that `a` and `b` are equal. But this interpretation is wrong.

First, equality is a symmetric relationship and assignment is not. For example, in mathematics, if $a = 7$ then $7 = a$. But in Python, the statement `a = 7` is legal and `7 = a` is not.

Also, in mathematics, a proposition of equality is either true or false for all time. If $a = b$ now, then a will always equal b . In Python, an assignment statement can make two variables equal, but they don't have to stay that way:

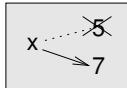


Figure 7.1: State diagram.

```

>>> a = 5
>>> b = a    # a and b are now equal
>>> a = 3    # a and b are no longer equal
>>> b
5

```

The third line changes the value of `a` but does not change the value of `b`, so they are no longer equal.

Reassigning variables is often useful, but you should use it with caution. If the values of variables change frequently, it can make the code difficult to read and debug.

7.2 Updating variables

A common kind of reassignment is an **update**, where the new value of the variable depends on the old.

```
>>> x = x + 1
```

This means “get the current value of `x`, add one, and then update `x` with the new value.”

If you try to update a variable that doesn’t exist, you get an error, because Python evaluates the right side before it assigns a value to `x`:

```
>>> x = x + 1
NameError: name 'x' is not defined
```

Before you can update a variable, you have to **initialize** it, usually with a simple assignment:

```
>>> x = 0
>>> x = x + 1
```

Updating a variable by adding 1 is called an **increment**; subtracting 1 is called a **decrement**.

7.3 The `while` statement

Computers are often used to automate repetitive tasks. Repeating identical or similar tasks without making errors is something that computers do well and people do poorly. In a computer program, repetition is also called **iteration**.

We have already seen two functions, `countdown` and `print_n`, that iterate using recursion. Because iteration is so common, Python provides language features to make it easier. One is the `for` statement we saw in Section 4.2. We’ll get back to that later.

Another is the `while` statement. Here is a version of `countdown` that uses a `while` statement:

```
def countdown(n):
    while n > 0:
        print(n)
        n = n - 1
    print('Blastoff!')
```

You can almost read the `while` statement as if it were English. It means, “While `n` is greater than 0, display the value of `n` and then decrement `n`. When you get to 0, display the word `Blastoff!`”

More formally, here is the flow of execution for a `while` statement:

1. Determine whether the condition is true or false.
2. If false, exit the `while` statement and continue execution at the next statement.
3. If the condition is true, run the body and then go back to step 1.

This type of flow is called a loop because the third step loops back around to the top.

The body of the loop should change the value of one or more variables so that the condition becomes false eventually and the loop terminates. Otherwise the loop will repeat forever, which is called an **infinite loop**. An endless source of amusement for computer scientists is the observation that the directions on shampoo, “Lather, rinse, repeat”, are an infinite loop.

In the case of `countdown`, we can prove that the loop terminates: if `n` is zero or negative, the loop never runs. Otherwise, `n` gets smaller each time through the loop, so eventually we have to get to 0.

For some other loops, it is not so easy to tell. For example:

```
def sequence(n):
    while n != 1:
        print(n)
        if n % 2 == 0:          # n is even
            n = n / 2
        else:                  # n is odd
            n = n*3 + 1
```

The condition for this loop is `n != 1`, so the loop will continue until `n` is 1, which makes the condition false.

Each time through the loop, the program outputs the value of `n` and then checks whether it is even or odd. If it is even, `n` is divided by 2. If it is odd, the value of `n` is replaced with `n*3 + 1`. For example, if the argument passed to `sequence` is 3, the resulting values of `n` are 3, 10, 5, 16, 8, 4, 2, 1.

Since `n` sometimes increases and sometimes decreases, there is no obvious proof that `n` will ever reach 1, or that the program terminates. For some particular values of `n`, we can prove termination. For example, if the starting value is a power of two, `n` will be even every time through the loop until it reaches 1. The previous example ends with such a sequence, starting with 16.

The hard question is whether we can prove that this program terminates for *all* positive values of `n`. So far, no one has been able to prove it *or* disprove it! (See http://en.wikipedia.org/wiki/Collatz_conjecture.)

As an exercise, rewrite the function `print_n` from Section 5.8 using iteration instead of recursion.

7.4 break

Sometimes you don't know it's time to end a loop until you get half way through the body. In that case you can use the `break` statement to jump out of the loop.

For example, suppose you want to take input from the user until they type `done`. You could write:

```
while True:
    line = input('> ')
    if line == 'done':
        break
    print(line)

print('Done!')
```

The loop condition is `True`, which is always true, so the loop runs until it hits the `break` statement.

Each time through, it prompts the user with an angle bracket. If the user types `done`, the `break` statement exits the loop. Otherwise the program echoes whatever the user types and goes back to the top of the loop. Here's a sample run:

```
> not done
not done
> done
Done!
```

This way of writing `while` loops is common because you can check the condition anywhere in the loop (not just at the top) and you can express the stop condition affirmatively ("stop when this happens") rather than negatively ("keep going until that happens").

7.5 Square roots

Loops are often used in programs that compute numerical results by starting with an approximate answer and iteratively improving it.

For example, one way of computing square roots is Newton's method. Suppose that you want to know the square root of a . If you start with almost any estimate, x , you can compute a better estimate with the following formula:

$$y = \frac{x + a/x}{2}$$

For example, if a is 4 and x is 3:

```
>>> a = 4
>>> x = 3
>>> y = (x + a/x) / 2
>>> y
2.166666666667
```

The result is closer to the correct answer ($\sqrt{4} = 2$). If we repeat the process with the new estimate, it gets even closer:

```
>>> x = y
>>> y = (x + a/x) / 2
>>> y
2.00641025641
```

After a few more updates, the estimate is almost exact:

```
>>> x = y
>>> y = (x + a/x) / 2
>>> y
2.00001024003
>>> x = y
>>> y = (x + a/x) / 2
>>> y
2.00000000003
```

In general we don't know ahead of time how many steps it takes to get to the right answer, but we know when we get there because the estimate stops changing:

```
>>> x = y
>>> y = (x + a/x) / 2
>>> y
2.0
>>> x = y
>>> y = (x + a/x) / 2
>>> y
2.0
```

When $y == x$, we can stop. Here is a loop that starts with an initial estimate, x , and improves it until it stops changing:

```
while True:
    print(x)
    y = (x + a/x) / 2
    if y == x:
        break
    x = y
```

For most values of a this works fine, but in general it is dangerous to test `float` equality. Floating-point values are only approximately right: most rational numbers, like $1/3$, and irrational numbers, like $\sqrt{2}$, can't be represented exactly with a `float`.

Rather than checking whether x and y are exactly equal, it is safer to use the built-in function `abs` to compute the absolute value, or magnitude, of the difference between them:

```
if abs(y-x) < epsilon:
    break
```

Where `epsilon` has a value like `0.0000001` that determines how close is close enough.

7.6 Algorithms

Newton's method is an example of an **algorithm**: it is a mechanical process for solving a category of problems (in this case, computing square roots).

To understand what an algorithm is, it might help to start with something that is not an algorithm. When you learned to multiply single-digit numbers, you probably memorized the multiplication table. In effect, you memorized 100 specific solutions. That kind of knowledge is not algorithmic.

But if you were “lazy”, you might have learned a few tricks. For example, to find the product of n and 9, you can write $n - 1$ as the first digit and $10 - n$ as the second digit. This trick is a general solution for multiplying any single-digit number by 9. That’s an algorithm!

Similarly, the techniques you learned for addition with carrying, subtraction with borrowing, and long division are all algorithms. One of the characteristics of algorithms is that they do not require any intelligence to carry out. They are mechanical processes where each step follows from the last according to a simple set of rules.

Executing algorithms is boring, but designing them is interesting, intellectually challenging, and a central part of computer science.

Some of the things that people do naturally, without difficulty or conscious thought, are the hardest to express algorithmically. Understanding natural language is a good example. We all do it, but so far no one has been able to explain *how* we do it, at least not in the form of an algorithm.

7.7 Debugging

As you start writing bigger programs, you might find yourself spending more time debugging. More code means more chances to make an error and more places for bugs to hide.

One way to cut your debugging time is “debugging by bisection”. For example, if there are 100 lines in your program and you check them one at a time, it would take 100 steps.

Instead, try to break the problem in half. Look at the middle of the program, or near it, for an intermediate value you can check. Add a print statement (or something else that has a verifiable effect) and run the program.

If the mid-point check is incorrect, there must be a problem in the first half of the program. If it is correct, the problem is in the second half.

Every time you perform a check like this, you halve the number of lines you have to search. After six steps (which is fewer than 100), you would be down to one or two lines of code, at least in theory.

In practice it is not always clear what the “middle of the program” is and not always possible to check it. It doesn’t make sense to count lines and find the exact midpoint. Instead, think about places in the program where there might be errors and places where it is easy to put a check. Then choose a spot where you think the chances are about the same that the bug is before or after the check.

7.8 Glossary

reassignment: Assigning a new value to a variable that already exists.

update: An assignment where the new value of the variable depends on the old.

initialization: An assignment that gives an initial value to a variable that will be updated.

increment: An update that increases the value of a variable (often by one).

decrement: An update that decreases the value of a variable.

iteration: Repeated execution of a set of statements using either a recursive function call or a loop.

infinite loop: A loop in which the terminating condition is never satisfied.

algorithm: A general process for solving a category of problems.

7.9 Exercises

Exercise 7.1. Copy the loop from Section 7.5 and encapsulate it in a function called `mysqrt` that takes `a` as a parameter, chooses a reasonable value of `x`, and returns an estimate of the square root of `a`.

To test it, write a function named `test_square_root` that prints a table like this:

a	mysqrt(a)	math.sqrt(a)	diff
1.0	1.0	1.0	0.0
2.0	1.41421356237	1.41421356237	2.22044604925e-16
3.0	1.73205080757	1.73205080757	0.0
4.0	2.0	2.0	0.0
5.0	2.2360679775	2.2360679775	0.0
6.0	2.44948974278	2.44948974278	0.0
7.0	2.64575131106	2.64575131106	0.0
8.0	2.82842712475	2.82842712475	4.4408920985e-16
9.0	3.0	3.0	0.0

The first column is a number, `a`; the second column is the square root of `a` computed with `mysqrt`; the third column is the square root computed by `math.sqrt`; the fourth column is the absolute value of the difference between the two estimates.

Exercise 7.2. The built-in function `eval` takes a string and evaluates it using the Python interpreter. For example:

```
>>> eval('1 + 2 * 3')
7
>>> import math
>>> eval('math.sqrt(5)')
2.2360679774997898
>>> eval('type(math.pi)')
<class 'float'>
```

Write a function called `eval_loop` that iteratively prompts the user, takes the resulting input and evaluates it using `eval`, and prints the result.

It should continue until the user enters 'done', and then return the value of the last expression it evaluated.

Exercise 7.3. The mathematician Srinivasa Ramanujan found an infinite series that can be used to generate a numerical approximation of $1/\pi$:

$$\frac{1}{\pi} = \frac{2\sqrt{2}}{9801} \sum_{k=0}^{\infty} \frac{(4k)!(1103 + 26390k)}{(k!)^4 396^{4k}}$$

Write a function called `estimate_pi` that uses this formula to compute and return an estimate of π . It should use a `while` loop to compute terms of the summation until the last term is smaller than `1e-15` (which is Python notation for 10^{-15}). You can check the result by comparing it to `math.pi`.

Solution: <http://thinkpython2.com/code/pi.py>.

Chapter 8

Strings

Strings are not like integers, floats, and booleans. A string is a **sequence**, which means it is an ordered collection of other values. In this chapter you'll see how to access the characters that make up a string, and you'll learn about some of the methods strings provide.

8.1 A string is a sequence

A string is a sequence of characters. You can access the characters one at a time with the bracket operator:

```
>>> fruit = 'banana'  
>>> letter = fruit[1]
```

The second statement selects character number 1 from `fruit` and assigns it to `letter`.

The expression in brackets is called an **index**. The index indicates which character in the sequence you want (hence the name).

But you might not get what you expect:

```
>>> letter  
'a'
```

For most people, the first letter of 'banana' is b, not a. But for computer scientists, the index is an offset from the beginning of the string, and the offset of the first letter is zero.

```
>>> letter = fruit[0]  
>>> letter  
'b'
```

So b is the 0th letter ("zero-eth") of 'banana', a is the 1th letter ("one-eth"), and n is the 2th letter ("two-eth").

As an index you can use an expression that contains variables and operators:

```
>>> i = 1  
>>> fruit[i]  
'a'  
>>> fruit[i+1]  
'n'
```

But the value of the index has to be an integer. Otherwise you get:

```
>>> letter = fruit[1.5]
TypeError: string indices must be integers
```

8.2 len

`len` is a built-in function that returns the number of characters in a string:

```
>>> fruit = 'banana'
>>> len(fruit)
6
```

To get the last letter of a string, you might be tempted to try something like this:

```
>>> length = len(fruit)
>>> last = fruit[length]
IndexError: string index out of range
```

The reason for the `IndexError` is that there is no letter in `'banana'` with the index 6. Since we started counting at zero, the six letters are numbered 0 to 5. To get the last character, you have to subtract 1 from `length`:

```
>>> last = fruit[length-1]
>>> last
'a'
```

Or you can use negative indices, which count backward from the end of the string. The expression `fruit[-1]` yields the last letter, `fruit[-2]` yields the second to last, and so on.

8.3 Traversal with a for loop

A lot of computations involve processing a string one character at a time. Often they start at the beginning, select each character in turn, do something to it, and continue until the end. This pattern of processing is called a **traversal**. One way to write a traversal is with a `while` loop:

```
index = 0
while index < len(fruit):
    letter = fruit[index]
    print(letter)
    index = index + 1
```

This loop traverses the string and displays each letter on a line by itself. The loop condition is `index < len(fruit)`, so when `index` is equal to the length of the string, the condition is false, and the body of the loop doesn't run. The last character accessed is the one with the index `len(fruit)-1`, which is the last character in the string.

As an exercise, write a function that takes a string as an argument and displays the letters backward, one per line.

Another way to write a traversal is with a `for` loop:

```
for letter in fruit:
    print(letter)
```

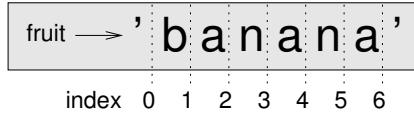


Figure 8.1: Slice indices.

Each time through the loop, the next character in the string is assigned to the variable `letter`. The loop continues until no characters are left.

The following example shows how to use concatenation (string addition) and a `for` loop to generate an abecedarian series (that is, in alphabetical order). In Robert McCloskey's book *Make Way for Ducklings*, the names of the ducklings are Jack, Kack, Lack, Mack, Nack, Ouack, Pack, and Quack. This loop outputs these names in order:

```
prefixes = 'JKLMNOPQ'
suffix = 'ack'

for letter in prefixes:
    print(letter + suffix)
```

The output is:

```
Jack
Kack
Lack
Mack
Nack
Oack
Pack
Qack
```

Of course, that's not quite right because "Ouack" and "Quack" are misspelled. As an exercise, modify the program to fix this error.

8.4 String slices

A segment of a string is called a **slice**. Selecting a slice is similar to selecting a character:

```
>>> s = 'Monty Python'
>>> s[0:5]
'Monty'
>>> s[6:12]
'Python'
```

The operator `[n:m]` returns the part of the string from the “n-eth” character to the “m-eth” character, including the first but excluding the last. This behavior is counterintuitive, but it might help to imagine the indices pointing *between* the characters, as in Figure 8.1.

If you omit the first index (before the colon), the slice starts at the beginning of the string. If you omit the second index, the slice goes to the end of the string:

```
>>> fruit = 'banana'
>>> fruit[:3]
```

```
'ban'
>>> fruit[3:]
'ana'
```

If the first index is greater than or equal to the second the result is an **empty string**, represented by two quotation marks:

```
>>> fruit = 'banana'
>>> fruit[3:3]
''
```

An empty string contains no characters and has length 0, but other than that, it is the same as any other string.

Continuing this example, what do you think `fruit[:]` means? Try it and see.

8.5 Strings are immutable

It is tempting to use the `[]` operator on the left side of an assignment, with the intention of changing a character in a string. For example:

```
>>> greeting = 'Hello, world!'
>>> greeting[0] = 'J'
TypeError: 'str' object does not support item assignment
```

The “object” in this case is the string and the “item” is the character you tried to assign. For now, an object is the same thing as a value, but we will refine that definition later (Section 10.10).

The reason for the error is that strings are **immutable**, which means you can’t change an existing string. The best you can do is create a new string that is a variation on the original:

```
>>> greeting = 'Hello, world!'
>>> new_greeting = 'J' + greeting[1:]
>>> new_greeting
'Jello, world!'
```

This example concatenates a new first letter onto a slice of `greeting`. It has no effect on the original string.

8.6 Searching

What does the following function do?

```
def find(word, letter):
    index = 0
    while index < len(word):
        if word[index] == letter:
            return index
        index = index + 1
    return -1
```

In a sense, `find` is the inverse of the `[]` operator. Instead of taking an index and extracting the corresponding character, it takes a character and finds the index where that character appears. If the character is not found, the function returns `-1`.

This is the first example we have seen of a `return` statement inside a loop. If `word[index] == letter`, the function breaks out of the loop and returns immediately.

If the character doesn't appear in the string, the program exits the loop normally and returns `-1`.

This pattern of computation—traversing a sequence and returning when we find what we are looking for—is called a **search**.

As an exercise, modify `find` so that it has a third parameter, the index in `word` where it should start looking.

8.7 Looping and counting

The following program counts the number of times the letter `a` appears in a string:

```
word = 'banana'  
count = 0  
for letter in word:  
    if letter == 'a':  
        count = count + 1  
print(count)
```

This program demonstrates another pattern of computation called a **counter**. The variable `count` is initialized to `0` and then incremented each time an `a` is found. When the loop exits, `count` contains the result—the total number of `a`'s.

As an exercise, encapsulate this code in a function named `count`, and generalize it so that it accepts the string and the letter as arguments.

Then rewrite the function so that instead of traversing the string, it uses the three-parameter version of `find` from the previous section.

8.8 String methods

Strings provide methods that perform a variety of useful operations. A method is similar to a function—it takes arguments and returns a value—but the syntax is different. For example, the method `upper` takes a string and returns a new string with all uppercase letters.

Instead of the function syntax `upper(word)`, it uses the method syntax `word.upper()`.

```
>>> word = 'banana'  
>>> new_word = word.upper()  
>>> new_word  
'BANANA'
```

This form of dot notation specifies the name of the method, `upper`, and the name of the string to apply the method to, `word`. The empty parentheses indicate that this method takes no arguments.

A method call is called an **invocation**; in this case, we would say that we are invoking `upper` on `word`.

As it turns out, there is a string method named `find` that is remarkably similar to the function we wrote:

```
>>> word = 'banana'
>>> index = word.find('a')
>>> index
1
```

In this example, we invoke `find` on `word` and pass the letter we are looking for as a parameter.

Actually, the `find` method is more general than our function; it can find substrings, not just characters:

```
>>> word.find('na')
2
```

By default, `find` starts at the beginning of the string, but it can take a second argument, the index where it should start:

```
>>> word.find('na', 3)
4
```

This is an example of an **optional argument**; `find` can also take a third argument, the index where it should stop:

```
>>> name = 'bob'
>>> name.find('b', 1, 2)
-1
```

This search fails because `b` does not appear in the index range from 1 to 2, not including 2. Searching up to, but not including, the second index makes `find` consistent with the slice operator.

8.9 The `in` operator

The word `in` is a boolean operator that takes two strings and returns `True` if the first appears as a substring in the second:

```
>>> 'a' in 'banana'
True
>>> 'seed' in 'banana'
False
```

For example, the following function prints all the letters from `word1` that also appear in `word2`:

```
def in_both(word1, word2):
    for letter in word1:
        if letter in word2:
            print(letter)
```

With well-chosen variable names, Python sometimes reads like English. You could read this loop, “for (each) letter in (the first) word, if (the) letter (appears) in (the second) word, print (the) letter.”

Here’s what you get if you compare apples and oranges:

```
>>> in_both('apples', 'oranges')
a
e
s
```

8.10 String comparison

The relational operators work on strings. To see if two strings are equal:

```
if word == 'banana':
    print('All right, bananas.')
```

Other relational operations are useful for putting words in alphabetical order:

```
if word < 'banana':
    print('Your word, ' + word + ', comes before banana.')
elif word > 'banana':
    print('Your word, ' + word + ', comes after banana.')
else:
    print('All right, bananas.')
```

Python does not handle uppercase and lowercase letters the same way people do. All the uppercase letters come before all the lowercase letters, so:

Your word, Pineapple, comes before banana.

A common way to address this problem is to convert strings to a standard format, such as all lowercase, before performing the comparison. Keep that in mind in case you have to defend yourself against a man armed with a Pineapple.

8.11 Debugging

When you use indices to traverse the values in a sequence, it is tricky to get the beginning and end of the traversal right. Here is a function that is supposed to compare two words and return True if one of the words is the reverse of the other, but it contains two errors:

```
def is_reverse(word1, word2):
    if len(word1) != len(word2):
        return False

    i = 0
    j = len(word2)

    while j > 0:
        if word1[i] != word2[j]:
            return False
        i = i+1
```

```
j = j-1
```

```
return True
```

The first if statement checks whether the words are the same length. If not, we can return `False` immediately. Otherwise, for the rest of the function, we can assume that the words are the same length. This is an example of the guardian pattern in Section 6.8.

`i` and `j` are indices: `i` traverses `word1` forward while `j` traverses `word2` backward. If we find two letters that don't match, we can return `False` immediately. If we get through the whole loop and all the letters match, we return `True`.

If we test this function with the words "pots" and "stop", we expect the return value `True`, but we get an `IndexError`:

```
>>> is_reverse('pots', 'stop')
...
File "reverse.py", line 15, in is_reverse
    if word1[i] != word2[j]:
IndexError: string index out of range
```

For debugging this kind of error, my first move is to print the values of the indices immediately before the line where the error appears.

```
while j > 0:
    print(i, j)          # print here

    if word1[i] != word2[j]:
        return False
    i = i+1
    j = j-1
```

Now when I run the program again, I get more information:

```
>>> is_reverse('pots', 'stop')
0 4
...
IndexError: string index out of range
```

The first time through the loop, the value of `j` is 4, which is out of range for the string 'pots'. The index of the last character is 3, so the initial value for `j` should be `len(word2)-1`.

If I fix that error and run the program again, I get:

```
>>> is_reverse('pots', 'stop')
0 3
1 2
2 1
True
```

This time we get the right answer, but it looks like the loop only ran three times, which is suspicious. To get a better idea of what is happening, it is useful to draw a state diagram. During the first iteration, the frame for `is_reverse` is shown in Figure 8.2.

I took some license by arranging the variables in the frame and adding dotted lines to show that the values of `i` and `j` indicate characters in `word1` and `word2`.

Starting with this diagram, run the program on paper, changing the values of `i` and `j` during each iteration. Find and fix the second error in this function.

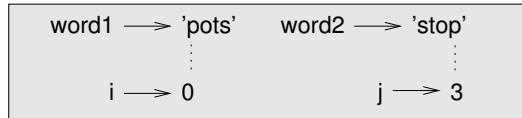


Figure 8.2: State diagram.

8.12 Glossary

object: Something a variable can refer to. For now, you can use “object” and “value” interchangeably.

sequence: An ordered collection of values where each value is identified by an integer index.

item: One of the values in a sequence.

index: An integer value used to select an item in a sequence, such as a character in a string. In Python indices start from 0.

slice: A part of a string specified by a range of indices.

empty string: A string with no characters and length 0, represented by two quotation marks.

immutable: The property of a sequence whose items cannot be changed.

traverse: To iterate through the items in a sequence, performing a similar operation on each.

search: A pattern of traversal that stops when it finds what it is looking for.

counter: A variable used to count something, usually initialized to zero and then incremented.

invocation: A statement that calls a method.

optional argument: A function or method argument that is not required.

8.13 Exercises

Exercise 8.1. Read the documentation of the string methods at <http://docs.python.org/3/library/stdtypes.html#string-methods>. You might want to experiment with some of them to make sure you understand how they work. `strip` and `replace` are particularly useful.

The documentation uses a syntax that might be confusing. For example, in `find(sub[, start[, end]])`, the brackets indicate optional arguments. So `sub` is required, but `start` is optional, and if you include `start`, then `end` is optional.

Exercise 8.2. There is a string method called `count` that is similar to the function in Section 8.7. Read the documentation of this method and write an invocation that counts the number of `a`'s in `'banana'`.

Exercise 8.3. A string slice can take a third index that specifies the “step size”; that is, the number of spaces between successive characters. A step size of 2 means every other character; 3 means every third, etc.

```
>>> fruit = 'banana'
>>> fruit[0:5:-2]
'bnn'
```

A step size of -1 goes through the word backwards, so the slice `[::-1]` generates a reversed string.

Use this idiom to write a one-line version of `is_palindrome` from Exercise 6.3.

Exercise 8.4. The following functions are all intended to check whether a string contains any lowercase letters, but at least some of them are wrong. For each function, describe what the function actually does (assuming that the parameter is a string).

```
def any_lowercase1(s):
    for c in s:
        if c.islower():
            return True
        else:
            return False

def any_lowercase2(s):
    for c in s:
        if 'c'.islower():
            return 'True'
        else:
            return 'False'

def any_lowercase3(s):
    for c in s:
        flag = c.islower()
    return flag

def any_lowercase4(s):
    flag = False
    for c in s:
        flag = flag or c.islower()
    return flag

def any_lowercase5(s):
    for c in s:
        if not c.islower():
            return False
    return True
```

Exercise 8.5. A Caesar cypher is a weak form of encryption that involves “rotating” each letter by a fixed number of places. To rotate a letter means to shift it through the alphabet, wrapping around to the beginning if necessary, so ‘A’ rotated by 3 is ‘D’ and ‘Z’ rotated by 1 is ‘A’.

To rotate a word, rotate each letter by the same amount. For example, “cheer” rotated by 7 is “jolly” and “melon” rotated by -10 is “cubed”. In the movie 2001: A Space Odyssey, the ship computer is called HAL, which is IBM rotated by -1.

Write a function called `rotate_word` that takes a string and an integer as parameters, and returns a new string that contains the letters from the original string rotated by the given amount.

You might want to use the built-in function `ord`, which converts a character to a numeric code, and

`chr`, which converts numeric codes to characters. Letters of the alphabet are encoded in alphabetical order, so for example:

```
>>> ord('c') - ord('a')
2
```

Because 'c' is the two-eth letter of the alphabet. But beware: the numeric codes for upper case letters are different.

Potentially offensive jokes on the Internet are sometimes encoded in ROT13, which is a Caesar cypher with rotation 13. If you are not easily offended, find and decode some of them. Solution: <http://thinkpython2.com/code/rotate.py>.

Chapter 9

Case study: word play

This chapter presents the second case study, which involves solving word puzzles by searching for words that have certain properties. For example, we'll find the longest palindromes in English and search for words whose letters appear in alphabetical order. And I will present another program development plan: reduction to a previously solved problem.

9.1 Reading word lists

For the exercises in this chapter we need a list of English words. There are lots of word lists available on the Web, but the one most suitable for our purpose is one of the word lists collected and contributed to the public domain by Grady Ward as part of the Moby lexicon project (see http://wikipedia.org/wiki/Moby_Project). It is a list of 113,809 official crosswords; that is, words that are considered valid in crossword puzzles and other word games. In the Moby collection, the filename is `113809of.fic`; you can download a copy, with the simpler name `words.txt`, from <http://thinkpython2.com/code/words.txt>.

This file is in plain text, so you can open it with a text editor, but you can also read it from Python. The built-in function `open` takes the name of the file as a parameter and returns a **file object** you can use to read the file.

```
>>> fin = open('words.txt')
```

`fin` is a common name for a file object used for input. The file object provides several methods for reading, including `readline`, which reads characters from the file until it gets to a newline and returns the result as a string:

```
>>> fin.readline()  
'aa\n'
```

The first word in this particular list is "aa", which is a kind of lava. The sequence `\n` represents the newline character that separates this word from the next.

The file object keeps track of where it is in the file, so if you call `readline` again, you get the next word:

```
>>> fin.readline()  
'aah\n'
```

The next word is “aah”, which is a perfectly legitimate word, so stop looking at me like that. Or, if it’s the newline character that’s bothering you, we can get rid of it with the string method `strip`:

```
>>> line = fin.readline()
>>> word = line.strip()
>>> word
'aahed'
```

You can also use a file object as part of a `for` loop. This program reads `words.txt` and prints each word, one per line:

```
fin = open('words.txt')
for line in fin:
    word = line.strip()
    print(word)
```

9.2 Exercises

There are solutions to these exercises in the next section. You should at least attempt each one before you read the solutions.

Exercise 9.1. Write a program that reads `words.txt` and prints only the words with more than 20 characters (not counting whitespace).

Exercise 9.2. In 1939 Ernest Vincent Wright published a 50,000 word novel called *Gadsby* that does not contain the letter “e”. Since “e” is the most common letter in English, that’s not easy to do.

In fact, it is difficult to construct a solitary thought without using that most common symbol. It is slow going at first, but with caution and hours of training you can gradually gain facility.

All right, I’ll stop now.

Write a function called `has_no_e` that returns True if the given word doesn’t have the letter “e” in it.

Write a program that reads `words.txt` and prints only the words that have no “e”. Compute the percentage of words in the list that have no “e”.

Exercise 9.3. Write a function named `avoids` that takes a word and a string of forbidden letters, and that returns True if the word doesn’t use any of the forbidden letters.

Write a program that prompts the user to enter a string of forbidden letters and then prints the number of words that don’t contain any of them. Can you find a combination of 5 forbidden letters that excludes the smallest number of words?

Exercise 9.4. Write a function named `uses_only` that takes a word and a string of letters, and that returns True if the word contains only letters in the list. Can you make a sentence using only the letters acefhlo? Other than “Hoe alfalfa”?

Exercise 9.5. Write a function named `uses_all` that takes a word and a string of required letters, and that returns True if the word uses all the required letters at least once. How many words are there that use all the vowels aeiou? How about aeiouy?

Exercise 9.6. Write a function called `is_abecedarian` that returns True if the letters in a word appear in alphabetical order (double letters are ok). How many abecedarian words are there?

9.3 Search

All of the exercises in the previous section have something in common; they can be solved with the search pattern we saw in Section 8.6. The simplest example is:

```
def has_no_e(word):
    for letter in word:
        if letter == 'e':
            return False
    return True
```

The `for` loop traverses the characters in `word`. If we find the letter “`e`”, we can immediately return `False`; otherwise we have to go to the next letter. If we exit the loop normally, that means we didn’t find an “`e`”, so we return `True`.

You could write this function more concisely using the `in` operator, but I started with this version because it demonstrates the logic of the search pattern.

`avoids` is a more general version of `has_no_e` but it has the same structure:

```
def avoids(word, forbidden):
    for letter in word:
        if letter in forbidden:
            return False
    return True
```

We can return `False` as soon as we find a forbidden letter; if we get to the end of the loop, we return `True`.

`uses_only` is similar except that the sense of the condition is reversed:

```
def uses_only(word, available):
    for letter in word:
        if letter not in available:
            return False
    return True
```

Instead of a list of forbidden letters, we have a list of available letters. If we find a letter in `word` that is not in `available`, we can return `False`.

`uses_all` is similar except that we reverse the role of the word and the string of letters:

```
def uses_all(word, required):
    for letter in required:
        if letter not in word:
            return False
    return True
```

Instead of traversing the letters in `word`, the loop traverses the required letters. If any of the required letters do not appear in the word, we can return `False`.

If you were really thinking like a computer scientist, you would have recognized that `uses_all` was an instance of a previously solved problem, and you would have written:

```
def uses_all(word, required):
    return uses_only(required, word)
```

This is an example of a program development plan called **reduction to a previously solved problem**, which means that you recognize the problem you are working on as an instance of a solved problem and apply an existing solution.

9.4 Looping with indices

I wrote the functions in the previous section with `for` loops because I only needed the characters in the strings; I didn't have to do anything with the indices.

For `is_abecedarian` we have to compare adjacent letters, which is a little tricky with a `for` loop:

```
def is_abecedarian(word):
    previous = word[0]
    for c in word:
        if c < previous:
            return False
        previous = c
    return True
```

An alternative is to use recursion:

```
def is_abecedarian(word):
    if len(word) <= 1:
        return True
    if word[0] > word[1]:
        return False
    return is_abecedarian(word[1:])
```

Another option is to use a `while` loop:

```
def is_abecedarian(word):
    i = 0
    while i < len(word)-1:
        if word[i+1] < word[i]:
            return False
        i = i+1
    return True
```

The loop starts at `i=0` and ends when `i=len(word)-1`. Each time through the loop, it compares the i th character (which you can think of as the current character) to the $i + 1$ th character (which you can think of as the next).

If the next character is less than (alphabetically before) the current one, then we have discovered a break in the abecedarian trend, and we return `False`.

If we get to the end of the loop without finding a fault, then the word passes the test. To convince yourself that the loop ends correctly, consider an example like '`flossy`'. The length of the word is 6, so the last time the loop runs is when `i` is 4, which is the index of the second-to-last character. On the last iteration, it compares the second-to-last character to the last, which is what we want.

Here is a version of `is_palindrome` (see Exercise 6.3) that uses two indices; one starts at the beginning and goes up; the other starts at the end and goes down.

```
def is_palindrome(word):
    i = 0
    j = len(word)-1

    while i < j:
        if word[i] != word[j]:
```

```
        return False
i = i+1
j = j-1

return True
```

Or we could reduce to a previously solved problem and write:

```
def is_palindrome(word):
    return is_reverse(word, word)
```

Using `is_reverse` from Section 8.11.

9.5 Debugging

Testing programs is hard. The functions in this chapter are relatively easy to test because you can check the results by hand. Even so, it is somewhere between difficult and impossible to choose a set of words that test for all possible errors.

Taking `has_no_e` as an example, there are two obvious cases to check: words that have an ‘e’ should return `False`, and words that don’t should return `True`. You should have no trouble coming up with one of each.

Within each case, there are some less obvious subcases. Among the words that have an “e”, you should test words with an “e” at the beginning, the end, and somewhere in the middle. You should test long words, short words, and very short words, like the empty string. The empty string is an example of a **special case**, which is one of the non-obvious cases where errors often lurk.

In addition to the test cases you generate, you can also test your program with a word list like `words.txt`. By scanning the output, you might be able to catch errors, but be careful: you might catch one kind of error (words that should not be included, but are) and not another (words that should be included, but aren’t).

In general, testing can help you find bugs, but it is not easy to generate a good set of test cases, and even if you do, you can’t be sure your program is correct. According to a legendary computer scientist:

Program testing can be used to show the presence of bugs, but never to show their absence!

— Edsger W. Dijkstra

9.6 Glossary

file object: A value that represents an open file.

reduction to a previously solved problem: A way of solving a problem by expressing it as an instance of a previously solved problem.

special case: A test case that is atypical or non-obvious (and less likely to be handled correctly).

9.7 Exercises

Exercise 9.7. This question is based on a Puzzler that was broadcast on the radio program Car Talk (<http://www.cartalk.com/content/puzzlers>):

Give me a word with three consecutive double letters. I'll give you a couple of words that almost qualify, but don't. For example, the word committee, c-o-m-m-i-t-t-e-e. It would be great except for the 'i' that sneaks in there. Or Mississippi: M-i-s-s-i-s-s-i-p-p-i. If you could take out those i's it would work. But there is a word that has three consecutive pairs of letters and to the best of my knowledge this may be the only word. Of course there are probably 500 more but I can only think of one. What is the word?

Write a program to find it. Solution: <http://thinkpython2.com/code/cartalk1.py>.

Exercise 9.8. Here's another Car Talk Puzzler (<http://www.cartalk.com/content/puzzlers>):

"I was driving on the highway the other day and I happened to notice my odometer. Like most odometers, it shows six digits, in whole miles only. So, if my car had 300,000 miles, for example, I'd see 3-0-0-0-0-0.

"Now, what I saw that day was very interesting. I noticed that the last 4 digits were palindromic; that is, they read the same forward as backward. For example, 5-4-4-5 is a palindrome, so my odometer could have read 3-1-5-4-4-5.

"One mile later, the last 5 numbers were palindromic. For example, it could have read 3-6-5-4-5-6. One mile after that, the middle 4 out of 6 numbers were palindromic. And you ready for this? One mile later, all 6 were palindromic!

"The question is, what was on the odometer when I first looked?"

Write a Python program that tests all the six-digit numbers and prints any numbers that satisfy these requirements. Solution: <http://thinkpython2.com/code/cartalk2.py>.

Exercise 9.9. Here's another Car Talk Puzzler you can solve with a search (<http://www.cartalk.com/content/puzzlers>):

"Recently I had a visit with my mom and we realized that the two digits that make up my age when reversed resulted in her age. For example, if she's 73, I'm 37. We wondered how often this has happened over the years but we got sidetracked with other topics and we never came up with an answer.

"When I got home I figured out that the digits of our ages have been reversible six times so far. I also figured out that if we're lucky it would happen again in a few years, and if we're really lucky it would happen one more time after that. In other words, it would have happened 8 times over all. So the question is, how old am I now?"

Write a Python program that searches for solutions to this Puzzler. Hint: you might find the string method `zfill` useful.

Solution: <http://thinkpython2.com/code/cartalk3.py>.

Chapter 10

Lists

This chapter presents one of Python’s most useful built-in types, lists. You will also learn more about objects and what can happen when you have more than one name for the same object.

10.1 A list is a sequence

Like a string, a **list** is a sequence of values. In a string, the values are characters; in a list, they can be any type. The values in a list are called **elements** or sometimes **items**.

There are several ways to create a new list; the simplest is to enclose the elements in square brackets ([and]):

```
[10, 20, 30, 40]  
['crunchy frog', 'ram bladder', 'lark vomit']
```

The first example is a list of four integers. The second is a list of three strings. The elements of a list don’t have to be the same type. The following list contains a string, a float, an integer, and (lo!) another list:

```
['spam', 2.0, 5, [10, 20]]
```

A list within another list is **nested**.

A list that contains no elements is called an empty list; you can create one with empty brackets, [].

As you might expect, you can assign list values to variables:

```
>>> cheeses = ['Cheddar', 'Edam', 'Gouda']  
>>> numbers = [42, 123]  
>>> empty = []  
>>> print(cheeses, numbers, empty)  
['Cheddar', 'Edam', 'Gouda'] [42, 123] []
```

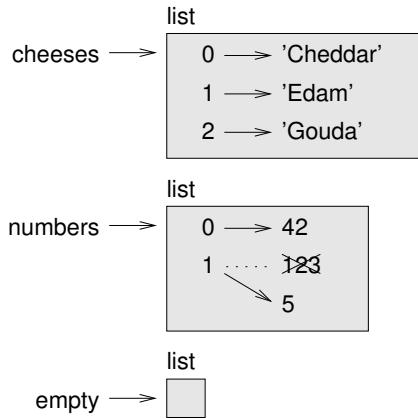


Figure 10.1: State diagram.

10.2 Lists are mutable

The syntax for accessing the elements of a list is the same as for accessing the characters of a string—the bracket operator. The expression inside the brackets specifies the index. Remember that the indices start at 0:

```
>>> cheeses[0]
'Cheddar'
```

Unlike strings, lists are mutable. When the bracket operator appears on the left side of an assignment, it identifies the element of the list that will be assigned.

```
>>> numbers = [42, 123]
>>> numbers[1] = 5
>>> numbers
[42, 5]
```

The one-eth element of `numbers`, which used to be 123, is now 5.

Figure 10.1 shows the state diagram for `cheeses`, `numbers` and `empty`.

Lists are represented by boxes with the word “list” outside and the elements of the list inside. `cheeses` refers to a list with three elements indexed 0, 1 and 2. `numbers` contains two elements; the diagram shows that the value of the second element has been reassigned from 123 to 5. `empty` refers to a list with no elements.

List indices work the same way as string indices:

- Any integer expression can be used as an index.
- If you try to read or write an element that does not exist, you get an `IndexError`.
- If an index has a negative value, it counts backward from the end of the list.

The `in` operator also works on lists.

```
>>> cheeses = ['Cheddar', 'Edam', 'Gouda']
>>> 'Edam' in cheeses
True
>>> 'Brie' in cheeses
False
```

10.3 Traversing a list

The most common way to traverse the elements of a list is with a `for` loop. The syntax is the same as for strings:

```
for cheese in cheeses:  
    print(cheese)
```

This works well if you only need to read the elements of the list. But if you want to write or update the elements, you need the indices. A common way to do that is to combine the built-in functions `range` and `len`:

```
for i in range(len(numbers)):  
    numbers[i] = numbers[i] * 2
```

This loop traverses the list and updates each element. `len` returns the number of elements in the list. `range` returns a list of indices from 0 to $n - 1$, where n is the length of the list. Each time through the loop `i` gets the index of the next element. The assignment statement in the body uses `i` to read the old value of the element and to assign the new value.

A `for` loop over an empty list never runs the body:

```
for x in []:  
    print('This never happens.')
```

Although a list can contain another list, the nested list still counts as a single element. The length of this list is four:

```
['spam', 1, ['Brie', 'Roquefort', 'Pol le Veq'], [1, 2, 3]]
```

10.4 List operations

The `+` operator concatenates lists:

```
>>> a = [1, 2, 3]  
>>> b = [4, 5, 6]  
>>> c = a + b  
>>> c  
[1, 2, 3, 4, 5, 6]
```

The `*` operator repeats a list a given number of times:

```
>>> [0] * 4  
[0, 0, 0, 0]  
>>> [1, 2, 3] * 3  
[1, 2, 3, 1, 2, 3, 1, 2, 3]
```

The first example repeats `[0]` four times. The second example repeats the list `[1, 2, 3]` three times.

10.5 List slices

The slice operator also works on lists:

```
>>> t = ['a', 'b', 'c', 'd', 'e', 'f']
>>> t[1:3]
['b', 'c']
>>> t[:4]
['a', 'b', 'c', 'd']
>>> t[3:]
['d', 'e', 'f']
```

If you omit the first index, the slice starts at the beginning. If you omit the second, the slice goes to the end. So if you omit both, the slice is a copy of the whole list.

```
>>> t[:]
['a', 'b', 'c', 'd', 'e', 'f']
```

Since lists are mutable, it is often useful to make a copy before performing operations that modify lists.

A slice operator on the left side of an assignment can update multiple elements:

```
>>> t = ['a', 'b', 'c', 'd', 'e', 'f']
>>> t[1:3] = ['x', 'y']
>>> t
['a', 'x', 'y', 'd', 'e', 'f']
```

10.6 List methods

Python provides methods that operate on lists. For example, `append` adds a new element to the end of a list:

```
>>> t = ['a', 'b', 'c']
>>> t.append('d')
>>> t
['a', 'b', 'c', 'd']
```

`extend` takes a list as an argument and appends all of the elements:

```
>>> t1 = ['a', 'b', 'c']
>>> t2 = ['d', 'e']
>>> t1.extend(t2)
>>> t1
['a', 'b', 'c', 'd', 'e']
```

This example leaves `t2` unmodified.

`sort` arranges the elements of the list from low to high:

```
>>> t = ['d', 'c', 'e', 'b', 'a']
>>> t.sort()
>>> t
['a', 'b', 'c', 'd', 'e']
```

Most list methods are void; they modify the list and return `None`. If you accidentally write `t = t.sort()`, you will be disappointed with the result.

10.7 Map, filter and reduce

To add up all the numbers in a list, you can use a loop like this:

```
def add_all(t):
    total = 0
    for x in t:
        total += x
    return total
```

`total` is initialized to 0. Each time through the loop, `x` gets one element from the list. The `+=` operator provides a short way to update a variable. This **augmented assignment statement**,

```
    total += x
```

is equivalent to

```
    total = total + x
```

As the loop runs, `total` accumulates the sum of the elements; a variable used this way is sometimes called an **accumulator**.

Adding up the elements of a list is such a common operation that Python provides it as a built-in function, `sum`:

```
>>> t = [1, 2, 3]
>>> sum(t)
6
```

An operation like this that combines a sequence of elements into a single value is sometimes called **reduce**.

Sometimes you want to traverse one list while building another. For example, the following function takes a list of strings and returns a new list that contains capitalized strings:

```
def capitalize_all(t):
    res = []
    for s in t:
        res.append(s.capitalize())
    return res
```

`res` is initialized with an empty list; each time through the loop, we append the next element. So `res` is another kind of accumulator.

An operation like `capitalize_all` is sometimes called a **map** because it “maps” a function (in this case the method `capitalize`) onto each of the elements in a sequence.

Another common operation is to select some of the elements from a list and return a sublist. For example, the following function takes a list of strings and returns a list that contains only the uppercase strings:

```
def only_upper(t):
    res = []
    for s in t:
        if s.isupper():
            res.append(s)
    return res
```

`isupper` is a string method that returns `True` if the string contains only upper case letters.

An operation like `only_upper` is called a **filter** because it selects some of the elements and filters out the others.

Most common list operations can be expressed as a combination of `map`, `filter` and `reduce`.

10.8 Deleting elements

There are several ways to delete elements from a list. If you know the index of the element you want, you can use `pop`:

```
>>> t = ['a', 'b', 'c']
>>> x = t.pop(1)
>>> t
['a', 'c']
>>> x
'b'
```

`pop` modifies the list and returns the element that was removed. If you don't provide an index, it deletes and returns the last element.

If you don't need the removed value, you can use the `del` operator:

```
>>> t = ['a', 'b', 'c']
>>> del t[1]
>>> t
['a', 'c']
```

If you know the element you want to remove (but not the index), you can use `remove`:

```
>>> t = ['a', 'b', 'c']
>>> t.remove('b')
>>> t
['a', 'c']
```

The return value from `remove` is `None`.

To remove more than one element, you can use `del` with a slice index:

```
>>> t = ['a', 'b', 'c', 'd', 'e', 'f']
>>> del t[1:5]
>>> t
['a', 'f']
```

As usual, the slice selects all the elements up to but not including the second index.

10.9 Lists and strings

A string is a sequence of characters and a list is a sequence of values, but a list of characters is not the same as a string. To convert from a string to a list of characters, you can use `list`:

```
>>> s = 'spam'
>>> t = list(s)
>>> t
['s', 'p', 'a', 'm']
```



Figure 10.2: State diagram.

Because `list` is the name of a built-in function, you should avoid using it as a variable name. I also avoid `l` because it looks too much like `1`. So that's why I use `t`.

The `list` function breaks a string into individual letters. If you want to break a string into words, you can use the `split` method:

```
>>> s = 'pining for the fjords'
>>> t = s.split()
>>> t
['pining', 'for', 'the', 'fjords']
```

An optional argument called a **delimiter** specifies which characters to use as word boundaries. The following example uses a hyphen as a delimiter:

```
>>> s = 'spam-spam-spam'
>>> delimiter = '-'
>>> t = s.split(delimiter)
>>> t
['spam', 'spam', 'spam']
```

`join` is the inverse of `split`. It takes a list of strings and concatenates the elements. `join` is a string method, so you have to invoke it on the delimiter and pass the list as a parameter:

```
>>> t = ['pining', 'for', 'the', 'fjords']
>>> delimiter = ' '
>>> s = delimiter.join(t)
>>> s
'pining for the fjords'
```

In this case the delimiter is a space character, so `join` puts a space between words. To concatenate strings without spaces, you can use the empty string, `''`, as a delimiter.

10.10 Objects and values

If we run these assignment statements:

```
a = 'banana'
b = 'banana'
```

We know that `a` and `b` both refer to a string, but we don't know whether they refer to the *same* string. There are two possible states, shown in Figure 10.2.

In one case, `a` and `b` refer to two different objects that have the same value. In the second case, they refer to the same object.

To check whether two variables refer to the same object, you can use the `is` operator.

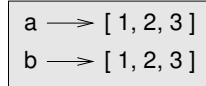


Figure 10.3: State diagram.

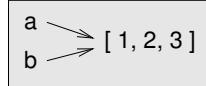


Figure 10.4: State diagram.

```

>>> a = 'banana'
>>> b = 'banana'
>>> a is b
True
  
```

In this example, Python only created one string object, and both `a` and `b` refer to it. But when you create two lists, you get two objects:

```

>>> a = [1, 2, 3]
>>> b = [1, 2, 3]
>>> a is b
False
  
```

So the state diagram looks like Figure 10.3.

In this case we would say that the two lists are **equivalent**, because they have the same elements, but not **identical**, because they are not the same object. If two objects are identical, they are also equivalent, but if they are equivalent, they are not necessarily identical.

Until now, we have been using “object” and “value” interchangeably, but it is more precise to say that an object has a value. If you evaluate `[1, 2, 3]`, you get a list object whose value is a sequence of integers. If another list has the same elements, we say it has the same value, but it is not the same object.

10.11 Aliasing

If `a` refers to an object and you assign `b = a`, then both variables refer to the same object:

```

>>> a = [1, 2, 3]
>>> b = a
>>> b is a
True
  
```

The state diagram looks like Figure 10.4.

The association of a variable with an object is called a **reference**. In this example, there are two references to the same object.

An object with more than one reference has more than one name, so we say that the object is **aliased**.

If the aliased object is mutable, changes made with one alias affect the other:

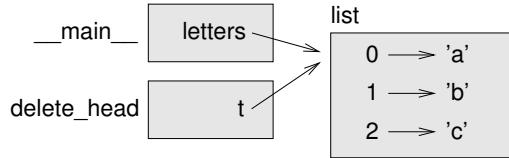


Figure 10.5: Stack diagram.

```
>>> b[0] = 42
>>> a
[42, 2, 3]
```

Although this behavior can be useful, it is error-prone. In general, it is safer to avoid aliasing when you are working with mutable objects.

For immutable objects like strings, aliasing is not as much of a problem. In this example:

```
a = 'banana'
b = 'banana'
```

It almost never makes a difference whether `a` and `b` refer to the same string or not.

10.12 List arguments

When you pass a list to a function, the function gets a reference to the list. If the function modifies the list, the caller sees the change. For example, `delete_head` removes the first element from a list:

```
def delete_head(t):
    del t[0]
```

Here's how it is used:

```
>>> letters = ['a', 'b', 'c']
>>> delete_head(letters)
>>> letters
['b', 'c']
```

The parameter `t` and the variable `letters` are aliases for the same object. The stack diagram looks like Figure 10.5.

Since the list is shared by two frames, I drew it between them.

It is important to distinguish between operations that modify lists and operations that create new lists. For example, the `append` method modifies a list, but the `+` operator creates a new list.

Here's an example using `append`:

```
>>> t1 = [1, 2]
>>> t2 = t1.append(3)
>>> t1
[1, 2, 3]
>>> t2
None
```

The return value from `append` is `None`.

Here's an example using the `+` operator:

```
>>> t3 = t1 + [4]
>>> t1
[1, 2, 3]
>>> t3
[1, 2, 3, 4]
```

The result of the operator is a new list, and the original list is unchanged.

This difference is important when you write functions that are supposed to modify lists. For example, this function *does not* delete the head of a list:

```
def bad_delete_head(t):
    t = t[1:]          # WRONG!
```

The slice operator creates a new list and the assignment makes `t` refer to it, but that doesn't affect the caller.

```
>>> t4 = [1, 2, 3]
>>> bad_delete_head(t4)
>>> t4
[1, 2, 3]
```

At the beginning of `bad_delete_head`, `t` and `t4` refer to the same list. At the end, `t` refers to a new list, but `t4` still refers to the original, unmodified list.

An alternative is to write a function that creates and returns a new list. For example, `tail` returns all but the first element of a list:

```
def tail(t):
    return t[1:]
```

This function leaves the original list unmodified. Here's how it is used:

```
>>> letters = ['a', 'b', 'c']
>>> rest = tail(letters)
>>> rest
['b', 'c']
```

10.13 Debugging

Careless use of lists (and other mutable objects) can lead to long hours of debugging. Here are some common pitfalls and ways to avoid them:

1. Most list methods modify the argument and return `None`. This is the opposite of the string methods, which return a new string and leave the original alone.

If you are used to writing string code like this:

```
word = word.strip()
```

It is tempting to write list code like this:

```
t = t.sort()          # WRONG!
```

Because `sort` returns `None`, the next operation you perform with `t` is likely to fail.

Before using list methods and operators, you should read the documentation carefully and then test them in interactive mode.

2. Pick an idiom and stick with it.

Part of the problem with lists is that there are too many ways to do things. For example, to remove an element from a list, you can use `pop`, `remove`, `del`, or even a slice assignment.

To add an element, you can use the `append` method or the `+` operator. Assuming that `t` is a list and `x` is a list element, these are correct:

```
t.append(x)
t = t + [x]
t += [x]
```

And these are wrong:

```
t.append([x])      # WRONG!
t = t.append(x)    # WRONG!
t + [x]            # WRONG!
t = t + x         # WRONG!
```

Try out each of these examples in interactive mode to make sure you understand what they do. Notice that only the last one causes a runtime error; the other three are legal, but they do the wrong thing.

3. Make copies to avoid aliasing.

If you want to use a method like `sort` that modifies the argument, but you need to keep the original list as well, you can make a copy.

```
>>> t = [3, 1, 2]
>>> t2 = t[:]
>>> t2.sort()
>>> t
[3, 1, 2]
>>> t2
[1, 2, 3]
```

In this example you could also use the built-in function `sorted`, which returns a new, sorted list and leaves the original alone.

```
>>> t2 = sorted(t)
>>> t
[3, 1, 2]
>>> t2
[1, 2, 3]
```

10.14 Glossary

list: A sequence of values.

element: One of the values in a list (or other sequence), also called items.

nested list: A list that is an element of another list.

accumulator: A variable used in a loop to add up or accumulate a result.

augmented assignment: A statement that updates the value of a variable using an operator like `+=`.

reduce: A processing pattern that traverses a sequence and accumulates the elements into a single result.

map: A processing pattern that traverses a sequence and performs an operation on each element.

filter: A processing pattern that traverses a list and selects the elements that satisfy some criterion.

object: Something a variable can refer to. An object has a type and a value.

equivalent: Having the same value.

identical: Being the same object (which implies equivalence).

reference: The association between a variable and its value.

aliasing: A circumstance where two or more variables refer to the same object.

delimiter: A character or string used to indicate where a string should be split.

10.15 Exercises

You can download solutions to these exercises from http://thinkpython2.com/code/list_exercises.py.

Exercise 10.1. Write a function called `nested_sum` that takes a list of lists of integers and adds up the elements from all of the nested lists. For example:

```
>>> t = [[1, 2], [3], [4, 5, 6]]  
>>> nested_sum(t)  
21
```

Exercise 10.2. Write a function called `cumsum` that takes a list of numbers and returns the cumulative sum; that is, a new list where the *i*th element is the sum of the first $i + 1$ elements from the original list. For example:

```
>>> t = [1, 2, 3]  
>>> cumsum(t)  
[1, 3, 6]
```

Exercise 10.3. Write a function called `middle` that takes a list and returns a new list that contains all but the first and last elements. For example:

```
>>> t = [1, 2, 3, 4]  
>>> middle(t)  
[2, 3]
```

Exercise 10.4. Write a function called `chop` that takes a list, modifies it by removing the first and last elements, and returns `None`. For example:

```
>>> t = [1, 2, 3, 4]
>>> chop(t)
>>> t
[2, 3]
```

Exercise 10.5. Write a function called `is_sorted` that takes a list as a parameter and returns `True` if the list is sorted in ascending order and `False` otherwise. For example:

```
>>> is_sorted([1, 2, 2])
True
>>> is_sorted(['b', 'a'])
False
```

Exercise 10.6. Two words are *anagrams* if you can rearrange the letters from one to spell the other. Write a function called `is_anagram` that takes two strings and returns `True` if they are anagrams.

Exercise 10.7. Write a function called `has_duplicates` that takes a list and returns `True` if there is any element that appears more than once. It should not modify the original list.

Exercise 10.8. This exercise pertains to the so-called *Birthday Paradox*, which you can read about at http://en.wikipedia.org/wiki/Birthday_paradox.

If there are 23 students in your class, what are the chances that two of them have the same birthday? You can estimate this probability by generating random samples of 23 birthdays and checking for matches. Hint: you can generate random birthdays with the `randint` function in the `random` module.

You can download my solution from <http://thinkpython2.com/code/birthday.py>.

Exercise 10.9. Write a function that reads the file `words.txt` and builds a list with one element per word. Write two versions of this function, one using the `append` method and the other using the idiom `t = t + [x]`. Which one takes longer to run? Why?

Solution: <http://thinkpython2.com/code/wordlist.py>.

Exercise 10.10. To check whether a word is in the word list, you could use the `in` operator, but it would be slow because it searches through the words in order.

Because the words are in alphabetical order, we can speed things up with a bisection search (also known as binary search), which is similar to what you do when you look a word up in the dictionary (the book, not the data structure). You start in the middle and check to see whether the word you are looking for comes before the word in the middle of the list. If so, you search the first half of the list the same way. Otherwise you search the second half.

Either way, you cut the remaining search space in half. If the word list has 113,809 words, it will take about 17 steps to find the word or conclude that it's not there.

Write a function called `in_bisect` that takes a sorted list and a target value and returns `True` if the word is in the list and `False` if it's not.

Or you could read the documentation of the `bisect` module and use that! Solution: <http://thinkpython2.com/code/inlist.py>.

Exercise 10.11. Two words are a “reverse pair” if each is the reverse of the other. Write a program that finds all the reverse pairs in the word list. Solution: http://thinkpython2.com/code/reverse_pair.py.

Exercise 10.12. Two words “interlock” if taking alternating letters from each forms a new word. For example, “shoe” and “cold” interlock to form “schooled”. Solution: <http://thinkpython2.com/code/interlock.py>.

thinkpython2.com/code/interlock.py. Credit: This exercise is inspired by an example at <http://puzzlers.org>.

1. Write a program that finds all pairs of words that interlock. Hint: don't enumerate all pairs!
2. Can you find any words that are three-way interlocked; that is, every third letter forms a word, starting from the first, second or third?

Chapter 11

Dictionaries

This chapter presents another built-in type called a dictionary. Dictionaries are one of Python’s best features; they are the building blocks of many efficient and elegant algorithms.

11.1 A dictionary is a mapping

A **dictionary** is like a list, but more general. In a list, the indices have to be integers; in a dictionary they can be (almost) any type.

A dictionary contains a collection of indices, which are called **keys**, and a collection of values. Each key is associated with a single value. The association of a key and a value is called a **key-value pair** or sometimes an **item**.

In mathematical language, a dictionary represents a **mapping** from keys to values, so you can also say that each key “maps to” a value. As an example, we’ll build a dictionary that maps from English to Spanish words, so the keys and the values are all strings.

The function `dict` creates a new dictionary with no items. Because `dict` is the name of a built-in function, you should avoid using it as a variable name.

```
>>> eng2sp = dict()  
>>> eng2sp  
{}
```

The squiggly-brackets, `{}`, represent an empty dictionary. To add items to the dictionary, you can use square brackets:

```
>>> eng2sp['one'] = 'uno'
```

This line creates an item that maps from the key `'one'` to the value `'uno'`. If we print the dictionary again, we see a key-value pair with a colon between the key and value:

```
>>> eng2sp  
{'one': 'uno'}
```

This output format is also an input format. For example, you can create a new dictionary with three items:

```
>>> eng2sp = {'one': 'uno', 'two': 'dos', 'three': 'tres'}
```

But if you print `eng2sp`, you might be surprised:

```
>>> eng2sp
{'one': 'uno', 'three': 'tres', 'two': 'dos'}
```

The order of the key-value pairs might not be the same. If you type the same example on your computer, you might get a different result. In general, the order of items in a dictionary is unpredictable.

But that's not a problem because the elements of a dictionary are never indexed with integer indices. Instead, you use the keys to look up the corresponding values:

```
>>> eng2sp['two']
'dos'
```

The key '`two`' always maps to the value '`dos`' so the order of the items doesn't matter.

If the key isn't in the dictionary, you get an exception:

```
>>> eng2sp['four']
KeyError: 'four'
```

The `len` function works on dictionaries; it returns the number of key-value pairs:

```
>>> len(eng2sp)
3
```

The `in` operator works on dictionaries, too; it tells you whether something appears as a *key* in the dictionary (appearing as a value is not good enough).

```
>>> 'one' in eng2sp
True
>>> 'uno' in eng2sp
False
```

To see whether something appears as a value in a dictionary, you can use the method `values`, which returns a collection of values, and then use the `in` operator:

```
>>> vals = eng2sp.values()
>>> 'uno' in vals
True
```

The `in` operator uses different algorithms for lists and dictionaries. For lists, it searches the elements of the list in order, as in Section 8.6. As the list gets longer, the search time gets longer in direct proportion.

Python dictionaries use a data structure called a **hashtable** that has a remarkable property: the `in` operator takes about the same amount of time no matter how many items are in the dictionary. I explain how that's possible in Section B.4, but the explanation might not make sense until you've read a few more chapters.

11.2 Dictionary as a collection of counters

Suppose you are given a string and you want to count how many times each letter appears. There are several ways you could do it:

1. You could create 26 variables, one for each letter of the alphabet. Then you could traverse the string and, for each character, increment the corresponding counter, probably using a chained conditional.
2. You could create a list with 26 elements. Then you could convert each character to a number (using the built-in function `ord`), use the number as an index into the list, and increment the appropriate counter.
3. You could create a dictionary with characters as keys and counters as the corresponding values. The first time you see a character, you would add an item to the dictionary. After that you would increment the value of an existing item.

Each of these options performs the same computation, but each of them implements that computation in a different way.

An **implementation** is a way of performing a computation; some implementations are better than others. For example, an advantage of the dictionary implementation is that we don't have to know ahead of time which letters appear in the string and we only have to make room for the letters that do appear.

Here is what the code might look like:

```
def histogram(s):  
    d = dict()  
    for c in s:  
        if c not in d:  
            d[c] = 1  
        else:  
            d[c] += 1  
    return d
```

The name of the function is `histogram`, which is a statistical term for a collection of counters (or frequencies).

The first line of the function creates an empty dictionary. The `for` loop traverses the string. Each time through the loop, if the character `c` is not in the dictionary, we create a new item with key `c` and the initial value 1 (since we have seen this letter once). If `c` is already in the dictionary we increment `d[c]`.

Here's how it works:

```
>>> h = histogram('brontosaurus')  
>>> h  
{'a': 1, 'b': 1, 'o': 2, 'n': 1, 's': 2, 'r': 2, 'u': 2, 't': 1}
```

The histogram indicates that the letters '`a`' and '`b`' appear once; '`o`' appears twice, and so on.

Dictionaries have a method called `get` that takes a key and a default value. If the key appears in the dictionary, `get` returns the corresponding value; otherwise it returns the default value. For example:

```
>>> h = histogram('a')  
>>> h  
{'a': 1}  
>>> h.get('a', 0)
```

```

1
>>> h.get('c', 0)
0

```

As an exercise, use `get` to write `histogram` more concisely. You should be able to eliminate the `if` statement.

11.3 Looping and dictionaries

If you use a dictionary in a `for` statement, it traverses the keys of the dictionary. For example, `print_hist` prints each key and the corresponding value:

```

def print_hist(h):
    for c in h:
        print(c, h[c])

```

Here's what the output looks like:

```

>>> h = histogram('parrot')
>>> print_hist(h)
a 1
p 1
r 2
t 1
o 1

```

Again, the keys are in no particular order. To traverse the keys in sorted order, you can use the built-in function `sorted`:

```

>>> for key in sorted(h):
...     print(key, h[key])
a 1
o 1
p 1
r 2
t 1

```

11.4 Reverse lookup

Given a dictionary `d` and a key `k`, it is easy to find the corresponding value `v = d[k]`. This operation is called a **lookup**.

But what if you have `v` and you want to find `k`? You have two problems: first, there might be more than one key that maps to the value `v`. Depending on the application, you might be able to pick one, or you might have to make a list that contains all of them. Second, there is no simple syntax to do a **reverse lookup**; you have to search.

Here is a function that takes a value and returns the first key that maps to that value:

```

def reverse_lookup(d, v):
    for k in d:
        if d[k] == v:
            return k
    raise LookupError()

```

This function is yet another example of the search pattern, but it uses a feature we haven't seen before, `raise`. The `raise statement` causes an exception; in this case it causes a `LookupError`, which is a built-in exception used to indicate that a lookup operation failed.

If we get to the end of the loop, that means `v` doesn't appear in the dictionary as a value, so we raise an exception.

Here is an example of a successful reverse lookup:

```
>>> h = histogram('parrot')
>>> key = reverse_lookup(h, 2)
>>> key
'r'
```

And an unsuccessful one:

```
>>> key = reverse_lookup(h, 3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    File "<stdin>", line 5, in reverse_lookup
LookupError
```

The effect when you raise an exception is the same as when Python raises one: it prints a traceback and an error message.

When you raise an exception, you can provide a detailed error message as an optional argument. For example:

```
>>> raise LookupError('value does not appear in the dictionary')
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
LookupError: value does not appear in the dictionary
```

A reverse lookup is much slower than a forward lookup; if you have to do it often, or if the dictionary gets big, the performance of your program will suffer.

11.5 Dictionaries and lists

Lists can appear as values in a dictionary. For example, if you are given a dictionary that maps from letters to frequencies, you might want to invert it; that is, create a dictionary that maps from frequencies to letters. Since there might be several letters with the same frequency, each value in the inverted dictionary should be a list of letters.

Here is a function that inverts a dictionary:

```
def invert_dict(d):
    inverse = dict()
    for key in d:
        val = d[key]
        if val not in inverse:
            inverse[val] = [key]
        else:
            inverse[val].append(key)
    return inverse
```

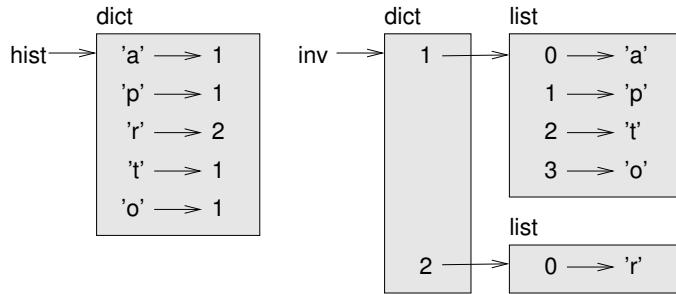


Figure 11.1: State diagram.

Each time through the loop, `key` gets a key from `d` and `val` gets the corresponding value. If `val` is not in `inverse`, that means we haven't seen it before, so we create a new item and initialize it with a **singleton** (a list that contains a single element). Otherwise we have seen this value before, so we append the corresponding key to the list.

Here is an example:

```
>>> hist = histogram('parrot')
>>> hist
{'a': 1, 'p': 1, 'r': 2, 't': 1, 'o': 1}
>>> inverse = invert_dict(hist)
>>> inverse
{1: ['a', 'p', 't', 'o'], 2: ['r']}
```

Figure 11.1 is a state diagram showing `hist` and `inverse`. A dictionary is represented as a box with the type `dict` above it and the key-value pairs inside. If the values are integers, floats or strings, I draw them inside the box, but I usually draw lists outside the box, just to keep the diagram simple.

Lists can be values in a dictionary, as this example shows, but they cannot be keys. Here's what happens if you try:

```
>>> t = [1, 2, 3]
>>> d = dict()
>>> d[t] = 'oops'
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: list objects are unhashable
```

I mentioned earlier that a dictionary is implemented using a hashtable and that means that the keys have to be **hashable**.

A **hash** is a function that takes a value (of any kind) and returns an integer. Dictionaries use these integers, called hash values, to store and look up key-value pairs.

This system works fine if the keys are immutable. But if the keys are mutable, like lists, bad things happen. For example, when you create a key-value pair, Python hashes the key and stores it in the corresponding location. If you modify the key and then hash it again, it would go to a different location. In that case you might have two entries for the same key, or you might not be able to find a key. Either way, the dictionary wouldn't work correctly.

That's why keys have to be hashable, and why mutable types like lists aren't. The simplest way to get around this limitation is to use tuples, which we will see in the next chapter.

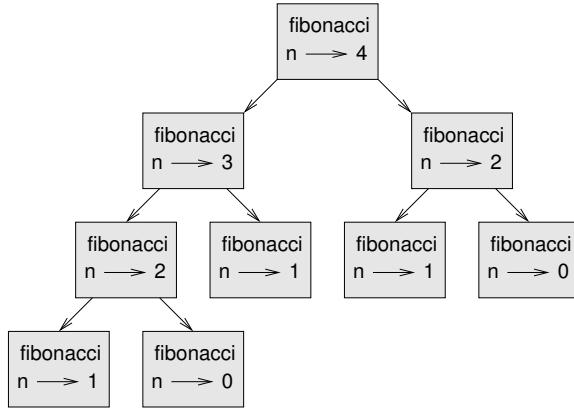


Figure 11.2: Call graph.

Since dictionaries are mutable, they can't be used as keys, but they *can* be used as values.

11.6 Memos

If you played with the `fibonacci` function from Section 6.7, you might have noticed that the bigger the argument you provide, the longer the function takes to run. Furthermore, the run time increases quickly.

To understand why, consider Figure 11.2, which shows the **call graph** for `fibonacci` with `n=4`:

A call graph shows a set of function frames, with lines connecting each frame to the frames of the functions it calls. At the top of the graph, `fibonacci` with `n=4` calls `fibonacci` with `n=3` and `n=2`. In turn, `fibonacci` with `n=3` calls `fibonacci` with `n=2` and `n=1`. And so on.

Count how many times `fibonacci(0)` and `fibonacci(1)` are called. This is an inefficient solution to the problem, and it gets worse as the argument gets bigger.

One solution is to keep track of values that have already been computed by storing them in a dictionary. A previously computed value that is stored for later use is called a **memo**. Here is a “memoized” version of `fibonacci`:

```

known = {0:0, 1:1}

def fibonacci(n):
    if n in known:
        return known[n]

    res = fibonacci(n-1) + fibonacci(n-2)
    known[n] = res
    return res
  
```

`known` is a dictionary that keeps track of the Fibonacci numbers we already know. It starts with two items: 0 maps to 0 and 1 maps to 1.

Whenever `fibonacci` is called, it checks `known`. If the result is already there, it can return immediately. Otherwise it has to compute the new value, add it to the dictionary, and return it.

If you run this version of `fibonacci` and compare it with the original, you will find that it is much faster.

11.7 Global variables

In the previous example, `known` is created outside the function, so it belongs to the special frame called `__main__`. Variables in `__main__` are sometimes called **global** because they can be accessed from any function. Unlike local variables, which disappear when their function ends, global variables persist from one function call to the next.

It is common to use global variables for **flags**; that is, boolean variables that indicate (“flag”) whether a condition is true. For example, some programs use a flag named `verbose` to control the level of detail in the output:

```
verbose = True
```

```
def example1():
    if verbose:
        print('Running example1')
```

If you try to reassign a global variable, you might be surprised. The following example is supposed to keep track of whether the function has been called:

```
been_called = False

def example2():
    been_called = True      # WRONG
```

But if you run it you will see that the value of `been_called` doesn’t change. The problem is that `example2` creates a new local variable named `been_called`. The local variable goes away when the function ends, and has no effect on the global variable.

To reassign a global variable inside a function you have to **declare** the global variable before you use it:

```
been_called = False

def example2():
    global been_called
    been_called = True
```

The **global statement** tells the interpreter something like, “In this function, when I say `been_called`, I mean the global variable; don’t create a local one.”

Here’s an example that tries to update a global variable:

```
count = 0

def example3():
    count = count + 1      # WRONG
```

If you run it you get:

```
UnboundLocalError: local variable 'count' referenced before assignment  
Python assumes that count is local, and under that assumption you are reading it before writing it. The solution, again, is to declare count global.
```

```
def example3():  
    global count  
    count += 1
```

If a global variable refers to a mutable value, you can modify the value without declaring the variable:

```
known = {0:0, 1:1}
```

```
def example4():  
    known[2] = 1
```

So you can add, remove and replace elements of a global list or dictionary, but if you want to reassign the variable, you have to declare it:

```
def example5():  
    global known  
    known = dict()
```

Global variables can be useful, but if you have a lot of them, and you modify them frequently, they can make programs hard to debug.

11.8 Debugging

As you work with bigger datasets it can become unwieldy to debug by printing and checking the output by hand. Here are some suggestions for debugging large datasets:

Scale down the input: If possible, reduce the size of the dataset. For example if the program reads a text file, start with just the first 10 lines, or with the smallest example you can find. You can either edit the files themselves, or (better) modify the program so it reads only the first n lines.

If there is an error, you can reduce n to the smallest value that manifests the error, and then increase it gradually as you find and correct errors.

Check summaries and types: Instead of printing and checking the entire dataset, consider printing summaries of the data: for example, the number of items in a dictionary or the total of a list of numbers.

A common cause of runtime errors is a value that is not the right type. For debugging this kind of error, it is often enough to print the type of a value.

Write self-checks: Sometimes you can write code to check for errors automatically. For example, if you are computing the average of a list of numbers, you could check that the result is not greater than the largest element in the list or less than the smallest. This is called a “sanity check” because it detects results that are “insane”.

Another kind of check compares the results of two different computations to see if they are consistent. This is called a “consistency check”.

Format the output: Formatting debugging output can make it easier to spot an error. We saw an example in Section 6.9. Another tool you might find useful is the `pprint` module, which provides a `pprint` function that displays built-in types in a more human-readable format (`pprint` stands for “pretty print”).

Again, time you spend building scaffolding can reduce the time you spend debugging.

11.9 Glossary

mapping: A relationship in which each element of one set corresponds to an element of another set.

dictionary: A mapping from keys to their corresponding values.

key-value pair: The representation of the mapping from a key to a value.

item: In a dictionary, another name for a key-value pair.

key: An object that appears in a dictionary as the first part of a key-value pair.

value: An object that appears in a dictionary as the second part of a key-value pair. This is more specific than our previous use of the word “value”.

implementation: A way of performing a computation.

hashtable: The algorithm used to implement Python dictionaries.

hash function: A function used by a hashtable to compute the location for a key.

hashable: A type that has a hash function. Immutable types like integers, floats and strings are hashable; mutable types like lists and dictionaries are not.

lookup: A dictionary operation that takes a key and finds the corresponding value.

reverse lookup: A dictionary operation that takes a value and finds one or more keys that map to it.

raise statement: A statement that (deliberately) raises an exception.

singleton: A list (or other sequence) with a single element.

call graph: A diagram that shows every frame created during the execution of a program, with an arrow from each caller to each callee.

memo: A computed value stored to avoid unnecessary future computation.

global variable: A variable defined outside a function. Global variables can be accessed from any function.

global statement: A statement that declares a variable name global.

flag: A boolean variable used to indicate whether a condition is true.

declaration: A statement like `global` that tells the interpreter something about a variable.

11.10 Exercises

Exercise 11.1. Write a function that reads the words in `words.txt` and stores them as keys in a dictionary. It doesn't matter what the values are. Then you can use the `in` operator as a fast way to check whether a string is in the dictionary.

If you did Exercise 10.10, you can compare the speed of this implementation with the list `in` operator and the bisection search.

Exercise 11.2. Read the documentation of the dictionary method `setdefault` and use it to write a more concise version of `invert_dict`. Solution: http://thinkpython2.com/code/invert_dict.py.

Exercise 11.3. Memoize the Ackermann function from Exercise 6.2 and see if memoization makes it possible to evaluate the function with bigger arguments. Hint: no. Solution: http://thinkpython2.com/code/ackermann_memo.py.

Exercise 11.4. If you did Exercise 10.7, you already have a function named `has_duplicates` that takes a list as a parameter and returns `True` if there is any object that appears more than once in the list.

Use a dictionary to write a faster, simpler version of `has_duplicates`. Solution: http://thinkpython2.com/code/has_duplicates.py.

Exercise 11.5. Two words are “rotate pairs” if you can rotate one of them and get the other (see `rotate_word` in Exercise 8.5).

Write a program that reads a wordlist and finds all the rotate pairs. Solution: http://thinkpython2.com/code/rotate_pairs.py.

Exercise 11.6. Here's another Puzzler from Car Talk (<http://www.cartalk.com/content/puzzlers>):

This was sent in by a fellow named Dan O'Leary. He came upon a common one-syllable, five-letter word recently that has the following unique property. When you remove the first letter, the remaining letters form a homophone of the original word, that is a word that sounds exactly the same. Replace the first letter, that is, put it back and remove the second letter and the result is yet another homophone of the original word. And the question is, what's the word?

Now I'm going to give you an example that doesn't work. Let's look at the five-letter word, 'wrack.' W-R-A-C-K, you know like to 'wrack with pain.' If I remove the first letter, I am left with a four-letter word, 'R-A-C-K.' As in, 'Holy cow, did you see the rack on that buck! It must have been a nine-pointer!' It's a perfect homophone. If you put the 'w' back, and remove the 'r,' instead, you're left with the word, 'wack,' which is a real word, it's just not a homophone of the other two words.

But there is, however, at least one word that Dan and we know of, which will yield two homophones if you remove either of the first two letters to make two, new four-letter words. The question is, what's the word?

You can use the dictionary from Exercise 11.1 to check whether a string is in the word list.

To check whether two words are homophones, you can use the CMU Pronouncing Dictionary. You can download it from <http://www.speech.cs.cmu.edu/cgi-bin/cmudict> or from <http://thinkpython2.com/code/c06d> and you can also download <http://thinkpython2.com/code/pronounce.py>, which provides a function named `read_dictionary` that reads the pronouncing dictionary and returns a Python dictionary that maps from each word to a string that describes its primary pronunciation.

Write a program that lists all the words that solve the Puzzler. Solution: <http://thinkpython2.com/code/homophone.py>.

Chapter 12

Tuples

This chapter presents one more built-in type, the tuple, and then shows how lists, dictionaries, and tuples work together. I also present a useful feature for variable-length argument lists, the gather and scatter operators.

One note: there is no consensus on how to pronounce “tuple”. Some people say “tuh-ple”, which rhymes with “supple”. But in the context of programming, most people say “too-ple”, which rhymes with “quadruple”.

12.1 Tuples are immutable

A tuple is a sequence of values. The values can be any type, and they are indexed by integers, so in that respect tuples are a lot like lists. The important difference is that tuples are immutable.

Syntactically, a tuple is a comma-separated list of values:

```
>>> t = 'a', 'b', 'c', 'd', 'e'
```

Although it is not necessary, it is common to enclose tuples in parentheses:

```
>>> t = ('a', 'b', 'c', 'd', 'e')
```

To create a tuple with a single element, you have to include a final comma:

```
>>> t1 = 'a',  
>>> type(t1)  
<class 'tuple'>
```

A value in parentheses is not a tuple:

```
>>> t2 = ('a')  
>>> type(t2)  
<class 'str'>
```

Another way to create a tuple is the built-in function `tuple`. With no argument, it creates an empty tuple:

```
>>> t = tuple()  
>>> t  
()
```

If the argument is a sequence (string, list or tuple), the result is a tuple with the elements of the sequence:

```
>>> t = tuple('lupins')
>>> t
('l', 'u', 'p', 'i', 'n', 's')
```

Because `tuple` is the name of a built-in function, you should avoid using it as a variable name.

Most list operators also work on tuples. The bracket operator indexes an element:

```
>>> t = ('a', 'b', 'c', 'd', 'e')
>>> t[0]
'a'
```

And the slice operator selects a range of elements.

```
>>> t[1:3]
('b', 'c')
```

But if you try to modify one of the elements of the tuple, you get an error:

```
>>> t[0] = 'A'
TypeError: object doesn't support item assignment
```

Because tuples are immutable, you can't modify the elements. But you can replace one tuple with another:

```
>>> t = ('A',) + t[1:]
>>> t
('A', 'b', 'c', 'd', 'e')
```

This statement makes a new tuple and then makes `t` refer to it.

The relational operators work with tuples and other sequences; Python starts by comparing the first element from each sequence. If they are equal, it goes on to the next elements, and so on, until it finds elements that differ. Subsequent elements are not considered (even if they are really big).

```
>>> (0, 1, 2) < (0, 3, 4)
True
>>> (0, 1, 2000000) < (0, 3, 4)
True
```

12.2 Tuple assignment

It is often useful to swap the values of two variables. With conventional assignments, you have to use a temporary variable. For example, to swap `a` and `b`:

```
>>> temp = a
>>> a = b
>>> b = temp
```

This solution is cumbersome; **tuple assignment** is more elegant:

```
>>> a, b = b, a
```

The left side is a tuple of variables; the right side is a tuple of expressions. Each value is assigned to its respective variable. All the expressions on the right side are evaluated before any of the assignments.

The number of variables on the left and the number of values on the right have to be the same:

```
>>> a, b = 1, 2, 3  
ValueError: too many values to unpack
```

More generally, the right side can be any kind of sequence (string, list or tuple). For example, to split an email address into a user name and a domain, you could write:

```
>>> addr = 'monty@python.org'  
>>> uname, domain = addr.split('@')
```

The return value from `split` is a list with two elements; the first element is assigned to `uname`, the second to `domain`.

```
>>> uname  
'monty'  
>>> domain  
'python.org'
```

12.3 Tuples as return values

Strictly speaking, a function can only return one value, but if the value is a tuple, the effect is the same as returning multiple values. For example, if you want to divide two integers and compute the quotient and remainder, it is inefficient to compute `x//y` and then `x%y`. It is better to compute them both at the same time.

The built-in function `divmod` takes two arguments and returns a tuple of two values, the quotient and remainder. You can store the result as a tuple:

```
>>> t = divmod(7, 3)  
>>> t  
(2, 1)
```

Or use tuple assignment to store the elements separately:

```
>>> quot, rem = divmod(7, 3)  
>>> quot  
2  
>>> rem  
1
```

Here is an example of a function that returns a tuple:

```
def min_max(t):  
    return min(t), max(t)
```

`max` and `min` are built-in functions that find the largest and smallest elements of a sequence. `min_max` computes both and returns a tuple of two values.

12.4 Variable-length argument tuples

Functions can take a variable number of arguments. A parameter name that begins with `*` gathers arguments into a tuple. For example, `printall` takes any number of arguments and prints them:

```
def printall(*args):
    print(args)
```

The gather parameter can have any name you like, but `args` is conventional. Here's how the function works:

```
>>> printall(1, 2.0, '3')
(1, 2.0, '3')
```

The complement of `gather` is `scatter`. If you have a sequence of values and you want to pass it to a function as multiple arguments, you can use the `*` operator. For example, `divmod` takes exactly two arguments; it doesn't work with a tuple:

```
>>> t = (7, 3)
>>> divmod(t)
TypeError: divmod expected 2 arguments, got 1
```

But if you scatter the tuple, it works:

```
>>> divmod(*t)
(2, 1)
```

Many of the built-in functions use variable-length argument tuples. For example, `max` and `min` can take any number of arguments:

```
>>> max(1, 2, 3)
3
```

But `sum` does not.

```
>>> sum(1, 2, 3)
TypeError: sum expected at most 2 arguments, got 3
```

As an exercise, write a function called `sum_all` that takes any number of arguments and returns their sum.

12.5 Lists and tuples

`zip` is a built-in function that takes two or more sequences and interleaves them. The name of the function refers to a zipper, which interleaves two rows of teeth.

This example zips a string and a list:

```
>>> s = 'abc'
>>> t = [0, 1, 2]
>>> zip(s, t)
<zip object at 0x7f7d0a9e7c48>
```

The result is a `zip object` that knows how to iterate through the pairs. The most common use of `zip` is in a `for` loop:

```
>>> for pair in zip(s, t):
...     print(pair)
...
('a', 0)
('b', 1)
('c', 2)
```

A `zip` object is a kind of **iterator**, which is any object that iterates through a sequence. Iterators are similar to lists in some ways, but unlike lists, you can't use an index to select an element from an iterator.

If you want to use list operators and methods, you can use a `zip` object to make a list:

```
>>> list(zip(s, t))
[('a', 0), ('b', 1), ('c', 2)]
```

The result is a list of tuples; in this example, each tuple contains a character from the string and the corresponding element from the list.

If the sequences are not the same length, the result has the length of the shorter one.

```
>>> list(zip('Anne', 'Elk'))
[('A', 'E'), ('n', 'l'), ('n', 'k')]
```

You can use tuple assignment in a `for` loop to traverse a list of tuples:

```
t = [('a', 0), ('b', 1), ('c', 2)]
for letter, number in t:
    print(number, letter)
```

Each time through the loop, Python selects the next tuple in the list and assigns the elements to `letter` and `number`. The output of this loop is:

```
0 a
1 b
2 c
```

If you combine `zip`, `for` and tuple assignment, you get a useful idiom for traversing two (or more) sequences at the same time. For example, `has_match` takes two sequences, `t1` and `t2`, and returns `True` if there is an index `i` such that `t1[i] == t2[i]`:

```
def has_match(t1, t2):
    for x, y in zip(t1, t2):
        if x == y:
            return True
    return False
```

If you need to traverse the elements of a sequence and their indices, you can use the built-in function `enumerate`:

```
for index, element in enumerate('abc'):
    print(index, element)
```

The result from `enumerate` is an `enumerate` object, which iterates a sequence of pairs; each pair contains an index (starting from 0) and an element from the given sequence. In this example, the output is

```
0 a
1 b
2 c
```

Again.

12.6 Dictionaries and tuples

Dictionaries have a method called `items` that returns a sequence of tuples, where each tuple is a key-value pair.

```
>>> d = {'a':0, 'b':1, 'c':2}
>>> t = d.items()
>>> t
dict_items([('c', 2), ('a', 0), ('b', 1)])
```

The result is a `dict_items` object, which is an iterator that iterates the key-value pairs. You can use it in a `for` loop like this:

```
>>> for key, value in d.items():
...     print(key, value)
...
c 2
a 0
b 1
```

As you should expect from a dictionary, the items are in no particular order.

Going in the other direction, you can use a list of tuples to initialize a new dictionary:

```
>>> t = [('a', 0), ('c', 2), ('b', 1)]
>>> d = dict(t)
>>> d
{'a': 0, 'c': 2, 'b': 1}
```

Combining `dict` with `zip` yields a concise way to create a dictionary:

```
>>> d = dict(zip('abc', range(3)))
>>> d
{'a': 0, 'c': 2, 'b': 1}
```

The `dictionary` method `update` also takes a list of tuples and adds them, as key-value pairs, to an existing dictionary.

It is common to use tuples as keys in dictionaries (primarily because you can't use lists). For example, a telephone directory might map from last-name, first-name pairs to telephone numbers. Assuming that we have defined `last`, `first` and `number`, we could write:

```
directory[last, first] = number
```

The expression in brackets is a tuple. We could use tuple assignment to traverse this dictionary.

```
for last, first in directory:
    print(first, last, directory[last,first])
```

This loop traverses the keys in `directory`, which are tuples. It assigns the elements of each tuple to `last` and `first`, then prints the name and corresponding telephone number.

There are two ways to represent tuples in a state diagram. The more detailed version shows the indices and elements just as they appear in a list. For example, the tuple `('Cleese', 'John')` would appear as in Figure 12.1.

But in a larger diagram you might want to leave out the details. For example, a diagram of the telephone directory might appear as in Figure 12.2.

Here the tuples are shown using Python syntax as a graphical shorthand. The telephone number in the diagram is the complaints line for the BBC, so please don't call it.

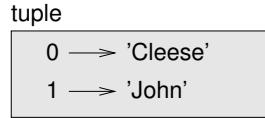


Figure 12.1: State diagram.

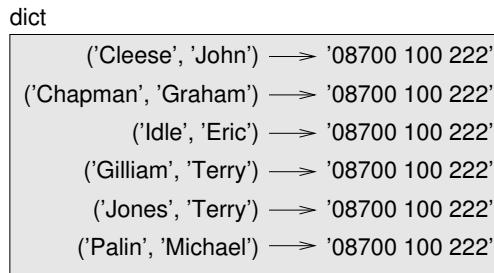


Figure 12.2: State diagram.

12.7 Sequences of sequences

I have focused on lists of tuples, but almost all of the examples in this chapter also work with lists of lists, tuples of tuples, and tuples of lists. To avoid enumerating the possible combinations, it is sometimes easier to talk about sequences of sequences.

In many contexts, the different kinds of sequences (strings, lists and tuples) can be used interchangeably. So how should you choose one over the others?

To start with the obvious, strings are more limited than other sequences because the elements have to be characters. They are also immutable. If you need the ability to change the characters in a string (as opposed to creating a new string), you might want to use a list of characters instead.

Lists are more common than tuples, mostly because they are mutable. But there are a few cases where you might prefer tuples:

1. In some contexts, like a `return` statement, it is syntactically simpler to create a tuple than a list.
2. If you want to use a sequence as a dictionary key, you have to use an immutable type like a tuple or string.
3. If you are passing a sequence as an argument to a function, using tuples reduces the potential for unexpected behavior due to aliasing.

Because tuples are immutable, they don't provide methods like `sort` and `reverse`, which modify existing lists. But Python provides the built-in function `sorted`, which takes any sequence and returns a new list with the same elements in sorted order, and `reversed`, which takes a sequence and returns an iterator that traverses the list in reverse order.

12.8 Debugging

Lists, dictionaries and tuples are examples of **data structures**; in this chapter we are starting to see compound data structures, like lists of tuples, or dictionaries that contain tuples as keys and lists as values. Compound data structures are useful, but they are prone to what I call **shape errors**; that is, errors caused when a data structure has the wrong type, size, or structure. For example, if you are expecting a list with one integer and I give you a plain old integer (not in a list), it won't work.

To help debug these kinds of errors, I have written a module called `structshape` that provides a function, also called `structshape`, that takes any kind of data structure as an argument and returns a string that summarizes its shape. You can download it from <http://thinkpython2.com/code/structshape.py>

Here's the result for a simple list:

```
>>> from structshape import structshape
>>> t = [1, 2, 3]
>>> structshape(t)
'list of 3 int'
```

A fancier program might write "list of 3 ints", but it was easier not to deal with plurals. Here's a list of lists:

```
>>> t2 = [[1,2], [3,4], [5,6]]
>>> structshape(t2)
'list of 3 list of 2 int'
```

If the elements of the list are not the same type, `structshape` groups them, in order, by type:

```
>>> t3 = [1, 2, 3, 4.0, '5', '6', [7], [8], 9]
>>> structshape(t3)
'list of (3 int, float, 2 str, 2 list of int, int)'
```

Here's a list of tuples:

```
>>> s = 'abc'
>>> lt = list(zip(t, s))
>>> structshape(lt)
'list of 3 tuple of (int, str)'
```

And here's a dictionary with 3 items that map integers to strings.

```
>>> d = dict(lt)
>>> structshape(d)
'dict of 3 int->str'
```

If you are having trouble keeping track of your data structures, `structshape` can help.

12.9 Glossary

tuple: An immutable sequence of elements.

tuple assignment: An assignment with a sequence on the right side and a tuple of variables on the left. The right side is evaluated and then its elements are assigned to the variables on the left.

gather: An operation that collects multiple arguments into a tuple.

scatter: An operation that makes a sequence behave like multiple arguments.

zip object: The result of calling a built-in function `zip`; an object that iterates through a sequence of tuples.

iterator: An object that can iterate through a sequence, but which does not provide list operators and methods.

data structure: A collection of related values, often organized in lists, dictionaries, tuples, etc.

shape error: An error caused because a value has the wrong shape; that is, the wrong type or size.

12.10 Exercises

Exercise 12.1. Write a function called `most_frequent` that takes a string and prints the letters in decreasing order of frequency. Find text samples from several different languages and see how letter frequency varies between languages. Compare your results with the tables at http://en.wikipedia.org/wiki/Letter_frequencies. Solution: http://thinkpython2.com/code/most_frequent.py.

Exercise 12.2. More anagrams!

1. Write a program that reads a word list from a file (see Section 9.1) and prints all the sets of words that are anagrams.

Here is an example of what the output might look like:

```
['deltas', 'desalt', 'lasted', 'salted', 'slated', 'staled']
['retainers', 'ternaries']
['generating', 'greatening']
['resmelts', 'smelters', 'termless']
```

Hint: you might want to build a dictionary that maps from a collection of letters to a list of words that can be spelled with those letters. The question is, how can you represent the collection of letters in a way that can be used as a key?

2. Modify the previous program so that it prints the longest list of anagrams first, followed by the second longest, and so on.
3. In Scrabble a “bingo” is when you play all seven tiles in your rack, along with a letter on the board, to form an eight-letter word. What collection of 8 letters forms the most possible bingos?

Solution: http://thinkpython2.com/code/anagram_sets.py.

Exercise 12.3. Two words form a “metathesis pair” if you can transform one into the other by swapping two letters; for example, “converse” and “conserve”. Write a program that finds all of the metathesis pairs in the dictionary. Hint: don’t test all pairs of words, and don’t test all possible swaps. Solution: <http://thinkpython2.com/code/metathesis.py>. Credit: This exercise is inspired by an example at <http://puzzlers.org>.

Exercise 12.4. Here's another Car Talk Puzzler (<http://www.cartalk.com/content/puzzlers>):

What is the longest English word, that remains a valid English word, as you remove its letters one at a time?

Now, letters can be removed from either end, or the middle, but you can't rearrange any of the letters. Every time you drop a letter, you wind up with another English word. If you do that, you're eventually going to wind up with one letter and that too is going to be an English word—one that's found in the dictionary. I want to know what's the longest word and how many letters does it have?

I'm going to give you a little modest example: Sprite. Ok? You start off with sprite, you take a letter off, one from the interior of the word, take the r away, and we're left with the word spite, then we take the e off the end, we're left with spit, we take the s off, we're left with pit, it, and I.

Write a program to find all words that can be reduced in this way, and then find the longest one.

This exercise is a little more challenging than most, so here are some suggestions:

1. You might want to write a function that takes a word and computes a list of all the words that can be formed by removing one letter. These are the "children" of the word.
2. Recursively, a word is reducible if any of its children are reducible. As a base case, you can consider the empty string reducible.
3. The wordlist I provided, `words.txt`, doesn't contain single letter words. So you might want to add "I", "a", and the empty string.
4. To improve the performance of your program, you might want to memoize the words that are known to be reducible.

Solution: <http://thinkpython2.com/code/reducible.py>.

Chapter 13

Case study: data structure selection

At this point you have learned about Python’s core data structures, and you have seen some of the algorithms that use them. If you would like to know more about algorithms, this might be a good time to read Chapter B. But you don’t have to read it before you go on; you can read it whenever you are interested.

This chapter presents a case study with exercises that let you think about choosing data structures and practice using them.

13.1 Word frequency analysis

As usual, you should at least attempt the exercises before you read my solutions.

Exercise 13.1. Write a program that reads a file, breaks each line into words, strips whitespace and punctuation from the words, and converts them to lowercase.

Hint: The string module provides a string named whitespace, which contains space, tab, new-line, etc., and punctuation which contains the punctuation characters. Let’s see if we can make Python swear:

```
>>> import string  
>>> string.punctuation  
'!"#$%&\'()*+,-./:;=>?@[\\"\\]^_`{|}~'
```

Also, you might consider using the string methods strip, replace and translate.

Exercise 13.2. Go to Project Gutenberg (<http://gutenberg.org>) and download your favorite out-of-copyright book in plain text format.

Modify your program from the previous exercise to read the book you downloaded, skip over the header information at the beginning of the file, and process the rest of the words as before.

Then modify the program to count the total number of words in the book, and the number of times each word is used.

Print the number of different words used in the book. Compare different books by different authors, written in different eras. Which author uses the most extensive vocabulary?

Exercise 13.3. Modify the program from the previous exercise to print the 20 most frequently used words in the book.

Exercise 13.4. Modify the previous program to read a word list (see Section 9.1) and then print all the words in the book that are not in the word list. How many of them are typos? How many of them are common words that should be in the word list, and how many of them are really obscure?

13.2 Random numbers

Given the same inputs, most computer programs generate the same outputs every time, so they are said to be **deterministic**. Determinism is usually a good thing, since we expect the same calculation to yield the same result. For some applications, though, we want the computer to be unpredictable. Games are an obvious example, but there are more.

Making a program truly nondeterministic turns out to be difficult, but there are ways to make it at least seem nondeterministic. One of them is to use algorithms that generate **pseudorandom** numbers. Pseudorandom numbers are not truly random because they are generated by a deterministic computation, but just by looking at the numbers it is all but impossible to distinguish them from random.

The `random` module provides functions that generate pseudorandom numbers (which I will simply call “random” from here on).

The function `random` returns a random float between 0.0 and 1.0 (including 0.0 but not 1.0). Each time you call `random`, you get the next number in a long series. To see a sample, run this loop:

```
import random

for i in range(10):
    x = random.random()
    print(x)
```

The function `randint` takes parameters `low` and `high` and returns an integer between `low` and `high` (including both).

```
>>> random.randint(5, 10)
5
>>> random.randint(5, 10)
9
```

To choose an element from a sequence at random, you can use `choice`:

```
>>> t = [1, 2, 3]
>>> random.choice(t)
2
>>> random.choice(t)
3
```

The `random` module also provides functions to generate random values from continuous distributions including Gaussian, exponential, gamma, and a few more.

Exercise 13.5. Write a function named `choose_from_hist` that takes a histogram as defined in Section 11.2 and returns a random value from the histogram, chosen with probability in proportion to frequency. For example, for this histogram:

```
>>> t = ['a', 'a', 'b']
>>> hist = histogram(t)
>>> hist
{'a': 2, 'b': 1}
```

your function should return 'a' with probability 2/3 and 'b' with probability 1/3.

13.3 Word histogram

You should attempt the previous exercises before you go on. You can download my solution from http://thinkpython2.com/code/analyze_book1.py. You will also need <http://thinkpython2.com/code/emma.txt>.

Here is a program that reads a file and builds a histogram of the words in the file:

```
import string

def process_file(filename):
    hist = dict()
    fp = open(filename)
    for line in fp:
        process_line(line, hist)
    return hist

def process_line(line, hist):
    line = line.replace('-', ' ')

    for word in line.split():
        word = word.strip(string.punctuation + string.whitespace)
        word = word.lower()
        hist[word] = hist.get(word, 0) + 1

hist = process_file('emma.txt')
```

This program reads `emma.txt`, which contains the text of *Emma* by Jane Austen.

`process_file` loops through the lines of the file, passing them one at a time to `process_line`. The histogram `hist` is being used as an accumulator.

`process_line` uses the string method `replace` to replace hyphens with spaces before using `split` to break the line into a list of strings. It traverses the list of words and uses `strip` and `lower` to remove punctuation and convert to lower case. (It is a shorthand to say that strings are “converted”; remember that strings are immutable, so methods like `strip` and `lower` return new strings.)

Finally, `process_line` updates the histogram by creating a new item or incrementing an existing one.

To count the total number of words in the file, we can add up the frequencies in the histogram:

```
def total_words(hist):
    return sum(hist.values())
```

The number of different words is just the number of items in the dictionary:

```
def different_words(hist):
    return len(hist)
```

Here is some code to print the results:

```
print('Total number of words:', total_words(hist))
print('Number of different words:', different_words(hist))
```

And the results:

```
Total number of words: 161080
Number of different words: 7214
```

13.4 Most common words

To find the most common words, we can make a list of tuples, where each tuple contains a word and its frequency, and sort it.

The following function takes a histogram and returns a list of word-frequency tuples:

```
def most_common(hist):
    t = []
    for key, value in hist.items():
        t.append((value, key))

    t.sort(reverse=True)
    return t
```

In each tuple, the frequency appears first, so the resulting list is sorted by frequency. Here is a loop that prints the ten most common words:

```
t = most_common(hist)
print('The most common words are:')
for freq, word in t[:10]:
    print(word, freq, sep='\t')
```

I use the keyword argument `sep` to tell `print` to use a tab character as a “separator”, rather than a space, so the second column is lined up. Here are the results from *Emma*:

The most common words are:

to	5242
the	5205
and	4897
of	4295
i	3191
a	3130
it	2529
her	2483
was	2400
she	2364

This code can be simplified using the `key` parameter of the `sort` function. If you are curious, you can read about it at <https://wiki.python.org/moin/HowTo/Sorting>.

13.5 Optional parameters

We have seen built-in functions and methods that take optional arguments. It is possible to write programmer-defined functions with optional arguments, too. For example, here is a function that prints the most common words in a histogram

```
def print_most_common(hist, num=10):
    t = most_common(hist)
    print('The most common words are:')
    for freq, word in t[:num]:
        print(word, freq, sep='\t')
```

The first parameter is required; the second is optional. The **default value** of num is 10.

If you only provide one argument:

```
print_most_common(hist)
```

num gets the default value. If you provide two arguments:

```
print_most_common(hist, 20)
```

num gets the value of the argument instead. In other words, the optional argument **overrides** the default value.

If a function has both required and optional parameters, all the required parameters have to come first, followed by the optional ones.

13.6 Dictionary subtraction

Finding the words from the book that are not in the word list from `words.txt` is a problem you might recognize as set subtraction; that is, we want to find all the words from one set (the words in the book) that are not in the other (the words in the list).

`subtract` takes dictionaries `d1` and `d2` and returns a new dictionary that contains all the keys from `d1` that are not in `d2`. Since we don't really care about the values, we set them all to `None`.

```
def subtract(d1, d2):
    res = dict()
    for key in d1:
        if key not in d2:
            res[key] = None
    return res
```

To find the words in the book that are not in `words.txt`, we can use `process_file` to build a histogram for `words.txt`, and then subtract:

```
words = process_file('words.txt')
diff = subtract(hist, words)

print("Words in the book that aren't in the word list:")
for word in diff:
    print(word, end=' ')
```

Here are some of the results from *Emma*:

Words in the book that aren't in the word list:
rencontre jane's blanche woodhouses disingenuousness
friend's venice apartment ...

Some of these words are names and possessives. Others, like "rencontre", are no longer in common use. But a few are common words that should really be in the list!

Exercise 13.6. *Python provides a data structure called set that provides many common set operations. You can read about them in Section 19.5, or read the documentation at <http://docs.python.org/3/library/stdtypes.html#types-set>.*

Write a program that uses set subtraction to find words in the book that are not in the word list.
Solution: http://thinkpython2.com/code/analyze_book2.py.

13.7 Random words

To choose a random word from the histogram, the simplest algorithm is to build a list with multiple copies of each word, according to the observed frequency, and then choose from the list:

```
def random_word(h):
    t = []
    for word, freq in h.items():
        t.extend([word] * freq)

    return random.choice(t)
```

The expression `[word] * freq` creates a list with `freq` copies of the string `word`. The `extend` method is similar to `append` except that the argument is a sequence.

This algorithm works, but it is not very efficient; each time you choose a random word, it rebuilds the list, which is as big as the original book. An obvious improvement is to build the list once and then make multiple selections, but the list is still big.

An alternative is:

1. Use `keys` to get a list of the words in the book.
2. Build a list that contains the cumulative sum of the word frequencies (see Exercise 10.2). The last item in this list is the total number of words in the book, n .
3. Choose a random number from 1 to n . Use a bisection search (See Exercise 10.10) to find the index where the random number would be inserted in the cumulative sum.
4. Use the index to find the corresponding word in the word list.

Exercise 13.7. *Write a program that uses this algorithm to choose a random word from the book.*
Solution: http://thinkpython2.com/code/analyze_book3.py.

13.8 Markov analysis

If you choose words from the book at random, you can get a sense of the vocabulary, but you probably won't get a sentence:

this the small regard harriet which knightley's it most things

A series of random words seldom makes sense because there is no relationship between successive words. For example, in a real sentence you would expect an article like “the” to be followed by an adjective or a noun, and probably not a verb or adverb.

One way to measure these kinds of relationships is Markov analysis, which characterizes, for a given sequence of words, the probability of the words that might come next. For example, the song *Eric, the Half a Bee* begins:

Half a bee, philosophically,
Must, ipso facto, half not be.
But half the bee has got to be
Vis a vis, its entity. D'you see?

But can a bee be said to be
Or not to be an entire bee
When half the bee is not a bee
Due to some ancient injury?

In this text, the phrase “half the” is always followed by the word “bee”, but the phrase “the bee” might be followed by either “has” or “is”.

The result of Markov analysis is a mapping from each prefix (like “half the” and “the bee”) to all possible suffixes (like “has” and “is”).

Given this mapping, you can generate a random text by starting with any prefix and choosing at random from the possible suffixes. Next, you can combine the end of the prefix and the new suffix to form the next prefix, and repeat.

For example, if you start with the prefix “Half a”, then the next word has to be “bee”, because the prefix only appears once in the text. The next prefix is “a bee”, so the next suffix might be “philosophically”, “be” or “due”.

In this example the length of the prefix is always two, but you can do Markov analysis with any prefix length.

Exercise 13.8. Markov analysis:

1. Write a program to read a text from a file and perform Markov analysis. The result should be a dictionary that maps from prefixes to a collection of possible suffixes. The collection might be a list, tuple, or dictionary; it is up to you to make an appropriate choice. You can test your program with prefix length two, but you should write the program in a way that makes it easy to try other lengths.
2. Add a function to the previous program to generate random text based on the Markov analysis. Here is an example from Emma with prefix length 2:

He was very clever, be it sweetness or be angry, ashamed or only amused, at such a stroke. She had never thought of Hannah till you were never meant for me?" "I cannot make speeches, Emma:" he soon cut it all himself.

For this example, I left the punctuation attached to the words. The result is almost syntactically correct, but not quite. Semantically, it almost makes sense, but not quite.

What happens if you increase the prefix length? Does the random text make more sense?

3. Once your program is working, you might want to try a mash-up: if you combine text from two or more books, the random text you generate will blend the vocabulary and phrases from the sources in interesting ways.

Credit: This case study is based on an example from Kernighan and Pike, The Practice of Programming, Addison-Wesley, 1999.

You should attempt this exercise before you go on; then you can download my solution from <http://thinkpython2.com/code/markov.py>. You will also need <http://thinkpython2.com/code/emma.txt>.

13.9 Data structures

Using Markov analysis to generate random text is fun, but there is also a point to this exercise: data structure selection. In your solution to the previous exercises, you had to choose:

- How to represent the prefixes.
- How to represent the collection of possible suffixes.
- How to represent the mapping from each prefix to the collection of possible suffixes.

The last one is easy: a dictionary is the obvious choice for a mapping from keys to corresponding values.

For the prefixes, the most obvious options are string, list of strings, or tuple of strings.

For the suffixes, one option is a list; another is a histogram (dictionary).

How should you choose? The first step is to think about the operations you will need to implement for each data structure. For the prefixes, we need to be able to remove words from the beginning and add to the end. For example, if the current prefix is “Half a”, and the next word is “bee”, you need to be able to form the next prefix, “a bee”.

Your first choice might be a list, since it is easy to add and remove elements, but we also need to be able to use the prefixes as keys in a dictionary, so that rules out lists. With tuples, you can’t append or remove, but you can use the addition operator to form a new tuple:

```
def shift(prefix, word):
    return prefix[1:] + (word,)
```

`shift` takes a tuple of words, `prefix`, and a string, `word`, and forms a new tuple that has all the words in `prefix` except the first, and `word` added to the end.

For the collection of suffixes, the operations we need to perform include adding a new suffix (or increasing the frequency of an existing one), and choosing a random suffix.

Adding a new suffix is equally easy for the list implementation or the histogram. Choosing a random element from a list is easy; choosing from a histogram is harder to do efficiently (see Exercise 13.7).

So far we have been talking mostly about ease of implementation, but there are other factors to consider in choosing data structures. One is run time. Sometimes there is a theoretical reason to expect one data structure to be faster than other; for example, I mentioned that

the `in` operator is faster for dictionaries than for lists, at least when the number of elements is large.

But often you don't know ahead of time which implementation will be faster. One option is to implement both of them and see which is better. This approach is called **benchmarking**. A practical alternative is to choose the data structure that is easiest to implement, and then see if it is fast enough for the intended application. If so, there is no need to go on. If not, there are tools, like the `profile` module, that can identify the places in a program that take the most time.

The other factor to consider is storage space. For example, using a histogram for the collection of suffixes might take less space because you only have to store each word once, no matter how many times it appears in the text. In some cases, saving space can also make your program run faster, and in the extreme, your program might not run at all if you run out of memory. But for many applications, space is a secondary consideration after run time.

One final thought: in this discussion, I have implied that we should use one data structure for both analysis and generation. But since these are separate phases, it would also be possible to use one structure for analysis and then convert to another structure for generation. This would be a net win if the time saved during generation exceeded the time spent in conversion.

13.10 Debugging

When you are debugging a program, and especially if you are working on a hard bug, there are five things to try:

Reading: Examine your code, read it back to yourself, and check that it says what you meant to say.

Running: Experiment by making changes and running different versions. Often if you display the right thing at the right place in the program, the problem becomes obvious, but sometimes you have to build scaffolding.

Ruminating: Take some time to think! What kind of error is it: syntax, runtime, or semantic? What information can you get from the error messages, or from the output of the program? What kind of error could cause the problem you're seeing? What did you change last, before the problem appeared?

Rubberducking: If you explain the problem to someone else, you sometimes find the answer before you finish asking the question. Often you don't need the other person; you could just talk to a rubber duck. And that's the origin of the well-known strategy called **rubber duck debugging**. I am not making this up; see https://en.wikipedia.org/wiki/Rubber_duck_debugging.

Retreating: At some point, the best thing to do is back off, undoing recent changes, until you get back to a program that works and that you understand. Then you can start rebuilding.

Beginning programmers sometimes get stuck on one of these activities and forget the others. Each activity comes with its own failure mode.

For example, reading your code might help if the problem is a typographical error, but not if the problem is a conceptual misunderstanding. If you don't understand what your program does, you can read it 100 times and never see the error, because the error is in your head.

Running experiments can help, especially if you run small, simple tests. But if you run experiments without thinking or reading your code, you might fall into a pattern I call "random walk programming", which is the process of making random changes until the program does the right thing. Needless to say, random walk programming can take a long time.

You have to take time to think. Debugging is like an experimental science. You should have at least one hypothesis about what the problem is. If there are two or more possibilities, try to think of a test that would eliminate one of them.

But even the best debugging techniques will fail if there are too many errors, or if the code you are trying to fix is too big and complicated. Sometimes the best option is to retreat, simplifying the program until you get to something that works and that you understand.

Beginning programmers are often reluctant to retreat because they can't stand to delete a line of code (even if it's wrong). If it makes you feel better, copy your program into another file before you start stripping it down. Then you can copy the pieces back one at a time.

Finding a hard bug requires reading, running, ruminating, and sometimes retreating. If you get stuck on one of these activities, try the others.

13.11 Glossary

deterministic: Pertaining to a program that does the same thing each time it runs, given the same inputs.

pseudorandom: Pertaining to a sequence of numbers that appears to be random, but is generated by a deterministic program.

default value: The value given to an optional parameter if no argument is provided.

override: To replace a default value with an argument.

benchmarking: The process of choosing between data structures by implementing alternatives and testing them on a sample of the possible inputs.

rubber duck debugging: Debugging by explaining your problem to an inanimate object such as a rubber duck. Articulating the problem can help you solve it, even if the rubber duck doesn't know Python.

13.12 Exercises

Exercise 13.9. *The "rank" of a word is its position in a list of words sorted by frequency: the most common word has rank 1, the second most common has rank 2, etc.*

Zipf's law describes a relationship between the ranks and frequencies of words in natural languages (http://en.wikipedia.org/wiki/Zipf's_law). Specifically, it predicts that the frequency, f , of the word with rank r is:

$$f = cr^{-s}$$

where s and c are parameters that depend on the language and the text. If you take the logarithm of both sides of this equation, you get:

$$\log f = \log c - s \log r$$

So if you plot $\log f$ versus $\log r$, you should get a straight line with slope $-s$ and intercept $\log c$.

Write a program that reads a text from a file, counts word frequencies, and prints one line for each word, in descending order of frequency, with $\log f$ and $\log r$. Use the graphing program of your choice to plot the results and check whether they form a straight line. Can you estimate the value of s ?

Solution: <http://thinkpython2.com/code/zipf.py>. To run my solution, you need the plotting module `matplotlib`. If you installed Anaconda, you already have `matplotlib`; otherwise you might have to install it.

Chapter 14

Files

This chapter introduces the idea of “persistent” programs that keep data in permanent storage, and shows how to use different kinds of permanent storage, like files and databases.

14.1 Persistence

Most of the programs we have seen so far are transient in the sense that they run for a short time and produce some output, but when they end, their data disappears. If you run the program again, it starts with a clean slate.

Other programs are **persistent**: they run for a long time (or all the time); they keep at least some of their data in permanent storage (a hard drive, for example); and if they shut down and restart, they pick up where they left off.

Examples of persistent programs are operating systems, which run pretty much whenever a computer is on, and web servers, which run all the time, waiting for requests to come in on the network.

One of the simplest ways for programs to maintain their data is by reading and writing text files. We have already seen programs that read text files; in this chapter we will see programs that write them.

An alternative is to store the state of the program in a database. In this chapter I will present a simple database and a module, `pickle`, that makes it easy to store program data.

14.2 Reading and writing

A text file is a sequence of characters stored on a permanent medium like a hard drive, flash memory, or CD-ROM. We saw how to open and read a file in Section 9.1.

To write a file, you have to open it with mode '`w`' as a second parameter:

```
>>> fout = open('output.txt', 'w')
```

If the file already exists, opening it in write mode clears out the old data and starts fresh, so be careful! If the file doesn't exist, a new one is created.

`open` returns a file object that provides methods for working with the file. The `write` method puts data into the file.

```
>>> line1 = "This here's the wattle,\n"
>>> fout.write(line1)
24
```

The return value is the number of characters that were written. The file object keeps track of where it is, so if you call `write` again, it adds the new data to the end of the file.

```
>>> line2 = "the emblem of our land.\n"
>>> fout.write(line2)
24
```

When you are done writing, you should close the file.

```
>>> fout.close()
```

If you don't close the file, it gets closed for you when the program ends.

14.3 Format operator

The argument of `write` has to be a string, so if we want to put other values in a file, we have to convert them to strings. The easiest way to do that is with `str`:

```
>>> x = 52
>>> fout.write(str(x))
```

An alternative is to use the **format operator**, `%`. When applied to integers, `%` is the modulus operator. But when the first operand is a string, `%` is the format operator.

The first operand is the **format string**, which contains one or more **format sequences**, which specify how the second operand is formatted. The result is a string.

For example, the format sequence '`%d`' means that the second operand should be formatted as a decimal integer:

```
>>> camels = 42
>>> '%d' % camels
'42'
```

The result is the string '`42`', which is not to be confused with the integer value 42.

A format sequence can appear anywhere in the string, so you can embed a value in a sentence:

```
>>> 'I have spotted %d camels.' % camels
'I have spotted 42 camels.'
```

If there is more than one format sequence in the string, the second argument has to be a tuple. Each format sequence is matched with an element of the tuple, in order.

The following example uses '`%d`' to format an integer, '`%g`' to format a floating-point number, and '`%s`' to format a string:

```
>>> 'In %d years I have spotted %g %s.' % (3, 0.1, 'camels')
'In 3 years I have spotted 0.1 camels.'
```

The number of elements in the tuple has to match the number of format sequences in the string. Also, the types of the elements have to match the format sequences:

```
>>> '%d %d %d' % (1, 2)
TypeError: not enough arguments for format string
>>> '%d' % 'dollars'
TypeError: %d format: a number is required, not str
```

In the first example, there aren't enough elements; in the second, the element is the wrong type.

For more information on the format operator, see <https://docs.python.org/3/library/stdtypes.html#printf-style-string-formatting>. A more powerful alternative is the string format method, which you can read about at <https://docs.python.org/3/library/stdtypes.html#str.format>.

14.4 Filenames and paths

Files are organized into **directories** (also called “folders”). Every running program has a “current directory”, which is the default directory for most operations. For example, when you open a file for reading, Python looks for it in the current directory.

The `os` module provides functions for working with files and directories (“`os`” stands for “operating system”). `os.getcwd` returns the name of the current directory:

```
>>> import os
>>> cwd = os.getcwd()
>>> cwd
'/home/dinsdale'
```

`cwd` stands for “current working directory”. The result in this example is `/home/dinsdale`, which is the home directory of a user named `dinsdale`.

A string like `'/home/dinsdale'` that identifies a file or directory is called a **path**.

A simple filename, like `memo.txt` is also considered a path, but it is a **relative path** because it relates to the current directory. If the current directory is `/home/dinsdale`, the filename `memo.txt` would refer to `/home/dinsdale/memo.txt`.

A path that begins with `/` does not depend on the current directory; it is called an **absolute path**. To find the absolute path to a file, you can use `os.path.abspath`:

```
>>> os.path.abspath('memo.txt')
'/home/dinsdale/memo.txt'
```

`os.path` provides other functions for working with filenames and paths. For example, `os.path.exists` checks whether a file or directory exists:

```
>>> os.path.exists('memo.txt')
True
```

If it exists, `os.path.isdir` checks whether it's a directory:

```
>>> os.path.isdir('memo.txt')
False
>>> os.path.isdir('/home/dinsdale')
True
```

Similarly, `os.path.isfile` checks whether it's a file.

`os.listdir` returns a list of the files (and other directories) in the given directory:

```
>>> os.listdir(cwd)
['music', 'photos', 'memo.txt']
```

To demonstrate these functions, the following example “walks” through a directory, prints the names of all the files, and calls itself recursively on all the directories.

```
def walk(dirname):
    for name in os.listdir(dirname):
        path = os.path.join(dirname, name)

        if os.path.isfile(path):
            print(path)
        else:
            walk(path)
```

`os.path.join` takes a directory and a file name and joins them into a complete path.

The `os` module provides a function called `walk` that is similar to this one but more versatile. As an exercise, read the documentation and use it to print the names of the files in a given directory and its subdirectories. You can download my solution from <http://thinkpython2.com/code/walk.py>.

14.5 Catching exceptions

A lot of things can go wrong when you try to read and write files. If you try to open a file that doesn't exist, you get an `FileNotFoundException`:

```
>>> fin = open('bad_file')
FileNotFoundException: [Errno 2] No such file or directory: 'bad_file'
```

If you don't have permission to access a file:

```
>>> fout = open('/etc/passwd', 'w')
PermissionError: [Errno 13] Permission denied: '/etc/passwd'
```

And if you try to open a directory for reading, you get

```
>>> fin = open('/home')
IsADirectoryError: [Errno 21] Is a directory: '/home'
```

To avoid these errors, you could use functions like `os.path.exists` and `os.path.isfile`, but it would take a lot of time and code to check all the possibilities (if “Errno 21” is any indication, there are at least 21 things that can go wrong).

It is better to go ahead and try—and deal with problems if they happen—which is exactly what the `try` statement does. The syntax is similar to an `if...else` statement:

```
try:
    fin = open('bad_file')
except:
    print('Something went wrong.')
```

Python starts by executing the `try` clause. If all goes well, it skips the `except` clause and proceeds. If an exception occurs, it jumps out of the `try` clause and runs the `except` clause.

Handling an exception with a `try` statement is called **catching** an exception. In this example, the `except` clause prints an error message that is not very helpful. In general, catching an exception gives you a chance to fix the problem, or try again, or at least end the program gracefully.

14.6 Databases

A **database** is a file that is organized for storing data. Many databases are organized like a dictionary in the sense that they map from keys to values. The biggest difference between a database and a dictionary is that the database is on disk (or other permanent storage), so it persists after the program ends.

The module `dbm` provides an interface for creating and updating database files. As an example, I'll create a database that contains captions for image files.

Opening a database is similar to opening other files:

```
>>> import dbm  
>>> db = dbm.open('captions', 'c')
```

The mode '`c`' means that the database should be created if it doesn't already exist. The result is a database object that can be used (for most operations) like a dictionary.

When you create a new item, `dbm` updates the database file.

```
>>> db['cleese.png'] = 'Photo of John Cleese.'
```

When you access one of the items, `dbm` reads the file:

```
>>> db['cleese.png']  
b'Photo of John Cleese.'
```

The result is a **bytes object**, which is why it begins with `b`. A bytes object is similar to a string in many ways. When you get farther into Python, the difference becomes important, but for now we can ignore it.

If you make another assignment to an existing key, `dbm` replaces the old value:

```
>>> db['cleese.png'] = 'Photo of John Cleese doing a silly walk.'  
>>> db['cleese.png']  
b'Photo of John Cleese doing a silly walk.'
```

Some dictionary methods, like `keys` and `items`, don't work with database objects. But iteration with a `for` loop works:

```
for key in db.keys():  
    print(key, db[key])
```

As with other files, you should close the database when you are done:

```
>>> db.close()
```

14.7 Pickling

A limitation of `dbm` is that the keys and values have to be strings or bytes. If you try to use any other type, you get an error.

The `pickle` module can help. It translates almost any type of object into a string suitable for storage in a database, and then translates strings back into objects.

`pickle.dumps` takes an object as a parameter and returns a string representation (`dumps` is short for “dump string”):

```
>>> import pickle
>>> t = [1, 2, 3]
>>> pickle.dumps(t)
b'\x80\x03]q\x00(K\x01K\x02K\x03e.'
```

The format isn’t obvious to human readers; it is meant to be easy for `pickle` to interpret. `pickle.loads` (“load string”) reconstitutes the object:

```
>>> t1 = [1, 2, 3]
>>> s = pickle.dumps(t1)
>>> t2 = pickle.loads(s)
>>> t2
[1, 2, 3]
```

Although the new object has the same value as the old, it is not (in general) the same object:

```
>>> t1 == t2
True
>>> t1 is t2
False
```

In other words, pickling and then unpickling has the same effect as copying the object.

You can use `pickle` to store non-strings in a database. In fact, this combination is so common that it has been encapsulated in a module called `shelve`.

14.8 Pipes

Most operating systems provide a command-line interface, also known as a `shell`. Shells usually provide commands to navigate the file system and launch applications. For example, in Unix you can change directories with `cd`, display the contents of a directory with `ls`, and launch a web browser by typing (for example) `firefox`.

Any program that you can launch from the shell can also be launched from Python using a `pipe object`, which represents a running program.

For example, the Unix command `ls -l` normally displays the contents of the current directory in long format. You can launch `ls` with `os.popen`¹:

```
>>> cmd = 'ls -l'
>>> fp = os.popen(cmd)
```

¹`popen` is deprecated now, which means we are supposed to stop using it and start using the `subprocess` module. But for simple cases, I find `subprocess` more complicated than necessary. So I am going to keep using `popen` until they take it away.

The argument is a string that contains a shell command. The return value is an object that behaves like an open file. You can read the output from the `ls` process one line at a time with `readline` or get the whole thing at once with `read`:

```
>>> res = fp.read()
```

When you are done, you close the pipe like a file:

```
>>> stat = fp.close()  
>>> print(stat)  
None
```

The return value is the final status of the `ls` process; `None` means that it ended normally (with no errors).

For example, most Unix systems provide a command called `md5sum` that reads the contents of a file and computes a “checksum”. You can read about MD5 at <http://en.wikipedia.org/wiki/Md5>. This command provides an efficient way to check whether two files have the same contents. The probability that different contents yield the same checksum is very small (that is, unlikely to happen before the universe collapses).

You can use a pipe to run `md5sum` from Python and get the result:

```
>>> filename = 'book.tex'  
>>> cmd = 'md5sum ' + filename  
>>> fp = os.popen(cmd)  
>>> res = fp.read()  
>>> stat = fp.close()  
>>> print(res)  
1e0033f0ed0656636de0d75144ba32e0  book.tex  
>>> print(stat)  
None
```

14.9 Writing modules

Any file that contains Python code can be imported as a module. For example, suppose you have a file named `wc.py` with the following code:

```
def linecount(filename):  
    count = 0  
    for line in open(filename):  
        count += 1  
    return count  
  
print(linecount('wc.py'))
```

If you run this program, it reads itself and prints the number of lines in the file, which is 7. You can also import it like this:

```
>>> import wc  
7
```

Now you have a module object `wc`:

```
>>> wc  
<module 'wc' from 'wc.py'>
```

The module object provides `linecount`:

```
>>> wc.linecount('wc.py')
7
```

So that's how you write modules in Python.

The only problem with this example is that when you import the module it runs the test code at the bottom. Normally when you import a module, it defines new functions but it doesn't run them.

Programs that will be imported as modules often use the following idiom:

```
if __name__ == '__main__':
    print(linecount('wc.py'))
```

`__name__` is a built-in variable that is set when the program starts. If the program is running as a script, `__name__` has the value '`__main__`'; in that case, the test code runs. Otherwise, if the module is being imported, the test code is skipped.

As an exercise, type this example into a file named `wc.py` and run it as a script. Then run the Python interpreter and `import wc`. What is the value of `__name__` when the module is being imported?

Warning: If you import a module that has already been imported, Python does nothing. It does not re-read the file, even if it has changed.

If you want to reload a module, you can use the built-in function `reload`, but it can be tricky, so the safest thing to do is restart the interpreter and then import the module again.

14.10 Debugging

When you are reading and writing files, you might run into problems with whitespace. These errors can be hard to debug because spaces, tabs and newlines are normally invisible:

```
>>> s = '1 2\t 3\n 4'
>>> print(s)
1 2 3
4
```

The built-in function `repr` can help. It takes any object as an argument and returns a string representation of the object. For strings, it represents whitespace characters with backslash sequences:

```
>>> print(repr(s))
'1 2\t 3\n 4'
```

This can be helpful for debugging.

One other problem you might run into is that different systems use different characters to indicate the end of a line. Some systems use a newline, represented `\n`. Others use a return character, represented `\r`. Some use both. If you move files between different systems, these inconsistencies can cause problems.

For most systems, there are applications to convert from one format to another. You can find them (and read more about this issue) at <http://en.wikipedia.org/wiki/Newline>. Or, of course, you could write one yourself.

14.11 Glossary

persistent: Pertaining to a program that runs indefinitely and keeps at least some of its data in permanent storage.

format operator: An operator, %, that takes a format string and a tuple and generates a string that includes the elements of the tuple formatted as specified by the format string.

format string: A string, used with the format operator, that contains format sequences.

format sequence: A sequence of characters in a format string, like %d, that specifies how a value should be formatted.

text file: A sequence of characters stored in permanent storage like a hard drive.

directory: A named collection of files, also called a folder.

path: A string that identifies a file.

relative path: A path that starts from the current directory.

absolute path: A path that starts from the topmost directory in the file system.

catch: To prevent an exception from terminating a program using the try and except statements.

database: A file whose contents are organized like a dictionary with keys that correspond to values.

bytes object: An object similar to a string.

shell: A program that allows users to type commands and then executes them by starting other programs.

pipe object: An object that represents a running program, allowing a Python program to run commands and read the results.

14.12 Exercises

Exercise 14.1. Write a function called sed that takes as arguments a pattern string, a replacement string, and two filenames; it should read the first file and write the contents into the second file (creating it if necessary). If the pattern string appears anywhere in the file, it should be replaced with the replacement string.

If an error occurs while opening, reading, writing or closing files, your program should catch the exception, print an error message, and exit. Solution: <http://thinkpython2.com/code/sed.py>.

Exercise 14.2. If you download my solution to Exercise 12.2 from http://thinkpython2.com/code/anagram_sets.py, you'll see that it creates a dictionary that maps from a sorted string of letters to the list of words that can be spelled with those letters. For example, 'opst' maps to the list ['opts', 'post', 'pots', 'spot', 'stop', 'tops'].

Write a module that imports anagram_sets and provides two new functions: store_anagrams should store the anagram dictionary in a "shelf"; read_anagrams should look up a word and return a list of its anagrams. Solution: http://thinkpython2.com/code/anagram_db.py.

Exercise 14.3. In a large collection of MP3 files, there may be more than one copy of the same song, stored in different directories or with different file names. The goal of this exercise is to search for duplicates.

1. Write a program that searches a directory and all of its subdirectories, recursively, and returns a list of complete paths for all files with a given suffix (like .mp3). Hint: `os.path` provides several useful functions for manipulating file and path names.
2. To recognize duplicates, you can use `md5sum` to compute a “checksum” for each files. If two files have the same checksum, they probably have the same contents.
3. To double-check, you can use the Unix command `diff`.

Solution: http://thinkpython2.com/code/find_duplicates.py.

Chapter 15

Classes and objects

At this point you know how to use functions to organize code and built-in types to organize data. The next step is to learn “object-oriented programming”, which uses programmer-defined types to organize both code and data. Object-oriented programming is a big topic; it will take a few chapters to get there.

Code examples from this chapter are available from <http://thinkpython2.com/code/Point1.py>; solutions to the exercises are available from http://thinkpython2.com/code/Point1_soln.py.

15.1 Programmer-defined types

We have used many of Python’s built-in types; now we are going to define a new type. As an example, we will create a type called `Point` that represents a point in two-dimensional space.

In mathematical notation, points are often written in parentheses with a comma separating the coordinates. For example, $(0, 0)$ represents the origin, and (x, y) represents the point x units to the right and y units up from the origin.

There are several ways we might represent points in Python:

- We could store the coordinates separately in two variables, `x` and `y`.
- We could store the coordinates as elements in a list or tuple.
- We could create a new type to represent points as objects.

Creating a new type is more complicated than the other options, but it has advantages that will be apparent soon.

A programmer-defined type is also called a `class`. A class definition looks like this:

```
class Point:  
    """Represents a point in 2-D space."""
```

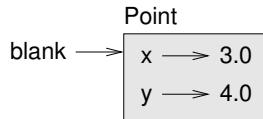


Figure 15.1: Object diagram.

The header indicates that the new class is called `Point`. The body is a docstring that explains what the class is for. You can define variables and methods inside a class definition, but we will get back to that later.

Defining a class named `Point` creates a **class object**.

```
>>> Point
```

```
<class '__main__.Point'>
```

Because `Point` is defined at the top level, its “full name” is `__main__.Point`.

The class object is like a factory for creating objects. To create a `Point`, you call `Point` as if it were a function.

```
>>> blank = Point()
```

```
>>> blank
```

```
<__main__.Point object at 0xb7e9d3ac>
```

The return value is a reference to a `Point` object, which we assign to `blank`.

Creating a new object is called **instantiation**, and the object is an **instance** of the class.

When you print an instance, Python tells you what class it belongs to and where it is stored in memory (the prefix `0x` means that the following number is in hexadecimal).

Every object is an instance of some class, so “object” and “instance” are interchangeable. But in this chapter I use “instance” to indicate that I am talking about a programmer-defined type.

15.2 Attributes

You can assign values to an instance using dot notation:

```
>>> blank.x = 3.0
```

```
>>> blank.y = 4.0
```

This syntax is similar to the syntax for selecting a variable from a module, such as `math.pi` or `string.whitespace`. In this case, though, we are assigning values to named elements of an object. These elements are called **attributes**.

As a noun, “AT-trib-ute” is pronounced with emphasis on the first syllable, as opposed to “a-TRIB-ute”, which is a verb.

Figure 15.1 is a state diagram that shows the result of these assignments. A state diagram that shows an object and its attributes is called an **object diagram**.

The variable `blank` refers to a `Point` object, which contains two attributes. Each attribute refers to a floating-point number.

You can read the value of an attribute using the same syntax:

```
>>> blank.y  
4.0  
>>> x = blank.x  
>>> x  
3.0
```

The expression `blank.x` means, “Go to the object `blank` refers to and get the value of `x`.” In the example, we assign that value to a variable named `x`. There is no conflict between the variable `x` and the attribute `x`.

You can use dot notation as part of any expression. For example:

```
>>> '(%g, %g)' % (blank.x, blank.y)  
'(3.0, 4.0)'  
>>> distance = math.sqrt(blank.x**2 + blank.y**2)  
>>> distance  
5.0
```

You can pass an instance as an argument in the usual way. For example:

```
def print_point(p):  
    print('(%g, %g)' % (p.x, p.y))
```

`print_point` takes a point as an argument and displays it in mathematical notation. To invoke it, you can pass `blank` as an argument:

```
>>> print_point(blank)  
(3.0, 4.0)
```

Inside the function, `p` is an alias for `blank`, so if the function modifies `p`, `blank` changes.

As an exercise, write a function called `distance_between_points` that takes two Points as arguments and returns the distance between them.

15.3 Rectangles

Sometimes it is obvious what the attributes of an object should be, but other times you have to make decisions. For example, imagine you are designing a class to represent rectangles. What attributes would you use to specify the location and size of a rectangle? You can ignore angle; to keep things simple, assume that the rectangle is either vertical or horizontal.

There are at least two possibilities:

- You could specify one corner of the rectangle (or the center), the width, and the height.
- You could specify two opposing corners.

At this point it is hard to say whether either is better than the other, so we’ll implement the first one, just as an example.

Here is the class definition:

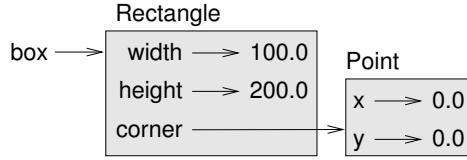


Figure 15.2: Object diagram.

```

class Rectangle:
    """Represents a rectangle.

    attributes: width, height, corner.
    """
  
```

The docstring lists the attributes: `width` and `height` are numbers; `corner` is a `Point` object that specifies the lower-left corner.

To represent a rectangle, you have to instantiate a `Rectangle` object and assign values to the attributes:

```

box = Rectangle()
box.width = 100.0
box.height = 200.0
box.corner = Point()
box.corner.x = 0.0
box.corner.y = 0.0
  
```

The expression `box.corner.x` means, “Go to the object `box` refers to and select the attribute named `corner`; then go to that object and select the attribute named `x`.”

Figure 15.2 shows the state of this object. An object that is an attribute of another object is **embedded**.

15.4 Instances as return values

Functions can return instances. For example, `find_center` takes a `Rectangle` as an argument and returns a `Point` that contains the coordinates of the center of the `Rectangle`:

```

def find_center(rect):
    p = Point()
    p.x = rect.corner.x + rect.width/2
    p.y = rect.corner.y + rect.height/2
    return p
  
```

Here is an example that passes `box` as an argument and assigns the resulting `Point` to `center`:

```

>>> center = find_center(box)
>>> print_point(center)
(50, 100)
  
```

15.5 Objects are mutable

You can change the state of an object by making an assignment to one of its attributes. For example, to change the size of a rectangle without changing its position, you can modify the values of `width` and `height`:

```
box.width = box.width + 50  
box.height = box.height + 100
```

You can also write functions that modify objects. For example, `grow_rectangle` takes a `Rectangle` object and two numbers, `dwidth` and `dheight`, and adds the numbers to the width and height of the rectangle:

```
def grow_rectangle(rect, dwidth, dheight):  
    rect.width += dwidth  
    rect.height += dheight
```

Here is an example that demonstrates the effect:

```
>>> box.width, box.height  
(150.0, 300.0)  
>>> grow_rectangle(box, 50, 100)  
>>> box.width, box.height  
(200.0, 400.0)
```

Inside the function, `rect` is an alias for `box`, so when the function modifies `rect`, `box` changes.

As an exercise, write a function named `move_rectangle` that takes a `Rectangle` and two numbers named `dx` and `dy`. It should change the location of the rectangle by adding `dx` to the `x` coordinate of `corner` and adding `dy` to the `y` coordinate of `corner`.

15.6 Copying

Aliasing can make a program difficult to read because changes in one place might have unexpected effects in another place. It is hard to keep track of all the variables that might refer to a given object.

Copying an object is often an alternative to aliasing. The `copy` module contains a function called `copy` that can duplicate any object:

```
>>> p1 = Point()  
>>> p1.x = 3.0  
>>> p1.y = 4.0  
  
>>> import copy  
>>> p2 = copy.copy(p1)
```

`p1` and `p2` contain the same data, but they are not the same `Point`.

```
>>> print_point(p1)  
(3, 4)  
>>> print_point(p2)  
(3, 4)  
>>> p1 is p2  
False
```

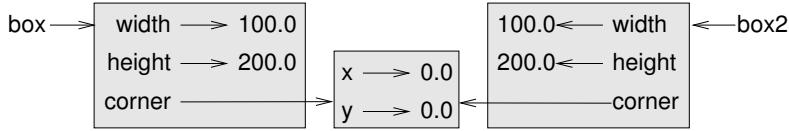


Figure 15.3: Object diagram.

```
>>> p1 == p2
False
```

The `is` operator indicates that `p1` and `p2` are not the same object, which is what we expected. But you might have expected `==` to yield `True` because these points contain the same data. In that case, you will be disappointed to learn that for instances, the default behavior of the `==` operator is the same as the `is` operator; it checks object identity, not object equivalence. That's because for programmer-defined types, Python doesn't know what should be considered equivalent. At least, not yet.

If you use `copy.copy` to duplicate a `Rectangle`, you will find that it copies the `Rectangle` object but not the embedded `Point`.

```
>>> box2 = copy.copy(box)
>>> box2 is box
False
>>> box2.corner is box.corner
True
```

Figure 15.3 shows what the object diagram looks like. This operation is called a **shallow copy** because it copies the object and any references it contains, but not the embedded objects.

For most applications, this is not what you want. In this example, invoking `grow_rectangle` on one of the `Rectangles` would not affect the other, but invoking `move_rectangle` on either would affect both! This behavior is confusing and error-prone.

Fortunately, the `copy` module provides a method named `deepcopy` that copies not only the object but also the objects it refers to, and the objects *they* refer to, and so on. You will not be surprised to learn that this operation is called a **deep copy**.

```
>>> box3 = copy.deepcopy(box)
>>> box3 is box
False
>>> box3.corner is box.corner
False
box3 and box are completely separate objects.
```

As an exercise, write a version of `move_rectangle` that creates and returns a new `Rectangle` instead of modifying the old one.

15.7 Debugging

When you start working with objects, you are likely to encounter some new exceptions. If you try to access an attribute that doesn't exist, you get an `AttributeError`:

```
>>> p = Point()
>>> p.x = 3
>>> p.y = 4
>>> p.z
AttributeError: Point instance has no attribute 'z'
```

If you are not sure what type an object is, you can ask:

```
>>> type(p)
<class '__main__.Point'>
```

You can also use `isinstance` to check whether an object is an instance of a class:

```
>>> isinstance(p, Point)
True
```

If you are not sure whether an object has a particular attribute, you can use the built-in function `hasattr`:

```
>>> hasattr(p, 'x')
True
>>> hasattr(p, 'z')
False
```

The first argument can be any object; the second argument is a *string* that contains the name of the attribute.

You can also use a `try` statement to see if the object has the attributes you need:

```
try:
    x = p.x
except AttributeError:
    x = 0
```

This approach can make it easier to write functions that work with different types; more on that topic is coming up in Section 17.9.

15.8 Glossary

class: A programmer-defined type. A class definition creates a new class object.

class object: An object that contains information about a programmer-defined type. The class object can be used to create instances of the type.

instance: An object that belongs to a class.

instantiate: To create a new object.

attribute: One of the named values associated with an object.

embedded object: An object that is stored as an attribute of another object.

shallow copy: To copy the contents of an object, including any references to embedded objects; implemented by the `copy` function in the `copy` module.

deep copy: To copy the contents of an object as well as any embedded objects, and any objects embedded in them, and so on; implemented by the `deepcopy` function in the `copy` module.

object diagram: A diagram that shows objects, their attributes, and the values of the attributes.

15.9 Exercises

Exercise 15.1. Write a definition for a class named `Circle` with attributes `center` and `radius`, where `center` is a `Point` object and `radius` is a number.

Instantiate a `Circle` object that represents a circle with its center at (150, 100) and radius 75.

Write a function named `point_in_circle` that takes a `Circle` and a `Point` and returns True if the `Point` lies in or on the boundary of the circle.

Write a function named `rect_in_circle` that takes a `Circle` and a `Rectangle` and returns True if the `Rectangle` lies entirely in or on the boundary of the circle.

Write a function named `rect_circle_overlap` that takes a `Circle` and a `Rectangle` and returns True if any of the corners of the `Rectangle` fall inside the `Circle`. Or as a more challenging version, return True if any part of the `Rectangle` falls inside the `Circle`.

Solution: <http://thinkpython2.com/code/Circle.py>.

Exercise 15.2. Write a function called `draw_rect` that takes a `Turtle` object and a `Rectangle` and uses the `Turtle` to draw the `Rectangle`. See Chapter 4 for examples using `Turtle` objects.

Write a function called `draw_circle` that takes a `Turtle` and a `Circle` and draws the `Circle`.

Solution: <http://thinkpython2.com/code/draw.py>.

Chapter 16

Classes and functions

Now that we know how to create new types, the next step is to write functions that take programmer-defined objects as parameters and return them as results. In this chapter I also present “functional programming style” and two new program development plans.

Code examples from this chapter are available from <http://thinkpython2.com/code/Time1.py>. Solutions to the exercises are at http://thinkpython2.com/code/Time1_soln.py.

16.1 Time

As another example of a programmer-defined type, we’ll define a class called `Time` that records the time of day. The class definition looks like this:

```
class Time:  
    """Represents the time of day.  
  
    attributes: hour, minute, second  
    """
```

We can create a new `Time` object and assign attributes for hours, minutes, and seconds:

```
time = Time()  
time.hour = 11  
time.minute = 59  
time.second = 30
```

The state diagram for the `Time` object looks like Figure 16.1.

As an exercise, write a function called `print_time` that takes a `Time` object and prints it in the form `hour:minute:second`. Hint: the format sequence `'%.2d'` prints an integer using at least two digits, including a leading zero if necessary.

Write a boolean function called `is_after` that takes two `Time` objects, `t1` and `t2`, and returns `True` if `t1` follows `t2` chronologically and `False` otherwise. Challenge: don’t use an `if` statement.

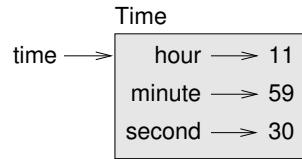


Figure 16.1: Object diagram.

16.2 Pure functions

In the next few sections, we'll write two functions that add time values. They demonstrate two kinds of functions: pure functions and modifiers. They also demonstrate a development plan I'll call **prototype and patch**, which is a way of tackling a complex problem by starting with a simple prototype and incrementally dealing with the complications.

Here is a simple prototype of `add_time`:

```
def add_time(t1, t2):
    sum = Time()
    sum.hour = t1.hour + t2.hour
    sum.minute = t1.minute + t2.minute
    sum.second = t1.second + t2.second
    return sum
```

The function creates a new `Time` object, initializes its attributes, and returns a reference to the new object. This is called a **pure function** because it does not modify any of the objects passed to it as arguments and it has no effect, like displaying a value or getting user input, other than returning a value.

To test this function, I'll create two `Time` objects: `start` contains the start time of a movie, like *Monty Python and the Holy Grail*, and `duration` contains the run time of the movie, which is one hour 35 minutes.

`add_time` figures out when the movie will be done.

```
>>> start = Time()
>>> start.hour = 9
>>> start.minute = 45
>>> start.second = 0

>>> duration = Time()
>>> duration.hour = 1
>>> duration.minute = 35
>>> duration.second = 0

>>> done = add_time(start, duration)
>>> print_time(done)
10:80:00
```

The result, `10:80:00` might not be what you were hoping for. The problem is that this function does not deal with cases where the number of seconds or minutes adds up to more than sixty. When that happens, we have to "carry" the extra seconds into the minute column or the extra minutes into the hour column.

Here's an improved version:

```
def add_time(t1, t2):
    sum = Time()
    sum.hour = t1.hour + t2.hour
    sum.minute = t1.minute + t2.minute
    sum.second = t1.second + t2.second

    if sum.second >= 60:
        sum.second -= 60
        sum.minute += 1

    if sum.minute >= 60:
        sum.minute -= 60
        sum.hour += 1

    return sum
```

Although this function is correct, it is starting to get big. We will see a shorter alternative later.

16.3 Modifiers

Sometimes it is useful for a function to modify the objects it gets as parameters. In that case, the changes are visible to the caller. Functions that work this way are called **modifiers**.

`increment`, which adds a given number of seconds to a `Time` object, can be written naturally as a modifier. Here is a rough draft:

```
def increment(time, seconds):
    time.second += seconds

    if time.second >= 60:
        time.second -= 60
        time.minute += 1

    if time.minute >= 60:
        time.minute -= 60
        time.hour += 1
```

The first line performs the basic operation; the remainder deals with the special cases we saw before.

Is this function correct? What happens if `seconds` is much greater than sixty?

In that case, it is not enough to carry once; we have to keep doing it until `time.second` is less than sixty. One solution is to replace the `if` statements with `while` statements. That would make the function correct, but not very efficient. As an exercise, write a correct version of `increment` that doesn't contain any loops.

Anything that can be done with modifiers can also be done with pure functions. In fact, some programming languages only allow pure functions. There is some evidence that programs that use pure functions are faster to develop and less error-prone than programs that use modifiers. But modifiers are convenient at times, and functional programs tend to be less efficient.

In general, I recommend that you write pure functions whenever it is reasonable and resort to modifiers only if there is a compelling advantage. This approach might be called a **functional programming style**.

As an exercise, write a “pure” version of `increment` that creates and returns a new `Time` object rather than modifying the parameter.

16.4 Prototyping versus planning

The development plan I am demonstrating is called “prototype and patch”. For each function, I wrote a prototype that performed the basic calculation and then tested it, patching errors along the way.

This approach can be effective, especially if you don’t yet have a deep understanding of the problem. But incremental corrections can generate code that is unnecessarily complicated—since it deals with many special cases—and unreliable—since it is hard to know if you have found all the errors.

An alternative is **designed development**, in which high-level insight into the problem can make the programming much easier. In this case, the insight is that a `Time` object is really a three-digit number in base 60 (see <http://en.wikipedia.org/wiki/Sexagesimal>). The `second` attribute is the “ones column”, the `minute` attribute is the “sixties column”, and the `hour` attribute is the “thirty-six hundreds column”.

When we wrote `add_time` and `increment`, we were effectively doing addition in base 60, which is why we had to carry from one column to the next.

This observation suggests another approach to the whole problem—we can convert `Time` objects to integers and take advantage of the fact that the computer knows how to do integer arithmetic.

Here is a function that converts `Times` to integers:

```
def time_to_int(time):
    minutes = time.hour * 60 + time.minute
    seconds = minutes * 60 + time.second
    return seconds
```

And here is a function that converts an integer to a `Time` (recall that `divmod` divides the first argument by the second and returns the quotient and remainder as a tuple).

```
def int_to_time(seconds):
    time = Time()
    minutes, time.second = divmod(seconds, 60)
    time.hour, time.minute = divmod(minutes, 60)
    return time
```

You might have to think a bit, and run some tests, to convince yourself that these functions are correct. One way to test them is to check that `time_to_int(int_to_time(x)) == x` for many values of `x`. This is an example of a consistency check.

Once you are convinced they are correct, you can use them to rewrite `add_time`:

```
def add_time(t1, t2):
    seconds = time_to_int(t1) + time_to_int(t2)
    return int_to_time(seconds)
```

This version is shorter than the original, and easier to verify. As an exercise, rewrite increment using `time_to_int` and `int_to_time`.

In some ways, converting from base 60 to base 10 and back is harder than just dealing with times. Base conversion is more abstract; our intuition for dealing with time values is better.

But if we have the insight to treat times as base 60 numbers and make the investment of writing the conversion functions (`time_to_int` and `int_to_time`), we get a program that is shorter, easier to read and debug, and more reliable.

It is also easier to add features later. For example, imagine subtracting two Times to find the duration between them. The naive approach would be to implement subtraction with borrowing. Using the conversion functions would be easier and more likely to be correct.

Ironically, sometimes making a problem harder (or more general) makes it easier (because there are fewer special cases and fewer opportunities for error).

16.5 Debugging

A Time object is well-formed if the values of `minute` and `second` are between 0 and 60 (including 0 but not 60) and if `hour` is positive. `hour` and `minute` should be integer values, but we might allow `second` to have a fraction part.

Requirements like these are called **invariants** because they should always be true. To put it a different way, if they are not true, something has gone wrong.

Writing code to check invariants can help detect errors and find their causes. For example, you might have a function like `valid_time` that takes a Time object and returns `False` if it violates an invariant:

```
def valid_time(time):
    if time.hour < 0 or time.minute < 0 or time.second < 0:
        return False
    if time.minute >= 60 or time.second >= 60:
        return False
    return True
```

At the beginning of each function you could check the arguments to make sure they are valid:

```
def add_time(t1, t2):
    if not valid_time(t1) or not valid_time(t2):
        raise ValueError('invalid Time object in add_time')
    seconds = time_to_int(t1) + time_to_int(t2)
    return int_to_time(seconds)
```

Or you could use an **assert statement**, which checks a given invariant and raises an exception if it fails:

```
def add_time(t1, t2):
    assert valid_time(t1) and valid_time(t2)
    seconds = time_to_int(t1) + time_to_int(t2)
    return int_to_time(seconds)
```

`assert` statements are useful because they distinguish code that deals with normal conditions from code that checks for errors.

16.6 Glossary

prototype and patch: A development plan that involves writing a rough draft of a program, testing, and correcting errors as they are found.

designed development: A development plan that involves high-level insight into the problem and more planning than incremental development or prototype development.

pure function: A function that does not modify any of the objects it receives as arguments.
Most pure functions are fruitful.

modifier: A function that changes one or more of the objects it receives as arguments. Most modifiers are void; that is, they return `None`.

functional programming style: A style of program design in which the majority of functions are pure.

invariant: A condition that should always be true during the execution of a program.

assert statement: A statement that checks a condition and raises an exception if it fails.

16.7 Exercises

Code examples from this chapter are available from <http://thinkpython2.com/code/Time1.py>; solutions to the exercises are available from http://thinkpython2.com/code/Time1_soln.py.

Exercise 16.1. Write a function called `mul_time` that takes a `Time` object and a number and returns a new `Time` object that contains the product of the original `Time` and the number.

Then use `mul_time` to write a function that takes a `Time` object that represents the finishing time in a race, and a number that represents the distance, and returns a `Time` object that represents the average pace (time per mile).

Exercise 16.2. The `datetime` module provides `time` objects that are similar to the `Time` objects in this chapter, but they provide a rich set of methods and operators. Read the documentation at <http://docs.python.org/3/library/datetime.html>.

1. Use the `datetime` module to write a program that gets the current date and prints the day of the week.
2. Write a program that takes a birthday as input and prints the user's age and the number of days, hours, minutes and seconds until their next birthday.
3. For two people born on different days, there is a day when one is twice as old as the other. That's their Double Day. Write a program that takes two birth dates and computes their Double Day.
4. For a little more challenge, write the more general version that computes the day when one person is n times older than the other.

Solution: <http://thinkpython2.com/code/double.py>

Chapter 17

Classes and methods

Although we are using some of Python’s object-oriented features, the programs from the last two chapters are not really object-oriented because they don’t represent the relationships between programmer-defined types and the functions that operate on them. The next step is to transform those functions into methods that make the relationships explicit.

Code examples from this chapter are available from <http://thinkpython2.com/code/Time2.py>, and solutions to the exercises are in http://thinkpython2.com/code/Point2_soln.py.

17.1 Object-oriented features

Python is an **object-oriented programming language**, which means that it provides features that support object-oriented programming, which has these defining characteristics:

- Programs include class and method definitions.
- Most of the computation is expressed in terms of operations on objects.
- Objects often represent things in the real world, and methods often correspond to the ways things in the real world interact.

For example, the `Time` class defined in Chapter 16 corresponds to the way people record the time of day, and the functions we defined correspond to the kinds of things people do with times. Similarly, the `Point` and `Rectangle` classes in Chapter 15 correspond to the mathematical concepts of a point and a rectangle.

So far, we have not taken advantage of the features Python provides to support object-oriented programming. These features are not strictly necessary; most of them provide alternative syntax for things we have already done. But in many cases, the alternative is more concise and more accurately conveys the structure of the program.

For example, in `Time1.py` there is no obvious connection between the class definition and the function definitions that follow. With some examination, it is apparent that every function takes at least one `Time` object as an argument.

This observation is the motivation for **methods**; a method is a function that is associated with a particular class. We have seen methods for strings, lists, dictionaries and tuples. In this chapter, we will define methods for programmer-defined types.

Methods are semantically the same as functions, but there are two syntactic differences:

- Methods are defined inside a class definition in order to make the relationship between the class and the method explicit.
- The syntax for invoking a method is different from the syntax for calling a function.

In the next few sections, we will take the functions from the previous two chapters and transform them into methods. This transformation is purely mechanical; you can do it by following a sequence of steps. If you are comfortable converting from one form to another, you will be able to choose the best form for whatever you are doing.

17.2 Printing objects

In Chapter 16, we defined a class named `Time` and in Section 16.1, you wrote a function named `print_time`:

```
class Time:
    """Represents the time of day."""

def print_time(time):
    print('%.2d:%.2d:%.2d' % (time.hour, time.minute, time.second))
```

To call this function, you have to pass a `Time` object as an argument:

```
>>> start = Time()
>>> start.hour = 9
>>> start.minute = 45
>>> start.second = 00
>>> print_time(start)
09:45:00
```

To make `print_time` a method, all we have to do is move the function definition inside the class definition. Notice the change in indentation.

```
class Time:
    def print_time(time):
        print('%.2d:%.2d:%.2d' % (time.hour, time.minute, time.second))
```

Now there are two ways to call `print_time`. The first (and less common) way is to use function syntax:

```
>>> Time.print_time(start)
09:45:00
```

In this use of dot notation, `Time` is the name of the class, and `print_time` is the name of the method. `start` is passed as a parameter.

The second (and more concise) way is to use method syntax:

```
>>> start.print_time()
09:45:00
```

In this use of dot notation, `print_time` is the name of the method (again), and `start` is the object the method is invoked on, which is called the **subject**. Just as the subject of a sentence is what the sentence is about, the subject of a method invocation is what the method is about.

Inside the method, the subject is assigned to the first parameter, so in this case `start` is assigned to `time`.

By convention, the first parameter of a method is called `self`, so it would be more common to write `print_time` like this:

```
class Time:  
    def print_time(self):  
        print('%.2d:%.2d:%.2d' % (self.hour, self.minute, self.second))
```

The reason for this convention is an implicit metaphor:

- The syntax for a function call, `print_time(start)`, suggests that the function is the active agent. It says something like, “Hey `print_time`! Here’s an object for you to print.”
- In object-oriented programming, the objects are the active agents. A method invocation like `start.print_time()` says “Hey `start`! Please print yourself.”

This change in perspective might be more polite, but it is not obvious that it is useful. In the examples we have seen so far, it may not be. But sometimes shifting responsibility from the functions onto the objects makes it possible to write more versatile functions (or methods), and makes it easier to maintain and reuse code.

As an exercise, rewrite `time_to_int` (from Section 16.4) as a method. You might be tempted to rewrite `int_to_time` as a method, too, but that doesn’t really make sense because there would be no object to invoke it on.

17.3 Another example

Here’s a version of `increment` (from Section 16.3) rewritten as a method:

```
# inside class Time:  
  
    def increment(self, seconds):  
        seconds += self.time_to_int()  
        return int_to_time(seconds)
```

This version assumes that `time_to_int` is written as a method. Also, note that it is a pure function, not a modifier.

Here’s how you would invoke `increment`:

```
>>> start.print_time()  
09:45:00  
>>> end = start.increment(1337)  
>>> end.print_time()  
10:07:17
```

The subject, `start`, gets assigned to the first parameter, `self`. The argument, `1337`, gets assigned to the second parameter, `seconds`.

This mechanism can be confusing, especially if you make an error. For example, if you invoke `increment` with two arguments, you get:

```
>>> end = start.increment(1337, 460)
TypeError: increment() takes 2 positional arguments but 3 were given
```

The error message is initially confusing, because there are only two arguments in parentheses. But the subject is also considered an argument, so all together that's three.

By the way, a **positional argument** is an argument that doesn't have a parameter name; that is, it is not a keyword argument. In this function call:

```
sketch(parrot, cage, dead=True)
```

`parrot` and `cage` are positional, and `dead` is a keyword argument.

17.4 A more complicated example

Rewriting `is_after` (from Section 16.1) is slightly more complicated because it takes two `Time` objects as parameters. In this case it is conventional to name the first parameter `self` and the second parameter `other`:

```
# inside class Time:
```

```
    def is_after(self, other):
        return self.time_to_int() > other.time_to_int()
```

To use this method, you have to invoke it on one object and pass the other as an argument:

```
>>> end.is_after(start)
True
```

One nice thing about this syntax is that it almost reads like English: “end is after start?”

17.5 The `init` method

The `init` method (short for “initialization”) is a special method that gets invoked when an object is instantiated. Its full name is `__init__` (two underscore characters, followed by `init`, and then two more underscores). An `init` method for the `Time` class might look like this:

```
# inside class Time:

    def __init__(self, hour=0, minute=0, second=0):
        self.hour = hour
        self.minute = minute
        self.second = second
```

It is common for the parameters of `__init__` to have the same names as the attributes. The statement

```
    self.hour = hour
```

stores the value of the parameter hour as an attribute of `self`.

The parameters are optional, so if you call `Time` with no arguments, you get the default values.

```
>>> time = Time()  
>>> time.print_time()  
00:00:00
```

If you provide one argument, it overrides hour:

```
>>> time = Time(9)  
>>> time.print_time()  
09:00:00
```

If you provide two arguments, they override hour and minute.

```
>>> time = Time(9, 45)  
>>> time.print_time()  
09:45:00
```

And if you provide three arguments, they override all three default values.

As an exercise, write an `init` method for the `Point` class that takes `x` and `y` as optional parameters and assigns them to the corresponding attributes.

17.6 The `__str__` method

`__str__` is a special method, like `__init__`, that is supposed to return a string representation of an object.

For example, here is a `str` method for `Time` objects:

```
# inside class Time:  
  
    def __str__(self):  
        return '%.2d:%.2d:%.2d' % (self.hour, self.minute, self.second)
```

When you print an object, Python invokes the `str` method:

```
>>> time = Time(9, 45)  
>>> print(time)  
09:45:00
```

When I write a new class, I almost always start by writing `__init__`, which makes it easier to instantiate objects, and `__str__`, which is useful for debugging.

As an exercise, write a `str` method for the `Point` class. Create a `Point` object and print it.

17.7 Operator overloading

By defining other special methods, you can specify the behavior of operators on programmer-defined types. For example, if you define a method named `__add__` for the `Time` class, you can use the `+` operator on `Time` objects.

Here is what the definition might look like:

```
# inside class Time:

    def __add__(self, other):
        seconds = self.time_to_int() + other.time_to_int()
        return int_to_time(seconds)
```

And here is how you could use it:

```
>>> start = Time(9, 45)
>>> duration = Time(1, 35)
>>> print(start + duration)
11:20:00
```

When you apply the `+` operator to `Time` objects, Python invokes `__add__`. When you print the result, Python invokes `__str__`. So there is a lot happening behind the scenes!

Changing the behavior of an operator so that it works with programmer-defined types is called **operator overloading**. For every operator in Python there is a corresponding special method, like `__add__`. For more details, see <http://docs.python.org/3/reference/datamodel.html#specialnames>.

As an exercise, write an `add` method for the `Point` class.

17.8 Type-based dispatch

In the previous section we added two `Time` objects, but you also might want to add an integer to a `Time` object. The following is a version of `__add__` that checks the type of `other` and invokes either `add_time` or `increment`:

```
# inside class Time:

    def __add__(self, other):
        if isinstance(other, Time):
            return self.add_time(other)
        else:
            return self.increment(other)

    def add_time(self, other):
        seconds = self.time_to_int() + other.time_to_int()
        return int_to_time(seconds)

    def increment(self, seconds):
        seconds += self.time_to_int()
        return int_to_time(seconds)
```

The built-in function `isinstance` takes a value and a class object, and returns `True` if the value is an instance of the class.

If `other` is a `Time` object, `__add__` invokes `add_time`. Otherwise it assumes that the parameter is a number and invokes `increment`. This operation is called a **type-based dispatch** because it dispatches the computation to different methods based on the type of the arguments.

Here are examples that use the `+` operator with different types:

```
>>> start = Time(9, 45)
>>> duration = Time(1, 35)
>>> print(start + duration)
11:20:00
>>> print(start + 1337)
10:07:17
```

Unfortunately, this implementation of addition is not commutative. If the integer is the first operand, you get

```
>>> print(1337 + start)
TypeError: unsupported operand type(s) for +: 'int' and 'instance'
```

The problem is, instead of asking the Time object to add an integer, Python is asking an integer to add a Time object, and it doesn't know how. But there is a clever solution for this problem: the special method `__radd__`, which stands for "right-side add". This method is invoked when a Time object appears on the right side of the `+` operator. Here's the definition:

```
# inside class Time:

    def __radd__(self, other):
        return self.__add__(other)
```

And here's how it's used:

```
>>> print(1337 + start)
10:07:17
```

As an exercise, write an `add` method for Points that works with either a Point object or a tuple:

- If the second operand is a Point, the method should return a new Point whose x coordinate is the sum of the x coordinates of the operands, and likewise for the y coordinates.
- If the second operand is a tuple, the method should add the first element of the tuple to the x coordinate and the second element to the y coordinate, and return a new Point with the result.

17.9 Polymorphism

Type-based dispatch is useful when it is necessary, but (fortunately) it is not always necessary. Often you can avoid it by writing functions that work correctly for arguments with different types.

Many of the functions we wrote for strings also work for other sequence types. For example, in Section 11.2 we used `histogram` to count the number of times each letter appears in a word.

```
def histogram(s):
    d = dict()
    for c in s:
        if c not in d:
            d[c] = 1
```

```

    else:
        d[c] = d[c]+1
    return d

```

This function also works for lists, tuples, and even dictionaries, as long as the elements of `s` are hashable, so they can be used as keys in `d`.

```

>>> t = ['spam', 'egg', 'spam', 'spam', 'bacon', 'spam']
>>> histogram(t)
{'bacon': 1, 'egg': 1, 'spam': 4}

```

Functions that work with several types are called **polymorphic**. Polymorphism can facilitate code reuse. For example, the built-in function `sum`, which adds the elements of a sequence, works as long as the elements of the sequence support addition.

Since Time objects provide an `add` method, they work with `sum`:

```

>>> t1 = Time(7, 43)
>>> t2 = Time(7, 41)
>>> t3 = Time(7, 37)
>>> total = sum([t1, t2, t3])
>>> print(total)
23:01:00

```

In general, if all of the operations inside a function work with a given type, the function works with that type.

The best kind of polymorphism is the unintentional kind, where you discover that a function you already wrote can be applied to a type you never planned for.

17.10 Debugging

It is legal to add attributes to objects at any point in the execution of a program, but if you have objects with the same type that don't have the same attributes, it is easy to make mistakes. It is considered a good idea to initialize all of an object's attributes in the `init` method.

If you are not sure whether an object has a particular attribute, you can use the built-in function `hasattr` (see Section 15.7).

Another way to access attributes is the built-in function `vars`, which takes an object and returns a dictionary that maps from attribute names (as strings) to their values:

```

>>> p = Point(3, 4)
>>> vars(p)
{'y': 4, 'x': 3}

```

For purposes of debugging, you might find it useful to keep this function handy:

```

def print_attributes(obj):
    for attr in vars(obj):
        print(attr, getattr(obj, attr))

```

`print_attributes` traverses the dictionary and prints each attribute name and its corresponding value.

The built-in function `getattr` takes an object and an attribute name (as a string) and returns the attribute's value.

17.11 Interface and implementation

One of the goals of object-oriented design is to make software more maintainable, which means that you can keep the program working when other parts of the system change, and modify the program to meet new requirements.

A design principle that helps achieve that goal is to keep interfaces separate from implementations. For objects, that means that the methods a class provides should not depend on how the attributes are represented.

For example, in this chapter we developed a class that represents a time of day. Methods provided by this class include `time_to_int`, `is_after`, and `add_time`.

We could implement those methods in several ways. The details of the implementation depend on how we represent time. In this chapter, the attributes of a `Time` object are `hour`, `minute`, and `second`.

As an alternative, we could replace these attributes with a single integer representing the number of seconds since midnight. This implementation would make some methods, like `is_after`, easier to write, but it makes other methods harder.

After you deploy a new class, you might discover a better implementation. If other parts of the program are using your class, it might be time-consuming and error-prone to change the interface.

But if you designed the interface carefully, you can change the implementation without changing the interface, which means that other parts of the program don't have to change.

17.12 Glossary

object-oriented language: A language that provides features, such as programmer-defined types and methods, that facilitate object-oriented programming.

object-oriented programming: A style of programming in which data and the operations that manipulate it are organized into classes and methods.

method: A function that is defined inside a class definition and is invoked on instances of that class.

subject: The object a method is invoked on.

positional argument: An argument that does not include a parameter name, so it is not a keyword argument.

operator overloading: Changing the behavior of an operator like `+` so it works with a programmer-defined type.

type-based dispatch: A programming pattern that checks the type of an operand and invokes different functions for different types.

polymorphic: Pertaining to a function that can work with more than one type.

17.13 Exercises

Exercise 17.1. Download the code from this chapter from <http://thinkpython2.com/code/Time2.py>. Change the attributes of Time to be a single integer representing seconds since midnight. Then modify the methods (and the function int_to_time) to work with the new implementation. You should not have to modify the test code in main. When you are done, the output should be the same as before. Solution: http://thinkpython2.com/code/Time2_soln.py.

Exercise 17.2. This exercise is a cautionary tale about one of the most common, and difficult to find, errors in Python. Write a definition for a class named Kangaroo with the following methods:

1. An `__init__` method that initializes an attribute named `pouch_contents` to an empty list.
2. A method named `put_in_pouch` that takes an object of any type and adds it to `pouch_contents`.
3. A `__str__` method that returns a string representation of the Kangaroo object and the contents of the pouch.

Test your code by creating two Kangaroo objects, assigning them to variables named `kanga` and `roo`, and then adding `roo` to the contents of `kanga`'s pouch.

Download <http://thinkpython2.com/code/BadKangaroo.py>. It contains a solution to the previous problem with one big, nasty bug. Find and fix the bug.

If you get stuck, you can download <http://thinkpython2.com/code/GoodKangaroo.py>, which explains the problem and demonstrates a solution.

Chapter 18

Inheritance

The language feature most often associated with object-oriented programming is **inheritance**. Inheritance is the ability to define a new class that is a modified version of an existing class. In this chapter I demonstrate inheritance using classes that represent playing cards, decks of cards, and poker hands.

If you don't play poker, you can read about it at <http://en.wikipedia.org/wiki/Poker>, but you don't have to; I'll tell you what you need to know for the exercises.

Code examples from this chapter are available from <http://thinkpython2.com/code/Card.py>.

18.1 Card objects

There are fifty-two cards in a deck, each of which belongs to one of four suits and one of thirteen ranks. The suits are Spades, Hearts, Diamonds, and Clubs (in descending order in bridge). The ranks are Ace, 2, 3, 4, 5, 6, 7, 8, 9, 10, Jack, Queen, and King. Depending on the game that you are playing, an Ace may be higher than King or lower than 2.

If we want to define a new object to represent a playing card, it is obvious what the attributes should be: `rank` and `suit`. It is not as obvious what type the attributes should be. One possibility is to use strings containing words like 'Spade' for suits and 'Queen' for ranks. One problem with this implementation is that it would not be easy to compare cards to see which had a higher rank or suit.

An alternative is to use integers to **encode** the ranks and suits. In this context, "encode" means that we are going to define a mapping between numbers and suits, or between numbers and ranks. This kind of encoding is not meant to be a secret (that would be "encryption").

For example, this table shows the suits and the corresponding integer codes:

Spades	↔	3
Hearts	↔	2
Diamonds	↔	1
Clubs	↔	0

This code makes it easy to compare cards; because higher suits map to higher numbers, we can compare suits by comparing their codes.

The mapping for ranks is fairly obvious; each of the numerical ranks maps to the corresponding integer, and for face cards:

Jack	\mapsto	11
Queen	\mapsto	12
King	\mapsto	13

I am using the \mapsto symbol to make it clear that these mappings are not part of the Python program. They are part of the program design, but they don't appear explicitly in the code.

The class definition for `Card` looks like this:

```
class Card:
    """Represents a standard playing card."""

    def __init__(self, suit=0, rank=2):
        self.suit = suit
        self.rank = rank
```

As usual, the `init` method takes an optional parameter for each attribute. The default card is the 2 of Clubs.

To create a `Card`, you call `Card` with the suit and rank of the card you want.

```
queen_of_diamonds = Card(1, 12)
```

18.2 Class attributes

In order to print `Card` objects in a way that people can easily read, we need a mapping from the integer codes to the corresponding ranks and suits. A natural way to do that is with lists of strings. We assign these lists to **class attributes**:

```
# inside class Card:

    suit_names = ['Clubs', 'Diamonds', 'Hearts', 'Spades']
    rank_names = [None, 'Ace', '2', '3', '4', '5', '6', '7',
                  '8', '9', '10', 'Jack', 'Queen', 'King']

    def __str__(self):
        return '%s of %s' % (Card.rank_names[self.rank],
                             Card.suit_names[self.suit])
```

Variables like `suit_names` and `rank_names`, which are defined inside a class but outside of any method, are called class attributes because they are associated with the class object `Card`.

This term distinguishes them from variables like `suit` and `rank`, which are called **instance attributes** because they are associated with a particular instance.

Both kinds of attribute are accessed using dot notation. For example, in `__str__`, `self` is a `Card` object, and `self.rank` is its rank. Similarly, `Card` is a class object, and `Card.rank_names` is a list of strings associated with the class.

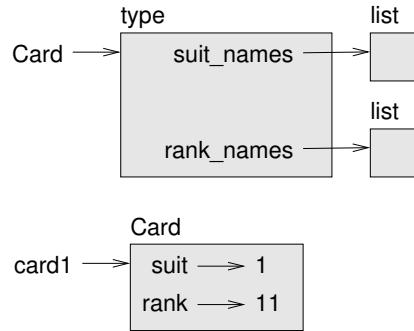


Figure 18.1: Object diagram.

Every card has its own suit and rank, but there is only one copy of `suit_names` and `rank_names`.

Putting it all together, the expression `Card.rank_names[self.rank]` means “use the attribute `rank` from the object `self` as an index into the list `rank_names` from the class `Card`, and select the appropriate string.”

The first element of `rank_names` is `None` because there is no card with rank zero. By including `None` as a place-keeper, we get a mapping with the nice property that the index 2 maps to the string '`2`', and so on. To avoid this tweak, we could have used a dictionary instead of a list.

With the methods we have so far, we can create and print cards:

```
>>> card1 = Card(2, 11)
>>> print(card1)
Jack of Hearts
```

Figure 18.1 is a diagram of the `Card` class object and one `Card` instance. `Card` is a class object; its type is `type`. `card1` is an instance of `Card`, so its type is `Card`. To save space, I didn’t draw the contents of `suit_names` and `rank_names`.

18.3 Comparing cards

For built-in types, there are relational operators (`<`, `>`, `==`, etc.) that compare values and determine when one is greater than, less than, or equal to another. For programmer-defined types, we can override the behavior of the built-in operators by providing a method named `__lt__`, which stands for “less than”.

`__lt__` takes two parameters, `self` and `other`, and returns `True` if `self` is strictly less than `other`.

The correct ordering for cards is not obvious. For example, which is better, the 3 of Clubs or the 2 of Diamonds? One has a higher rank, but the other has a higher suit. In order to compare cards, you have to decide whether rank or suit is more important.

The answer might depend on what game you are playing, but to keep things simple, we’ll make the arbitrary choice that suit is more important, so all of the Spades outrank all of the Diamonds, and so on.

With that decided, we can write `__lt__`:

```
# inside class Card:

def __lt__(self, other):
    # check the suits
    if self.suit < other.suit: return True
    if self.suit > other.suit: return False

    # suits are the same... check ranks
    return self.rank < other.rank
```

You can write this more concisely using tuple comparison:

```
# inside class Card:

def __lt__(self, other):
    t1 = self.suit, self.rank
    t2 = other.suit, other.rank
    return t1 < t2
```

As an exercise, write an `__lt__` method for Time objects. You can use tuple comparison, but you also might consider comparing integers.

18.4 Decks

Now that we have Cards, the next step is to define Decks. Since a deck is made up of cards, it is natural for each Deck to contain a list of cards as an attribute.

The following is a class definition for Deck. The `__init__` method creates the attribute `cards` and generates the standard set of fifty-two cards:

```
class Deck:

def __init__(self):
    self.cards = []
    for suit in range(4):
        for rank in range(1, 14):
            card = Card(suit, rank)
            self.cards.append(card)
```

The easiest way to populate the deck is with a nested loop. The outer loop enumerates the suits from 0 to 3. The inner loop enumerates the ranks from 1 to 13. Each iteration creates a new Card with the current suit and rank, and appends it to `self.cards`.

18.5 Printing the deck

Here is a `__str__` method for Deck:

```
# inside class Deck:
```

```
def __str__(self):
    res = []
```

```
        for card in self.cards:
            res.append(str(card))
        return '\n'.join(res)
```

This method demonstrates an efficient way to accumulate a large string: building a list of strings and then using the string method `join`. The built-in function `str` invokes the `__str__` method on each card and returns the string representation.

Since we invoke `join` on a newline character, the cards are separated by newlines. Here's what the result looks like:

```
>>> deck = Deck()
>>> print(deck)
Ace of Clubs
2 of Clubs
3 of Clubs
...
10 of Spades
Jack of Spades
Queen of Spades
King of Spades
```

Even though the result appears on 52 lines, it is one long string that contains newlines.

18.6 Add, remove, shuffle and sort

To deal cards, we would like a method that removes a card from the deck and returns it. The list method `pop` provides a convenient way to do that:

```
# inside class Deck:
```

```
    def pop_card(self):
        return self.cards.pop()
```

Since `pop` removes the *last* card in the list, we are dealing from the bottom of the deck.

To add a card, we can use the list method `append`:

```
# inside class Deck:
```

```
    def add_card(self, card):
        self.cards.append(card)
```

A method like this that uses another method without doing much work is sometimes called a **veneer**. The metaphor comes from woodworking, where a veneer is a thin layer of good quality wood glued to the surface of a cheaper piece of wood to improve the appearance.

In this case `add_card` is a “thin” method that expresses a list operation in terms appropriate for decks. It improves the appearance, or interface, of the implementation.

As another example, we can write a `Deck` method named `shuffle` using the function `shuffle` from the `random` module:

```
# inside class Deck:
```

```
    def shuffle(self):
        random.shuffle(self.cards)
```

Don't forget to import random.

As an exercise, write a Deck method named `sort` that uses the list method `sort` to sort the cards in a Deck. `sort` uses the `__lt__` method we defined to determine the order.

18.7 Inheritance

Inheritance is the ability to define a new class that is a modified version of an existing class. As an example, let's say we want a class to represent a "hand", that is, the cards held by one player. A hand is similar to a deck: both are made up of a collection of cards, and both require operations like adding and removing cards.

A hand is also different from a deck; there are operations we want for hands that don't make sense for a deck. For example, in poker we might compare two hands to see which one wins. In bridge, we might compute a score for a hand in order to make a bid.

This relationship between classes—similar, but different—lends itself to inheritance. To define a new class that inherits from an existing class, you put the name of the existing class in parentheses:

```
class Hand(Deck):
    """Represents a hand of playing cards."""
```

This definition indicates that `Hand` inherits from `Deck`; that means we can use methods like `pop_card` and `add_card` for `Hands` as well as `Decks`.

When a new class inherits from an existing one, the existing one is called the **parent** and the new class is called the **child**.

In this example, `Hand` inherits `__init__` from `Deck`, but it doesn't really do what we want: instead of populating the hand with 52 new cards, the `init` method for `Hands` should initialize `cards` with an empty list.

If we provide an `init` method in the `Hand` class, it overrides the one in the `Deck` class:

```
# inside class Hand:

    def __init__(self, label=''):
        self.cards = []
        self.label = label
```

When you create a `Hand`, Python invokes this `init` method, not the one in `Deck`.

```
>>> hand = Hand('new hand')
>>> hand.cards
[]
>>> hand.label
'new hand'
```

The other methods are inherited from `Deck`, so we can use `pop_card` and `add_card` to deal a card:

```
>>> deck = Deck()
>>> card = deck.pop_card()
>>> hand.add_card(card)
>>> print(hand)
King of Spades
```

A natural next step is to encapsulate this code in a method called `move_cards`:

```
# inside class Deck:

    def move_cards(self, hand, num):
        for i in range(num):
            hand.add_card(self.pop_card())
```

`move_cards` takes two arguments, a `Hand` object and the number of cards to deal. It modifies both `self` and `hand`, and returns `None`.

In some games, cards are moved from one hand to another, or from a hand back to the deck. You can use `move_cards` for any of these operations: `self` can be either a `Deck` or a `Hand`, and `hand`, despite the name, can also be a `Deck`.

Inheritance is a useful feature. Some programs that would be repetitive without inheritance can be written more elegantly with it. Inheritance can facilitate code reuse, since you can customize the behavior of parent classes without having to modify them. In some cases, the inheritance structure reflects the natural structure of the problem, which makes the design easier to understand.

On the other hand, inheritance can make programs difficult to read. When a method is invoked, it is sometimes not clear where to find its definition. The relevant code may be spread across several modules. Also, many of the things that can be done using inheritance can be done as well or better without it.

18.8 Class diagrams

So far we have seen stack diagrams, which show the state of a program, and object diagrams, which show the attributes of an object and their values. These diagrams represent a snapshot in the execution of a program, so they change as the program runs.

They are also highly detailed; for some purposes, too detailed. A class diagram is a more abstract representation of the structure of a program. Instead of showing individual objects, it shows classes and the relationships between them.

There are several kinds of relationship between classes:

- Objects in one class might contain references to objects in another class. For example, each `Rectangle` contains a reference to a `Point`, and each `Deck` contains references to many `Cards`. This kind of relationship is called **HAS-A**, as in, “a `Rectangle` has a `Point`.”
- One class might inherit from another. This relationship is called **IS-A**, as in, “a `Hand` is a kind of a `Deck`.”
- One class might depend on another in the sense that objects in one class take objects in the second class as parameters, or use objects in the second class as part of a computation. This kind of relationship is called a **dependency**.

A **class diagram** is a graphical representation of these relationships. For example, Figure 18.2 shows the relationships between `Card`, `Deck` and `Hand`.

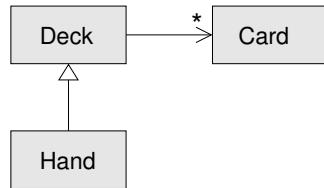


Figure 18.2: Class diagram.

The arrow with a hollow triangle head represents an IS-A relationship; in this case it indicates that Hand inherits from Deck.

The standard arrow head represents a HAS-A relationship; in this case a Deck has references to Card objects.

The star (*) near the arrow head is a **multiplicity**; it indicates how many Cards a Deck has. A multiplicity can be a simple number, like 52, a range, like 5..7 or a star, which indicates that a Deck can have any number of Cards.

There are no dependencies in this diagram. They would normally be shown with a dashed arrow. Or if there are a lot of dependencies, they are sometimes omitted.

A more detailed diagram might show that a Deck actually contains a *list* of Cards, but built-in types like list and dict are usually not included in class diagrams.

18.9 Debugging

Inheritance can make debugging difficult because when you invoke a method on an object, it might be hard to figure out which method will be invoked.

Suppose you are writing a function that works with Hand objects. You would like it to work with all kinds of Hands, like PokerHands, BridgeHands, etc. If you invoke a method like `shuffle`, you might get the one defined in Deck, but if any of the subclasses override this method, you'll get that version instead. This behavior is usually a good thing, but it can be confusing.

Any time you are unsure about the flow of execution through your program, the simplest solution is to add print statements at the beginning of the relevant methods. If `Deck.shuffle` prints a message that says something like `Running Deck.shuffle`, then as the program runs it traces the flow of execution.

As an alternative, you could use this function, which takes an object and a method name (as a string) and returns the class that provides the definition of the method:

```

def find_defining_class(obj, meth_name):
    for ty in type(obj).mro():
        if meth_name in ty.__dict__:
            return ty
  
```

Here's an example:

```

>>> hand = Hand()
>>> find_defining_class(hand, 'shuffle')
<class '__main__.Deck'>
  
```

So the `shuffle` method for this Hand is the one in Deck.

`find_defining_class` uses the `mro` method to get the list of class objects (types) that will be searched for methods. “MRO” stands for “method resolution order”, which is the sequence of classes Python searches to “resolve” a method name.

Here’s a design suggestion: when you override a method, the interface of the new method should be the same as the old. It should take the same parameters, return the same type, and obey the same preconditions and postconditions. If you follow this rule, you will find that any function designed to work with an instance of a parent class, like a Deck, will also work with instances of child classes like a Hand and PokerHand.

If you violate this rule, which is called the “Liskov substitution principle”, your code will collapse like (sorry) a house of cards.

18.10 Data encapsulation

The previous chapters demonstrate a development plan we might call “object-oriented design”. We identified objects we needed—like `Point`, `Rectangle` and `Time`—and defined classes to represent them. In each case there is an obvious correspondence between the object and some entity in the real world (or at least a mathematical world).

But sometimes it is less obvious what objects you need and how they should interact. In that case you need a different development plan. In the same way that we discovered function interfaces by encapsulation and generalization, we can discover class interfaces by **data encapsulation**.

Markov analysis, from Section 13.8, provides a good example. If you download my code from <http://thinkpython2.com/code/markov.py>, you’ll see that it uses two global variables—`suffix_map` and `prefix`—that are read and written from several functions.

```
suffix_map = {}
prefix = ()
```

Because these variables are global, we can only run one analysis at a time. If we read two texts, their prefixes and suffixes would be added to the same data structures (which makes for some interesting generated text).

To run multiple analyses, and keep them separate, we can encapsulate the state of each analysis in an object. Here’s what that looks like:

```
class Markov:

    def __init__(self):
        self.suffix_map = {}
        self.prefix = ()
```

Next, we transform the functions into methods. For example, here’s `process_word`:

```
def process_word(self, word, order=2):
    if len(self.prefix) < order:
        self.prefix += (word,)
    return
```

```

try:
    self.suffix_map[self.prefix].append(word)
except KeyError:
    # if there is no entry for this prefix, make one
    self.suffix_map[self.prefix] = [word]

self.prefix = shift(self.prefix, word)

```

Transforming a program like this—changing the design without changing the behavior—is another example of refactoring (see Section 4.7).

This example suggests a development plan for designing objects and methods:

1. Start by writing functions that read and write global variables (when necessary).
2. Once you get the program working, look for associations between global variables and the functions that use them.
3. Encapsulate related variables as attributes of an object.
4. Transform the associated functions into methods of the new class.

As an exercise, download my Markov code from <http://thinkpython2.com/code/markov.py>, and follow the steps described above to encapsulate the global variables as attributes of a new class called `Markov`. Solution: <http://thinkpython2.com/code/markov2.py>.

18.11 Glossary

encode: To represent one set of values using another set of values by constructing a mapping between them.

class attribute: An attribute associated with a class object. Class attributes are defined inside a class definition but outside any method.

instance attribute: An attribute associated with an instance of a class.

veneer: A method or function that provides a different interface to another function without doing much computation.

inheritance: The ability to define a new class that is a modified version of a previously defined class.

parent class: The class from which a child class inherits.

child class: A new class created by inheriting from an existing class; also called a “sub-class”.

IS-A relationship: A relationship between a child class and its parent class.

HAS-A relationship: A relationship between two classes where instances of one class contain references to instances of the other.

dependency: A relationship between two classes where instances of one class use instances of the other class, but do not store them as attributes.

class diagram: A diagram that shows the classes in a program and the relationships between them.

multiplicity: A notation in a class diagram that shows, for a HAS-A relationship, how many references there are to instances of another class.

data encapsulation: A program development plan that involves a prototype using global variables and a final version that makes the global variables into instance attributes.

18.12 Exercises

Exercise 18.1. For the following program, draw a UML class diagram that shows these classes and the relationships among them.

```
class PingPongParent:
    pass

class Ping(PingPongParent):
    def __init__(self, pong):
        self.pong = pong

class Pong(PingPongParent):
    def __init__(self, pings=None):
        if pings is None:
            self.pings = []
        else:
            self.pings = pings

    def add_ping(self, ping):
        self.pings.append(ping)

pong = Pong()
ping = Ping(pong)
pong.add_ping(ping)
```

Exercise 18.2. Write a Deck method called `deal_hands` that takes two parameters, the number of hands and the number of cards per hand. It should create the appropriate number of Hand objects, deal the appropriate number of cards per hand, and return a list of Hands.

Exercise 18.3. The following are the possible hands in poker, in increasing order of value and decreasing order of probability:

pair: two cards with the same rank

two pair: two pairs of cards with the same rank

three of a kind: three cards with the same rank

straight: five cards with ranks in sequence (aces can be high or low, so Ace-2-3-4-5 is a straight and so is 10-Jack-Queen-King-Ace, but Queen-King-Ace-2-3 is not.)

flush: five cards with the same suit

full house: three cards with one rank, two cards with another

four of a kind: four cards with the same rank

straight flush: five cards in sequence (as defined above) and with the same suit

The goal of these exercises is to estimate the probability of drawing these various hands.

1. Download the following files from <http://thinkpython2.com/code>:
`Card.py` : A complete version of the Card, Deck and Hand classes in this chapter.
`PokerHand.py` : An incomplete implementation of a class that represents a poker hand, and some code that tests it.
2. If you run `PokerHand.py`, it deals seven 7-card poker hands and checks to see if any of them contains a flush. Read this code carefully before you go on.
3. Add methods to `PokerHand.py` named `has_pair`, `has_twopair`, etc. that return True or False according to whether or not the hand meets the relevant criteria. Your code should work correctly for “hands” that contain any number of cards (although 5 and 7 are the most common sizes).
4. Write a method named `classify` that figures out the highest-value classification for a hand and sets the `label` attribute accordingly. For example, a 7-card hand might contain a flush and a pair; it should be labeled “flush”.
5. When you are convinced that your classification methods are working, the next step is to estimate the probabilities of the various hands. Write a function in `PokerHand.py` that shuffles a deck of cards, divides it into hands, classifies the hands, and counts the number of times various classifications appear.
6. Print a table of the classifications and their probabilities. Run your program with larger and larger numbers of hands until the output values converge to a reasonable degree of accuracy. Compare your results to the values at http://en.wikipedia.org/wiki/Hand_rankings.

Solution: <http://thinkpython2.com/code/PokerHandSoln.py>.

Chapter 19

The Goodies

One of my goals for this book has been to teach you as little Python as possible. When there were two ways to do something, I picked one and avoided mentioning the other. Or sometimes I put the second one into an exercise.

Now I want to go back for some of the good bits that got left behind. Python provides a number of features that are not really necessary—you can write good code without them—but with them you can sometimes write code that’s more concise, readable or efficient, and sometimes all three.

19.1 Conditional expressions

We saw conditional statements in Section 5.4. Conditional statements are often used to choose one of two values; for example:

```
if x > 0:  
    y = math.log(x)  
else:  
    y = float('nan')
```

This statement checks whether `x` is positive. If so, it computes `math.log`. If not, `math.log` would raise a `ValueError`. To avoid stopping the program, we generate a “`NaN`”, which is a special floating-point value that represents “Not a Number”.

We can write this statement more concisely using a **conditional expression**:

```
y = math.log(x) if x > 0 else float('nan')
```

You can almost read this line like English: “`y` gets `log-x` if `x` is greater than 0; otherwise it gets `NaN`”.

Recursive functions can sometimes be rewritten using conditional expressions. For example, here is a recursive version of `factorial`:

```
def factorial(n):  
    if n == 0:  
        return 1  
    else:  
        return n * factorial(n-1)
```

We can rewrite it like this:

```
def factorial(n):
    return 1 if n == 0 else n * factorial(n-1)
```

Another use of conditional expressions is handling optional arguments. For example, here is the init method from GoodKangaroo (see Exercise 17.2):

```
def __init__(self, name, contents=None):
    self.name = name
    if contents == None:
        contents = []
    self.pouch_contents = contents
```

We can rewrite this one like this:

```
def __init__(self, name, contents=None):
    self.name = name
    self.pouch_contents = [] if contents == None else contents
```

In general, you can replace a conditional statement with a conditional expression if both branches contain simple expressions that are either returned or assigned to the same variable.

19.2 List comprehensions

In Section 10.7 we saw the map and filter patterns. For example, this function takes a list of strings, maps the string method capitalize to the elements, and returns a new list of strings:

```
def capitalize_all(t):
    res = []
    for s in t:
        res.append(s.capitalize())
    return res
```

We can write this more concisely using a **list comprehension**:

```
def capitalize_all(t):
    return [s.capitalize() for s in t]
```

The bracket operators indicate that we are constructing a new list. The expression inside the brackets specifies the elements of the list, and the for clause indicates what sequence we are traversing.

The syntax of a list comprehension is a little awkward because the loop variable, `s` in this example, appears in the expression before we get to the definition.

List comprehensions can also be used for filtering. For example, this function selects only the elements of `t` that are upper case, and returns a new list:

```
def only_upper(t):
    res = []
    for s in t:
        if s.isupper():
            res.append(s)
    return res
```

We can rewrite it using a list comprehension

```
def only_upper(t):
    return [s for s in t if s.isupper()]
```

List comprehensions are concise and easy to read, at least for simple expressions. And they are usually faster than the equivalent for loops, sometimes much faster. So if you are mad at me for not mentioning them earlier, I understand.

But, in my defense, list comprehensions are harder to debug because you can't put a print statement inside the loop. I suggest that you use them only if the computation is simple enough that you are likely to get it right the first time. And for beginners that means never.

19.3 Generator expressions

Generator expressions are similar to list comprehensions, but with parentheses instead of square brackets:

```
>>> g = (x**2 for x in range(5))
>>> g
<generator object <genexpr> at 0x7f4c45a786c0>
```

The result is a generator object that knows how to iterate through a sequence of values. But unlike a list comprehension, it does not compute the values all at once; it waits to be asked. The built-in function `next` gets the next value from the generator:

```
>>> next(g)
0
>>> next(g)
1
```

When you get to the end of the sequence, `next` raises a `StopIteration` exception. You can also use a `for` loop to iterate through the values:

```
>>> for val in g:
...     print(val)
4
9
16
```

The generator object keeps track of where it is in the sequence, so the `for` loop picks up where `next` left off. Once the generator is exhausted, it continues to raise `StopIteration`:

```
>>> next(g)
StopIteration
```

Generator expressions are often used with functions like `sum`, `max`, and `min`:

```
>>> sum(x**2 for x in range(5))
30
```

19.4 any and all

Python provides a built-in function, `any`, that takes a sequence of boolean values and returns True if any of the values are True. It works on lists:

```
>>> any([False, False, True])
True
```

But it is often used with generator expressions:

```
>>> any(letter == 't' for letter in 'monty')
True
```

That example isn't very useful because it does the same thing as the `in` operator. But we could use `any` to rewrite some of the search functions we wrote in Section 9.3. For example, we could write `avoids` like this:

```
def avoids(word, forbidden):
    return not any(letter in forbidden for letter in word)
```

The function almost reads like English, “`word` avoids `forbidden` if there are not any forbidden letters in `word`.”

Using `any` with a generator expression is efficient because it stops immediately if it finds a `True` value, so it doesn't have to evaluate the whole sequence.

Python provides another built-in function, `all`, that returns `True` if every element of the sequence is `True`. As an exercise, use `all` to re-write `uses_all` from Section 9.3.

19.5 Sets

In Section 13.6 I use dictionaries to find the words that appear in a document but not in a word list. The function I wrote takes `d1`, which contains the words from the document as keys, and `d2`, which contains the list of words. It returns a dictionary that contains the keys from `d1` that are not in `d2`.

```
def subtract(d1, d2):
    res = dict()
    for key in d1:
        if key not in d2:
            res[key] = None
    return res
```

In all of these dictionaries, the values are `None` because we never use them. As a result, we waste some storage space.

Python provides another built-in type, called a `set`, that behaves like a collection of dictionary keys with no values. Adding elements to a set is fast; so is checking membership. And sets provide methods and operators to compute common set operations.

For example, set subtraction is available as a method called `difference` or as an operator, `-`. So we can rewrite `subtract` like this:

```
def subtract(d1, d2):
    return set(d1) - set(d2)
```

The result is a set instead of a dictionary, but for operations like iteration, the behavior is the same.

Some of the exercises in this book can be done concisely and efficiently with sets. For example, here is a solution to `has_duplicates`, from Exercise 10.7, that uses a dictionary:

```
def has_duplicates(t):
    d = {}
    for x in t:
        if x in d:
            return True
        d[x] = True
    return False
```

When an element appears for the first time, it is added to the dictionary. If the same element appears again, the function returns `True`.

Using sets, we can write the same function like this:

```
def has_duplicates(t):
    return len(set(t)) < len(t)
```

An element can only appear in a set once, so if an element in `t` appears more than once, the set will be smaller than `t`. If there are no duplicates, the set will be the same size as `t`.

We can also use sets to do some of the exercises in Chapter 9. For example, here's a version of `uses_only` with a loop:

```
def uses_only(word, available):
    for letter in word:
        if letter not in available:
            return False
    return True
```

`uses_only` checks whether all letters in `word` are in `available`. We can rewrite it like this:

```
def uses_only(word, available):
    return set(word) <= set(available)
```

The `<=` operator checks whether one set is a subset of another, including the possibility that they are equal, which is true if all the letters in `word` appear in `available`.

As an exercise, rewrite `avoids` using sets.

19.6 Counters

A Counter is like a set, except that if an element appears more than once, the Counter keeps track of how many times it appears. If you are familiar with the mathematical idea of a **multiset**, a Counter is a natural way to represent a multiset.

Counter is defined in a standard module called `collections`, so you have to import it. You can initialize a Counter with a string, list, or anything else that supports iteration:

```
>>> from collections import Counter
>>> count = Counter('parrot')
>>> count
Counter({'r': 2, 't': 1, 'o': 1, 'p': 1, 'a': 1})
```

Counters behave like dictionaries in many ways; they map from each key to the number of times it appears. As in dictionaries, the keys have to be hashable.

Unlike dictionaries, Counters don't raise an exception if you access an element that doesn't appear. Instead, they return 0:

```
>>> count['d']
0
```

We can use Counters to rewrite `is_anagram` from Exercise 10.6:

```
def is_anagram(word1, word2):
    return Counter(word1) == Counter(word2)
```

If two words are anagrams, they contain the same letters with the same counts, so their Counters are equivalent.

Counters provide methods and operators to perform set-like operations, including addition, subtraction, union and intersection. And they provide an often-useful method, `most_common`, which returns a list of value-frequency pairs, sorted from most common to least:

```
>>> count = Counter('parrot')
>>> for val, freq in count.most_common(3):
...     print(val, freq)
r 2
p 1
a 1
```

19.7 defaultdict

The `collections` module also provides `defaultdict`, which is like a dictionary except that if you access a key that doesn't exist, it can generate a new value on the fly.

When you create a `defaultdict`, you provide a function that's used to create new values. A function used to create objects is sometimes called a **factory**. The built-in functions that create lists, sets, and other types can be used as factories:

```
>>> from collections import defaultdict
>>> d = defaultdict(list)
```

Notice that the argument is `list`, which is a class object, not `list()`, which is a new list. The function you provide doesn't get called unless you access a key that doesn't exist.

```
>>> t = d['new key']
>>> t
[]
```

The new list, which we're calling `t`, is also added to the dictionary. So if we modify `t`, the change appears in `d`:

```
>>> t.append('new value')
>>> d
defaultdict(<class 'list'>, {'new key': ['new value']})
```

If you are making a dictionary of lists, you can often write simpler code using `defaultdict`. In my solution to Exercise 12.2, which you can get from http://thinkpython2.com/code/anagram_sets.py, I make a dictionary that maps from a sorted string of letters to the list of words that can be spelled with those letters. For example, 'opst' maps to the list ['opts', 'post', 'pots', 'spot', 'stop', 'tops'].

Here's the original code:

```
def all_anagrams(filename):
    d = {}
    for line in open(filename):
        word = line.strip().lower()
        t = signature(word)
        if t not in d:
            d[t] = [word]
        else:
            d[t].append(word)
    return d
```

This can be simplified using `setdefault`, which you might have used in Exercise 11.2:

```
def all_anagrams(filename):
    d = {}
    for line in open(filename):
        word = line.strip().lower()
        t = signature(word)
        d.setdefault(t, []).append(word)
    return d
```

This solution has the drawback that it makes a new list every time, regardless of whether it is needed. For lists, that's no big deal, but if the factory function is complicated, it might be.

We can avoid this problem and simplify the code using a `defaultdict`:

```
def all_anagrams(filename):
    d = defaultdict(list)
    for line in open(filename):
        word = line.strip().lower()
        t = signature(word)
        d[t].append(word)
    return d
```

My solution to Exercise 18.3, which you can download from <http://thinkpython2.com/code/PokerHandSoln.py>, uses `setdefault` in the function `has_straightflush`. This solution has the drawback of creating a `Hand` object every time through the loop, whether it is needed or not. As an exercise, rewrite it using a `defaultdict`.

19.8 Named tuples

Many simple objects are basically collections of related values. For example, the `Point` object defined in Chapter 15 contains two numbers, `x` and `y`. When you define a class like this, you usually start with an `__init__` method and a `str` method:

```
class Point:

    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y

    def __str__(self):
        return '(%g, %g)' % (self.x, self.y)
```

This is a lot of code to convey a small amount of information. Python provides a more concise way to say the same thing:

```
from collections import namedtuple
Point = namedtuple('Point', ['x', 'y'])
```

The first argument is the name of the class you want to create. The second is a list of the attributes Point objects should have, as strings. The return value from `namedtuple` is a class object:

```
>>> Point
<class '__main__.Point'>
```

`Point` automatically provides methods like `__init__` and `__str__` so you don't have to write them.

To create a `Point` object, you use the `Point` class as a function:

```
>>> p = Point(1, 2)
>>> p
Point(x=1, y=2)
```

The `init` method assigns the arguments to attributes using the names you provided. The `str` method prints a representation of the `Point` object and its attributes.

You can access the elements of the named tuple by name:

```
>>> p.x, p.y
(1, 2)
```

But you can also treat a named tuple as a tuple:

```
>>> p[0], p[1]
(1, 2)
```

```
>>> x, y = p
>>> x, y
(1, 2)
```

Named tuples provide a quick way to define simple classes. The drawback is that simple classes don't always stay simple. You might decide later that you want to add methods to a named tuple. In that case, you could define a new class that inherits from the named tuple:

```
class Pointier(Point):
    # add more methods here
```

Or you could switch to a conventional class definition.

19.9 Gathering keyword args

In Section 12.4, we saw how to write a function that gathers its arguments into a tuple:

```
def printall(*args):
    print(args)
```

You can call this function with any number of positional arguments (that is, arguments that don't have keywords):

```
>>> printall(1, 2.0, '3')
(1, 2.0, '3')
```

But the * operator doesn't gather keyword arguments:

```
>>> printall(1, 2.0, third='3')
TypeError: printall() got an unexpected keyword argument 'third'
```

To gather keyword arguments, you can use the ** operator:

```
def printall(*args, **kwargs):
    print(args, kwargs)
```

You can call the keyword gathering parameter anything you want, but kwargs is a common choice. The result is a dictionary that maps from keywords to values:

```
>>> printall(1, 2.0, third='3')
(1, 2.0) {'third': '3'}
```

If you have a dictionary of keywords and values, you can use the scatter operator, ** to call a function:

```
>>> d = dict(x=1, y=2)
>>> Point(**d)
Point(x=1, y=2)
```

Without the scatter operator, the function would treat d as a single positional argument, so it would assign d to x and complain because there's nothing to assign to y:

```
>>> d = dict(x=1, y=2)
>>> Point(d)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: __new__() missing 1 required positional argument: 'y'
```

When you are working with functions that have a large number of parameters, it is often useful to create and pass around dictionaries that specify frequently used options.

19.10 Glossary

conditional expression: An expression that has one of two values, depending on a condition.

list comprehension: An expression with a for loop in square brackets that yields a new list.

generator expression: An expression with a for loop in parentheses that yields a generator object.

multiset: A mathematical entity that represents a mapping between the elements of a set and the number of times they appear.

factory: A function, usually passed as a parameter, used to create objects.

19.11 Exercises

Exercise 19.1. *The following is a function that computes the binomial coefficient recursively.*

```
def binomial_coeff(n, k):
    """Compute the binomial coefficient "n choose k".

    n: number of trials
    k: number of successes

    returns: int
    """
    if k == 0:
        return 1
    if n == 0:
        return 0

    res = binomial_coeff(n-1, k) + binomial_coeff(n-1, k-1)
    return res
```

Rewrite the body of the function using nested conditional expressions.

One note: this function is not very efficient because it ends up computing the same values over and over. You could make it more efficient by memoizing (see Section 11.6). But you will find that it's harder to memoize if you write it using conditional expressions.

Appendix A

Debugging

When you are debugging, you should distinguish among different kinds of errors in order to track them down more quickly:

- Syntax errors are discovered by the interpreter when it is translating the source code into byte code. They indicate that there is something wrong with the structure of the program. Example: Omitting the colon at the end of a `def` statement generates the somewhat redundant message `SyntaxError: invalid syntax`.
- Runtime errors are produced by the interpreter if something goes wrong while the program is running. Most runtime error messages include information about where the error occurred and what functions were executing. Example: An infinite recursion eventually causes the runtime error “maximum recursion depth exceeded”.
- Semantic errors are problems with a program that runs without producing error messages but doesn’t do the right thing. Example: An expression may not be evaluated in the order you expect, yielding an incorrect result.

The first step in debugging is to figure out which kind of error you are dealing with. Although the following sections are organized by error type, some techniques are applicable in more than one situation.

A.1 Syntax errors

Syntax errors are usually easy to fix once you figure out what they are. Unfortunately, the error messages are often not helpful. The most common messages are `SyntaxError: invalid syntax` and `SyntaxError: invalid token`, neither of which is very informative.

On the other hand, the message does tell you where in the program the problem occurred. Actually, it tells you where Python noticed a problem, which is not necessarily where the error is. Sometimes the error is prior to the location of the error message, often on the preceding line.

If you are building the program incrementally, you should have a good idea about where the error is. It will be in the last line you added.

If you are copying code from a book, start by comparing your code to the book's code very carefully. Check every character. At the same time, remember that the book might be wrong, so if you see something that looks like a syntax error, it might be.

Here are some ways to avoid the most common syntax errors:

1. Make sure you are not using a Python keyword for a variable name.
2. Check that you have a colon at the end of the header of every compound statement, including `for`, `while`, `if`, and `def` statements.
3. Make sure that any strings in the code have matching quotation marks. Make sure that all quotation marks are "straight quotes", not "curly quotes".
4. If you have multiline strings with triple quotes (single or double), make sure you have terminated the string properly. An unterminated string may cause an invalid token error at the end of your program, or it may treat the following part of the program as a string until it comes to the next string. In the second case, it might not produce an error message at all!
5. An unclosed opening operator—`(`, `{`, or `[`—makes Python continue with the next line as part of the current statement. Generally, an error occurs almost immediately in the next line.
6. Check for the classic `=` instead of `==` inside a conditional.
7. Check the indentation to make sure it lines up the way it is supposed to. Python can handle space and tabs, but if you mix them it can cause problems. The best way to avoid this problem is to use a text editor that knows about Python and generates consistent indentation.
8. If you have non-ASCII characters in the code (including strings and comments), that might cause a problem, although Python 3 usually handles non-ASCII characters. Be careful if you paste in text from a web page or other source.

If nothing works, move on to the next section...

A.1.1 I keep making changes and it makes no difference.

If the interpreter says there is an error and you don't see it, that might be because you and the interpreter are not looking at the same code. Check your programming environment to make sure that the program you are editing is the one Python is trying to run.

If you are not sure, try putting an obvious and deliberate syntax error at the beginning of the program. Now run it again. If the interpreter doesn't find the new error, you are not running the new code.

There are a few likely culprits:

- You edited the file and forgot to save the changes before running it again. Some programming environments do this for you, but some don't.

- You changed the name of the file, but you are still running the old name.
- Something in your development environment is configured incorrectly.
- If you are writing a module and using `import`, make sure you don't give your module the same name as one of the standard Python modules.
- If you are using `import` to read a module, remember that you have to restart the interpreter or use `reload` to read a modified file. If you import the module again, it doesn't do anything.

If you get stuck and you can't figure out what is going on, one approach is to start again with a new program like "Hello, World!", and make sure you can get a known program to run. Then gradually add the pieces of the original program to the new one.

A.2 Runtime errors

Once your program is syntactically correct, Python can read it and at least start running it. What could possibly go wrong?

A.2.1 My program does absolutely nothing.

This problem is most common when your file consists of functions and classes but does not actually invoke a function to start execution. This may be intentional if you only plan to import this module to supply classes and functions.

If it is not intentional, make sure there is a function call in the program, and make sure the flow of execution reaches it (see "Flow of Execution" below).

A.2.2 My program hangs.

If a program stops and seems to be doing nothing, it is "hanging". Often that means that it is caught in an infinite loop or infinite recursion.

- If there is a particular loop that you suspect is the problem, add a `print` statement immediately before the loop that says "entering the loop" and another immediately after that says "exiting the loop".

Run the program. If you get the first message and not the second, you've got an infinite loop. Go to the "Infinite Loop" section below.

- Most of the time, an infinite recursion will cause the program to run for a while and then produce a "RuntimeError: Maximum recursion depth exceeded" error. If that happens, go to the "Infinite Recursion" section below.

If you are not getting this error but you suspect there is a problem with a recursive method or function, you can still use the techniques in the "Infinite Recursion" section.

- If neither of those steps works, start testing other loops and other recursive functions and methods.
- If that doesn't work, then it is possible that you don't understand the flow of execution in your program. Go to the "Flow of Execution" section below.

Infinite Loop

If you think you have an infinite loop and you think you know what loop is causing the problem, add a `print` statement at the end of the loop that prints the values of the variables in the condition and the value of the condition.

For example:

```
while x > 0 and y < 0 :  
    # do something to x  
    # do something to y  
  
    print('x: ', x)  
    print('y: ', y)  
    print("condition: ", (x > 0 and y < 0))
```

Now when you run the program, you will see three lines of output for each time through the loop. The last time through the loop, the condition should be `False`. If the loop keeps going, you will be able to see the values of `x` and `y`, and you might figure out why they are not being updated correctly.

Infinite Recursion

Most of the time, infinite recursion causes the program to run for a while and then produce a `Maximum recursion depth exceeded` error.

If you suspect that a function is causing an infinite recursion, make sure that there is a base case. There should be some condition that causes the function to return without making a recursive invocation. If not, you need to rethink the algorithm and identify a base case.

If there is a base case but the program doesn't seem to be reaching it, add a `print` statement at the beginning of the function that prints the parameters. Now when you run the program, you will see a few lines of output every time the function is invoked, and you will see the parameter values. If the parameters are not moving toward the base case, you will get some ideas about why not.

Flow of Execution

If you are not sure how the flow of execution is moving through your program, add `print` statements to the beginning of each function with a message like "entering function `foo`", where `foo` is the name of the function.

Now when you run the program, it will print a trace of each function as it is invoked.

A.2.3 When I run the program I get an exception.

If something goes wrong during runtime, Python prints a message that includes the name of the exception, the line of the program where the problem occurred, and a traceback.

The traceback identifies the function that is currently running, and then the function that called it, and then the function that called *that*, and so on. In other words, it traces the

sequence of function calls that got you to where you are, including the line number in your file where each call occurred.

The first step is to examine the place in the program where the error occurred and see if you can figure out what happened. These are some of the most common runtime errors:

NameError: You are trying to use a variable that doesn't exist in the current environment.

Check if the name is spelled right, or at least consistently. And remember that local variables are local; you cannot refer to them from outside the function where they are defined.

TypeError: There are several possible causes:

- You are trying to use a value improperly. Example: indexing a string, list, or tuple with something other than an integer.
- There is a mismatch between the items in a format string and the items passed for conversion. This can happen if either the number of items does not match or an invalid conversion is called for.
- You are passing the wrong number of arguments to a function. For methods, look at the method definition and check that the first parameter is `self`. Then look at the method invocation; make sure you are invoking the method on an object with the right type and providing the other arguments correctly.

KeyError: You are trying to access an element of a dictionary using a key that the dictionary does not contain. If the keys are strings, remember that capitalization matters.

AttributeError: You are trying to access an attribute or method that does not exist. Check the spelling! You can use the built-in function `vars` to list the attributes that do exist.

If an `AttributeError` indicates that an object has `NoneType`, that means that it is `None`. So the problem is not the attribute name, but the object.

The reason the object is `None` might be that you forgot to return a value from a function; if you get to the end of a function without hitting a `return` statement, it returns `None`. Another common cause is using the result from a list method, like `sort`, that returns `None`.

IndexError: The index you are using to access a list, string, or tuple is greater than its length minus one. Immediately before the site of the error, add a `print` statement to display the value of the index and the length of the array. Is the array the right size? Is the index the right value?

The Python debugger (`pdb`) is useful for tracking down exceptions because it allows you to examine the state of the program immediately before the error. You can read about `pdb` at <https://docs.python.org/3/library/pdb.html>.

A.2.4 I added so many print statements I get inundated with output.

One of the problems with using `print` statements for debugging is that you can end up buried in output. There are two ways to proceed: simplify the output or simplify the program.

To simplify the output, you can remove or comment out print statements that aren't helping, or combine them, or format the output so it is easier to understand.

To simplify the program, there are several things you can do. First, scale down the problem the program is working on. For example, if you are searching a list, search a *small* list. If the program takes input from the user, give it the simplest input that causes the problem.

Second, clean up the program. Remove dead code and reorganize the program to make it as easy to read as possible. For example, if you suspect that the problem is in a deeply nested part of the program, try rewriting that part with simpler structure. If you suspect a large function, try splitting it into smaller functions and testing them separately.

Often the process of finding the minimal test case leads you to the bug. If you find that a program works in one situation but not in another, that gives you a clue about what is going on.

Similarly, rewriting a piece of code can help you find subtle bugs. If you make a change that you think shouldn't affect the program, and it does, that can tip you off.

A.3 Semantic errors

In some ways, semantic errors are the hardest to debug, because the interpreter provides no information about what is wrong. Only you know what the program is supposed to do.

The first step is to make a connection between the program text and the behavior you are seeing. You need a hypothesis about what the program is actually doing. One of the things that makes that hard is that computers run so fast.

You will often wish that you could slow the program down to human speed, and with some debuggers you can. But the time it takes to insert a few well-placed print statements is often short compared to setting up the debugger, inserting and removing breakpoints, and "stepping" the program to where the error is occurring.

A.3.1 My program doesn't work.

You should ask yourself these questions:

- Is there something the program was supposed to do but which doesn't seem to be happening? Find the section of the code that performs that function and make sure it is executing when you think it should.
- Is something happening that shouldn't? Find code in your program that performs that function and see if it is executing when it shouldn't.
- Is a section of code producing an effect that is not what you expected? Make sure that you understand the code in question, especially if it involves functions or methods in other Python modules. Read the documentation for the functions you call. Try them out by writing simple test cases and checking the results.

In order to program, you need a mental model of how programs work. If you write a program that doesn't do what you expect, often the problem is not in the program; it's in your mental model.

The best way to correct your mental model is to break the program into its components (usually the functions and methods) and test each component independently. Once you find the discrepancy between your model and reality, you can solve the problem.

Of course, you should be building and testing components as you develop the program. If you encounter a problem, there should be only a small amount of new code that is not known to be correct.

A.3.2 I've got a big hairy expression and it doesn't do what I expect.

Writing complex expressions is fine as long as they are readable, but they can be hard to debug. It is often a good idea to break a complex expression into a series of assignments to temporary variables.

For example:

```
self.hands[i].addCard(self.hands[self.findNeighbor(i)].popCard())
```

This can be rewritten as:

```
neighbor = self.findNeighbor(i)
pickedCard = self.hands[neighbor].popCard()
self.hands[i].addCard(pickedCard)
```

The explicit version is easier to read because the variable names provide additional documentation, and it is easier to debug because you can check the types of the intermediate variables and display their values.

Another problem that can occur with big expressions is that the order of evaluation may not be what you expect. For example, if you are translating the expression $\frac{x}{2\pi}$ into Python, you might write:

```
y = x / 2 * math.pi
```

That is not correct because multiplication and division have the same precedence and are evaluated from left to right. So this expression computes $x\pi/2$.

A good way to debug expressions is to add parentheses to make the order of evaluation explicit:

```
y = x / (2 * math.pi)
```

Whenever you are not sure of the order of evaluation, use parentheses. Not only will the program be correct (in the sense of doing what you intended), it will also be more readable for other people who haven't memorized the order of operations.

A.3.3 I've got a function that doesn't return what I expect.

If you have a `return` statement with a complex expression, you don't have a chance to print the result before returning. Again, you can use a temporary variable. For example, instead of:

```
return self.hands[i].removeMatches()
```

you could write:

```
count = self.hands[i].removeMatches()  
return count
```

Now you have the opportunity to display the value of count before returning.

A.3.4 I'm really, really stuck and I need help.

First, try getting away from the computer for a few minutes. Computers emit waves that affect the brain, causing these symptoms:

- Frustration and rage.
- Superstitious beliefs (“the computer hates me”) and magical thinking (“the program only works when I wear my hat backward”).
- Random walk programming (the attempt to program by writing every possible program and choosing the one that does the right thing).

If you find yourself suffering from any of these symptoms, get up and go for a walk. When you are calm, think about the program. What is it doing? What are some possible causes of that behavior? When was the last time you had a working program, and what did you do next?

Sometimes it just takes time to find a bug. I often find bugs when I am away from the computer and let my mind wander. Some of the best places to find bugs are trains, showers, and in bed, just before you fall asleep.

A.3.5 No, I really need help.

It happens. Even the best programmers occasionally get stuck. Sometimes you work on a program so long that you can't see the error. You need a fresh pair of eyes.

Before you bring someone else in, make sure you are prepared. Your program should be as simple as possible, and you should be working on the smallest input that causes the error. You should have print statements in the appropriate places (and the output they produce should be comprehensible). You should understand the problem well enough to describe it concisely.

When you bring someone in to help, be sure to give them the information they need:

- If there is an error message, what is it and what part of the program does it indicate?
- What was the last thing you did before this error occurred? What were the last lines of code that you wrote, or what is the new test case that fails?
- What have you tried so far, and what have you learned?

When you find the bug, take a second to think about what you could have done to find it faster. Next time you see something similar, you will be able to find the bug more quickly.

Remember, the goal is not just to make the program work. The goal is to learn how to make the program work.

Appendix B

Analysis of Algorithms

This appendix is an edited excerpt from *Think Complexity*, by Allen B. Downey, also published by O'Reilly Media (2012). When you are done with this book, you might want to move on to that one.

Analysis of algorithms is a 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” (http://en.wikipedia.org/wiki/Radix_sort)¹.

The goal of algorithm analysis is to make meaningful comparisons between algorithms, but there are some problems:

- 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

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.”

run slower in this case. A common way to avoid this problem is to analyze the **worst case** scenario. It is sometimes useful to analyze average case performance, but that's usually harder, and it might not be obvious 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 group functions into categories depending on how quickly they grow as problem size increases.

The good thing about this kind of comparison is 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, at least for large values of n .

This kind of analysis comes with some caveats, but we'll get to that later.

B.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	100 010 001

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 $an^2 > bn$, for any values of a and b .

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 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 called **quadratic**.

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
$O(n \log_b n)$	linearithmic
$O(n^2)$	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 B.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 $af + b$, where a and b are constants?
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 $f_1 \cdot f_2$?

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. Sometimes the details of the hardware, the programming language, and the characteristics of the input make a big difference. And for small problems, order of growth is irrelevant.

But if you keep those caveats in mind, algorithmic analysis is a useful tool. At least for large problems, the “better” algorithm 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!

B.2 Analysis of basic Python operations

In Python, 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.

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.

String concatenation is linear; the run time depends on the sum of the lengths of the operands.

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. The string method `join` is linear; the run time depends on the total length of the strings.

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 the average time for each 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 `update` is proportional to the size of the dictionary passed as a parameter, not the dictionary being updated.
- `keys`, `values` and `items` are constant time because they return iterators. But if you loop through the iterators, the loop will be linear.

The performance of dictionaries is one of the minor miracles of computer science. We will see how they work in Section B.4.

Exercise B.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 Python use an $O(n \log n)$ comparison sort?

B.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 might use to look a word up in a dictionary (a paper 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.

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 hash tables, which is why most dictionary operations, including the `in` operator, are constant time.

B.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(k, v)`: Add a new item that maps from key `k` to value `v`. With a Python dictionary, `d`, this operation is written `d[k] = v`.

`get(k)`: Look up and return the value that corresponds to key `k`. With a Python dictionary, `d`, this operation is written `d[k]` or `d.get(k)`.

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:
```

```
    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 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:
```

```
    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 equivalent return the same hash value, but the converse is not necessarily true: two objects with different values 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:

    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

```

`__init__` creates a `BetterMap` 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 hash into the same `LinearMap` will get split up (which is what we wanted, right?).

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 B.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 unit each, the third costs 3 units, etc.

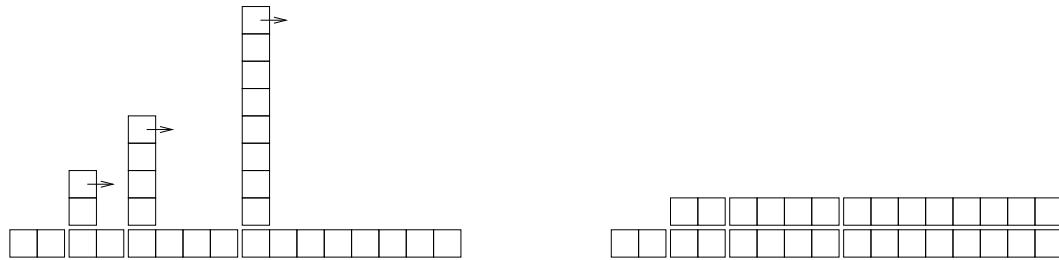


Figure B.1: The cost of a hashtable add.

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, spreading 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 <http://thinkpython2.com/code/Map.py>, but remember that there is no reason to use it; if you want a map, just use a Python dictionary.

B.5 Glossary

analysis of algorithms: A way to compare algorithms in terms of their run time and/or space requirements.

machine model: A simplified representation of a computer used to describe algorithms.

worst case: The input that makes a given algorithm run slowest (or require the most space).

leading term: In a polynomial, the term with the highest exponent.

crossover point: The problem size where two algorithms require the same run time or space.

order of growth: A set of functions that all grow in a way considered equivalent for purposes of analysis of algorithms. For example, all functions that grow linearly belong to the same order of growth.

Big-Oh notation: Notation for representing an order of growth; for example, $O(n)$ represents the set of functions that grow linearly.

linear: An algorithm whose run time is proportional to problem size, at least for large problem sizes.

quadratic: An algorithm whose run time is proportional to n^2 , where n is a measure of problem size.

search: The problem of locating an element of a collection (like a list or dictionary) or determining that it is not present.

hashtable: A data structure that represents a collection of key-value pairs and performs search in constant time.

Index

- abecedarian, 73, 84
- abs function, 52
- absolute path, 139, 145
- access, 90
- accumulator, 100
 - histogram, 127
 - list, 93
 - string, 175
 - sum, 93
- Ackermann function, 61, 113
- add method, 165
- addition with carrying, 68
- algorithm, 67, 69, 130, 201
 - MD5, 146
 - square root, 69
- aliasing, 95, 96, 100, 149, 151, 170
 - copying to avoid, 99
- all, 186
- alphabet, 37
- alternative execution, 41
- ambiguity, 5
- anagram, 101
- anagram set, 123, 145
- analysis of algorithms, 201, 209
- analysis of primitives, 204
- and operator, 40
- any, 185
- append method, 92, 97, 101, 174, 175
- arc function, 31
- Archimedian spiral, 38
- argument, 17, 19, 21, 22, 26, 97
 - gather, 118
 - keyword, 33, 36, 191
 - list, 97
 - optional, 76, 79, 95, 107, 184
 - positional, 164, 169, 190
 - variable-length tuple, 118
- argument scatter, 118
- arithmetic operator, 3
- assert statement, 159, 160
- assignment, 14, 63, 89
 - augmented, 93, 100
 - item, 74, 90, 116
 - tuple, 116, 117, 119, 122
- assignment statement, 9
- attribute, 153, 169
 - __dict__, 168
 - class, 172, 180
 - initializing, 168
 - instance, 148, 153, 172, 180
- AttributeError, 152, 197
- augmented assignment, 93, 100
- Austen, Jane, 127
- average case, 202
- average cost, 208
- badness, 203
- base case, 44, 47
- benchmarking, 133, 134
- BetterMap, 206
- big, hairy expression, 199
- Big-Oh notation, 209
- big-oh notation, 203
- binary search, 101
- bingo, 123
- birthday, 160
- birthday paradox, 101
- bisect module, 101
- bisection search, 101, 205
- bisection, debugging by, 68
- bitwise operator, 3
- body, 19, 26, 65
- bool type, 40
- boolean expression, 40, 47
- boolean function, 54
- boolean operator, 76
- borrowing, subtraction with, 68, 159
- bounded, 207
- bracket
 - squiggly, 103
- bracket operator, 71, 90, 116
- branch, 41, 47

break statement, 66
bubble sort, 201
bug, 6, 7, 13
 worst, 170
built-in function
 any, 185, 186
bytes object, 141, 145

calculator, 8, 15
call graph, 109, 112
Car Talk, 88, 113, 124
Card class, 172
card, playing, 171
carrying, addition with, 68, 156, 158
catch, 145
chained conditional, 41, 47
character, 71
checksum, 143, 146
child class, 176, 180
choice function, 126
circle function, 31
circular definition, 55
class, 4, 147, 153
 Card, 172
 child, 176, 180
 Deck, 174
 Hand, 176
 Kangaroo, 170
 parent, 176
 Point, 148, 165
 Rectangle, 149
 Time, 155
class attribute, 172, 180
class definition, 147
class diagram, 177, 181
class object, 148, 153, 190
close method, 138, 141, 143
 `__cmp__` method, 173
Collatz conjecture, 65
collections, 187, 188, 190
colon, 19, 194
comment, 12, 14
commutativity, 12, 167
compare function, 52
comparing algorithms, 201
comparison
 string, 77
 tuple, 116, 174
comparison sort, 205
composition, 19, 22, 26, 54, 174

compound statement, 41, 47
concatenation, 12, 14, 22, 73, 74, 95
 list, 91, 97, 101
condition, 41, 47, 65, 196
conditional, 194
 chained, 41, 47
 nested, 42, 47
conditional execution, 41
conditional expression, 183, 191
conditional statement, 41, 47, 55, 184
consistency check, 111, 158
constant time, 208
contributors, vii
conversion
 type, 17
copy
 deep, 152
 shallow, 152
 slice, 74, 92
 to avoid aliasing, 99
copy module, 151
copying objects, 151
count method, 79
Counter, 187
counter, 75, 79, 104, 111
counting and looping, 75
Creative Commons, vi
crossover point, 202, 209
crosswords, 83
cumulative sum, 100

data encapsulation, 179, 181
data structure, 122, 123, 132
database, 141, 145
database object, 141
datetime module, 160
dbm module, 141
dead code, 52, 60, 198
debugger (pdb), 197
debugging, 6, 7, 13, 36, 46, 59, 77, 87, 98, 111,
 122, 133, 144, 152, 159, 168, 178,
 185, 193
by bisection, 68
emotional response, 6, 200
experimental, 25
rubber duck, 134
superstition, 200
deck, 171
Deck class, 174
deck, playing cards, 174

declaration, 110, 112
decrement, 64, 69
deep copy, 152, 153
deepcopy function, 152
def keyword, 19
default value, 129, 134, 165
 avoiding mutable, 170
defaultdict, 188
definition
 circular, 55
 class, 147
 function, 19
 recursive, 124
del operator, 94
deletion, element of list, 94
delimiter, 95, 100
designed development, 160
deterministic, 126, 134
development plan, 36
 data encapsulation, 179, 181
 designed, 158
 encapsulation and generalization, 35
 incremental, 52, 193
 prototype and patch, 156, 158
 random walk programming, 134, 200
 reduction, 85, 87
diagram
 call graph, 112
 class, 177, 181
 object, 148, 150, 152, 153, 155, 173
 stack, 23, 97
 state, 9, 63, 78, 90, 96, 108, 120, 148, 150,
 152, 155, 173
__dict__ attribute, 168
dict function, 103
dictionary, 103, 112, 120, 197
 initialize, 120
 invert, 107
 lookup, 106
 looping with, 106
 reverse lookup, 106
 subtraction, 129
 traversal, 120, 168
dictionary methods, 204
 dbm module, 141
dictionary subtraction, 186
diff, 146
Dijkstra, Edsger, 87
dir function, 197
directory, 139, 145
 walk, 140
 working, 139
dispatch
 type-based, 167
dispatch, type-based, 166
divisibility, 39
division
 floating-point, 39
 floor, 39, 46, 47
divmod, 117, 158
docstring, 35, 37, 148
dot notation, 18, 26, 76, 148, 162, 172
Double Day, 160
double letters, 88
Doyle, Arthur Conan, 25
duplicate, 101, 113, 146, 187
element, 89, 100
element deletion, 94
elif keyword, 42
Elkner, Jeff, v, vi
ellipses, 19
else keyword, 41
email address, 117
embedded object, 150, 153, 170
 copying, 152
emotional debugging, 6, 200
empty list, 89
empty string, 79, 95
encapsulation, 32, 36, 54, 69, 75, 177
encode, 171, 180
encrypt, 171
end of line character, 144
enumerate function, 119
enumerate object, 119
epsilon, 67
equality and assignment, 63
equivalence, 96, 152
equivalent, 100
error
 runtime, 13, 44, 46, 193
 semantic, 13, 193, 198
 shape, 122
 syntax, 13, 193
error checking, 58
error message, 7, 13, 193
eval function, 69
evaluate, 10
exception, 13, 14, 193, 196
 AttributeError, 152, 197

FileNotFoundException, 140
IndexError, 72, 78, 90, 197
KeyError, 104, 197
LookupError, 107
NameError, 22, 197
OverflowError, 46
RuntimeError, 45
StopIteration, 185
SyntaxError, 19
TypeError, 72, 74, 108, 116, 118, 139, 164, 197
UnboundLocalError, 110
ValueError, 46, 117
exception, catching, 140
execute, 11, 14
exists function, 139
experimental debugging, 25, 134
exponent, 202
exponential growth, 203
expression, 10, 14
 big and hairy, 199
 boolean, 40, 47
 conditional, 183, 191
 generator, 185, 186, 191
extend method, 92

factorial, 183
factorial function, 56, 58
factory, 191
factory function, 188, 189
False special value, 40
Fermat's Last Theorem, 48
fibonacci function, 57, 109
file, 137
 permission, 140
 reading and writing, 137
file object, 83, 87
filename, 139
FileNotFoundException, 140
filter pattern, 93, 100, 184
find function, 74
flag, 110, 112
float function, 17
float type, 4
floating-point, 4, 7, 67, 183
floating-point division, 39
floor division, 39, 46, 47
flow of execution, 21, 26, 58, 59, 65, 178, 196
flower, 37
folder, 139

for loop, 30, 44, 72, 91, 119, 184
formal language, 4, 7
format operator, 138, 145, 197
format sequence, 138, 145
format string, 138, 145
frame, 23, 26, 44, 56, 109
Free Documentation License, GNU, v, vi
frequency, 105
 letter, 123
 word, 125, 134
fruitful function, 24, 26
frustration, 200
function, 3, 17, 19, 25, 161
 abs, 52
 ack, 61, 113
 arc, 31
 choice, 126
 circle, 31
 compare, 52
 deepcopy, 152
 dict, 103
 dir, 197
 enumerate, 119
 eval, 69
 exists, 139
 factorial, 56, 183
 fibonacci, 57, 109
 find, 74
 float, 17
 fruitful, 24
 getattr, 168
 getcwd, 139
 hasattr, 153, 168
 input, 45
 int, 17
 isinstance, 58, 153, 166
 len, 26, 72, 104
 list, 94
 log, 18
 math, 18
 max, 117, 118
 min, 117, 118
 open, 83, 84, 137, 140, 141
 polygon, 31
 popen, 142
 programmer defined, 22, 129
 randint, 101, 126
 random, 126
 reasons for, 24
 recursive, 43

reload, 144, 195
repr, 144
reversed, 121
shuffle, 175
sorted, 99, 106, 121
sqrt, 18, 53
str, 18
sum, 118, 185
trigonometric, 18
tuple, 115
tuple as return value, 117
type, 153
void, 24
zip, 118
function argument, 21
function call, 17, 26
function composition, 54
function definition, 19, 20, 25
function frame, 23, 26, 44, 56, 109
function object, 27
function parameter, 21
function syntax, 162
function type, 20
 modifier, 157
 pure, 156
functional programming style, 158, 160

gamma function, 58
gather, 118, 123, 190
GCD (greatest common divisor), 61
generalization, 32, 36, 85, 159
generator expression, 185, 186, 191
generator object, 185
geometric resizing, 209
get method, 105
getattr function, 168
getcwd function, 139
global statement, 110, 112
global variable, 110, 112
 update, 110
GNU Free Documentation License, v, vi
greatest common divisor (GCD), 61
grid, 27
guardian pattern, 59, 60, 78

Hand class, 176
hanging, 195
HAS-A relationship, 177, 180
hasattr function, 153, 168
hash function, 108, 112, 207

hashable, 108, 112, 120
HashMap, 207
hashtable, 112, 206, 210
header, 19, 25, 194
Hello, World, 3
hexadecimal, 148
high-level language, 6
histogram, 105
 random choice, 126, 130
 word frequencies, 127
Holmes, Sherlock, 25
homophone, 113
hypotenuse, 54

identical, 100
identity, 96, 152
if statement, 41
immutability, 74, 79, 97, 108, 115, 121
implementation, 105, 112, 132, 169
import statement, 26, 144
in operator, 205
in operator, 76, 85, 90, 104
increment, 64, 69, 157, 163
incremental development, 60, 193
indentation, 19, 162, 194
index, 71, 78, 79, 90, 103, 197
 looping with, 86, 91
 negative, 72
 slice, 73, 91
 starting at zero, 71, 90
IndexError, 72, 78, 90, 197
indexing, 204
infinite loop, 65, 69, 195, 196
infinite recursion, 44, 47, 58, 195, 196
inheritance, 176, 178, 180, 190
init method, 164, 168, 172, 174, 176
initialization
 variable, 69
initialization (before update), 64
input function, 45
instance, 148, 153
 as argument, 149
 as return value, 150
instance attribute, 148, 153, 172, 180
instantiate, 153
instantiation, 148
int function, 17
int type, 4
integer, 4, 7
interactive mode, 11, 14, 24

interface, 33, 36, 169, 179
 interlocking words, 101
 interpret, 6
 interpreter, 2
 invariant, 159, 160
 invert dictionary, 107
 invocation, 76, 79
 is operator, 95, 152
 IS-A relationship, 177, 180
 isinstance function, 58, 153, 166
 item, 74, 79, 89, 103
 dictionary, 112
 item assignment, 74, 90, 116
 item update, 91
 items method, 120
 iteration, 64, 69
 iterator, 119–121, 123, 204
 join, 204
 join method, 95, 175
 Kangaroo class, 170
 key, 103, 112
 key-value pair, 103, 112, 120
 keyboard input, 45
 KeyError, 104, 197
 KeyError, 206
 keyword, 10, 14, 194
 def, 19
 elif, 42
 else, 41
 keyword argument, 33, 36, 191
 Koch curve, 49
 language
 formal, 4
 natural, 4
 safe, 13
 Turing complete, 55
 leading coefficient, 202
 leading term, 202, 209
 leap of faith, 57
 len function, 26, 72, 104
 letter frequency, 123
 letter rotation, 80, 113
 linear, 209
 linear growth, 203
 linear search, 205
 LinearMap, 206
 Linux, 25
 lipogram, 84
 Liskov substitution principle, 179
 list, 89, 94, 100, 121, 184
 as argument, 97
 concatenation, 91, 97, 101
 copy, 92
 element, 90
 empty, 89
 function, 94
 index, 90
 membership, 90
 method, 92
 nested, 89, 91
 of objects, 174
 of tuples, 119
 operation, 91
 repetition, 91
 slice, 91
 traversal, 91
 list comprehension, 184, 191
 list methods, 204
 literalness, 5
 local variable, 22, 26
 log function, 18
 logarithm, 135
 logarithmic growth, 203
 logical operator, 40
 lookup, 112
 lookup, dictionary, 106
 LookupError, 107
 loop, 31, 36, 65, 119
 condition, 196
 for, 30, 44, 72, 91
 infinite, 65, 196
 nested, 174
 traversal, 72
 while, 64
 loop variable, 184
 looping
 with dictionaries, 106
 with indices, 86, 91
 with strings, 75
 looping and counting, 75
 low-level language, 6
 ls (Unix command), 142
 machine model, 201, 209
 main, 23, 43, 110, 144
 maintainable, 169
 map pattern, 93, 100
 map to, 171

- mapping, 112, 131
- Markov analysis, 130
- mash-up, 132
- math function, 18
- matplotlib, 135
- max function, 117, 118
- McCloskey, Robert, 73
- md5, 143
- MD5 algorithm, 146
- md5sum, 146
- membership
 - binary search, 101
 - bisection search, 101
 - dictionary, 104
 - list, 90
 - set, 113
- memo, 109, 112
- mental model, 199
- metaphor, method invocation, 163
- metathesis, 123
- method, 36, 75, 161, 169
 - `__cmp__`, 173
 - `__str__`, 165, 174
 - add, 165
 - append, 92, 97, 101, 174, 175
 - close, 138, 141, 143
 - count, 79
 - extend, 92
 - get, 105
 - init, 164, 172, 174, 176
 - items, 120
 - join, 95, 175
 - mro, 179
 - pop, 94, 175
 - radd, 167
 - read, 143
 - readline, 83, 143
 - remove, 94
 - replace, 125
 - setdefault, 113
 - sort, 92, 99, 176
 - split, 95, 117
 - string, 79
 - strip, 84, 125
 - translate, 125
 - update, 120
 - values, 104
 - void, 92
- method resolution order, 179
- method syntax, 162
- method, list, 92
- Meyers, Chris, vi
- min function, 117, 118
- Moby Project, 83
- model, mental, 199
- modifier, 157, 160
- module, 18, 26
 - bisect, 101
 - collections, 187, 188, 190
 - copy, 151
 - datetime, 160
 - dbm, 141
 - os, 139
 - pickle, 137, 142
 - pprint, 112
 - profile, 133
 - random, 101, 126, 175
 - reload, 144, 195
 - shelve, 142
 - string, 125
 - structshape, 122
 - time, 101
- module object, 18, 143
- module, writing, 143
- modulus operator, 39, 47
- Monty Python and the Holy Grail, 156
- MP3, 146
- mro method, 179
- multiline string, 35, 194
- multiplicity (in class diagram), 178, 181
- multiset, 187
- mutability, 74, 90, 92, 96, 111, 115, 121, 151
- mutable object, as default value, 170
- name built-in variable, 144
- namedtuple, 190
- NameError, 22, 197
- NaN, 183
- natural language, 4, 7
- negative index, 72
- nested conditional, 42, 47
- nested list, 89, 91, 100
- newline, 45, 175
- Newton's method, 66
- None special value, 24, 26, 52, 92, 94
- NoneType type, 24
- not operator, 40
- number, random, 126
- Obama, Barack, 201

object, 74, 79, 95, 96, 100
 bytes, 141, 145
 class, 147, 148, 153, 190
 copying, 151
 Counter, 187
 database, 141
 defaultdict, 188
 embedded, 150, 153, 170
 enumerate, 119
 file, 83, 87
 function, 27
 generator, 185
 module, 143
 mutable, 151
 namedtuple, 190
 pipe, 145
 printing, 162
 set, 186
 zip, 123
object diagram, 148, 150, 152, 153, 155, 173
object-oriented design, 169
object-oriented language, 169
object-oriented programming, 147, 161, 169, 176
odometer, 88
Olin College, v
open function, 83, 84, 137, 140, 141
operand, 14
operator, 7
 and, 40
 arithmetic, 3
 bitwise, 3
 boolean, 76
 bracket, 71, 90, 116
 del, 94
 format, 138, 145, 197
 in, 76, 85, 90, 104
 is, 95, 152
 logical, 40
 modulus, 39, 47
 not, 40
 or, 40
 overloading, 169
 relational, 40, 173
 slice, 73, 79, 91, 98, 116
 string, 12
 update, 93
operator overloading, 166, 173
optional argument, 76, 79, 95, 107, 184
optional parameter, 129, 165
or operator, 40
order of growth, 202, 209
order of operations, 11, 14, 199
os module, 139
other (parameter name), 164
OverflowError, 46
overloading, 169
override, 129, 134, 165, 173, 176, 179
palindrome, 61, 80, 86, 88
parameter, 21, 23, 26, 97
 gather, 118
 optional, 129, 165
 other, 164
 self, 163
parent class, 176, 180
parentheses
 argument in, 17
 empty, 19, 76
 parameters in, 21, 22
 parent class in, 176
 tuples in, 115
parse, 5, 7
pass statement, 41
path, 139, 145
 absolute, 139
 relative, 139
pattern
 filter, 93, 100, 184
 guardian, 59, 60, 78
 map, 93, 100
 reduce, 93, 100
 search, 75, 79, 85, 107, 186
 swap, 116
pdb (Python debugger), 197
PEMDAS, 11
permission, file, 140
persistence, 137, 145
pi, 18, 70
pickle module, 137, 142
pickling, 142
pie, 37
pipe, 142
pipe object, 145
plain text, 83, 125
planned development, 158
poetry, 5
Point class, 148, 165
point, mathematical, 147
poker, 171, 181

- polygon function, 31
- polymorphism, 168, 169
- pop method, 94, 175
- popen function, 142
- portability, 6
- positional argument, 164, 169, 190
- postcondition, 36, 59, 179
- pprint module, 112
- precedence, 199
- precondition, 36, 37, 59, 179
- prefix, 131
- pretty print, 112
- print function, 3
- print statement, 3, 7, 165, 197
- problem solving, 1, 6
- profile module, 133
- program, 1, 6
- program testing, 87
- programmer-defined function, 22, 129
- programmer-defined type, 147, 153, 155, 162, 165, 173
- Project Gutenberg, 125
- prompt, 2, 6, 45
- prose, 5
- prototype and patch, 156, 158, 160
- pseudorandom, 126, 134
- pure function, 156, 160
- Puzzler, 88, 113, 124
- Pythagorean theorem, 52
- Python
 - running, 2
 - Python 2, 2, 3, 33, 40, 45
 - Python in a browser, 2
 - PythonAnywhere, 2
- quadratic, 209
- quadratic growth, 203
- quotation mark, 3, 4, 35, 74, 194
- radd method, 167
- radian, 18
- radix sort, 201
- rage, 200
- raise statement, 107, 112, 159
- Ramanujan, Srinivasa, 70
- randint function, 101, 126
- random function, 126
- random module, 101, 126, 175
- random number, 126
- random text, 131
- random walk programming, 134, 200
- rank, 171
- read method, 143
- readline method, 83, 143
- reassignment, 63, 68, 90, 110
- Rectangle class, 149
- recursion, 43, 47, 55, 57
 - base case, 44
 - infinite, 44, 58, 196
- recursive definition, 56, 124
- red-black tree, 206
- reduce pattern, 93, 100
- reducible word, 113, 124
- reduction to a previously solved problem, 85
- reduction to a previously solved problem, 87
- redundancy, 5
- refactoring, 34–36, 180
- reference, 96, 97, 100
 - aliasing, 96
- rehashing, 208
- relational operator, 40, 173
- relative path, 139, 145
- reload function, 144, 195
- remove method, 94
- repetition, 30
 - list, 91
- replace method, 125
- repr function, 144
- representation, 147, 149, 171
- return statement, 44, 51, 199
- return value, 17, 26, 51, 150
 - tuple, 117
- reverse lookup, 112
- reverse lookup, dictionary, 106
- reverse word pair, 101
- reversed function, 121
- rotation
 - letters, 113
- rotation, letter, 80
- rubber duck debugging, 134
- running pace, 8, 15, 160
- running Python, 2
- runtime error, 13, 44, 46, 193, 196
- RuntimeError, 45, 58
- safe language, 13
- sanity check, 111
- scaffolding, 53, 60, 112

scatter, 118, 123, 191
Schmidt, Eric, 201
Scrabble, 123
script, 11, 14
script mode, 11, 14, 24
search, 107, 205, 209
search pattern, 75, 79, 85, 186
search, binary, 101
search, bisection, 101
self (parameter name), 163
semantic error, 13, 14, 193, 198
semantics, 14, 162
sequence, 4, 71, 79, 89, 94, 115, 121
set, 130, 186
 anagram, 123, 145
set membership, 113
set subtraction, 186
setdefault, 189
setdefault method, 113
sexagesimal, 158
shallow copy, 152, 153
shape, 123
shape error, 122
shell, 142, 145
shelve module, 142
shuffle function, 175
sine function, 18
singleton, 108, 112, 115
slice, 79
 copy, 74, 92
 list, 91
 string, 73
 tuple, 116
 update, 92
slice operator, 73, 79, 91, 98, 116
sort method, 92, 99, 176
sorted
 function, 99, 106
sorted function, 121
sorting, 204, 205
special case, 87, 157
special value
 False, 40
 None, 24, 26, 52, 92, 94
 True, 40
spiral, 38
split method, 95, 117
sqrt, 53
sqrt function, 18
square root, 66
squiggly bracket, 103
stable sort, 205
stack diagram, 23, 26, 37, 44, 56, 60, 97
state diagram, 9, 14, 63, 78, 90, 96, 108, 120, 148, 150, 152, 155, 173
statement, 10, 14
 assert, 159, 160
 assignment, 9, 63
 break, 66
 compound, 41
 conditional, 41, 47, 55, 184
 for, 30, 72, 91
 global, 110, 112
 if, 41
 import, 26, 144
 pass, 41
 print, 3, 7, 165, 197
 raise, 107, 112, 159
 return, 44, 51, 199
 try, 140, 153
 while, 64
step size, 79
StopIteration, 185
str function, 18
__str__ method, 165, 174
string, 4, 7, 94, 121
 accumulator, 175
 comparison, 77
 empty, 95
 immutable, 74
 method, 75
 multiline, 35, 194
 operation, 12
 slice, 73
 triple-quoted, 35
string concatenation, 204
string method, 79
string methods, 204
string module, 125
string representation, 144, 165
string type, 4
strip method, 84, 125
structshape module, 122
structure, 5
subject, 163, 169
subset, 187
subtraction
 dictionary, 129
 with borrowing, 68
subtraction with borrowing, 159

- suffix, 131
- suit, 171
- sum, 185
- sum function, 118
- superstitious debugging, 200
- swap pattern, 116
- syntax, 5, 7, 13, 162, 194
- syntax error, 13, 14, 193
- SyntaxError, 19
- temporary variable, 51, 60, 199
- test case, minimal, 198
- testing
 - and absence of bugs, 87
 - incremental development, 52
 - is hard, 87
 - knowing the answer, 53
 - leap of faith, 57
 - minimal test case, 198
- text
 - plain, 83, 125
 - random, 131
- text file, 145
- Time class, 155
- time module, 101
- token, 5, 7
- traceback, 24, 26, 44, 46, 107, 196
- translate method, 125
- traversal, 72, 75, 77, 79, 85, 93, 100, 105, 106, 119, 127
 - dictionary, 168
 - list, 91
- traverse
 - dictionary, 120
- triangle, 48
- trigonometric function, 18
- triple-quoted string, 35
- True special value, 40
- try statement, 140, 153
- tuple, 115, 117, 121, 122
 - as key in dictionary, 120, 132
 - assignment, 116
 - comparison, 116, 174
 - in brackets, 120
 - singleton, 115
 - slice, 116
- tuple assignment, 117, 119, 122
- tuple function, 115
- tuple methods, 204
- Turing complete language, 55
- Turing Thesis, 55
- Turing, Alan, 55
- turtle module, 48
- turtle typewriter, 37
- type, 4, 7
 - bool, 40
 - dict, 103
 - file, 137
 - float, 4
 - function, 20
 - int, 4
 - list, 89
 - NoneType, 24
 - programmer-defined, 147, 153, 155, 162, 165, 173
 - set, 130
 - str, 4
 - tuple, 115
- type checking, 58
- type conversion, 17
- type function, 153
- type-based dispatch, 166, 167, 169
- TypeError, 72, 74, 108, 116, 118, 139, 164, 197
- typewriter, turtle, 37
- typographical error, 134
- UnboundLocalError, 110
- underscore character, 10
- uniqueness, 101
- Unix command
 - ls, 142
- update, 64, 67, 69
 - database, 141
 - global variable, 110
 - histogram, 127
 - item, 91
 - slice, 92
- update method, 120
- update operator, 93
- use before def, 20
- value, 4, 7, 95, 96, 112
 - default, 129
 - tuple, 117
- ValueError, 46, 117
- values method, 104
- variable, 9, 14
 - global, 110
 - local, 22
- temporary, 51, 60, 199

updating, 64
variable-length argument tuple, 118
veneer, 175, 180
void function, 24, 26
void method, 92
vorpal, 55

walk, directory, 140
while loop, 64
whitespace, 46, 84, 144, 194
word count, 143
word frequency, 125, 134
word, reducible, 113, 124
working directory, 139
worst bug, 170
worst case, 202, 209

zero, index starting at, 71
zero, index starting at, 90
zip function, 118
 use with dict, 120
zip object, 123
Zipf's law, 134