

# LEARN PYTHON THE RIGHT WAY



HOW TO THINK LIKE A  
COMPUTER SCIENTIST



**Ritza**

Marketing as Education

# **Learn Python the right way**

How to think like a computer scientist

Ritza

© 2021 - 2022 Ritza

# Contents

<b>Copyright Notice . . . . .</b>	<b>1</b>
<b>Foreword . . . . .</b>	<b>2</b>
<b>Preface . . . . .</b>	<b>4</b>
How and why I came to use Python . . . . .	4
Finding a textbook . . . . .	5
Introducing programming with Python . . . . .	6
Building a community . . . . .	7
<b>Contributor List . . . . .</b>	<b>8</b>
Second Edition . . . . .	8
First Edition . . . . .	9
<b>Chapter 1: The way of the program . . . . .</b>	<b>11</b>
1.1. The Python programming language . . . . .	11
1.2. What is a program? . . . . .	13
1.3. What is debugging? . . . . .	13
1.4. Syntax errors . . . . .	14
1.5. Runtime errors . . . . .	14
1.6. Semantic errors . . . . .	14
1.7. Experimental debugging . . . . .	14
1.8. Formal and natural languages . . . . .	15
1.9. The first program . . . . .	17
1.10. Comments . . . . .	18
1.11. Glossary . . . . .	18
1.12. Exercises . . . . .	20
<b>Chapter 2: Variables, expressions and statements . . . . .</b>	<b>23</b>
2.1. Values and data types . . . . .	23
2.2. Variables . . . . .	25
2.3. Variable names and keywords . . . . .	27
2.4. Statements . . . . .	28
2.5. Evaluating expressions . . . . .	28
2.6. Operators and operands . . . . .	29

## CONTENTS

2.7. Type converter functions . . . . .	30
2.8. Order of operations . . . . .	31
2.9. Operations on strings . . . . .	32
2.10. Input . . . . .	33
2.11. Composition . . . . .	34
2.12. The modulus operator . . . . .	35
2.13. Glossary . . . . .	35
2.14. Exercises . . . . .	38
<b>Chapter 3: Hello, little turtles! . . . . .</b>	<b>40</b>
3.1. Our first turtle program . . . . .	40
3.2. Instances — a herd of turtles . . . . .	43
3.3. The for loop . . . . .	45
3.4. Flow of Execution of the for loop . . . . .	46
3.5. The loop simplifies our turtle program . . . . .	47
3.6. A few more turtle methods and tricks . . . . .	49
3.7. Glossary . . . . .	52
3.8. Exercises . . . . .	53
<b>Chapter 4: Functions . . . . .</b>	<b>56</b>
4.1. Functions . . . . .	56
4.2. Functions can call other functions . . . . .	59
4.3. Flow of execution . . . . .	60
4.4. Functions that require arguments . . . . .	63
4.5. Functions that return values . . . . .	63
4.6. Variables and parameters are local . . . . .	65
4.7. Turtles Revisited . . . . .	66
4.8. Glossary . . . . .	67
4.9. Exercises . . . . .	69
<b>Chapter 5: Conditionals . . . . .</b>	<b>73</b>
5.1. Boolean values and expressions . . . . .	73
5.2. Logical operators . . . . .	74
5.3. Truth Tables . . . . .	75
5.4. Simplifying Boolean Expressions . . . . .	75
5.5. Conditional execution . . . . .	76
5.6. Omitting the else clause . . . . .	78
5.7. Chained conditionals . . . . .	79
5.8. Nested conditionals . . . . .	81
5.9. The return statement . . . . .	82
5.10. Logical opposites . . . . .	82
5.11. Type conversion . . . . .	84
5.12. A Turtle Bar Chart . . . . .	85
5.13. Glossary . . . . .	89

## CONTENTS

5.14. Exercises . . . . .	90
<b>Chapter 6: Fruitful functions . . . . .</b>	<b>93</b>
6.1. Return values . . . . .	93
6.2. Program development . . . . .	95
6.3. Debugging with print . . . . .	98
6.4. Composition . . . . .	99
6.5. Boolean functions . . . . .	100
6.6. Programming with style . . . . .	101
6.7. Unit testing . . . . .	102
6.8. Glossary . . . . .	104
6.9. Exercises . . . . .	105
<b>Chapter 7: Iteration . . . . .</b>	<b>110</b>
7.1. Assignment . . . . .	110
7.2. Updating variables . . . . .	111
7.3. The <code>for</code> loop revisited . . . . .	112
7.4. The <code>while</code> statement . . . . .	112
7.5. The Collatz $3n + 1$ sequence . . . . .	114
7.6. Tracing a program . . . . .	116
7.7. Counting digits . . . . .	117
7.8. Abbreviated assignment . . . . .	118
7.9. Help and meta-notation . . . . .	119
7.10. Tables . . . . .	121
7.11. Two-dimensional tables . . . . .	122
7.12. Encapsulation and generalization . . . . .	123
7.13. More encapsulation . . . . .	124
7.14. Local variables . . . . .	124
7.15. The <code>break</code> statement . . . . .	125
7.16. Other flavours of loops . . . . .	126
7.17. An example . . . . .	128
7.18. The <code>continue</code> statement . . . . .	129
7.19. More generalization . . . . .	130
7.20. Functions . . . . .	131
7.21. Paired Data . . . . .	132
7.22. Nested Loops for Nested Data . . . . .	133
7.23. Newton's method for finding square roots . . . . .	134
7.24. Algorithms . . . . .	135
7.25. Glossary . . . . .	136
7.26. Exercises . . . . .	138
<b>Chapter 8: Strings . . . . .</b>	<b>143</b>
8.1. A compound data type . . . . .	143
8.2. Working with strings as single things . . . . .	143

## CONTENTS

8.3. Working with the parts of a string . . . . .	145
8.4. Length . . . . .	146
8.5. Traversal and the for loop . . . . .	146
8.6. Slices . . . . .	147
8.7. String comparison . . . . .	148
8.8. Strings are immutable . . . . .	149
8.9. The <code>in</code> and <code>not in</code> operators . . . . .	149
8.10. A <code>find</code> function . . . . .	150
8.11. Looping and counting . . . . .	151
8.12. Optional parameters . . . . .	151
8.13. The built-in <code>find</code> method . . . . .	153
8.14. The <code>split</code> method . . . . .	153
8.15. Cleaning up your strings . . . . .	154
8.16. The string format method . . . . .	155
8.17. Summary . . . . .	158
8.18. Glossary . . . . .	159
8.19. Exercises . . . . .	160
<b>Chapter 9: Tuples . . . . .</b>	<b>165</b>
9.1. Tuples are used for grouping data . . . . .	165
9.2. Tuple assignment . . . . .	166
9.3. Tuples as return values . . . . .	167
9.4. Composability of Data Structures . . . . .	168
9.5. Glossary . . . . .	168
9.6. Exercises . . . . .	169
<b>Chapter 10: Event handling . . . . .</b>	<b>170</b>
10.1. Event-driven programming . . . . .	170
10.2. Keypress events . . . . .	170
10.3. Mouse events . . . . .	171
10.4. Automatic events from a timer . . . . .	173
10.5. An example: state machines . . . . .	174
10.6. Glossary . . . . .	177
10.7. Exercises . . . . .	177
<b>Chapter 11: Lists . . . . .</b>	<b>180</b>
11.1. List values . . . . .	180
11.2. Accessing elements . . . . .	180
11.3. List length . . . . .	181
11.4. List membership . . . . .	182
11.5. List operations . . . . .	183
11.6. List slices . . . . .	183
11.7. Lists are mutable . . . . .	183
11.8. List deletion . . . . .	185

## CONTENTS

11.9. Objects and references . . . . .	185
11.10. Aliasing . . . . .	186
11.11. Cloning lists . . . . .	187
11.12. Lists and for loops . . . . .	188
11.13. List parameters . . . . .	189
11.14. List methods . . . . .	190
11.15. Pure functions and modifiers . . . . .	191
11.16. Functions that produce lists . . . . .	192
11.17. Strings and lists . . . . .	193
11.18. list and range . . . . .	194
11.19. Nested lists . . . . .	195
11.20. Matrices . . . . .	196
11.21. Glossary . . . . .	196
11.22. Exercises . . . . .	198
<b>Chapter 12: Modules . . . . .</b>	<b>202</b>
12.1. Random numbers . . . . .	202
12.2. The time module . . . . .	205
12.3. The math module . . . . .	206
12.4. Creating your own modules . . . . .	207
12.5. Namespaces . . . . .	208
12.6. Scope and lookup rules . . . . .	210
12.7. Attributes and the dot operator . . . . .	211
12.8. Three import statement variants . . . . .	212
12.9. Turn your unit tester into a module . . . . .	213
12.10. Glossary . . . . .	213
12.11. Exercises . . . . .	214
<b>Chapter 13: Files . . . . .</b>	<b>219</b>
13.1. About files . . . . .	219
13.2. Writing our first file . . . . .	219
13.3. Reading a file line-at-a-time . . . . .	220
13.4. Turning a file into a list of lines . . . . .	221
13.5. Reading the whole file at once . . . . .	222
13.6. Working with binary files . . . . .	222
13.7. An example . . . . .	223
13.8. Directories . . . . .	224
13.9. What about fetching something from the web? . . . . .	225
13.10. Glossary . . . . .	226
13.11. Exercises . . . . .	227
<b>Chapter 14: List Algorithms . . . . .</b>	<b>228</b>
14.1. Test-driven development . . . . .	228
14.2. The linear search algorithm . . . . .	228

## CONTENTS

14.3. A more realistic problem . . . . .	230
14.4. Binary Search . . . . .	234
14.5. Removing adjacent duplicates from a list . . . . .	238
14.6. Merging sorted lists . . . . .	239
14.7. Alice in Wonderland, again! . . . . .	241
14.8. Eight Queens puzzle, part 1 . . . . .	243
14.9. Eight Queens puzzle, part 2 . . . . .	247
14.10. Glossary . . . . .	249
14.11. Exercises . . . . .	250
<b>Chapter 15: Classes and Objects — the Basics . . . . .</b>	<b>254</b>
15.1. Object-oriented programming . . . . .	254
15.2. User-defined compound data types . . . . .	254
15.3. Attributes . . . . .	256
15.4. Improving our initializer . . . . .	257
15.5. Adding other methods to our class . . . . .	258
15.6. Instances as arguments and parameters . . . . .	260
15.7. Converting an instance to a string . . . . .	260
15.8. Instances as return values . . . . .	261
15.9. A change of perspective . . . . .	263
15.10. Objects can have state . . . . .	263
15.11. Glossary . . . . .	264
15.12. Exercises . . . . .	265
<b>Chapter 16: Classes and Objects — Digging a little deeper . . . . .</b>	<b>267</b>
16.1. Rectangles . . . . .	267
16.2. Objects are mutable . . . . .	268
16.3. Sameness . . . . .	269
16.4. Copying . . . . .	271
16.5. Glossary . . . . .	272
16.6. Exercises . . . . .	273
<b>Chapter 17: PyGame . . . . .</b>	<b>275</b>
17.1. The game loop . . . . .	275
17.2. Displaying images and text . . . . .	279
17.3. Drawing a board for the N queens puzzle . . . . .	282
17.4. Sprites . . . . .	288
17.5. Events . . . . .	292
17.6. A wave of animation . . . . .	295
17.7. Aliens - a case study . . . . .	300
17.8. Reflections . . . . .	301
17.9. Glossary . . . . .	301
17.10. Exercises . . . . .	302

## CONTENTS

<b>Chapter 18: Recursion . . . . .</b>	<b>303</b>
18.1. Drawing Fractals . . . . .	303
18.2. Recursive data structures . . . . .	306
18.3. Processing recursive number lists . . . . .	307
18.4. Case study: Fibonacci numbers . . . . .	309
18.5. Example with recursive directories and files . . . . .	310
18.6. An animated fractal, using PyGame . . . . .	311
18.7. Glossary . . . . .	314
18.8. Exercises . . . . .	315
<b>Chapter 19: Exceptions . . . . .</b>	<b>319</b>
19.1. Catching exceptions . . . . .	319
19.2. Raising our own exceptions . . . . .	321
19.3. Revisiting an earlier example . . . . .	322
19.4. The <code>finally</code> clause of the <code>try</code> statement . . . . .	322
19.5. Glossary . . . . .	323
19.6. Exercises . . . . .	324
<b>Chapter 20: Dictionaries . . . . .</b>	<b>325</b>
20.1. Dictionary operations . . . . .	326
20.2. Dictionary methods . . . . .	327
20.3. Aliasing and copying . . . . .	329
20.4. Sparse matrices . . . . .	330
20.5. Memoization . . . . .	331
20.6. Counting letters . . . . .	332
20.7. Glossary . . . . .	333
20.8. Exercises . . . . .	334
<b>Chapter 21: A Case Study: Indexing your files . . . . .</b>	<b>337</b>
21.1. The Crawler . . . . .	337
21.2. Saving the dictionary to disk . . . . .	340
21.3. The Query Program . . . . .	340
21.4. Compressing the serialized dictionary . . . . .	342
21.5. Glossary . . . . .	343
<b>Chapter 22: Even more OOP . . . . .</b>	<b>344</b>
22.1. MyTime . . . . .	344
22.2. Pure functions . . . . .	344
22.3. Modifiers . . . . .	346
22.4. Converting <code>increment</code> to a method . . . . .	347
22.5. An “Aha!” insight . . . . .	347
22.6. Generalization . . . . .	349
22.7. Another example . . . . .	350
22.8. Operator overloading . . . . .	351

## CONTENTS

22.9. Polymorphism . . . . .	353
22.10. Glossary . . . . .	354
22.11. Exercises . . . . .	355
<b>Chapter 23: Collections of objects . . . . .</b>	<b>357</b>
23.1. Composition . . . . .	357
23.2. Card objects . . . . .	357
23.3. Class attributes and the <code>__str__</code> method . . . . .	358
23.4. Comparing cards . . . . .	360
23.5. Decks . . . . .	362
23.6. Printing the deck . . . . .	362
23.7. Shuffling the deck . . . . .	364
23.8. Removing and dealing cards . . . . .	365
23.9. Glossary . . . . .	366
23.10. Exercises . . . . .	366
<b>Chapter 24: Inheritance . . . . .</b>	<b>367</b>
24.1. Inheritance . . . . .	367
24.2. A hand of cards . . . . .	367
24.3. Dealing cards . . . . .	368
24.4. Printing a Hand . . . . .	369
24.5. The <code>CardGame</code> class . . . . .	370
24.6. <code>OldMaidHand</code> class . . . . .	371
24.7. <code>OldMaidGame</code> class . . . . .	372
24.8. Glossary . . . . .	376
24.9. Exercises . . . . .	377
<b>Chapter 25: Linked lists . . . . .</b>	<b>378</b>
25.1. Embedded references . . . . .	378
25.2. The <code>Node</code> class . . . . .	378
25.3. Lists as collections . . . . .	379
25.4. Lists and recursion . . . . .	380
25.5. Infinite lists . . . . .	381
25.6. The fundamental ambiguity theorem . . . . .	382
25.7. Modifying lists . . . . .	383
25.8. Wrappers and helpers . . . . .	384
25.9. The <code>LinkedList</code> class . . . . .	384
25.10. Invariants . . . . .	386
25.11. Glossary . . . . .	386
25.12. Exercises . . . . .	387
<b>Chapter 26: Stacks . . . . .</b>	<b>388</b>
26.1. Abstract data types . . . . .	388
26.2. The Stack ADT . . . . .	388

## CONTENTS

26.3. Implementing stacks with Python lists . . . . .	389
26.4. Pushing and popping . . . . .	390
26.5. Using a stack to evaluate postfix . . . . .	390
26.6. Parsing . . . . .	391
26.7. Evaluating postfix . . . . .	391
26.8. Clients and providers . . . . .	392
26.9. Glossary . . . . .	393
26.10. Exercises . . . . .	394
<b>Chapter 27: Queues . . . . .</b>	<b>395</b>
27.1. The Queue ADT . . . . .	395
27.2. Linked Queue . . . . .	395
27.3. Performance characteristics . . . . .	397
27.4. Improved Linked Queue . . . . .	397
27.5. Priority queue . . . . .	398
27.6. The Golfer class . . . . .	400
27.7. Glossary . . . . .	401
27.8. Exercises . . . . .	402
<b>Chapter 28: Trees . . . . .</b>	<b>403</b>
28.1. Building trees . . . . .	404
28.2. Traversing trees . . . . .	404
28.3. Expression trees . . . . .	405
28.4. Tree traversal . . . . .	406
28.5. Building an expression tree . . . . .	407
28.6. Handling errors . . . . .	411
28.7. The animal tree . . . . .	412
28.8. Glossary . . . . .	414
28.9. Exercises . . . . .	415
<b>Appendix A: Debugging . . . . .</b>	<b>416</b>
A.1. Syntax errors . . . . .	416
A.2. I can't get my program to run no matter what I do. . . . .	417
A.3. Runtime errors . . . . .	417
A.4. My program does absolutely nothing. . . . .	418
A.5. My program hangs. . . . .	418
A.6. Infinite Loop . . . . .	418
A.7. Infinite Recursion . . . . .	419
A.8. Flow of Execution . . . . .	419
A.9. When I run the program I get an exception. . . . .	420
A.10. I added so many print statements I get inundated with output. . . . .	421
A.11. Semantic errors . . . . .	421
A.12. My program doesn't work. . . . .	422
A.13. I've got a big hairy expression and it doesn't do what I expect. . . . .	423

## CONTENTS

A.14. I've got a function or method that doesn't return what I expect. . . . .	423
A.15. I'm really, really stuck and I need help. . . . .	424
A.16. No, I really need help. . . . .	424
<b>Appendix B: An odds-and-ends Workbook</b> . . . . .	<b>426</b>
B.1. The Five Strands of Proficiency . . . . .	426
B.2. Sending Email . . . . .	427
B.3. Write your own Web Server . . . . .	428
B.4. Using a Database . . . . .	430
<b>Appendix C: Configuring Ubuntu for Python Development</b> . . . . .	<b>433</b>
C.1. Vim . . . . .	433
C.2. \$HOME environment . . . . .	434
C.3. Making a Python script executable and runnable from anywhere . . . . .	435
<b>Appendix D: Customizing and Contributing to the Book</b> . . . . .	<b>436</b>
D.1. Getting the Source . . . . .	436
D.2. Making the HTML Version . . . . .	437
<b>Appendix E: Some Tips, Tricks, and Common Errors</b> . . . . .	<b>438</b>
E.1. Functions . . . . .	438
E.2. Problems with logic and flow of control . . . . .	439
E.3. Local variables . . . . .	441
E.4. Event handler functions . . . . .	442
E.5. String handling . . . . .	442
E.6. Looping and lists . . . . .	444

# **Copyright Notice**

Copyright (C) Peter Wentworth, Jeffrey Elkner, Allen B. Downey and Chris Meyers. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation; with Invariant Sections being Foreword, Preface, and Contributor List, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled “GNU Free Documentation License”.

# Foreword

By David Beazley

As an educator, researcher, and book author, I am delighted to see the completion of this book. Python is a fun and extremely easy-to-use programming language that has steadily gained in popularity over the last few years. Developed over ten years ago by Guido van Rossum, Python's simple syntax and overall feel is largely derived from ABC, a teaching language that was developed in the 1980's. However, Python was also created to solve real problems and it borrows a wide variety of features from programming languages such as C++, Java, Modula-3, and Scheme. Because of this, one of Python's most remarkable features is its broad appeal to professional software developers, scientists, researchers, artists, and educators.

Despite Python's appeal to many different communities, you may still wonder why Python? or why teach programming with Python? Answering these questions is no simple task—especially when popular opinion is on the side of more masochistic alternatives such as C++ and Java. However, I think the most direct answer is that programming in Python is simply a lot of fun and more productive.

When I teach computer science courses, I want to cover important concepts in addition to making the material interesting and engaging to students. Unfortunately, there is a tendency for introductory programming courses to focus far too much attention on mathematical abstraction and for students to become frustrated with annoying problems related to low-level details of syntax, compilation, and the enforcement of seemingly arcane rules. Although such abstraction and formalism is important to professional software engineers and students who plan to continue their study of computer science, taking such an approach in an introductory course mostly succeeds in making computer science boring. When I teach a course, I don't want to have a room of uninspired students. I would much rather see them trying to solve interesting problems by exploring different ideas, taking unconventional approaches, breaking the rules, and learning from their mistakes. In doing so, I don't want to waste half of the semester trying to sort out obscure syntax problems, unintelligible compiler error messages, or the several hundred ways that a program might generate a general protection fault.

One of the reasons why I like Python is that it provides a really nice balance between the practical and the conceptual. Since Python is interpreted, beginners can pick up the language and start doing neat things almost immediately without getting lost in the problems of compilation and linking. Furthermore, Python comes with a large library of modules that can be used to do all sorts of tasks ranging from web-programming to graphics. Having such a practical focus is a great way to engage students and it allows them to complete significant projects. However, Python can also serve as an excellent foundation for introducing important computer science concepts. Since Python fully supports procedures and classes, students can be gradually introduced to topics such as procedural abstraction, data structures, and object-oriented programming — all of which are applicable to later

courses on Java or C++. Python even borrows a number of features from functional programming languages and can be used to introduce concepts that would be covered in more detail in courses on Scheme and Lisp.

In reading Jeffrey's preface, I am struck by his comments that Python allowed him to see a higher level of success and a lower level of frustration and that he was able to move faster with better results. Although these comments refer to his introductory course, I sometimes use Python for these exact same reasons in advanced graduate level computer science courses at the University of Chicago. In these courses, I am constantly faced with the daunting task of covering a lot of difficult course material in a blistering nine week quarter. Although it is certainly possible for me to inflict a lot of pain and suffering by using a language like C++, I have often found this approach to be counterproductive—especially when the course is about a topic unrelated to just programming. I find that using Python allows me to better focus on the actual topic at hand while allowing students to complete substantial class projects.

Although Python is still a young and evolving language, I believe that it has a bright future in education. This book is an important step in that direction. David Beazley University of Chicago  
Author of the Python Essential Reference

# Preface

By Jeffrey Elkner

This book owes its existence to the collaboration made possible by the Internet and the free software movement. Its three authors—a college professor, a high school teacher, and a professional programmer—never met face to face to work on it, but we have been able to collaborate closely, aided by many other folks who have taken the time and energy to send us their feedback.

We think this book is a testament to the benefits and future possibilities of this kind of collaboration, the framework for which has been put in place by Richard Stallman and the Free Software Foundation.

## How and why I came to use Python

In 1999, the College Board’s Advanced Placement (AP) Computer Science exam was given in C++ for the first time. As in many high schools throughout the country, the decision to change languages had a direct impact on the computer science curriculum at Yorktown High School in Arlington, Virginia, where I teach. Up to this point, Pascal was the language of instruction in both our first-year and AP courses. In keeping with past practice of giving students two years of exposure to the same language, we made the decision to switch to C++ in the first year course for the 1997-98 school year so that we would be in step with the College Board’s change for the AP course the following year.

Two years later, I was convinced that C++ was a poor choice to use for introducing students to computer science. While it is certainly a very powerful programming language, it is also an extremely difficult language to learn and teach. I found myself constantly fighting with C++’s difficult syntax and multiple ways of doing things, and I was losing many students unnecessarily as a result. Convinced there had to be a better language choice for our first-year class, I went looking for an alternative to C++.

I needed a language that would run on the machines in our GNU/Linux lab as well as on the Windows and Macintosh platforms most students have at home. I wanted it to be free software, so that students could use it at home regardless of their income. I wanted a language that was used by professional programmers, and one that had an active developer community around it. It had to support both procedural and object-oriented programming. And most importantly, it had to be easy to learn and teach. When I investigated the choices with these goals in mind, Python stood out as the best candidate for the job.

I asked one of Yorktown’s talented students, Matt Ahrens, to give Python a try. In two months he not only learned the language but wrote an application called pyTicket that enabled our staff to report technology problems via the Web. I knew that Matt could not have finished an application of that

scale in so short a time in C++, and this accomplishment, combined with Matt's positive assessment of Python, suggested that Python was the solution I was looking for.

## Finding a textbook

Having decided to use Python in both of my introductory computer science classes the following year, the most pressing problem was the lack of an available textbook.

Free documents came to the rescue. Earlier in the year, Richard Stallman had introduced me to Allen Downey. Both of us had written to Richard expressing an interest in developing free educational materials. Allen had already written a first-year computer science textbook, *How to Think Like a Computer Scientist*. When I read this book, I knew immediately that I wanted to use it in my class. It was the clearest and most helpful computer science text I had seen. It emphasized the processes of thought involved in programming rather than the features of a particular language. Reading it immediately made me a better teacher.

*How to Think Like a Computer Scientist* was not just an excellent book, but it had been released under the GNU public license, which meant it could be used freely and modified to meet the needs of its user. Once I decided to use Python, it occurred to me that I could translate Allen's original Java version of the book into the new language. While I would not have been able to write a textbook on my own, having Allen's book to work from made it possible for me to do so, at the same time demonstrating that the cooperative development model used so well in software could also work for educational materials.

Working on this book for the last two years has been rewarding for both my students and me, and my students played a big part in the process. Since I could make instant changes whenever someone found a spelling error or difficult passage, I encouraged them to look for mistakes in the book by giving them a bonus point each time they made a suggestion that resulted in a change in the text. This had the double benefit of encouraging them to read the text more carefully and of getting the text thoroughly reviewed by its most important critics, students using it to learn computer science.

For the second half of the book on object-oriented programming, I knew that someone with more real programming experience than I had would be needed to do it right. The book sat in an unfinished state for the better part of a year until the open source community once again provided the needed means for its completion.

I received an email from Chris Meyers expressing interest in the book. Chris is a professional programmer who started teaching a programming course last year using Python at Lane Community College in Eugene, Oregon. The prospect of teaching the course had led Chris to the book, and he started helping out with it immediately. By the end of the school year he had created a companion project on our Website at <http://openbookproject.net> called *Python for Fun* and was working with some of my most advanced students as a master teacher, guiding them beyond where I could take them.

# Introducing programming with Python

The process of translating and using How to Think Like a Computer Scientist for the past two years has confirmed Python's suitability for teaching beginning students. Python greatly simplifies programming examples and makes important programming ideas easier to teach.

The first example from the text illustrates this point. It is the traditional `Hello, world` program, which in the Java version of the book looks like this:

```
1 class Hello {  
2  
3     public static void main (String[] args) {  
4         System.out.println ("Hello, world.");  
5     }  
6 }
```

in the Python version it becomes:

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

Even though this is a trivial example, the advantages of Python stand out. Yorktown's Computer Science I course has no prerequisites, so many of the students seeing this example are looking at their first program. Some of them are undoubtedly a little nervous, having heard that computer programming is difficult to learn. The Java version has always forced me to choose between two unsatisfying options: either to explain the class Hello, public static void main, String[] args, statements and risk confusing or intimidating some of the students right at the start, or to tell them, Just don't worry about all of that stuff now; we will talk about it later, and risk the same thing. The educational objectives at this point in the course are to introduce students to the idea of a programming statement and to get them to write their first program, thereby introducing them to the programming environment. The Python program has exactly what is needed to do these things, and nothing more.

Comparing the explanatory text of the program in each version of the book further illustrates what this means to the beginning student. There are seven paragraphs of explanation of `Hello, world!` in the Java version; in the Python version, there are only a few sentences. More importantly, the missing six paragraphs do not deal with the big ideas in computer programming but with the minutia of Java syntax. I found this same thing happening throughout the book. Whole paragraphs simply disappear from the Python version of the text because Python's much clearer syntax renders them unnecessary.

Using a very high-level language like Python allows a teacher to postpone talking about low-level details of the machine until students have the background that they need to better make sense of the details. It thus creates the ability to put first things first pedagogically. One of the best examples of this is the way in which Python handles variables. In Java a variable is a name for a place that

holds a value if it is a built-in type, and a reference to an object if it is not. Explaining this distinction requires a discussion of how the computer stores data. Thus, the idea of a variable is bound up with the hardware of the machine. The powerful and fundamental concept of a variable is already difficult enough for beginning students (in both computer science and algebra). Bytes and addresses do not help the matter. In Python a variable is a name that refers to a thing. This is a far more intuitive concept for beginning students and is much closer to the meaning of variable that they learned in their math courses. I had much less difficulty teaching variables this year than I did in the past, and I spent less time helping students with problems using them.

Another example of how Python aids in the teaching and learning of programming is in its syntax for functions. My students have always had a great deal of difficulty understanding functions. The main problem centers around the difference between a function definition and a function call, and the related distinction between a parameter and an argument. Python comes to the rescue with syntax that is nothing short of beautiful. Function definitions begin with the keyword `def`, so I simply tell my students, When you define a function, begin with `def`, followed by the name of the function that you are defining; when you call a function, simply call (type) out its name. Parameters go with definitions; arguments go with calls. There are no return types, parameter types, or reference and value parameters to get in the way, so I am now able to teach functions in less than half the time that it previously took me, with better comprehension.

Using Python improved the effectiveness of our computer science program for all students. I saw a higher general level of success and a lower level of frustration than I experienced teaching with either C++ or Java. I moved faster with better results. More students left the course with the ability to create meaningful programs and with the positive attitude toward the experience of programming that this engenders.

## Building a community

I have received emails from all over the globe from people using this book to learn or to teach programming. A user community has begun to emerge, and many people have been contributing to the project by sending in materials for the companion Website at <http://openbookproject.net/pybiblio>.

With the continued growth of Python, I expect the growth in the user community to continue and accelerate. The emergence of this user community and the possibility it suggests for similar collaboration among educators have been the most exciting parts of working on this project for me. By working together, we can increase the quality of materials available for our use and save valuable time. I invite you to join our community and look forward to hearing from you. Please write to me at [jeff@elkner.net](mailto:jeff@elkner.net).

Jeffrey Elkner  
Governor's Career and Technical Academy in Arlington  
Arlington, Virginia

# Contributor List

To paraphrase the philosophy of the Free Software Foundation, this book is free like free speech, but not necessarily free like free pizza. It came about because of a collaboration that would not have been possible without the GNU Free Documentation License. So we would like to thank the Free Software Foundation for developing this license and, of course, making it available to us.

We would also like to thank the more than 100 sharp-eyed and thoughtful readers who have sent us suggestions and corrections over the past few years. In the spirit of free software, we decided to express our gratitude in the form of a contributor list. Unfortunately, this list is not complete, but we are doing our best to keep it up to date. It was also getting too large to include everyone who sends in a typo or two. You have our gratitude, and you have the personal satisfaction of making a book you found useful better for you and everyone else who uses it. New additions to the list for the 2nd edition will be those who have made on-going contributions.

If you have a chance to look through the list, you should realize that each person here has spared you and all subsequent readers from the confusion of a technical error or a less-than-transparent explanation, just by sending us a note.

Impossible as it may seem after so many corrections, there may still be errors in this book. If you should stumble across one, we hope you will take a minute to contact us. The email address (for the Python 3 version of the book) is [p.wentworth@ru.ac.za](mailto:p.wentworth@ru.ac.za). Substantial changes made due to your suggestions will add you to the next version of the contributor list (unless you ask to be omitted). Thank you!

## Second Edition

- An email from Mike MacHenry set me straight on tail recursion. He not only pointed out an error in the presentation, but suggested how to correct it.
- It wasn't until 5th Grade student Owen Davies came to me in a Saturday morning Python enrichment class and said he wanted to write the card game, Gin Rummy, in Python that I finally knew what I wanted to use as the case study for the object oriented programming chapters.
- A special thanks to pioneering students in Jeff's Python Programming class at GCTAA during the 2009-2010 school year: Safath Ahmed, Howard Batiste, Louis Elkner-Alfaro, and Rachel Hancock. Your continual and thoughtfull feedback led to changes in most of the chapters of the book. You set the standard for the active and engaged learners that will help make the new Governor's Academy what it is to become. Thanks to you this is truly a student tested text.
- Thanks in a similar vein to the students in Jeff's Computer Science class at the HB-Woodlawn program during the 2007-2008 school year: James Crowley, Joshua Eddy, Eric Larson, Brian McGrail, and Iliana Vazuka.

- Ammar Nabulsi sent in numerous corrections from Chapters 1 and 2.
- Aldric Giacomoni pointed out an error in our definition of the Fibonacci sequence in Chapter 5.
- Roger Sperberg sent in several spelling corrections and pointed out a twisted piece of logic in Chapter 3.
- Adele Goldberg sat down with Jeff at PyCon 2007 and gave him a list of suggestions and corrections from throughout the book.
- Ben Bruno sent in corrections for chapters 4, 5, 6, and 7.
- Carl LaCombe pointed out that we incorrectly used the term commutative in chapter 6 where symmetric was the correct term.
- Alessandro Montanile sent in corrections for errors in the code examples and text in chapters 3, 12, 15, 17, 18, 19, and 20.
- Emanuele Rusconi found errors in chapters 4, 8, and 15.
- Michael Vogt reported an indentation error in an example in chapter 6, and sent in a suggestion for improving the clarity of the shell vs. script section in chapter 1.

## First Edition

- 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.
- Benoit 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 is helping us prepare to 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.

# Chapter 1: The way of the program

(Watch a video based on this chapter [here on YouTube<sup>1</sup>](#).)

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. The Python programming language

The programming language you will be learning is Python. Python is an example of a **high-level language**; other **high-level languages** you might have heard of are C++, PHP, Pascal, C#, and Java.

As you might infer from the name **high-level language**, there are also **low-level languages**, sometimes referred to as machine languages or assembly languages. Loosely speaking, computers can only execute programs written in low-level languages. Thus, programs written in a high-level language have to be translated into something more suitable before they can run.

Almost all programs are written in high-level languages because of their advantages. It is much easier to program in a high-level language so programs take less time to write, they are shorter and easier to read, and they are more likely to be correct. Second, high-level languages are **portable**, meaning that they can run on different kinds of computers with few or no modifications.

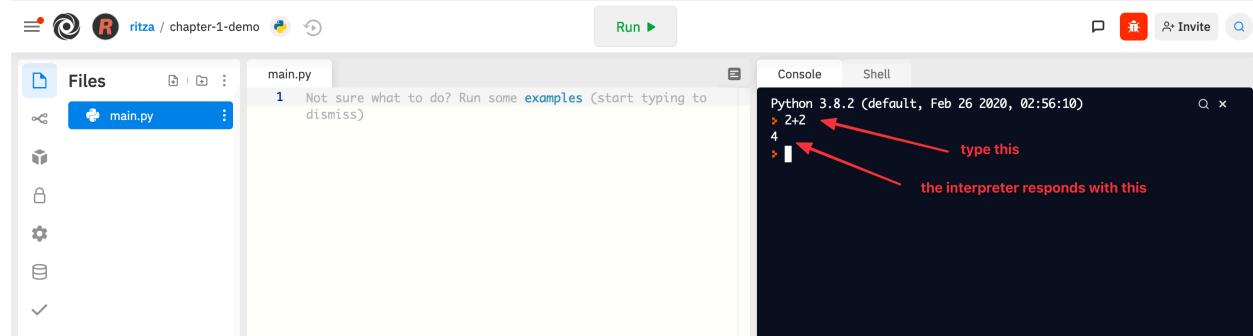
In this edition of the textbook, we use an online programming environment called **Repl.it**. To follow along with the examples and complete the exercises, all you need is a free account - just navigate to <https://replit.com> and complete the sign up process.

Once you have an account, create a new repl and choose Python as the language from the dropdown. You'll see it automatically creates a file called `main.py`. By convention, files that contain Python programs have names that end with `.py`.

---

<sup>1</sup><https://youtu.be/lhtUREG6vAg>

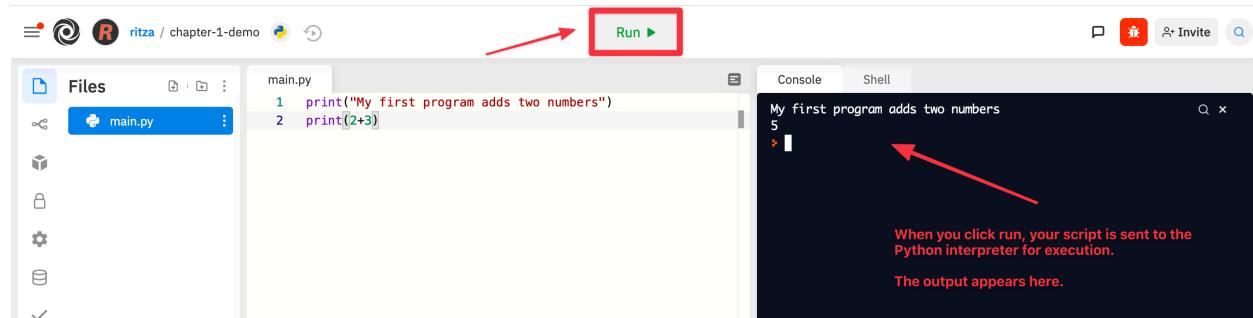
The engine that translates and runs Python is called the **Python Interpreter**: There are two ways to use it: *immediate mode* and *script mode*. In immediate mode, you type Python expressions into the Python Interpreter window, and the interpreter immediately shows the result:



Running code in the interpreter (immediate mode)

The `>>>` or `>` is called the **Python prompt**. The interpreter uses the prompt to indicate that it is ready for instructions. We typed `2 + 2`, and the interpreter evaluated our expression, and replied `4`, and on the next line it gave a new prompt, indicating that it is ready for more input.

Working directly in the interpreter is convenient for testing short bits of code because you get immediate feedback. Think of it as scratch paper used to help you work out problems. Anything longer than a few lines should be put into a script. Scripts have the advantage that they can be saved to disk, printed, and so on. To create a script, you can enter the code into the middle pane, as shown below



Running code from a file (script mode)

```
1 print("My first program adds two numbers")
2 print(2+3)
```

To execute the program, click the **Run** button in Replit. You're now a computer programmer! Let's take a look at some more theory before we start writing more advanced programs.

## 1.2. 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 (strangely enough) compiling a program.

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, or some other device.

### output

- Display data on the screen or send data to a file or other device.

### math

- Perform basic mathematical operations like addition and multiplication.

### conditional execution

- Check for certain conditions and execute the appropriate sequence of statements.

### 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 more or less like these. Thus, we can describe 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 sequences of these basic instructions.

That may be a little vague, but we will come back to this topic later when we talk about **algorithms**.

## 1.3. What is debugging?

Programming is a complex process, and because it is done by human beings, it often leads to errors. Programming errors are called **bugs** and the process of tracking them down and correcting them is called **debugging**. Use of the term bug to describe small engineering difficulties dates back to at least 1889, when Thomas Edison had a bug with his phonograph.

Three kinds of errors can occur in a program: [syntax errors<sup>2</sup>](#), [runtime errors<sup>3</sup>](#), and [semantic errors<sup>4</sup>](#).

<sup>2</sup>[https://en.wikipedia.org/wiki/Syntax\\_error](https://en.wikipedia.org/wiki/Syntax_error)

<sup>3</sup>[https://en.wikipedia.org/wiki/Runtime\\_\(program.lifecycle.phase\)](https://en.wikipedia.org/wiki/Runtime_(program.lifecycle.phase))

<sup>4</sup>[https://en.wikipedia.org/wiki/Logic\\_error](https://en.wikipedia.org/wiki/Logic_error)

It is useful to distinguish between them in order to track them down more quickly.

## 1.4. Syntax errors

Python can only execute a program if the program is syntactically correct; otherwise, the process fails and returns an error message. **Syntax** refers to the structure of a program and the rules about that structure. For example, in English, a sentence must begin with a capital letter and end with a period. This sentence contains a **syntax error**. So does this one

For most readers, a few syntax errors are not a significant problem, which is why we can read the poetry of E. E. Cummings without problems. Python is not so forgiving. If there is a single syntax error anywhere in your program, Python will display an error message and quit, and you will not be able to run your program. During the first few weeks of your programming career, you will probably spend a lot of time tracking down syntax errors. As you gain experience, though, you will make fewer errors and find them faster.

## 1.5. Runtime errors

The second type of error is a runtime error, so called because the error does not appear until you run the program. 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.

## 1.6. Semantic errors

The third type of error is the **semantic error**. If there is a semantic error in your program, it will run successfully, in the sense that the computer will not generate any 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.

The problem is that the program you wrote is not the program you wanted to write. The meaning of the program (its semantics) is wrong. 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.

## 1.7. Experimental 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 what is going wrong, you modify your program and try again. If your hypothesis was correct, then 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 program that does something and make small modifications, debugging them as you go, so that you always have a working program.

For example, Linux is an operating system kernel 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 displaying AAAA and BBBB. This later evolved to Linux (*The Linux Users' Guide Beta Version 1*).

Later chapters will make more suggestions about debugging and other programming practices.

## 1.8. Formal and natural languages

**Natural languages** are the languages that 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 rules about syntax. For example,  $3+3=6$  is a syntactically correct mathematical statement, but  $3=+6\$$  is not.  $H_2O$  is a syntactically correct chemical name, 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, parentheses, commas, and so on. In Python, a statement like `print("Happy New Year for ", 2013)` has 6 tokens: a function name, an open parenthesis (round bracket), a string, a comma, a number, and a close parenthesis.

It is possible to make errors in the way one constructs tokens. One of the problems with  $3=+6\$$  is that  $\$$  is not a legal token in mathematics (at least as far as we know). Similarly,  $2Zz$  is not a legal token in chemistry notation because there is no element with the abbreviation  $Zz$ .

The second type of syntax rule pertains to the **structure** of a statement—that is, the way the tokens are arranged. The statement `3=+6$` is structurally illegal because you can't place a plus sign immediately after an equal sign. Similarly, molecular formulas have to have subscripts after the element name, not before. And in our Python example, if we omitted the comma, or if we changed the two parentheses around to say `print("Happy New Year for ", 2013)` our statement would still have six legal and valid tokens, but the structure is illegal.

When you read a sentence in English or a statement in a formal language, you have to figure out what the structure of the sentence is (although in a natural language you do this subconsciously). This process is called **parsing**.

For example, when you hear the sentence, “The other shoe fell”, you understand that the other shoe is the subject and fell is the verb. Once you have parsed a sentence, you can figure out what it means, or the **semantics** of the sentence. Assuming that you know what a shoe is and what it means to fall, you will understand the general implication of this sentence.

Although formal and natural languages have many features in common—tokens, structure, syntax, and semantics—there are many 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

- Formal languages mean exactly what they say. On the other hand, natural languages are full of idiom and metaphor. If someone says, “The other shoe fell”, there is probably no shoe and nothing falling. You’ll need to find the original joke to understand the idiomatic meaning of the other shoe falling. Yahoo! Answers thinks it knows!

People who grow up speaking a natural language—everyone—often have a hard time adjusting to formal languages. In some ways, 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.

## program

- The meaning of a computer program is unambiguous and literal, and can be understood entirely by analysis of the tokens and structure.

Here are some suggestions for reading programs (and other formal languages). First, remember that formal languages are much more dense than natural languages, so it takes longer to read them. Also, the structure is very important, so it is usually not a good idea 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. Little things like spelling errors and bad punctuation, which you can get away with in natural languages, can make a big difference in a formal language.

## 1.9. The first program

Traditionally, the first program written in a new language is called Hello, World! because all it does is display the words, Hello, World! In Python, the script looks like this: (For scripts, we'll show line numbers to the left of the Python statements.)

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

This is an example of using the **print function**, which doesn't actually print anything on paper. It displays a value on the screen. In this case, the result shown is

```
1 Hello, World!
```

The quotation marks in the program mark the beginning and end of the value; they don't appear in the result.

Some people judge the quality of a programming language by the simplicity of the Hello, World! program. By this standard, Python does about as well as possible.

## 1.10. 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.

A **comment** in a computer program is text that is intended only for the human reader — it is completely ignored by the interpreter.

In Python, the `#` token starts a comment. The rest of the line is ignored. Here is a new version of Hello, World!.

```
1 #-----
2 # This demo program shows off how elegant Python is!
3 # Written by Joe Soap, December 2010.
4 # Anyone may freely copy or modify this program.
5 #-----
6
7 print("Hello, World!")      # Isn't this easy!
```

You'll also notice that we've left a blank line in the program. Blank lines are also ignored by the interpreter, but comments and blank lines can make your programs much easier for humans to parse. Use them liberally!

## 1.11. Glossary

### algorithm

A set of specific steps for solving a category of problems.

### bug

An error in a program.

### 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.

### debugging

The process of finding and removing any of the three kinds of programming errors.

### exception

Another name for a runtime error.

**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.

**high-level language**

A programming language like Python that is designed to be easy for humans to read and write.

**immediate mode**

A style of using Python where we type expressions at the command prompt, and the results are shown immediately. Contrast with script, and see the entry under Python shell.

**interpreter**

The engine that executes your Python scripts or expressions.

**low-level language**

A programming language that is designed to be easy for a computer to execute; also called machine language or assembly language.

**natural language**

Any one of the languages that people speak that evolved naturally.

**object code**

The output of the compiler after it translates the program.

**parse**

To examine a program and analyze the syntactic structure.

**portability**

A property of a program that can run on more than one kind of computer.

**print function**

A function used in a program or script that causes the Python interpreter to display a value on its output device.

**problem solving**

The process of formulating a problem, finding a solution, and expressing the solution.

**program**

a sequence of instructions that specifies to a computer actions and computations to be performed.

**Python shell**

An interactive user interface to the Python interpreter. The user of a Python shell types commands at the prompt (>>>), and presses the return key to send these commands immediately to the interpreter

for processing. The word shell comes from Unix. In the PyScripter used in this RLE version of the book, the Interpreter Window is where we'd do the immediate mode interaction.

**runtime error**

An error that does not occur until the program has started to execute but that prevents the program from continuing.

**script**

A program stored in a file (usually one that will be interpreted).

**semantic error**

An error in a program that makes it do something other than what the programmer intended.

**semantics**

The meaning of a program.

**source code**

A program in a high-level language before being compiled.

**syntax**

The structure of a program.

**syntax error**

An error in a program that makes it impossible to parse — and therefore impossible to interpret.

**token**

One of the basic elements of the syntactic structure of a program, analogous to a word in a natural language.

## 1.12. Exercises

1. Write an English sentence with understandable semantics but incorrect syntax. Write another English sentence which has correct syntax but has semantic errors.
2. Using the Python interpreter, type `1 + 2` and then hit return. Python evaluates this expression, displays the result, and then shows another prompt. `*` is the multiplication operator, and `**` is the exponentiation operator. Experiment by entering different expressions and recording what is displayed by the Python interpreter.
3. Type `1 2` and then hit return. Python tries to evaluate the expression, but it can't because the expression is not syntactically legal. Instead, it shows the error message:

```
1  File "<interactive input>", line 1
2      1 2
3          ^
4  SyntaxError: invalid syntax
```

In many cases, Python indicates where the syntax error occurred, but it is not always right, and it doesn't give you much information about what is wrong.

So, for the most part, the burden is on you to learn the syntax rules.

In this case, Python is complaining because there is no operator between the numbers.

See if you can find a few more examples of things that will produce error messages when you enter them at the Python prompt. Write down what you enter at the prompt and the last line of the error message that Python reports back to you.

4. Type `print("he11o")`. Python executes this, which has the effect of printing the letters h-e-l-l-o. Notice that the quotation marks that you used to enclose the string are not part of the output. Now type "he11o" and describe your result. Make notes of when you see the quotation marks and when you don't.
5. Type cheese without the quotation marks. The output will look something like this:

```
1  Traceback (most recent call last):
2      File "<interactive input>", line 1, in ?
3  NameError: name 'cheese' is not defined
```

This is a run-time error; specifically, it is a `NameError`, and even more specifically, it is an error because the name `cheese` is not defined. If you don't know what that means yet, you will soon.

6. Type `6 + 4 * 9` at the Python prompt and hit enter. Record what happens.

Now create a Python script with the following contents:

```
1  6 + 4 * 9
```

What happens when you run this script? Now change the script contents to:

```
1  print(6 + 4 * 9)
```

and run it again.

What happened this time?

Whenever an expression is typed at the Python prompt, it is evaluated and the result is automatically shown on the line below. (Like on your calculator, if you type this expression you'll get the result 42.)

A script is different, however. Evaluations of expressions are not automatically displayed, so it is necessary to use the **print** function to make the answer show up.

It is hardly ever necessary to use the print function in immediate mode at the command prompt.

# Chapter 2: Variables, expressions and statements

(Watch a video based on this chapter [here on YouTube<sup>5</sup>](#).)

## 2.1. Values and data types

A **value** is one of the fundamental things — like a letter or a number — that a program manipulates. The values we have seen so far are 4 (the result when we added `2 + 2`), and "Hello, World!".

These values are classified into different **classes**, or **data types**: 4 is an *integer*, and "Hello, World!" is a *string*, so-called because it contains a string of letters. You (and the interpreter) can identify strings because they are enclosed in quotation marks.

If you are not sure what class a value falls into, Python has a function called **type** which can tell you.

```
1 >>> type("Hello, World!")
2 <class 'str'>
3 >>> type(17)
4 <class 'int'>
```

Not surprisingly, strings belong to the class **str** and integers belong to the class **int**. Less obviously, numbers with a decimal point belong to a class called **float**, because these numbers are represented in a format called *floating-point*. At this stage, you can treat the words *class* and *type* interchangeably. We'll come back to a deeper understanding of what a class is in later chapters.

```
1 >>> type(3.2)
2 <class 'float'>
```

What about values like "17" and "3.2"? They look like numbers, but they are in quotation marks like strings.

---

<sup>5</sup><https://youtu.be/gIvstR16coI>

```
1 >>> type("17")
2 <class 'str'>
3 >>> type("3.2")
4 <class 'str'>
```

They're strings!

Strings in Python can be enclosed in either single quotes ('') or double quotes (""), or three of each (''' or ''''')

```
1 >>> type('This is a string.')
2 <class 'str'>
3 >>> type("And so is this.")
4 <class 'str'>
5 >>> type("""and this.""")
6 <class 'str'>
7 >>> type('''and even this...'''')
8 <class 'str'>
```

Double quoted strings can contain single quotes inside them, as in "Bruce's beard", and single quoted strings can have double quotes inside them, as in 'The knights who say "Ni!"'.

Strings enclosed with three occurrences of either quote symbol are called triple quoted strings. They can contain either single or double quotes:

```
1 >>> print('''Oh no'', she exclaimed, "Ben's bike is broken!'''')
2 "Oh no", she exclaimed, "Ben's bike is broken!"
3 >>>
```

Triple quoted strings can even span multiple lines:

```
1 >>> message = """This message will
2 ... span several
3 ... lines."""
4 >>> print(message)
5 This message will
6 span several
7 lines.
8 >>>
```

Python doesn't care whether you use single or double quotes or the three-of-a-kind quotes to surround your strings: once it has parsed the text of your program or command, the way it stores the value is identical in all cases, and the surrounding quotes are not part of the value. But when the interpreter wants to display a string, it has to decide which quotes to use to make it look like a string.

```
1 >>> 'This is a string.'  
2 'This is a string.'  
3 >>> """And so is this."""  
4 'And so is this.'
```

So the Python language designers usually chose to surround their strings by single quotes. What do you think would happen if the string already contained single quotes?

When you type a large integer, you might be tempted to use commas between groups of three digits, as in 42,000. This is not a legal integer in Python, but it does mean something else, which is legal:

```
1 >>> 42000  
2 42000  
3 >>> 42,000  
4 (42, 0)
```

Well, that's not what we expected at all! Because of the comma, Python chose to treat this as a pair of values. We'll come back to learn about pairs later. But, for the moment, remember not to put commas or spaces in your integers, no matter how big they are. Also revisit what we said in the previous chapter: formal languages are strict, the notation is concise, and even the smallest change might mean something quite different from what you intended.

## 2.2. Variables

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.

The **assignment statement** gives a value to a variable:

```
1 >>> message = "What's up, Doc?"  
2 >>> n = 17  
3 >>> pi = 3.14159
```

This example makes three assignments. The first assigns the string value "What's up, Doc?" to a variable named `message`. The second gives the integer 17 to `n`, and the third assigns the floating-point number 3.14159 to a variable called `pi`.

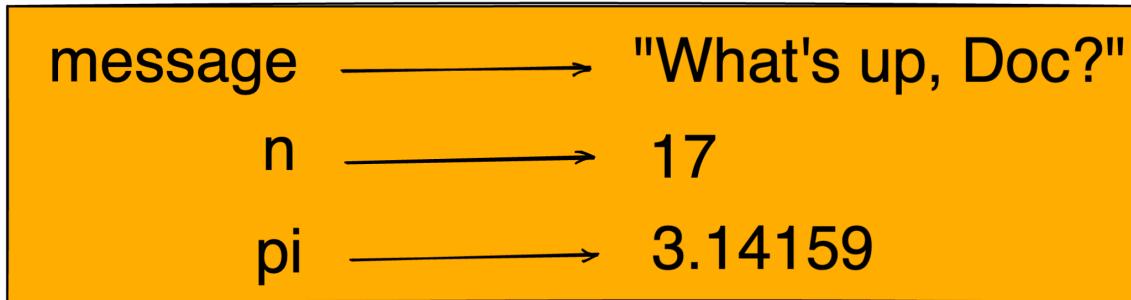
The assignment token, `=`, should not be confused with *equals*, which uses the token `==`. The assignment statement binds a *name*, on the left-hand side of the operator, to a *value*, on the right-hand side. This is why you will get an error if you enter:

```
1 >>> 17 = n
2 File "<interactive input>", line 1
3 SyntaxError: can't assign to literal
```

**Tip:**

When reading or writing code, say to yourself “*n* is assigned 17” or “*n* gets the value 17”. Don’t say “*n* equals 17”.

A common way to represent variables on paper is to write the name with an arrow pointing to the variable’s value. This kind of figure is called a **state snapshot** because it shows what state each of the variables is in at a particular instant in time. (Think of it as the variable’s state of mind). This diagram shows the result of executing the assignment statements:



State Snapshot

If you ask the interpreter to evaluate a variable, it will produce the value that is currently linked to the variable:

```
1 >>> message
2 "What's up, Doc?"
3 >>> n
4 17
5 >>> pi
6 3.14159
```

We use variables in a program to “remember” things, perhaps the current score at the football game. But variables are *variable*. This means they can change over time, just like the scoreboard at a football game. You can assign a value to a variable, and later assign a different value to the same variable. (*This is different from maths. In maths, if you give x the value 3, it cannot change to link to a different value half-way through your calculations!*)

```
1 >>> day = "Thursday"
2 >>> day
3 'Thursday'
4 >>> day = "Friday"
5 >>> day
6 'Friday'
7 >>> day = 21
8 >>> day
9 21
```

You'll notice we changed the value of `day` three times, and on the third assignment we even made it refer to a value that was of a different type.

A great deal of programming is about having the computer remember things, e.g. *The number of missed calls on your phone*, and then arranging to update or change the variable when you miss another call.

## 2.3. Variable names and keywords

**Variable names** can be arbitrarily long. They can contain both letters and digits, but they have to begin with a letter or an underscore. Although it is legal to use uppercase letters, by convention we don't. If you do, remember that case matters. `Bruce` and `bruce` are different variables.

The underscore character (`_`) can appear in a name. It is often used in names with multiple words, such as `my_name` or `price_of_tea_in_china`.

There are some situations in which names beginning with an underscore have special meaning, so a safe rule for beginners is to start all names with a letter.

If you give a variable an illegal name, you get a syntax error:

```
1 >>> 76trombones = "big parade"
2 SyntaxError: invalid syntax
3 >>> more$ = 1000000
4 SyntaxError: invalid syntax
5 >>> class = "Computer Science 101"
6 SyntaxError: invalid syntax
```

`76trombones` is illegal because it does not begin with a letter. `more$` is illegal because it contains an illegal character, the dollar sign. But what's wrong with `class`?

It turns out that `class` is one of the Python **keywords**. Keywords define the language's syntax rules and structure, and they cannot be used as variable names.

Python has thirty-something keywords (and every now and again improvements to Python introduce or eliminate one or two):

and	as	assert	break	class	continue
def	del	elif	else	except	exec
finally	for	from	global	if	import
in	is	lambda	nonlocal	not	or
pass	raise	return	try	while	with
yield	True	False	None		

You might want to keep this list handy. If the interpreter complains about one of your variable names and you don't know why, see if it is on this list.

Programmers generally choose names for their variables that are meaningful to the human readers of the program — they help the programmer document, or remember, what the variable is used for.

### *Caution*

*Beginners sometimes confuse “meaningful to the human readers” with “meaningful to the computer”. So they’ll wrongly think that because they’ve called some variable `average` or `pi`, it will somehow magically calculate an average, or magically know that the variable `pi` should have a value like 3.14159. No! The computer doesn’t understand what you intend the variable to mean.*

*So you’ll find some instructors who deliberately don’t choose meaningful names when they teach beginners — not because we don’t think it is a good habit, but because we’re trying to reinforce the message that you — the programmer — must write the program code to calculate the average, and you must write an assignment statement to give the variable `pi` the value you want it to have.*

## 2.4. Statements

A **statement** is an instruction that the Python interpreter can execute. We have only seen the assignment statement so far. Some other kinds of statements that we’ll see shortly are `while` statements, `for` statements, `if` statements, and `import` statements. (There are other kinds too!)

When you type a statement on the command line, Python executes it. Statements don’t produce any result.

## 2.5. Evaluating expressions

An **expression** is a combination of values, variables, operators, and calls to functions. If you type an expression at the Python prompt, the interpreter **evaluates** it and displays the result:

```

1  >>> 1 + 1
2  2
3  >>> len("hello")
4  5

```

In this example `len` is a built-in Python function that returns the number of characters in a string. We've previously seen the `print` and the `type` functions, so this is our third example of a function!

The *evaluation* of an *expression* produces a value, which is why expressions can appear on the right hand side of assignment statements. A value all by itself is a simple expression, and so is a variable.

```

1  >>> 17
2  17
3  >>> y = 3.14
4  >>> x = len("hello")
5  >>> x
6  5
7  >>> y
8  3.14

```

## 2.6. Operators and operands

**Operators** are special tokens that represent computations like addition, multiplication and division. The values the operator uses are called **operands**.

The following are all legal Python expressions whose meaning is more or less clear:

```
1  20+32    hour-1    hour*60+minute    minute/60    5**2    (5+9)*(15-7)
```

The tokens `+`, `-`, and `*`, and the use of parenthesis for grouping, mean in Python what they mean in mathematics. The asterisk (`*`) is the token for multiplication, and `**` is the token for exponentiation.

```

1  >>> 2 ** 3
2  8
3  >>> 3 ** 2
4  9

```

When a variable name appears in the place of an operand, it is replaced with its value before the operation is performed.

Addition, subtraction, multiplication, and exponentiation all do what you expect.

Example: so let us convert 645 minutes into hours:

```
1 >>> minutes = 645
2 >>> hours = minutes / 60
3 >>> hours
4 10.75
```

Oops! In Python 3, the division operator `/` always yields a floating point result. What we might have wanted to know was how many whole hours there are, and how many minutes remain. Python gives us two different flavors of the division operator. The second, called **floor division** uses the token `//`. Its result is always a whole number — and if it has to adjust the number it always moves it to the left on the number line. So `6 // 4` yields `1`, but `-6 // 4` might surprise you!

```
1 >>> 7 / 4
2 1.75
3 >>> 7 // 4
4 1
5 >>> minutes = 645
6 >>> hours = minutes // 60
7 >>> hours
8 10
```

Take care that you choose the correct flavor of the division operator. If you’re working with expressions where you need floating point values, use the division operator that does the division accurately.

## 2.7. Type converter functions

Here we’ll look at three more Python functions, `int`, `float` and `str`, which will (attempt to) convert their arguments into types `int`, `float` and `str` respectively. We call these **type converter** functions.

The `int` function can take a floating point number or a string, and turn it into an `int`. For floating point numbers, it discards the decimal portion of the number — a process we call truncation towards zero on the number line. Let us see this in action:

```
1 >>> int(3.14)
2 3
3 >>> int(3.9999)          # This doesn't round to the closest int!
4 3
5 >>> int(3.0)
6 3
7 >>> int(-3.999)         # Note that the result is closer to zero
8 -3
9 >>> int(minutes / 60)
10 10
11 >>> int("2345")        # Parse a string to produce an int
12 2345
13 >>> int(17)            # It even works if arg is already an int
14 17
15 >>> int("23 bottles")
```

This last case doesn't look like a number — what do we expect?

```
1 Traceback (most recent call last):
2 File "<interactive input>", line 1, in <module>
3 ValueError: invalid literal for int() with base 10: '23 bottles'
```

The type converter `float` can turn an integer, a float, or a syntactically legal string into a float:

```
1 >>> float(17)
2 17.0
3 >>> float("123.45")
4 123.45
```

The type converter `str` turns its argument into a string:

```
1 >>> str(17)
2 '17'
3 >>> str(123.45)
4 '123.45'
```

## 2.8. Order of operations

When more than one operator appears in an expression, the order of evaluation depends on the **rules of precedence**. Python follows the same precedence rules for its mathematical operators that mathematics does. The acronym PEMDAS is a useful way to remember the order of operations:

1. 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 though it doesn't change the result.
2. Exponentiation has the next highest precedence, so  $2**1+1$  is 3 and not 4, and  $3*1**3$  is 3 and not 27.
3. Multiplication and both Division operators have the same precedence, which is higher than Addition and Subtraction, which also have the same precedence. So  $2*3-1$  yields 5 rather than 4, and  $5-2*2$  is 1, not 6.

Operators with the same precedence are evaluated from left-to-right. In algebra we say they are left-associative. So in the expression  $6-3+2$ , the subtraction happens first, yielding 3. We then add 2 to get the result 5. If the operations had been evaluated from right to left, the result would have been  $6-(3+2)$ , which is 1. (The acronym PEDMAS could mislead you to thinking that division has higher precedence than multiplication, and addition is done ahead of subtraction - don't be misled. Subtraction and addition are at the same precedence, and the left-to-right rule applies.)

Due to some historical quirk, an exception to the left-to-right left-associative rule is the exponentiation operator  $**$ , so a useful hint is to always use parentheses to force exactly the order you want when exponentiation is involved:

```

1 >>> 2 ** 3 ** 2      # The right-most ** operator gets done first!
2 512
3 >>> (2 ** 3) ** 2   # Use parentheses to force the order you want!
4 64

```

The immediate mode command prompt of Python is great for exploring and experimenting with expressions like this.

## 2.9. Operations on strings

In general, you cannot perform mathematical operations on strings, even if the strings look like numbers. The following are illegal (assuming that message has type string):

```

1 >>> message - 1          # Error
2 >>> "Hello" / 123        # Error
3 >>> message * "Hello"   # Error
4 >>> "15" + 2            # Error

```

Interestingly, the  $+$  operator does work with strings, but for strings, the  $+$  operator represents **concatenation**, not addition. Concatenation means joining the two operands by linking them end-to-end. For example:

```

1 fruit = "banana"
2 baked_good = " nut bread"
3 print(fruit + baked_good)

```

The output of this program is banana nut bread. The space before the word nut is part of the string, and is necessary to produce the space between the concatenated strings.

The \* operator also works on strings; it performs repetition. For example, 'Fun'\*3 is 'FunFunFun'. One of the operands has to be a string; the other has to be an integer.

On one hand, this interpretation of + and \* makes sense by analogy with addition and multiplication. Just as  $4*3$  is equivalent to  $4+4+4$ , we expect "Fun"\*3 to be the same as "Fun"+"Fun"+"Fun", 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 and multiplication have that string concatenation and repetition do not?

## 2.10. Input

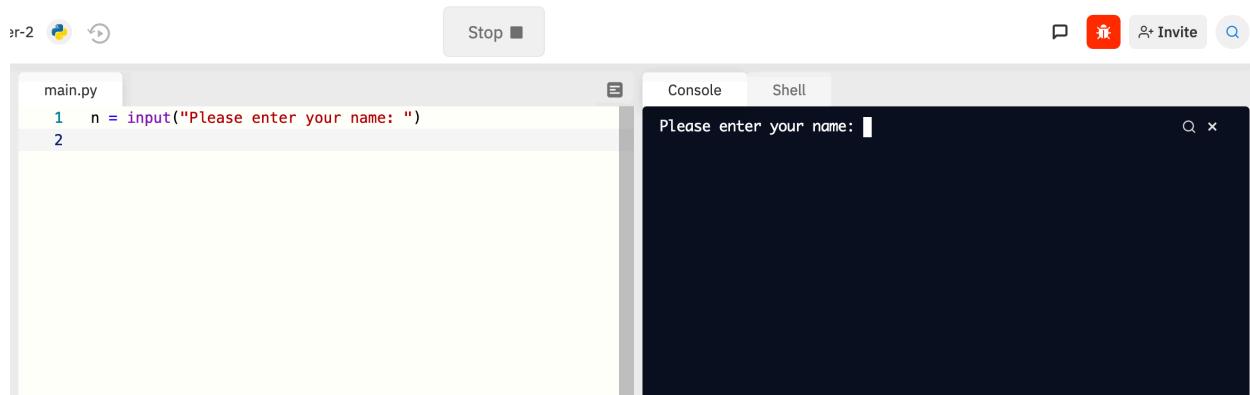
There is a built-in function in Python for getting input from the user:

```

1 n = input("Please enter your name: ")

```

A sample run of this script in Replit would populate your input question in the console to the left like this:



Input Prompt

The user of the program can enter the name and press enter, and when this happens the text that has been entered is returned from the input function, and in this case assigned to the variable n.

Even if you asked the user to enter their age, you would get back a string like "17". It would be your job, as the programmer, to convert that string into a int or a float, using the int or float converter functions we saw earlier.

## 2.11. Composition

So far, we have looked at the elements of a program — variables, expressions, statements, and function calls — 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 into larger chunks.

For example, we know how to get the user to enter some input, we know how to convert the string we get into a float, we know how to write a complex expression, and we know how to print values. Let's put these together in a small four-step program that asks the user to input a value for the radius of a circle, and then computes the area of the circle from the formula

$$A = \pi r^2$$

Area of a circle

Firstly, we'll do the four steps one at a time:

```

1 response = input("What is your radius? ")
2 r = float(response)
3 area = 3.14159 * r**2
4 print("The area is ", area)
```

Now let's compose the first two lines into a single line of code, and compose the second two lines into another line of code.

```

1 r = float( input("What is your radius? " ) )
2 print("The area is ", 3.14159 * r**2)
```

If we really wanted to be tricky, we could write it all in one statement:

```
1 print("The area is ", 3.14159*float(input("What is your radius?"))**2)
```

Such compact code may not be most understandable for humans, but it does illustrate how we can compose bigger chunks from our building blocks.

If you're ever in doubt about whether to compose code or fragment it into smaller steps, try to make it as simple as you can for the human to follow. My choice would be the first case above, with four separate steps.

## 2.12. The modulus operator

The modulus operator works on integers (and integer expressions) and gives the remainder when the first number is divided by the second. In Python, the modulus operator is a percent sign (%). The syntax is the same as for other operators. It has the same precedence as the multiplication operator.

```
1 >>> q = 7 // 3      # This is integer division operator
2 >>> print(q)
3 2
4 >>> r = 7 % 3
5 >>> print(r)
6 1
```

So 7 divided by 3 is 2 with a remainder of 1.

The modulus operator turns out to be surprisingly useful. 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.

It is also extremely useful for doing conversions, say from seconds, to hours, minutes and seconds. So let's write a program to ask the user to enter some seconds, and we'll convert them into hours, minutes, and remaining seconds.

```
1 total_secs = int(input("How many seconds, in total?"))
2 hours = total_secs // 3600
3 secs_still_remaining = total_secs % 3600
4 minutes = secs_still_remaining // 60
5 secs_finally_remaining = secs_still_remaining % 60
6
7 print("Hrs=", hours, " mins=", minutes,
8                  "secs=", secs_finally_remaining)
```

## 2.13. Glossary

**assignment statement**

A statement that assigns a value to a name (variable). To the left of the assignment operator, `=`, is a name. To the right of the assignment token is an expression which is evaluated by the Python interpreter and then assigned to the name. The difference between the left and right hand sides of the assignment statement is often confusing to new programmers. In the following assignment:

```
1 n = n + 1
```

`n` plays a very different role on each side of the `=`. On the right it is a value and makes up part of the expression which will be evaluated by the Python interpreter before assigning it to the name on the left.

### **assignment token**

`=` is Python's assignment token. Do not confuse it with *equals*, which is an operator for comparing values.

### **composition**

The ability to combine simple expressions and statements into compound statements and expressions in order to represent complex computations concisely.

### **concatenate**

To join two strings end-to-end.

### **data type**

A set of values. The type of a value determines how it can be used in expressions. So far, the types you have seen are integers (`int`), floating-point numbers (`float`), and strings (`str`).

### **evaluate**

To simplify an expression by performing the operations in order to yield a single value.

### **expression**

A combination of variables, operators, and values that represents a single result value.

### **float**

A Python data type which stores *floating-point* numbers. Floating-point numbers are stored internally in two parts: a *base* and an *exponent*. When printed in the standard format, they look like decimal numbers. Beware of rounding errors when you use `floats`, and remember that they are only approximate values.

### **floor division**

An operator (denoted by the token `//`) that divides one number by another and yields an integer, or, if the result is not already an integer, it yields the next smallest integer.

### **int**

A Python data type that holds positive and negative whole numbers.

**keyword**

A reserved word that is used by the compiler to parse programs; you cannot use keywords like `if`, `def`, and `while` as variable names.

**modulus operator**

An operator, denoted with a percent sign (%), that works on integers and yields the remainder when one number is divided by another.

**operand**

One of the values on which an operator operates.

**operator**

A special symbol that represents a simple computation like addition, multiplication, or string concatenation.

**rules of precedence**

The set of rules governing the order in which expressions involving multiple operators and operands are evaluated.

**state snapshot**

A graphical representation of a set of variables and the values to which they refer, taken at a particular instant during the program's execution.

**statement**

An instruction that the Python interpreter can execute. So far we have only seen the assignment statement, but we will soon meet the `import` statement and the `for` statement.

**str**

A Python data type that holds a string of characters.

**value**

A number or string (or other things to be named later) that can be stored in a variable or computed in an expression.

**variable**

A name that refers to a value.

**variable name**

A name given to a variable. Variable names in Python consist of a sequence of letters (`a..z`, `A..Z`, and `_`) and digits (`0..9`) that begins with a letter. In best programming practice, variable names should be chosen so that they describe their use in the program, making the program *self documenting*.

## 2.14. Exercises

1. Take the sentence: All work and no play makes Jack a dull boy. Store each word in a separate variable, then print out the sentence on one line using print.
2. Add parenthesis to the expression  $6 * 1 - 2$  to change its value from 4 to -6.
3. Place a comment before a line of code that previously worked, and record what happens when you rerun the program.
4. Start the Python interpreter and enter `bruce + 4` at the prompt. This will give you an error:

```
1  NameError: name 'bruce' is not defined
```

Assign a value to bruce so that `bruce + 4` evaluates to 10.

5. The formula for computing the final amount if one is earning compound interest is given on Wikipedia as

Compounded Interest Formula

$$P' = P \left(1 + \frac{r}{n}\right)^{nt}$$

where:

$P$  is the original principal sum

$P'$  is the new principal sum

$r$  is the nominal annual interest rate

$n$  is the compounding frequency

$t$  is the overall length of time the interest is applied (expressed using the same time units as  $r$ , usually years).

Compounded Interest Formula

Write a Python program that assigns the principal amount of \$10000 to variable  $P$ , assign to  $n$  the value 12, and assign to  $r$  the interest rate of 8%. Then have the program prompt the user for the number of years  $t$  that the money will be compounded for. Calculate and print the final amount after  $t$  years.

6. Evaluate the following numerical expressions in your head, then use the Python interpreter to check your results:

```
1  >>> 5 % 2
2  >>> 9 % 5
3  >>> 15 % 12
4  >>> 12 % 15
5  >>> 6 % 6
6  >>> 0 % 7
7  >>> 7 % 0
```

What happened with the last example? Why? If you were able to correctly anticipate the computer's response in all but the last one, it is time to move on. If not, take time now to

make up examples of your own. Explore the modulus operator until you are confident you understand how it works.

7. You look at the clock and it is exactly 2pm. You set an alarm to go off in 51 hours. At what time does the alarm go off? (*Hint: you could count on your fingers, but this is not what we're after. If you are tempted to count on your fingers, change the 51 to 5100.*)
8. Write a Python program to solve the general version of the above problem. Ask the user for the time now (in hours), and ask for the number of hours to wait. Your program should output what the time will be on the clock when the alarm goes off.

# Chapter 3: Hello, little turtles!

There are many *modules* in Python that provide very powerful features that we can use in our own programs. Some of these can send email, or fetch web pages. The one we'll look at in this chapter allows us to create turtles and get them to draw shapes and patterns.

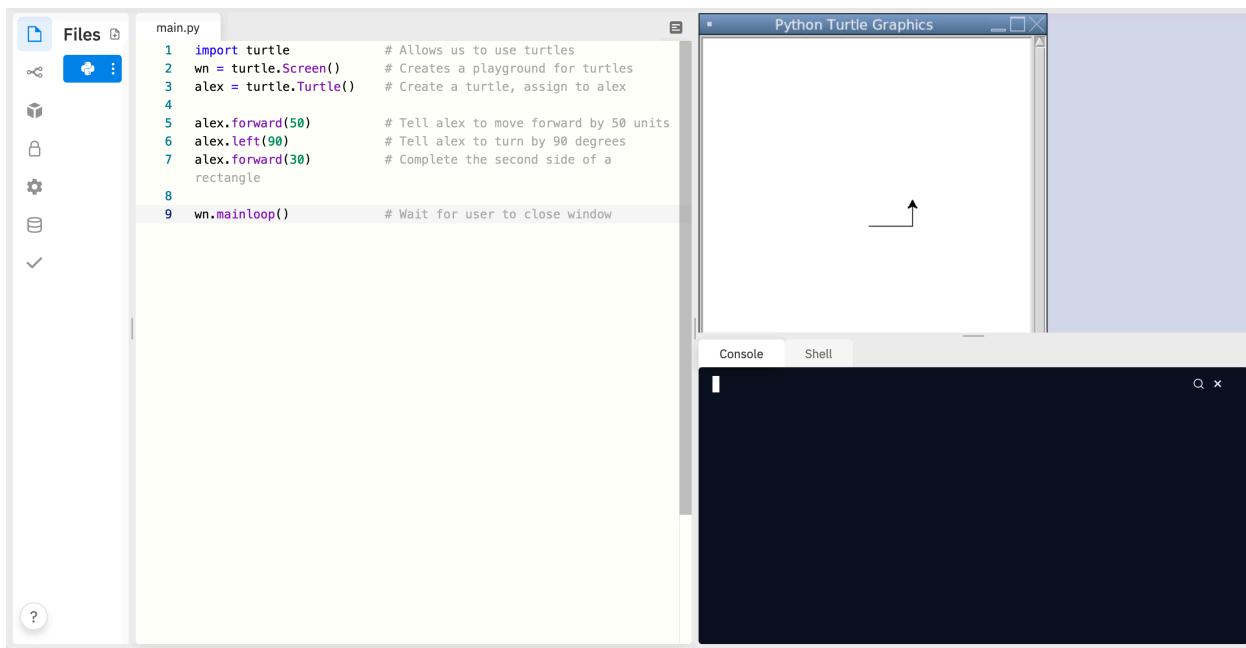
The turtles are fun, but the real purpose of the chapter is to teach ourselves a little more Python, and to develop our theme of *computational thinking*, or *thinking like a computer scientist*. Most of the Python covered here will be explored in more depth later.

## 3.1. Our first turtle program

Let's write a couple of lines of Python program to create a new turtle and start drawing a rectangle. (We'll call the variable that refers to our first turtle `alex`, but we can choose another name if we follow the naming rules from the previous chapter).

```
1 import turtle          # Allows us to use turtles
2 wn = turtle.Screen()   # Creates a playground for turtles
3 alex = turtle.Turtle() # Create a turtle, assign to alex
4
5 alex.forward(50)       # Tell alex to move forward by 50 units
6 alex.left(90)          # Tell alex to turn by 90 degrees
7 alex.forward(30)        # Complete the second side of a rectangle
8
9 wn.mainloop()          # Wait for user to close window
```

When we run this program, a new window pops up:



Turtle Window

Here are a couple of things we'll need to understand about this program.

The first line tells Python to load a module named `turtle`. That module brings us two new types that we can use: the `Turtle` type, and the `Screen` type. The dot notation `turtle.Turtle` means “*The Turtle type that is defined within the turtle module*”. (Remember that Python is case sensitive, so the module name, with a lowercase `t`, is different from the type `Turtle`.)

We then create and open what it calls a screen (we would prefer to call it a window), which we assign to variable `wn`. Every window contains a `canvas`, which is the area inside the window on which we can draw.

In line 3 we create a turtle. The variable `alex` is made to refer to this turtle.

So these first three lines have set things up, we're ready to get our turtle to draw on our canvas.

In lines 5-7, we instruct the **object** `alex` to move, and to turn. We do this by **invoking**, or activating, `alex`'s **methods** — these are the instructions that all turtles know how to respond to.

The last line plays a part too: the `wn` variable refers to the window shown above. When we invoke its `mainloop` method, it enters a state where it waits for events (like keypresses, or mouse movement and clicks). The program will terminate when the user closes the window.

An object can have various methods — things it can do — and it can also have **attributes** — (sometimes called properties). For example, each turtle has a `color` attribute. The method invocation `alex.color("red")` will make `alex` red, and drawing will be red too. (Note the word `color` is spelled the American way!)

The color of the turtle, the width of its pen, the position of the turtle within the window, which way it is facing, and so on are all part of its current **state**. Similarly, the window object has a background

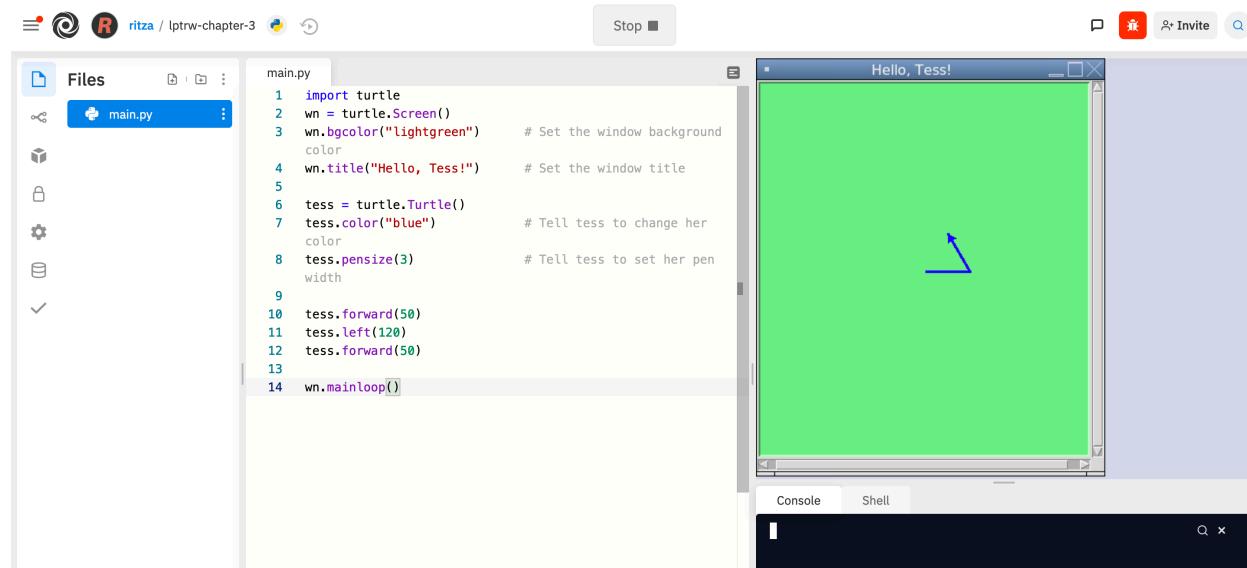
color, and some text in the title bar, and a size and position on the screen. These are all part of the state of the window object.

Quite a number of methods exist that allow us to modify the turtle and the window objects. We'll just show a couple. In this program we've only commented those lines that are different from the previous example (and we've used a different variable name for this turtle):

```

1 import turtle
2 wn = turtle.Screen()
3 wn.bgcolor("lightgreen")      # Set the window background color
4 wn.title("Hello, Tess!")     # Set the window title
5
6 tess = turtle.Turtle()
7 tess.color("blue")           # Tell tess to change her color
8 tess.pensize(3)              # Tell tess to set her pen width
9
10 tess.forward(50)
11 tess.left(120)
12 tess.forward(50)
13
14 wn.mainloop()
```

When we run this program, this new window pops up, and will remain on the screen until we close it.



**Extend this program ...**

1. Modify this program so that before it creates the window, it prompts the user to enter the desired background color. It should store the user's responses in a variable, and modify the color of the window according to the user's wishes. (*Hint: you can find a list of permitted color names at <http://www.tcl.tk/man/tcl8.4/TkCmd/colors.htm>. It includes some quite unusual ones, like “peach puff” and “HotPink”.*)
2. Do similar changes to allow the user, at runtime, to set tess' color.
3. Do the same for the width of tess' pen. *Hint: your dialog with the user will return a string, but tess' pensize method expects its argument to be an int. So you'll need to convert the string to an int before you pass it to pensize.*

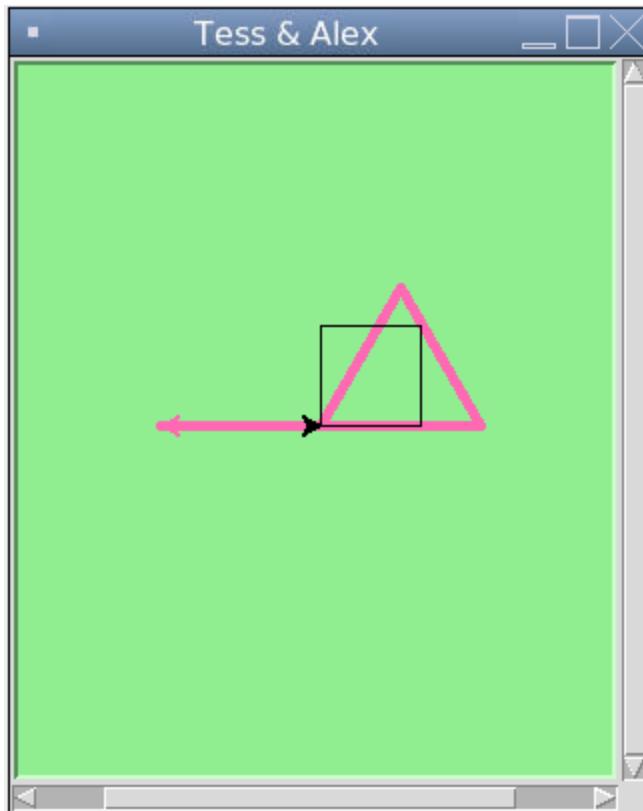
## 3.2. Instances — a herd of turtles

Just like we can have many different integers in a program, we can have many turtles. Each of them is called an **instance**. Each instance has its own attributes and methods — so alex might draw with a thin black pen and be at some position, while tess might be going in her own direction with a fat pink pen.

```
1 import turtle
2 wn = turtle.Screen()           # Set up the window and its attributes
3 wn.bgcolor("lightgreen")
4 wn.title("Tess & Alex")
5
6 tess = turtle.Turtle()         # Create tess and set some attributes
7 tess.color("hotpink")
8 tess.pensize(5)
9
10 alex = turtle.Turtle()        # Create alex
11
12 tess.forward(80)              # Make tess draw equilateral triangle
13 tess.left(120)
14 tess.forward(80)
15 tess.left(120)
16 tess.forward(80)
17 tess.left(120)                # Complete the triangle
18
19 tess.right(180)               # Turn tess around
20 tess.forward(80)               # Move her away from the origin
21
22 alex.forward(50)              # Make alex draw a square
23 alex.left(90)
24 alex.forward(50)
25 alex.left(90)
```

```
26 alex.forward(50)
27 alex.left(90)
28 alex.forward(50)
29 alex.left(90)
30
31 wn.mainloop()
```

Here is what happens when `alex` completes his rectangle, and `tess` completes her triangle:



Alex and Tess

Here are some *How to think like a computer scientist* observations:

- There are 360 degrees in a full circle. If we add up all the turns that a turtle makes, no matter what steps occurred between the turns, we can easily figure out if they add up to some multiple of 360. This should convince us that `alex` is facing in exactly the same direction as he was when he was first created. (Geometry conventions have 0 degrees facing East, and that is the case here too!)
- We could have left out the last turn for `alex`, but that would not have been as satisfying. If we're asked to draw a closed shape like a square or a rectangle, it is a good idea to complete all the turns and to leave the turtle back where it started, facing the same direction as it started in. This makes reasoning about the program and composing chunks of code into bigger programs easier for us humans!

- We did the same with tess: she drew her triangle, and turned through a full 360 degrees. Then we turned her around and moved her aside. Even the blank line 18 is a hint about how the programmer’s mental chunking is working: in big terms, tess’ movements were chunked as “draw the triangle” (lines 12-17) and then “move away from the origin” (lines 19 and 20).
- One of the key uses for comments is to record our mental chunking, and big ideas. They’re not always explicit in the code.
- And, uh-huh, two turtles may not be enough for a herd. But the important idea is that the turtle module gives us a kind of factory that lets us create as many turtles as we need. Each instance has its own state and behaviour.

### 3.3. The for loop

When we drew the square, it was quite tedious. We had to explicitly repeat the steps of moving and turning four times. If we were drawing a hexagon, or an octagon, or a polygon with 42 sides, it would have been worse.

So a basic building block of all programs is to be able to repeat some code, over and over again.

Python’s `for` loop solves this for us. Let’s say we have some friends, and we’d like to send them each an email inviting them to our party. We don’t quite know how to send email yet, so for the moment we’ll just print a message for each friend:

```
1 for f in ["Joe", "Zoe", "Brad", "Angelina", "Zuki", "Thandi", "Paris"]:
2     invite = "Hi " + f + ". Please come to my party on Saturday!"
3     print(invite)
4 # more code can follow here ...
```

When we run this, the output looks like this:

```
1 Hi Joe. Please come to my party on Saturday!
2 Hi Zoe. Please come to my party on Saturday!
3 Hi Brad. Please come to my party on Saturday!
4 Hi Angelina. Please come to my party on Saturday!
5 Hi Zuki. Please come to my party on Saturday!
6 Hi Thandi. Please come to my party on Saturday!
7 Hi Paris. Please come to my party on Saturday!
```

- The variable `f` in the `for` statement at line 1 is called the **loop variable**. We could have chosen any other variable name instead.

Lines 2 and 3 are the **loop body**. The loop body is always indented. The indentation determines exactly what statements are “in the body of the loop”.

- On each *iteration* or *pass* of the loop, first a check is done to see if there are still more items to be processed. If there are none left (this is called the **terminating condition** of the loop), the loop has finished. Program execution continues at the next statement after the loop body, (e.g. in this case the next statement below the comment in line 4).
- If there are items still to be processed, the loop variable is updated to refer to the next item in the list. This means, in this case, that the loop body is executed here 7 times, and each time `f` will refer to a different friend.
- At the end of each execution of the body of the loop, Python returns to the `for` statement, to see if there are more items to be handled, and to assign the next one to `f`.

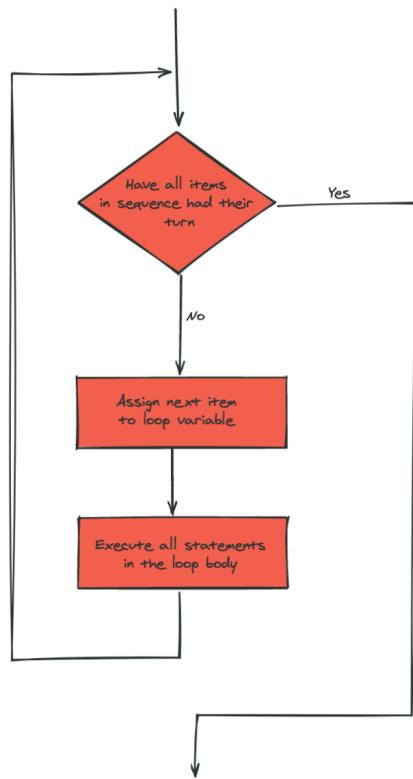
## 3.4. Flow of Execution of the for loop

As a program executes, the interpreter always keeps track of which statement is about to be executed. We call this the **control flow**, or the **flow of execution** of the program. When humans execute programs, they often use their finger to point to each statement in turn. So we could think of control flow as “Python’s moving finger”.

Control flow until now has been strictly top to bottom, one statement at a time. The `for` loop changes this.

### Flowchart of a for loop

Control flow is often easy to visualize and understand if we draw a flowchart. This shows the exact steps and logic of how the `for` statement executes.



For loop flowchart

### 3.5. The loop simplifies our turtle program

To draw a square we'd like to do the same thing four times — move the turtle, and turn. We previously used 8 lines to have alex draw the four sides of a square. This does exactly the same, but using just three lines:

```

1 for i in [0,1,2,3]:
2     alex.forward(50)
3     alex.left(90)
  
```

Some observations:

- While “saving some lines of code” might be convenient, it is not the big deal here. What is much more important is that we've found a “repeating pattern” of statements, and reorganized our program to repeat the pattern. Finding the chunks and somehow getting our programs arranged around those chunks is a vital skill in computational thinking.

- The values `[0, 1, 2, 3]` were provided to make the loop body execute 4 times. We could have used any four values, but these are the conventional ones to use. In fact, they are so popular that Python gives us special built-in range objects:

```

1 for i in range(4):
2     # Executes the body with i = 0, then 1, then 2, then 3
3 for x in range(10):
4     # Sets x to each of ... [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

```

- Computer scientists like to count from 0!
- range can deliver a sequence of values to the loop variable in the for loop. They start at 0, and in these cases do not include the 4 or the 10.
- Our little trick earlier to make sure that alex did the final turn to complete 360 degrees has paid off: if we had not done that, then we would not have been able to use a loop for the fourth side of the square. It would have become a “special case”, different from the other sides. When possible, we’d much prefer to make our code fit a general pattern, rather than have to create a special case.

So to repeat something four times, a good Python programmer would do this:

```

1 for i in range(4):
2     alex.forward(50)
3     alex.left(90)

```

By now you should be able to see how to change our previous program so that tess can also use a for loop to draw her equilateral triangle.

But now, what would happen if we made this change?

```

1 for c in ["yellow", "red", "purple", "blue"]:
2     alex.color(c)
3     alex.forward(50)
4     alex.left(90)

```

A variable can also be assigned a value that is a list. So lists can also be used in more general situations, not only in the for loop. The code above could be rewritten like this:

```
1 # Assign a list to a variable
2 clrs = ["yellow", "red", "purple", "blue"]
3 for c in clrs:
4     alex.color(c)
5     alex.forward(50)
6     alex.left(90)
```

## 3.6. A few more turtle methods and tricks

Turtle methods can use negative angles or distances. So `tess.forward(-100)` will move tess backwards, and `tess.left(-30)` turns her to the right. Additionally, because there are 360 degrees in a circle, turning 30 to the left will get tess facing in the same direction as turning 330 to the right! (The on-screen animation will differ, though — you will be able to tell if tess is turning clockwise or counter-clockwise!)

This suggests that we don't need both a left and a right turn method — we could be minimalists, and just have one method. There is also a *backward* method. (If you are very nerdy, you might enjoy saying `alex.backward(-100)` to move `alex` forward!)

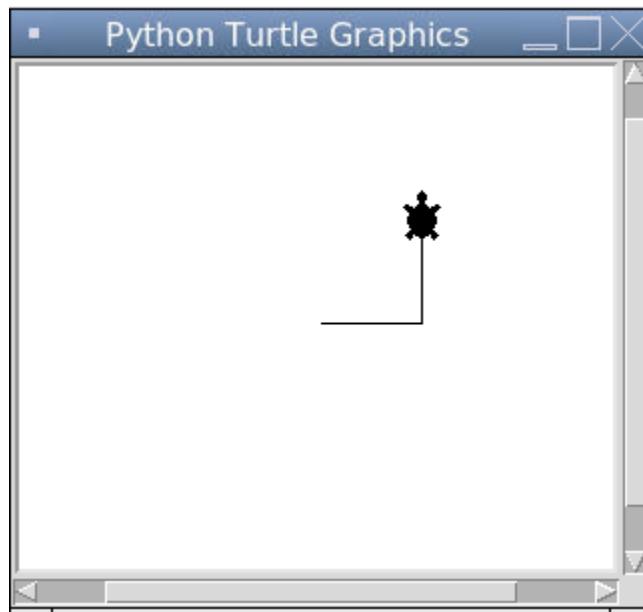
Part of thinking like a scientist is to understand more of the structure and rich relationships in our field. So revising a few basic facts about geometry and number lines, and spotting the relationships between left, right, backward, forward, negative and positive distances or angles values is a good start if we're going to play with turtles.

A turtle's pen can be picked up or put down. This allows us to move a turtle to a different place without drawing a line. The methods are

```
1 alex.penup()
2 alex.forward(100)      # This moves alex, but no line is drawn
3 alex.pendown()
```

Every turtle can have its own shape. The ones available “out of the box” are `arrow`, `blank`, `circle`, `classic`, `square`, `triangle`, `turtle`.

```
1 alex.shape("turtle")
```



Turtle Shape

We can speed up or slow down the turtle's animation speed. (Animation controls how quickly the turtle turns and moves forward). Speed settings can be set between 1 (slowest) to 10 (fastest). But if we set the speed to 0, it has a special meaning — turn off animation and go as fast as possible.

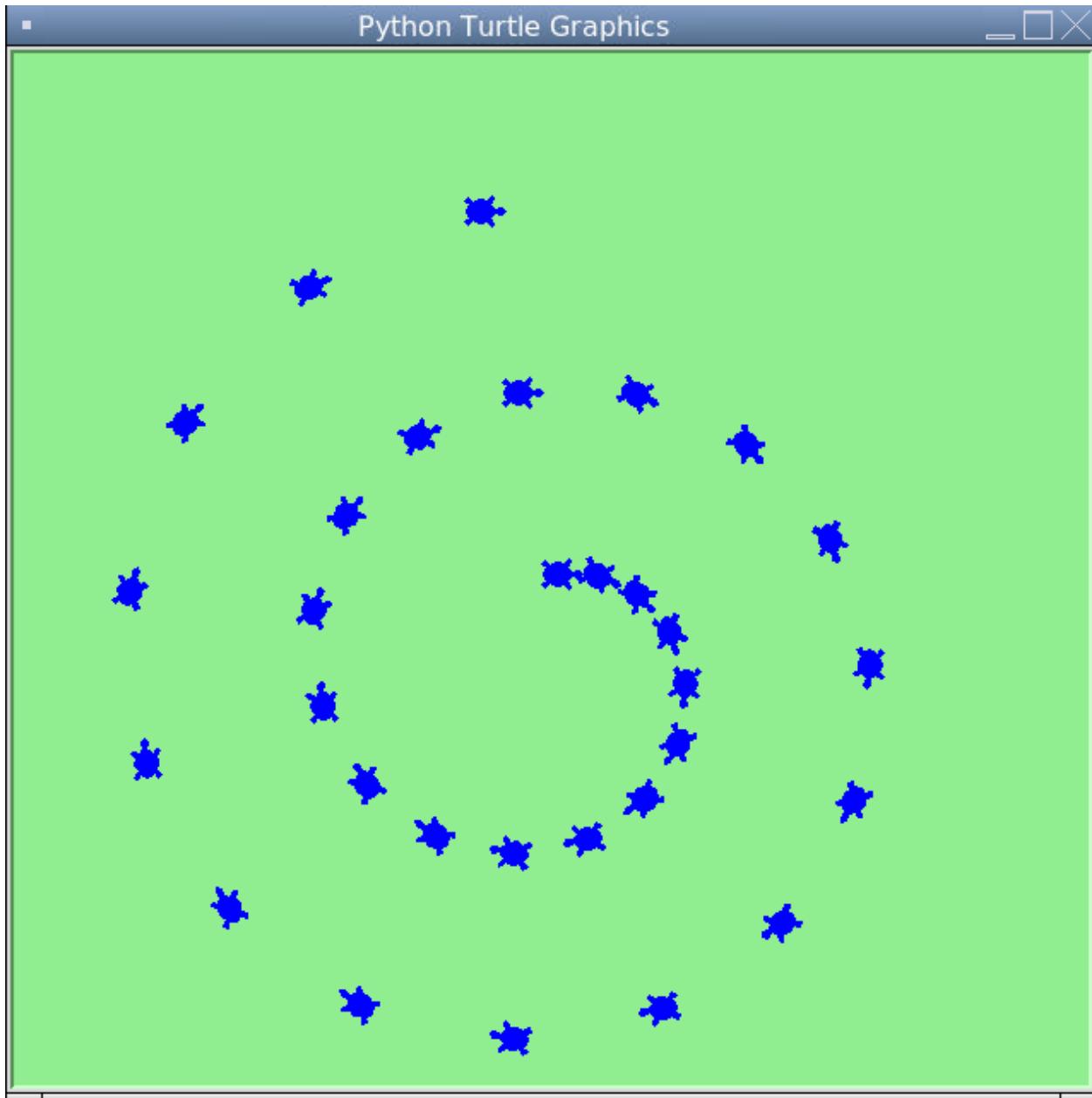
```
1 alex.speed(10)
```

A turtle can “stamp” its footprint onto the canvas, and this will remain after the turtle has moved somewhere else. Stamping works, even when the pen is up.

Let's do an example that shows off some of these new features:

```
1 import turtle
2 wn = turtle.Screen()
3 wn.bgcolor("lightgreen")
4 tess = turtle.Turtle()
5 tess.shape("turtle")
6 tess.color("blue")
7
8 tess.penup()                      # This is new
9 size = 20
10 for i in range(30):
11     tess.stamp()                  # Leave an impression on the canvas
12     size = size + 3              # Increase the size on every iteration
13     tess.forward(size)          # Move tess along
14     tess.right(24)              # ... and turn her
```

```
15  
16 wn.mainloop()
```



Turtle Spiral

Be careful now! How many times was the body of the loop executed? How many turtle images do we see on the screen? All except one of the shapes we see on the screen here are footprints created by stamp. But the program still only has one turtle instance — can you figure out which one here is the real tess? (*Hint: if you're not sure, write a new line of code after the for loop to change tess' color, or to put her pen down and draw a line, or to change her shape, etc.*)

## 3.7. Glossary

**attribute**

Some state or value that belongs to a particular object. For example, tess has a color.

**canvas**

A surface within a window where drawing takes place.

**control flow**

See flow of execution in the next chapter.

**for loop**

A statement in Python for convenient repetition of statements in the body of the loop.

**loop body**

Any number of statements nested inside a loop. The nesting is indicated by the fact that the statements are indented under the for loop statement.

**loop variable**

A variable used as part of a for loop. It is assigned a different value on each iteration of the loop.

**instance**

An object of a certain type, or class. tess and alex are different instances of the class Turtle.

**method**

A function that is attached to an object. Invoking or activating the method causes the object to respond in some way, e.g. forward is the method when we say tess.forward(100).

**invoke**

An object has methods. We use the verb invoke to mean activate the method. Invoking a method is done by putting parentheses after the method name, with some possible arguments. So tess.forward() is an invocation of the forward method.

**module**

A file containing Python definitions and statements intended for use in other Python programs. The contents of a module are made available to the other program by using the import statement.

**object**

A “thing” to which a variable can refer. This could be a screen window, or one of the turtles we have created.

**range**

A built-in function in Python for generating sequences of integers. It is especially useful when we need to write a for loop that executes a fixed number of times.

### terminating condition

A condition that occurs which causes a loop to stop repeating its body. In the for loops we saw in this chapter, the terminating condition has been when there are no more elements to assign to the loop variable.

## 3.8. Exercises

1. Write a program that prints We like Python's turtles! 1000 times.
2. Give three attributes of your cellphone object. Give three methods of your cellphone.
3. Write a program that uses a for loop to print
  - 1 One of the months of the year is January
  - 2 One of the months of the year is February
  - 3 ...
4. Suppose our turtle tess is at heading  $0$  — facing east. We execute the statement `tess.left(3645)`. What does tess do, and what is her final heading?
5. Assume you have the assignment `xs = [12, 10, 32, 3, 66, 17, 42, 99, 20]`
  - a. Write a loop that prints each of the numbers on a new line.
  - b. Write a loop that prints each number and its square on a new line.
  - c. Write a loop that adds all the numbers from the list into a variable called `total`. You should set the `total` variable to have the value  $0$  before you start adding them up, and print the value in `total` after the loop has completed.
  - d. Print the product of all the numbers in the list. (product means all multiplied together)
6. Use for loops to make a turtle draw these regular polygons (regular means all sides the same lengths, all angles the same):
  - An equilateral triangle
  - A square
  - A hexagon (six sides)
  - An octagon (eight sides)
7. A drunk pirate makes a random turn and then takes 100 steps forward, makes another random turn, takes another 100 steps, turns another random amount, etc. A social science student records the angle of each turn before the next 100 steps are taken. Her experimental data is `[160, -43, 270, -97, -43, 200, -940, 17, -86]`. (Positive angles are counter-clockwise.) Use a turtle to draw the path taken by our drunk friend.
8. Enhance your program above to also tell us what the drunk pirate's heading is after he has finished stumbling around. (Assume he begins at heading  $0$ ).
9. If you were going to draw a regular polygon with 18 sides, what angle would you need to turn the turtle at each corner?
10. At the interactive prompt, anticipate what each of the following lines will do, and then record what happens. Score yourself, giving yourself one point for each one you anticipate correctly:

```
1  >>> import turtle  
2  >>> wn = turtle.Screen()  
3  >>> tess = turtle.Turtle()  
4  >>> tess.right(90)  
5  >>> tess.left(3600)  
6  >>> tess.right(-90)  
7  >>> tess.speed(10)  
8  >>> tess.left(3600)  
9  >>> tess.speed(0)  
10 >>> tess.left(3645)  
11 >>> tess.forward(-100)
```

11. Write a program to draw a shape like this:



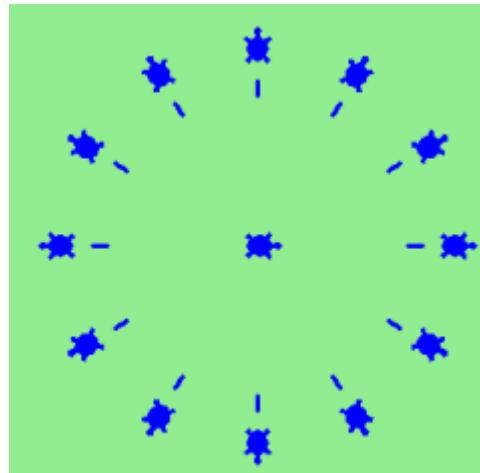
Star

Hints:

- Try this on a piece of paper, moving and turning your cellphone as if it was a turtle. Watch how many complete rotations your cellphone makes before you complete the star. Since each full rotation is 360 degrees, you can figure out the total number of degrees that your phone was rotated through. If you divide that by 5, because there are five points to the star, you'll know how many degrees to turn the turtle at each point.

- You can hide a turtle behind its invisibility cloak if you don't want it shown. It will still draw its lines if its pen is down. The method is invoked as `tess.hideturtle()`. To make the turtle visible again, use `tess.showturtle()`.

12. Write a program to draw a face of a clock that looks something like this:



Clock face

13. Create a turtle, and assign it to a variable. When you ask for its type, what do you get?
14. What is the collective noun for turtles? (Hint: they don't come in *herds*.)
15. What is the collective noun for pythons? Is a python a viper? Is a python venomous?

# Chapter 4: Functions

## 4.1. Functions

In Python, a **function** is a named sequence of statements that belong together. Their primary purpose is to help us organize programs into chunks that match how we think about the problem.

The syntax for a **function definition** is:

```
1 def NAME( PARAMETERS ):
2     STATEMENTS
```

We can make up any names we want for the functions we create, except that we can't use a name that is a Python keyword, and the names must follow the rules for legal identifiers.

There can be any number of statements inside the function, but they have to be indented from the `def`. In the examples in this book, we will use the standard indentation of four spaces. Function definitions are the second of several **compound statements** we will see, all of which have the same pattern:

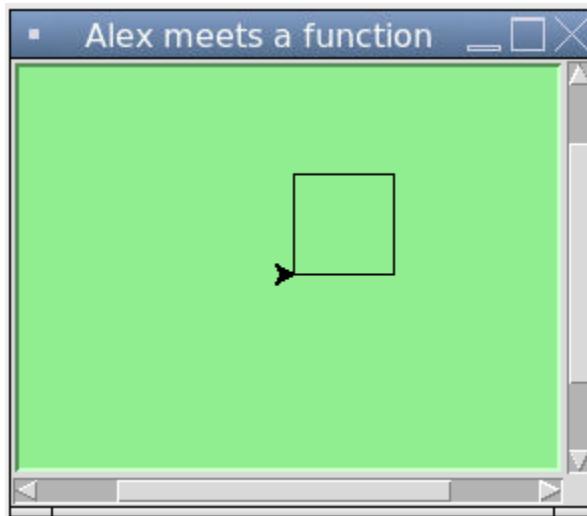
1. A header line which begins with a keyword and ends with a colon.
2. A **body** consisting of one or more Python statements, each indented the same amount — the *Python style guide recommends 4 spaces* — from the header line.

We've already seen the `for` loop which follows this pattern.

So looking again at the function definition, the keyword in the header is `def`, which is followed by the name of the function and some *parameters* enclosed in parentheses. The parameter list may be empty, or it may contain any number of parameters separated from one another by commas. In either case, the parentheses are required. The parameters specifies what information, if any, we have to provide in order to use the new function.

Suppose we're working with turtles, and a common operation we need is to draw squares. "Draw a square" is an abstraction, or a mental chunk, of a number of smaller steps. So let's write a function to capture the pattern of this "building block":

```
1 import turtle  
2  
3 def draw_square(t, sz):  
4     """Make turtle t draw a square of sz."""  
5     for i in range(4):  
6         t.forward(sz)  
7         t.left(90)  
8  
9  
10    wn = turtle.Screen()      # Set up the window and its attributes  
11    wn.bgcolor("lightgreen")  
12    wn.title("Alex meets a function")  
13  
14    alex = turtle.Turtle()    # Create alex  
15    draw_square(alex, 50)      # Call the function to draw the square  
16    wn.mainloop()
```



alex function

This function is named `draw_square`. It has two parameters: one to tell the function which turtle to move around, and the other to tell it the size of the square we want drawn. Make sure you know where the body of the function ends — it depends on the indentation, and the blank lines don't count for this purpose!

### Docstrings for documentation

If the first thing after the function header is a string, it is treated as a **docstring** and gets special treatment in Python and in some programming tools. For example, when we type a built-in function name with an unclosed parenthesis in Repl.it, a tooltip pops up, telling us what arguments the function takes, and it shows us any other text contained in the docstring.

Docstrings are the key way to document our functions in Python and the documentation part is important. Because whoever calls our function shouldn't have to need to know what is going on in the function or how it works; they just need to know what arguments our function takes, what it does, and what the expected result is. Enough to be able to use the function without having to look underneath. This goes back to the concept of abstraction of which we'll talk more about.

Docstrings are usually formed using triple-quoted strings as they allow us to easily expand the docstring later on should we want to write more than a one-liner.

Just to differentiate from comments, a string at the start of a function (a docstring) is retrievable by Python tools at runtime. By contrast, comments are completely eliminated when the program is parsed.

Defining a new function does not make the function run. To do that we need a **function call**. We've already seen how to call some built-in functions like `print`, `range` and `int`. Function calls contain the name of the function being executed followed by a list of values, called *arguments*, which are assigned to the parameters in the function definition. So in the second last line of the program, we call the function, and pass `alex` as the turtle to be manipulated, and `50` as the size of the square we want. While the function is executing, then, the variable `sz` refers to the value `50`, and the variable `t` refers to the same turtle instance that the variable `alex` refers to.

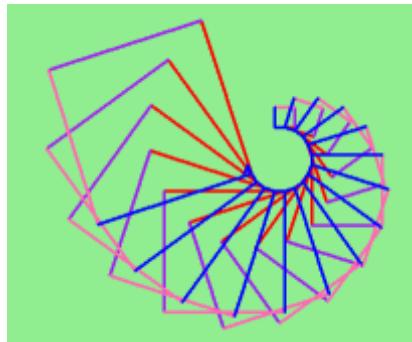
Once we've defined a function, we can call it as often as we like, and its statements will be executed each time we call it. And we could use it to get any of our turtles to draw a square. In the next example, we've changed the `draw_square` function a little, and we get `tess` to draw 15 squares, with some variations.

```
1 import turtle
2
3 def draw_multicolor_square(t, sz):
4     """Make turtle t draw a multi-color square of sz."""
5     for i in ["red", "purple", "hotpink", "blue"]:
6         t.color(i)
7         t.forward(sz)
8         t.left(90)
9
10    wn = turtle.Screen()          # Set up the window and its attributes
11    wn.bgcolor("lightgreen")
12
13    tess = turtle.Turtle()        # Create tess and set some attributes
14    tess.pensize(3)
15
16    size = 20                    # Size of the smallest square
17    for i in range(15):
18        draw_multicolor_square(tess, size)
19        size = size + 10         # Increase the size for next time
```

```

20     tess.forward(10)      # Move tess along a little
21     tess.right(18)       #   and give her some turn
22
23 wn.mainloop()

```



Draw multicolor square

## 4.2. Functions can call other functions

Let's assume now we want a function to draw a rectangle. We need to be able to call the function with different arguments for width and height. And, unlike the case of the square, we cannot repeat the same thing 4 times, because the four sides are not equal.

So we eventually come up with this rather nice code that can draw a rectangle.

```

1 def draw_rectangle(t, w, h):
2     """Get turtle t to draw a rectangle of width w and height h."""
3     for i in range(2):
4         t.forward(w)
5         t.left(90)
6         t.forward(h)
7         t.left(90)

```

The parameter names are deliberately chosen as single letters to ensure they're not misunderstood. In real programs, once we've had more experience, we will insist on better variable names than this. But the point is that the program doesn't "understand" that we're drawing a rectangle, or that the parameters represent the width and the height. Concepts like rectangle, width, and height are the meaning we humans have, not concepts that the program or the computer understands.

*Thinking like a scientist* involves looking for patterns and relationships. In the code above, we've done that to some extent. We did not just draw four sides. Instead, we spotted that we could draw the rectangle as two halves, and used a loop to repeat that pattern twice.

But now we might spot that a square is a special kind of rectangle. We already have a function that draws a rectangle, so we can use that to draw our square.

```

1 def draw_square(tx, sz):      # A new version of draw_square
2     draw_rectangle(tx, sz, sz)

```

There are some points worth noting here:

- Functions can call other functions.
- Rewriting `draw_square` like this captures the relationship that we've spotted between squares and rectangles.
- A caller of this function might say `draw_square(tess, 50)`. The parameters of this function, `tx` and `sz`, are assigned the values of the `tess` object, and the `int` `50` respectively.
- In the body of the function they are just like any other variable.
- When the call is made to `draw_rectangle`, the values in variables `tx` and `sz` are fetched first, then the call happens. So as we enter the top of function `draw_rectangle`, its variable `t` is assigned the `tess` object, and `w` and `h` in that function are both given the value `50`.

So far, it may not be clear why it is worth the trouble to create all of these new functions. Actually, there are a lot of reasons, but this example demonstrates two:

1. Creating a new function gives us an opportunity to name a group of statements. Functions can simplify a program by hiding a complex computation behind a single command. The function (including its name) can capture our mental chunking, or *abstraction*, of the problem.
2. Creating a new function can make a program smaller by eliminating repetitive code.

As we might expect, we have to create a function before we can execute it. In other words, the function definition has to be executed before the function is called.

## 4.3. Flow of execution

In order to ensure that a function is defined before its first use, we have to know the order in which statements are executed, which is called the **flow of execution**. We've already talked about this a little in the previous chapter.

Execution always begins at the first statement of the program. Statements are executed 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 are not executed until the function is called. Although it is not common, we can define one function inside another. In this case, the inner definition isn't executed until the outer function is called.

Function calls are like a detour in the flow of execution. Instead of going to the next statement, the flow jumps to the first line of the called function, executes all the statements there, and then comes back to pick up where it left off.

That sounds simple enough, until we remember that one function can call another. While in the middle of one function, the program might have to execute the statements in another function. But while executing that new function, the program might have to execute yet another function!

Fortunately, Python is adept 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.

What's the moral of this sordid tale? When we read a program, don't read from top to bottom. Instead, follow the flow of execution.

### Watch the flow of execution in action

Repl.it does not have “single-stepping” functionality. For this we would recommend a different IDE like [PyScripter](#)<sup>6</sup>.

In PyScripter, we can watch the flow of execution by “single-stepping” through any program. PyScripter will highlight each line of code just before it is about to be executed.

PyScripter also lets us hover the mouse over any variable in the program, and it will pop up the current value of that variable. So this makes it easy to inspect the “state snapshot” of the program — the current values that are assigned to the program’s variables.

This is a powerful mechanism for building a deep and thorough understanding of what is happening at each step of the way. Learn to use the single-stepping feature well, and be mentally proactive: as you work through the code, challenge yourself before each step: *“What changes will this line make to any variables in the program?”* and *“Where will flow of execution go next?”*

Let us go back and see how this works with the program above that draws 15 multicolor squares. First, we’re going to add one line of magic below the import statement — not strictly necessary, but it will make our lives much simpler, because it prevents stepping into the module containing the turtle code.

```
1 import turtle  
2 __import__("turtle").__traceable__ = False
```

Now we’re ready to begin. Put the mouse cursor on the line of the program where we create the turtle screen, and press the F4 key. This will run the Python program up to, but not including, the line where we have the cursor. Our program will “break” now, and provide a highlight on the next line to be executed, something like this:

---

<sup>6</sup><https://sourceforge.net/projects/pyscripter/>

```

• 1 import turtle
• 2 __import__("turtle").__traceable__ = False
3
4 def draw_multicolor_square(t, sz):
• 5     """Make turtle t draw a multi-color square of sz."""
• 6     for i in ["red", "purple", "hotpink", "blue"]:
• 7         t.color(i)
• 8         t.forward(sz)
• 9         t.left(90)
10
• 11 wn = turtle.Screen()          # Set up the window and its attributes
• 12 wn.bgcolor("lightgreen")
13
• 14 tess = turtle.Turtle()        # Create tess and set some attributes
• 15 tess.pensize(3)
16
• 17 size = 20                    # Size of the smallest square
• 18 for i in range(15):
• 19     draw_multicolor_square(tess, size)
• 20     size = size + 10           # Increase the size for next time
• 21     tess.forward(10)          # Move tess along a little
• 22     tess.right(18)            # ... and give her some extra turn
23
• 24 wn.mainloop()
25

```

## PyScripter Breakpoint

At this point we can press the F7 key (*step into*) repeatedly to single step through the code. Observe as we execute lines 10, 11, 12, ... how the turtle window gets created, how its canvas color is changed, how the title gets changed, how the turtle is created on the canvas, and then how the flow of execution gets into the loop, and from there into the function, and into the function's loop, and then repeatedly through the body of that loop.

While we do this, we can also hover our mouse over some of the variables in the program, and confirm that their values match our conceptual model of what is happening.

After a few loops, when we're about to execute line 20 and we're starting to get bored, we can use the key F8 to “step over” the function we are calling. This executes all the statements in the function, but without having to step through each one. We always have the choice to either “go for the detail”, or to “take the high-level view” and execute the function as a single chunk.

There are some other options, including one that allow us to resume execution without further stepping. Find them under the *Run* menu of PyScripter.

## 4.4. Functions that require arguments

Most functions require arguments: the arguments provide for generalization. For example, if we want to find the absolute value of a number, we have to indicate what the number is. Python has a built-in function for computing the absolute value:

```
1 >>> abs(5)
2 5
3 >>> abs(-5)
4 5
```

In this example, the arguments to the `abs` function are 5 and -5.

Some functions take more than one argument. For example the built-in function `pow` takes two arguments, the base and the exponent. Inside the function, the values that are passed get assigned to variables called **parameters**.

```
1 >>> pow(2, 3)
2 8
3 >>> pow(7, 4)
4 2401
```

Another built-in function that takes more than one argument is `max`.

```
1 >>> max(7, 11)
2 11
3 >>> max(4, 1, 17, 2, 12)
4 17
5 >>> max(3 * 11, 5**3, 512 - 9, 1024**0)
6 503
```

`max` can be passed any number of arguments, separated by commas, and will return the largest value passed. The arguments can be either simple values or expressions. In the last example, 503 is returned, since it is larger than 33, 125, and 1.

## 4.5. Functions that return values

All the functions in the previous section return values. Furthermore, functions like `range`, `int`, `abs` all return values that can be used to build more complex expressions.

So an important difference between these functions and one like `draw_square` is that `draw_square` was not executed because we wanted it to compute a value — on the contrary, we wrote `draw_square` because we wanted it to execute a sequence of steps that caused the turtle to draw.

A function that returns a value is called a **fruitful function** in this book. The opposite of a fruitful function is **void function** — one that is not executed for its resulting value, but is executed because it does something useful. (Languages like Java, C#, C and C++ use the term “void function”, other languages like Pascal call it a **procedure**.) Even though void functions are not executed for their resulting value, Python always wants to return something. So if the programmer doesn’t arrange to return a value, Python will automatically return the value `None`.

How do we write our own fruitful function? In the exercises at the end of chapter 2 we saw the standard formula for compound interest, which we’ll now write as a fruitful function:

$$P' = P \left(1 + \frac{r}{n}\right)^{nt}$$

where:

*P* is the original principal sum

*P'* is the new principal sum

*r* is the nominal annual interest rate

*n* is the compounding frequency

*t* is the overall length of time the interest is applied (expressed using the same time units as *r*, usually years).

### Compound interest

```

1 def final_amt(p, r, n, t):
2     """
3         Apply the compound interest formula to p
4             to produce the final amount.
5     """
6
7     a = p * (1 + r/n) ** (n*t)
8     return a          # This is new, and makes the function fruitful.
9
10 # now that we have the function above, let us call it.
11 toInvest = float(input("How much do you want to invest?"))
12 fnl = final_amt(toInvest, 0.08, 12, 5)
13 print("At the end of the period you'll have", fnl)
```

- The return statement is followed by an expression (`a` in this case). This expression will be evaluated and returned to the caller as the “fruit” of calling this function.
- We prompted the user for the principal amount. The type of `toInvest` is a string, but we need a number before we can work with it. Because it is money, and could have decimal places, we’ve used the `float` type converter function to parse the string and return a float.
- Notice how we entered the arguments for 8% interest, compounded 12 times per year, for 5 years.
- When we run this, we get the output

```
1 At the end of the period you'll have 14898.457083
```

This is a bit messy with all these decimal places, but remember that Python doesn't understand that we're working with money: it just does the calculation to the best of its ability, without rounding. Later we'll see how to format the string that is printed in such a way that it does get nicely rounded to two decimal places before printing.

- The line `toInvest = float(input("How much do you want to invest?"))` also shows yet another example of *composition* — we can call a function like `float`, and its arguments can be the results of other function calls (like `input`) that we've called along the way.

Notice something else very important here. The name of the variable we pass as an argument — `toInvest` — has nothing to do with the name of the parameter — `p`. It is as if `p = toInvest` is executed when `final_amt` is called. It doesn't matter what the value was named in the caller, in `final_amt` its name is `p`.

These short variable names are getting quite tricky, so perhaps we'd prefer one of these versions instead:

```
1 def final_amt_v2(principalAmount, nominalPercentageRate,
2                   numTimesPerYear, years):
3     a = principalAmount * (1 + nominalPercentageRate /
4                           numTimesPerYear) ** (numTimesPerYear*years)
5     return a
6
7 def final_amt_v3(amt, rate, compounded, years):
8     a = amt * (1 + rate/compounded) ** (compounded*years)
9     return a
```

They all do the same thing. Use your judgement to write code that can be best understood by other humans! Short variable names are more economical and sometimes make code easier to read: `E = mc2` would not be nearly so memorable if Einstein had used longer variable names! If you do prefer short names, make sure you also have some comments to enlighten the reader about what the variables are used for.

## 4.6. Variables and parameters are local

When we create a **local variable** inside a function, it only exists inside the function, and we cannot use it outside. For example, consider again this function:

```

1 def final_amt(p, r, n, t):
2     a = p * (1 + r/n) ** (n*t)
3     return a

```

If we try to use `a`, outside the function, we'll get an error:

```

1 >>> a
2 NameError: name 'a' is not defined

```

The variable `a` is local to `final_amt`, and is not visible outside the function.

Additionally, `a` only exists while the function is being executed — we call this its **lifetime**. When the execution of the function terminates, the local variables are destroyed.

Parameters are also local, and act like local variables. For example, the lifetimes of `p`, `r`, `n`, `t` begin when `final_amt` is called, and the lifetime ends when the function completes its execution.

So it is not possible for a function to set some local variable to a value, complete its execution, and then when it is called again next time, recover the local variable. Each call of the function creates new local variables, and their lifetimes expire when the function returns to the caller.

## 4.7. Turtles Revisited

Now that we have fruitful functions, we can focus our attention on reorganizing our code so that it fits more nicely into our mental chunks. This process of rearrangement is called **refactoring** the code.

Two things we're always going to want to do when working with turtles is to create the window for the turtle, and to create one or more turtles. We could write some functions to make these tasks easier in future:

```

1 def make_window(colr, ttle):
2     """
3         Set up the window with the given background color and title.
4         Returns the new window.
5     """
6     w = turtle.Screen()
7     w.bgcolor(colr)
8     w.title(ttle)
9     return w
10
11
12 def make_turtle(colr, sz):

```

```
13      """
14      Set up a turtle with the given color and pensize.
15      Returns the new turtle.
16      """
17      t = turtle.Turtle()
18      t.color(colr)
19      t.pensize(sz)
20      return t
21
22
23 wn = make_window("lightgreen", "Tess and Alex dancing")
24 tess = make_turtle("hotpink", 5)
25 alex = make_turtle("black", 1)
26 dave = make_turtle("yellow", 2)
```

The trick about refactoring code is to anticipate which things we are likely to want to change each time we call the function: these should become the parameters, or changeable parts, of the functions we write.

## 4.8. Glossary

### argument

A value provided to a function when the function is called. This value is assigned to the corresponding parameter in the function. The argument can be the result of an expression which may involve operators, operands and calls to other fruitful functions.

### body

The second part of a compound statement. The body consists of a sequence of statements all indented the same amount from the beginning of the header. The standard amount of indentation used within the Python community is 4 spaces.

### compound statement

A statement that consists of two parts:

1. header - which begins with a keyword determining the statement type, and ends with a colon.
2. body - containing one or more statements indented the same amount from the header.

The syntax of a compound statement looks like this:

```
1 keyword ... :  
2     statement  
3     statement ...
```

### **docstring**

A special string that is attached to a function as its `__doc__` attribute. Tools like Repl.it can use docstrings to provide documentation or hints for the programmer. When we get to modules, classes, and methods, we'll see that docstrings can also be used there.

### **flow of execution**

The order in which statements are executed during a program run.

### **frame**

A box in a stack diagram that represents a function call. It contains the local variables and parameters of the function.

### **function**

A named sequence of statements that performs some useful operation. Functions may or may not take parameters and may or may not produce a result.

### **function call**

A statement that executes a function. It consists of the name of the function followed by a list of arguments enclosed in parentheses.

### **function composition**

Using the output from one function call as the input to another.

### **function definition**

A statement that creates a new function, specifying its name, parameters, and the statements it executes.

### **fruitful function**

A function that returns a value when it is called.

### **header line**

The first part of a compound statement. A header line begins with a keyword and ends with a colon (:)

### **import statement**

A statement which permits functions and variables defined in another Python module to be brought into the environment of another script. To use the features of the turtle, we need to first import the turtle module.

### **lifetime**

Variables and objects have lifetimes — they are created at some point during program execution, and will be destroyed at some time.

### local variable

A variable defined inside a function. A local variable can only be used inside its function. Parameters of a function are also a special kind of local variable.

### parameter

A name used inside a function to refer to the value which was passed to it as an argument.

### refactor

A fancy word to describe reorganizing our program code, usually to make it more understandable. Typically, we have a program that is already working, then we go back to “tidy it up”. It often involves choosing better variable names, or spotting repeated patterns and moving that code into a function.

### stack diagram

A graphical representation of a stack of functions, their variables, and the values to which they refer.

### traceback

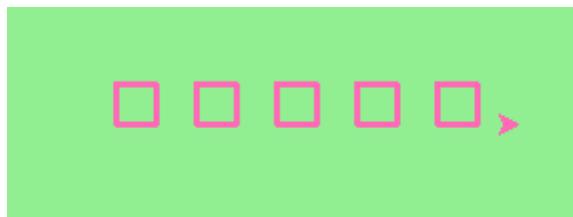
A list of the functions that are executing, printed when a runtime error occurs. A traceback is also commonly referred to as a *stack trace*, since it lists the functions in the order in which they are stored in the [runtime stack](#).<sup>7</sup>

### void function

The opposite of a fruitful function: one that does not return a value. It is executed for the work it does, rather than for the value it returns.

## 4.9. Exercises

1. Write a void (non-fruitful) function to draw a square. Use it in a program to draw the image shown below. Assume each side is 20 units. (*Hint: notice that the turtle has already moved away from the ending point of the last square when the program ends.*)



Five Squares

---

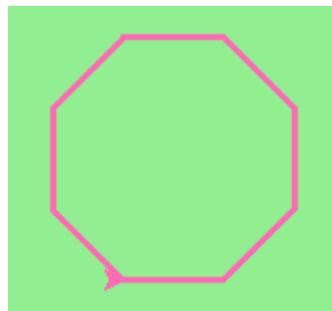
<sup>7</sup>[http://en.wikipedia.org/wiki/Runtime\\_stack](http://en.wikipedia.org/wiki/Runtime_stack)

2. Write a program to draw this. Assume the innermost square is 20 units per side, and each successive square is 20 units bigger, per side, than the one inside it.



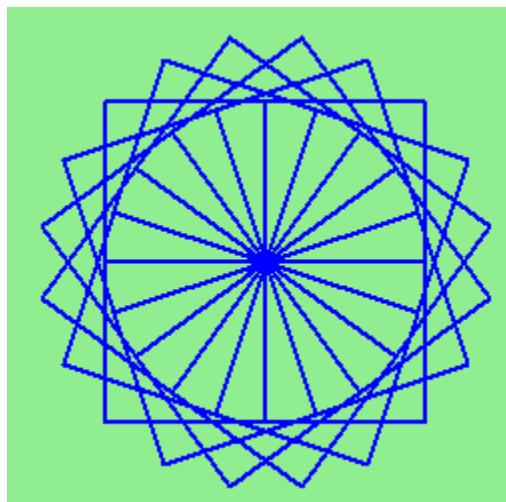
Nested Squares

3. Write a void function `draw_poly(t, n, sz)` which makes a turtle draw a regular polygon. When called with `draw_poly(tess, 8, 50)`, it will draw a shape like this:



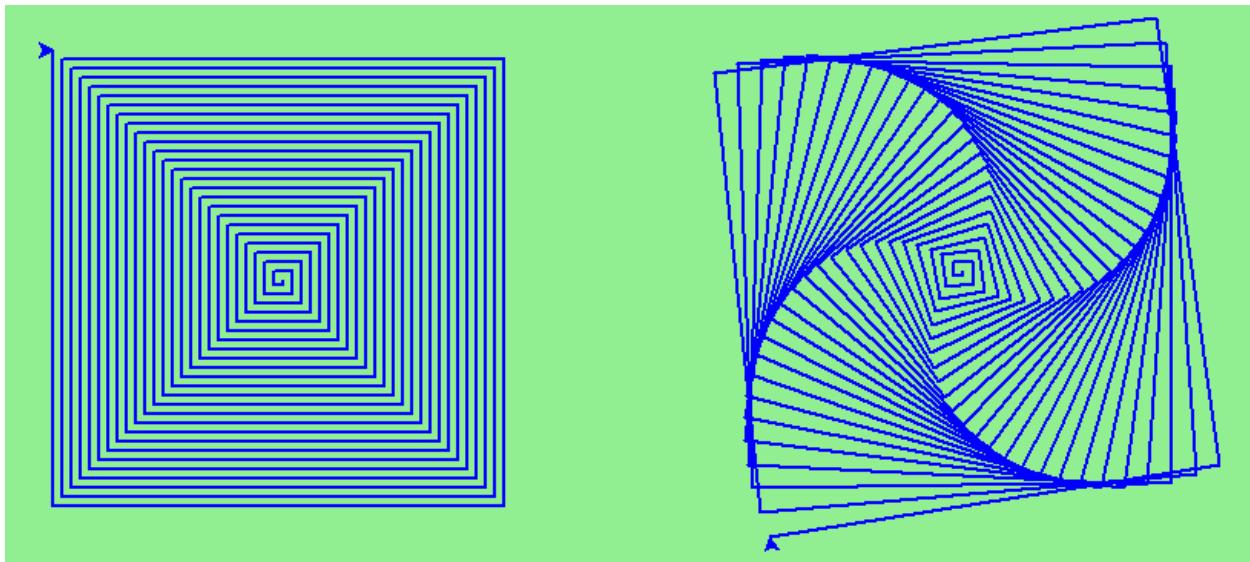
Regular polygon

4. Draw this pretty pattern.



Regular Polygon

5. The two spirals in this picture differ only by the turn angle. Draw both.



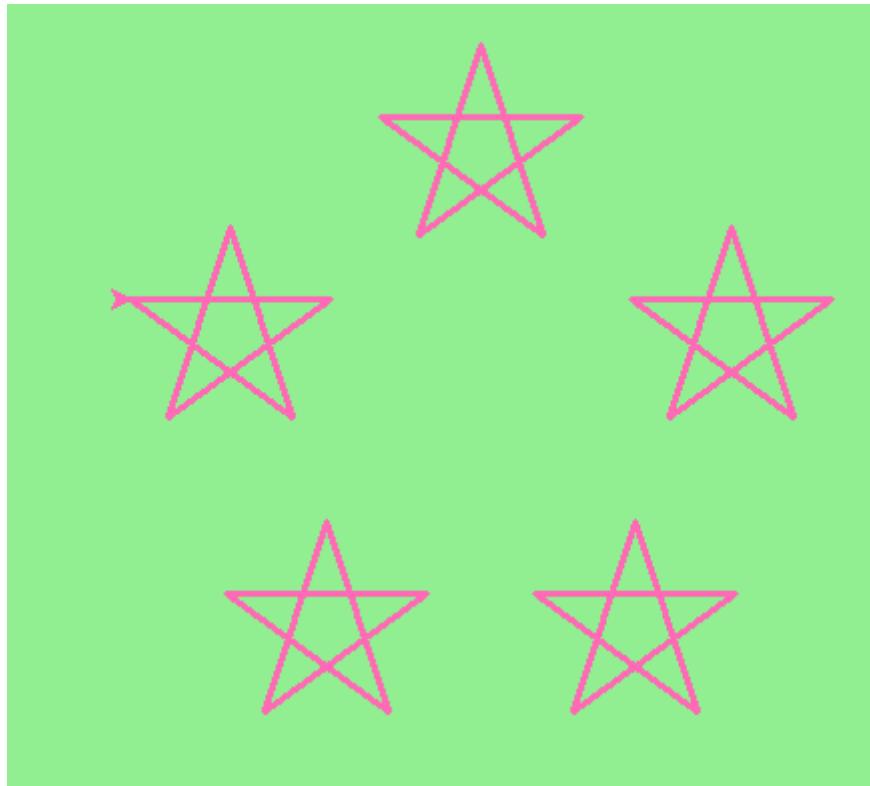
Spirals

6. Write a void function `draw_equitriangle(t, sz)` which calls `draw_poly` from the previous question to have its turtle draw a equilateral triangle.
7. Write a fruitful function `sum_to(n)` that returns the sum of all integer numbers up to and including  $n$ . So `sum_to(10)` would be  $1+2+3\dots+10$  which would return the value 55.
8. Write a function `area_of_circle(r)` which returns the area of a circle of radius  $r$ .
9. Write a void function to draw a star, where the length of each side is 100 units. (*Hint: You should turn the turtle by 144 degrees at each point.*)



Star

10. Extend your program above. Draw five stars, but between each, pick up the pen, move forward by 350 units, turn right by 144, put the pen down, and draw the next star. You'll get something like this:



Five Stars

What would it look like if you didn't pick up the pen?

# Chapter 5: Conditionals

Programs get really interesting when we can test conditions and change the program behaviour depending on the outcome of the tests. That's what this chapter is about.

## 5.1. Boolean values and expressions

A Boolean value is either true or false. It is named after the British mathematician, George Boole, who first formulated Boolean algebra — some rules for reasoning about and combining these values. This is the basis of all modern computer logic.

In Python, the two Boolean values are `True` and `False` (the capitalization must be exactly as shown), and the Python type is `bool`.

```
1 >>> type(True)
2 <class 'bool'>
3 >>> type(true)
4 Traceback (most recent call last):
5   File "<interactive input>", line 1, in <module>
6 NameError: name 'true' is not defined
```

A **Boolean expression** is an expression that evaluates to produce a result which is a Boolean value. For example, the operator `==` tests if two values are equal. It produces (or *yields*) a Boolean value:

```
1 >>> 5 == (3 + 2)    # Is 5 equal to the result of 3 + 2?
2 True
3 >>> 5 == 6
4 False
5 >>> j = "hel"
6 >>> j + "lo" == "hello"
7 True
```

In the first statement, the two operands evaluate to equal values, so the expression evaluates to `True`; in the second statement, 5 is not equal to 6, so we get `False`.

The `==` operator is one of six common **comparison operators** which all produce a `bool` result; here are all six:

```

1 x == y          # Produce True if ... x is equal to y
2 x != y          # ... x is not equal to y
3 x > y           # ... x is greater than y
4 x < y           # ... x is less than y
5 x >= y          # ... x is greater than or equal to y
6 x <= y          # ... x is less than or equal to y

```

Although these operations are probably familiar, 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 comparison operator. Also, there is no such thing as =< or =>.

Like any other types we've seen so far, Boolean values can be assigned to variables, printed, etc.

```

1 >>> age = 18
2 >>> old_enough_to_get_driving_licence = age >= 17
3 >>> print(old_enough_to_get_driving_licence)
4 True
5 >>> type(old_enough_to_get_driving_licence)
6 <class 'bool'>

```

## 5.2. Logical operators

There are three **logical operators**, and, or, and not, that allow us to build more complex Boolean expressions from simpler Boolean expressions. The semantics (meaning) of these operators is similar to their meaning in English. For example,  $x > 0$  and  $x < 10$  produces True only if  $x$  is greater than 0 and at the same time,  $x$  is less than 10.

$n \% 2 == 0$  or  $n \% 3 == 0$  is True if *either* of the conditions is True, that is, if the number  $n$  is divisible by 2 *or* it is divisible by 3. (What do you think happens if  $n$  is divisible by both 2 and by 3 at the same time? Will the expression yield True or False? Try it in your Python interpreter.)

Finally, the not operator negates a Boolean value, so not  $(x > y)$  is True if  $(x > y)$  is False, that is, if  $x$  is less than or equal to  $y$ .

The expression on the left of the or operator is evaluated first: if the result is True, Python does not (and need not) evaluate the expression on the right — this is called *short-circuit evaluation*. Similarly, for the and operator, if the expression on the left yields False, Python does not evaluate the expression on the right.

So there are no unnecessary evaluations.

## 5.3. Truth Tables

A truth table is a small table that allows us to list all the possible inputs, and to give the results for the logical operators. Because the `and` and `or` operators each have two operands, there are only four rows in a truth table that describes the semantics of `and`.

a	b	a and b
False	False	False
False	True	False
True	False	False
True	True	True

In a Truth Table, we sometimes use `T` and `F` as shorthand for the two Boolean values: here is the truth table describing `or`:

a	b	a or b
F	F	F
F	T	T
T	F	T
T	T	T

The third logical operator, `not`, only takes a single operand, so its truth table only has two rows:

a	not a
F	T
T	F

## 5.4. Simplifying Boolean Expressions

A set of rules for simplifying and rearranging expressions is called an algebra. For example, we are all familiar with school *algebra* rules, such as:

1 `n * 0 == 0`

Here we see a different algebra — the Boolean algebra — which provides rules for working with Boolean values.

First, the `and` operator:

```

1 x and False == False
2 False and x == False
3 y and x == x and y
4 x and True == x
5 True and x == x
6 x and x == x

```

Here are some corresponding rules for the or operator:

```

1 x or False == x
2 False or x == x
3 y or x == x or y
4 x or True == True
5 True or x == True
6 x or x == x

```

Two not operators cancel each other:

```
1 not (not x) == x
```

## 5.5. 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:

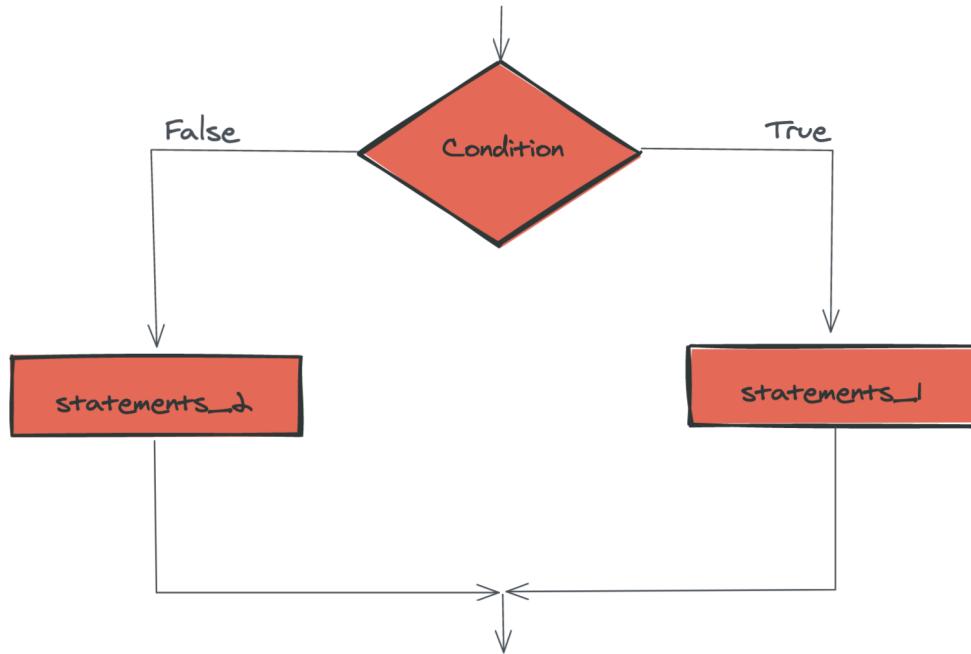
```

1 if x % 2 == 0:
2     print(x, " is even.")
3     print("Did you know that 2 is the only even number that is prime?")
4 else:
5     print(x, " is odd.")
6     print("Did you know that multiplying two odd numbers " +
7           "always gives an odd result?")

```

The Boolean expression after the **if** statement is called the **condition**. If it is true, then all the indented statements get executed. If not, then all the statements indented under the **else** clause get executed.

**Flowchart of an if statement with an else clause**



Flowchart - if else

The syntax for an if statement looks like this:

```

1  if BOOLEAN EXPRESSION:
2      STATEMENTS_1          # Executed if condition evaluates to True
3  else:
4      STATEMENTS_2          # Executed if condition evaluates to False
  
```

As with the function definition from the last chapter and other compound statements like `for`, the `if` statement consists of a header line and a body. The header line begins with the keyword `if` followed by a *Boolean expression* and ends with a colon (`:`).

The indented statements that follow are called a **block**. The first unindented statement marks the end of the block.

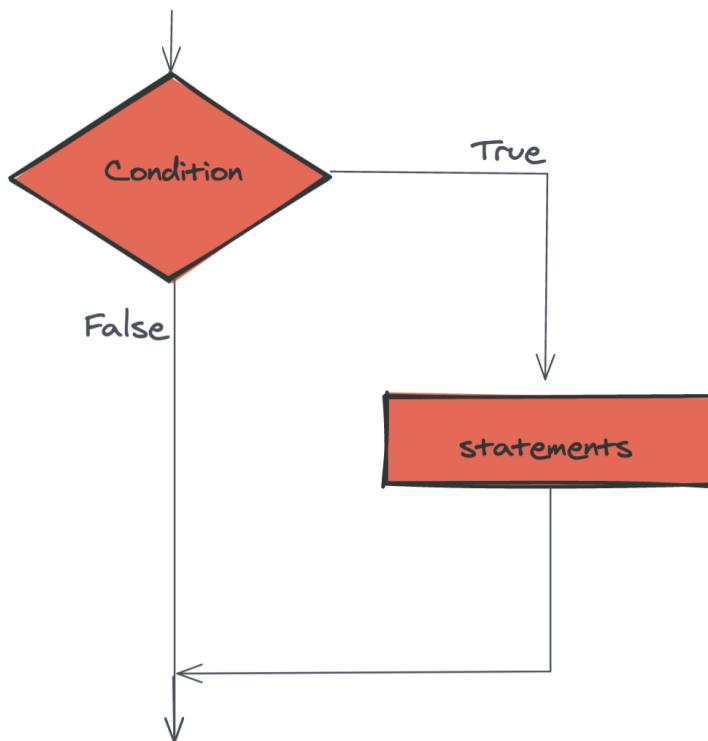
Each of the statements inside the first block of statements are executed in order if the Boolean expression evaluates to `True`. The entire first block of statements is skipped if the Boolean expression evaluates to `False`, and instead all the statements indented under the `else` clause are executed.

There is no limit on the number of statements that can appear under the two clauses of an `if` statement, but there has to be at least one statement in each block. Occasionally, it is useful to have a section with no statements (usually as a place keeper, or scaffolding, for code we haven't written yet). In that case, we can use the `pass` statement, which does nothing except act as a placeholder.

```
1 if True:          # This is always True,  
2     pass           # so this is always executed, but it does nothing  
3 else:  
4     pass
```

## 5.6. Omitting the else clause

Flowchart of an `if` statement with no `else` clause



Flowchart - if only

Another form of the `if` statement is one in which the `else` clause is omitted entirely. In this case, when the condition evaluates to True, the statements are executed, otherwise the flow of execution continues to the statement after the `if`.

```
1 if x < 0:  
2     print("The negative number ", x, " is not valid here.")  
3     x = 42  
4     print("I've decided to use the number 42 instead.")  
5  
6 print("The square root of ", x, "is", math.sqrt(x))
```

In this case, the `print` function that outputs the square root is the one after the `if` — not because we left a blank line, but because of the way the code is indented. Note too that the function call `math.sqrt(x)` will give an error unless we have an `import math` statement, usually placed near the top of our script.

### Python terminology

Python documentation sometimes uses the term **suite** of statements to mean what we have called a *block* here. They mean the same thing, and since most other languages and computer scientists use the word *block*, we'll stick with that.

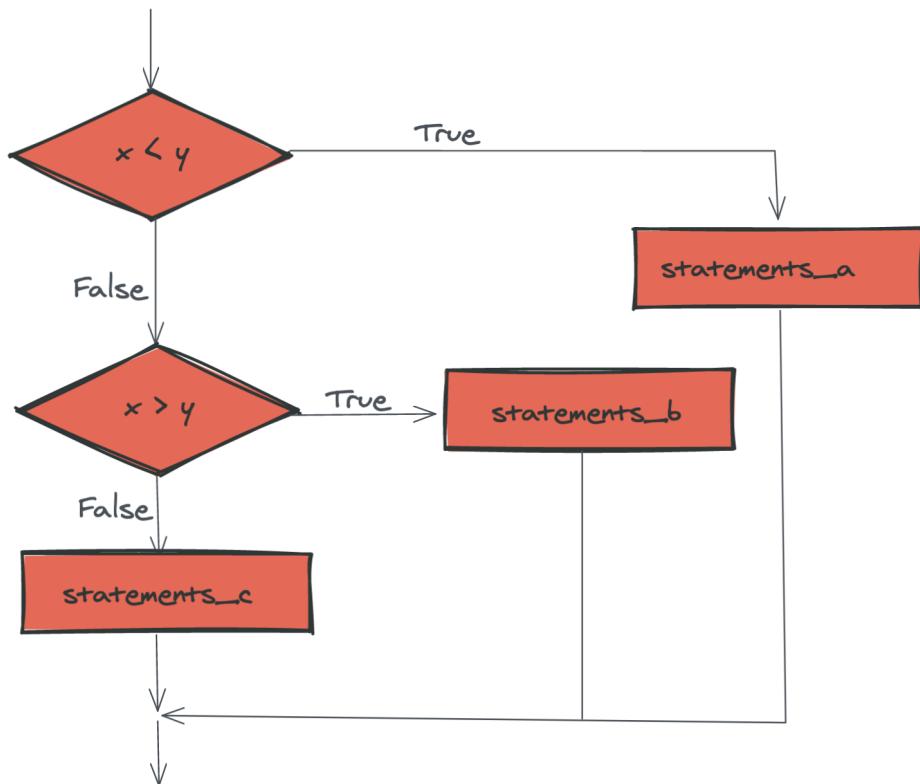
Notice too that `else` is not a statement. The `if` statement has two clauses, one of which is the (optional) `else` clause.

## 5.7. 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**:

```
1 if x < y:  
2     STATEMENTS_A  
3 elif x > y:  
4     STATEMENTS_B  
5 else:  
6     STATEMENTS_C
```

### Flowchart of this chained conditional



Flowchart - chained conditional

`elif` is an abbreviation of `else if`. Again, exactly one branch will be executed. There is no limit of the number of `elif` statements but only a single (and optional) final `else` statement is allowed and it must be the last branch in the statement:

```

1  if choice == "a":
2      function_one()
3  elif choice == "b":
4      function_two()
5  elif choice == "c":
6      function_three()
7 else:
8     print("Invalid choice.")

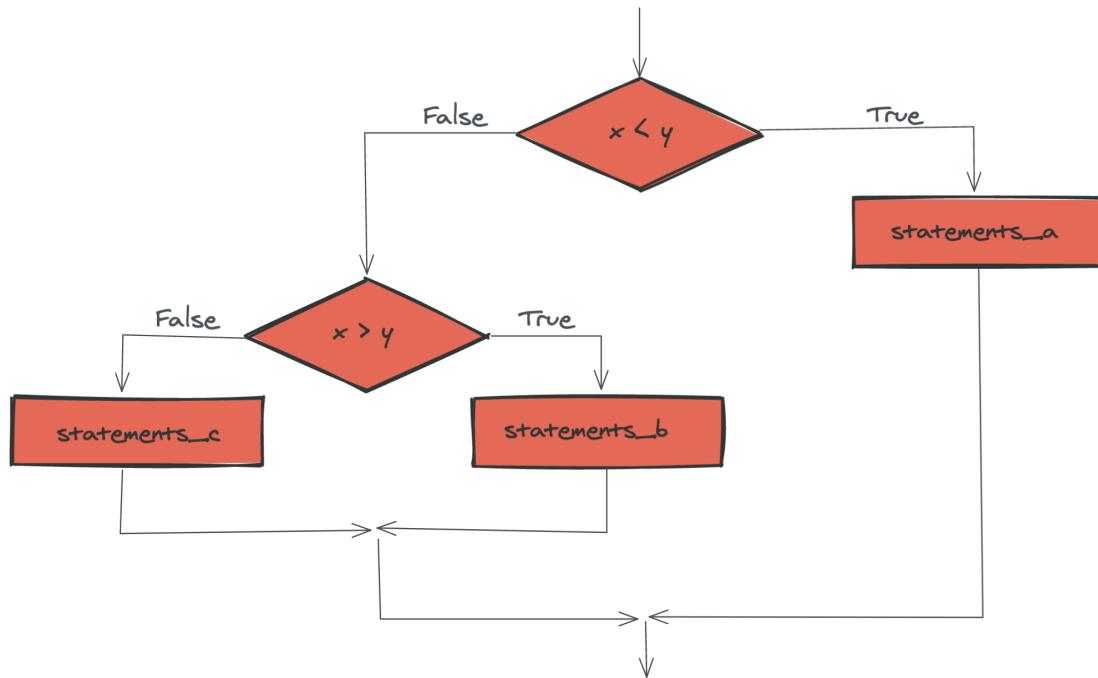
```

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 executes, and the statement ends. Even if more than one condition is true, only the first true branch executes.

## 5.8. Nested conditionals

One conditional can also be **nested** within another. (It is the same theme of composability, again!) We could have written the previous example as follows:

Flowchart of this nested conditional



Flowchart - nested conditional

```

1 if x < y:
2     STATEMENTS_A
3 else:
4     if x > y:
5         STATEMENTS_B
6     else:
7         STATEMENTS_C
  
```

The outer conditional contains two branches. The second branch contains another `if` statement, which has two branches of its own. Those two branches could contain conditional statements as well.

Although the indentation of the statements makes the structure apparent, nested conditionals very quickly become difficult to read. In general, it is a good idea to avoid them when we can.

Logical operators often provide a way to simplify nested conditional statements. For example, we can rewrite the following code using a single conditional:

```
1 if 0 < x:          # Assume x is an int here
2     if x < 10:
3         print("x is a positive single digit.")
```

The `print` function is called only if we make it past both the conditionals, so instead of the above which uses two `if` statements each with a simple condition, we could make a more complex condition using the `and` operator. Now we only need a single `if` statement:

```
1 if 0 < x and x < 10:
2     print("x is a positive single digit.")
```

## 5.9. The return statement

The `return` statement, with or without a value, depending on whether the function is fruitful or void, allows us to terminate the execution of a function before (or when) we reach the end. One reason to use an *early return* is if we detect an error condition:

```
1 def print_square_root(x):
2     if x <= 0:
3         print("Positive numbers only, please.")
4         return
5
6     result = x**0.5
7     print("The square root of", x, "is", result)
```

The function `print_square_root` has a parameter named `x`. The first thing it does is check whether `x` is less than or equal to 0, in which case it displays an error message and then uses `return` to exit the function. The flow of execution immediately returns to the caller, and the remaining lines of the function are not executed.

## 5.10. Logical opposites

Each of the six relational operators has a logical opposite: for example, suppose we can get a driving licence when our age is greater or equal to 17, we can not get the driving licence when we are less than 17.

Notice that the opposite of `>=` is `<`.

operator	logical opposite
<code>==</code>	<code>!=</code>
<code>!=</code>	<code>==</code>
<code>&lt;</code>	<code>&gt;=</code>
<code>&lt;=</code>	<code>&gt;</code>
<code>&gt;</code>	<code>&lt;=</code>
<code>&gt;=</code>	<code>&lt;</code>

Understanding these logical opposites allows us to sometimes get rid of `not` operators. `not` operators are often quite difficult to read in computer code, and our intentions will usually be clearer if we can eliminate them.

For example, if we wrote this Python:

```
1 if not (age >= 17):
2     print("Hey, you're too young to get a driving licence!")
```

it would probably be clearer to use the simplification laws, and to write instead:

```
1 if age < 17:
2     print("Hey, you're too young to get a driving licence!")
```

Two powerful simplification laws (called de Morgan's laws) that are often helpful when dealing with complicated Boolean expressions are:

```
1 not (x and y) == (not x) or (not y)
2 not (x or y) == (not x) and (not y)
```

For example, suppose we can slay the dragon only if our magic lightsabre sword is charged to 90% or higher, and we have 100 or more energy units in our protective shield. We find this fragment of Python code in the game:

```
1 if not ((sword_charge >= 0.90) and (shield_energy >= 100)):
2     print("Your attack has no effect, the dragon fries you to a crisp!")
3 else:
4     print("The dragon crumples in a heap. You rescue the gorgeous princess!")
```

de Morgan's laws together with the logical opposites would let us rework the condition in a (perhaps) easier to understand way like this:

```

1 if (sword_charge < 0.90) or (shield_energy < 100):
2     print("Your attack has no effect, the dragon fries you to a crisp!")
3 else:
4     print("The dragon crumples in a heap. You rescue the gorgeous princess!")

```

We could also get rid of the `not` by swapping around the `then` and `else` parts of the conditional. So here is a third version, also equivalent:

```

1 if (sword_charge >= 0.90) and (shield_energy >= 100):
2     print("The dragon crumples in a heap. You rescue the gorgeous princess!")
3 else:
4     print("Your attack has no effect, the dragon fries you to a crisp!")

```

This version is probably the best of the three, because it very closely matches the initial English statement. Clarity of our code (for other humans), and making it easy to see that the code does what was expected should always be a high priority.

As our programming skills develop we'll find we have more than one way to solve any problem. So good programs are *designed*. We make choices that favour clarity, simplicity, and elegance. The job title *software architect* says a lot about what we do — we are *architects* who engineer our products to balance beauty, functionality, simplicity and clarity in our creations.

### *Tip*

*Once our program works, we should play around a bit trying to polish it up. Write good comments. Think about whether the code would be clearer with different variable names. Could we have done it more elegantly? Should we rather use a function? Can we simplify the conditionals?*

*We think of our code as our creation, our work of art! We make it great.*

## 5.11. Type conversion

We've had a first look at this in an earlier chapter. Seeing it again won't hurt!

Many Python types come with a built-in function that attempts to convert values of another type into its own type. The `int` function, for example, takes any value and converts it to an integer, if possible, or complains otherwise:

```

1 >>> int("32")
2 32
3 >>> int("Hello")
4 ValueError: invalid literal for int() with base 10: 'Hello'

```

`int` can also convert floating-point values to integers, but remember that it truncates the fractional part:

```
1 >>> int(-2.3)
2 -2
3 >>> int(3.99999)
4 3
5 >>> int("42")
6 42
7 >>> int(1.0)
8 1
```

The float function converts integers and strings to floating-point numbers:

```
1 >>> float(32)
2 32.0
3 >>> float("3.14159")
4 3.14159
5 >>> float(1)
6 1.0
```

It may seem odd that Python distinguishes the integer value 1 from the floating-point value 1.0. They may represent the same number, but they belong to different types. The reason is that they are represented differently inside the computer.

The str function converts any argument given to it to type string:

```
1 >>> str(32)
2 '32'
3 >>> str(3.14149)
4 '3.14149'
5 >>> str(True)
6 'True'
7 >>> str(true)
8 Traceback (most recent call last):
9   File "<interactive input>", line 1, in <module>
10  NameError: name 'true' is not defined
```

str will work with any value and convert it into a string. As mentioned earlier, True is a Boolean value; true is just an ordinary variable name, and is not defined here, so we get an error.

## 5.12. A Turtle Bar Chart

The turtle has a lot more power than we've seen so far. The full documentation can be found at <http://docs.python.org/py3k/library/turtle.html>.

Here are a couple of new tricks for our turtles:

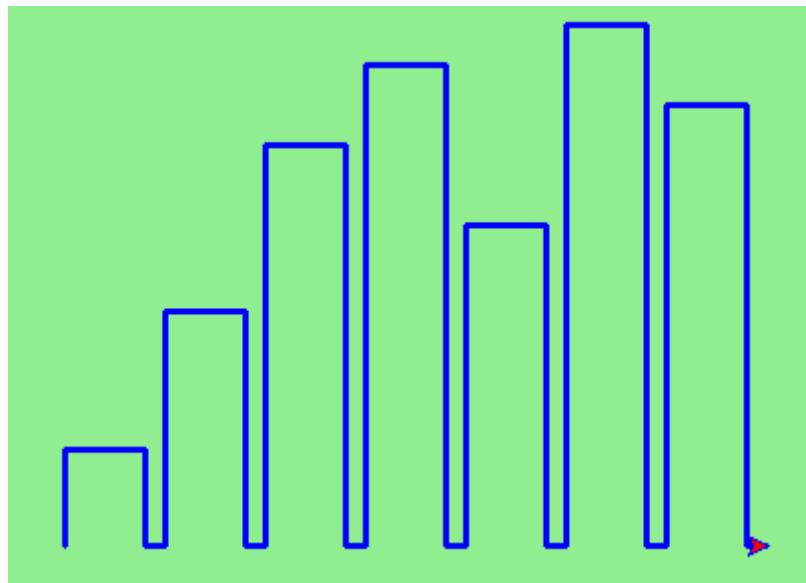
- We can get a turtle to display text on the canvas at the turtle's current position. The method to do that is `alex.write("Hello")`.
- We can fill a shape (circle, semicircle, triangle, etc.) with a color. It is a two-step process. First we call the method `alex.begin_fill()`, then we draw the shape, then we call `alex.end_fill()`.
- We've previously set the color of our turtle — we can now also set its fill color, which need not be the same as the turtle and the pen color. We use `alex.color("blue", "red")` to set the turtle to draw in blue, and fill in red.

Ok, so can we get tess to draw a bar chart? Let us start with some data to be charted,

```
1 xs = [48, 117, 200, 240, 160, 260, 220]
```

Corresponding to each data measurement, we'll draw a simple rectangle of that height, with a fixed width.

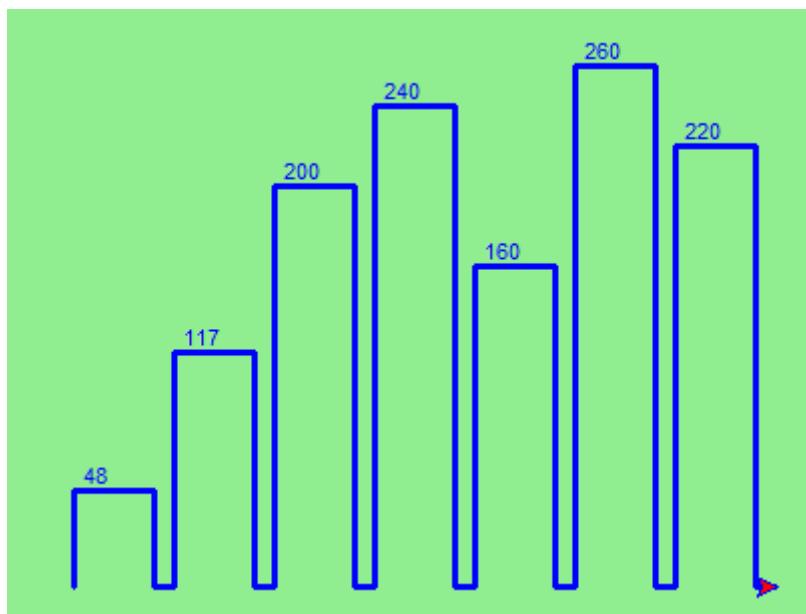
```
1 def draw_bar(t, height):
2     """ Get turtle t to draw one bar, of height. """
3     t.left(90)
4     t.forward(height)      # Draw up the left side
5     t.right(90)
6     t.forward(40)         # Width of bar, along the top
7     t.right(90)
8     t.forward(height)    # And down again!
9     t.left(90)            # Put the turtle facing the way we found it.
10    t.forward(10)         # Leave small gap after each bar
11
12 ...
13 for v in xs:           # Assume xs and tess are ready
14     draw_bar(tess, v)
```



Simple bar chart

Ok, not fantastically impressive, but it is a nice start! The important thing here was the mental chunking, or how we broke the problem into smaller pieces. Our chunk is to draw one bar, and we wrote a function to do that. Then, for the whole chart, we repeatedly called our function.

Next, at the top of each bar, we'll print the value of the data. We'll do this in the body of `draw_bar`, by adding `t.write(' ' + str(height))` as the new third line of the body. We've put a little space in front of the number, and turned the number into a string. Without this extra space we tend to cramp our text awkwardly against the bar to the left. The result looks a lot better now:

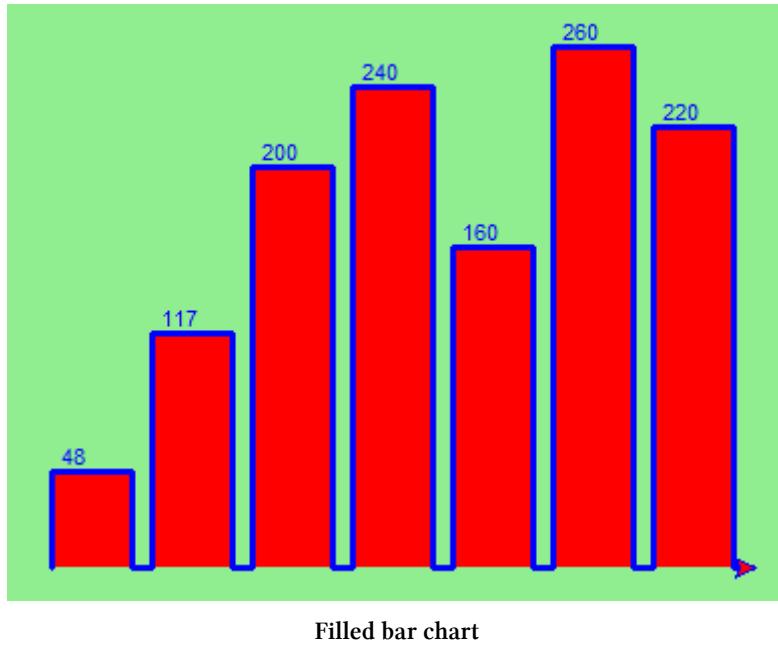


Numbered bar chart

And now we'll add two lines to fill each bar. Our final program now looks like this:

```
1 import turtle
2
3 def draw_bar(t, height):
4     """ Get turtle t to draw one bar, of height. """
5     t.begin_fill()          # Added this line
6     t.left(90)
7     t.forward(height)
8     t.write(" " + str(height))
9     t.right(90)
10    t.forward(40)
11    t.right(90)
12    t.forward(height)
13    t.left(90)
14    t.end_fill()           # Added this line
15    t.forward(10)
16
17 wn = turtle.Screen()      # Set up the window and its attributes
18 wn.bgcolor("lightgreen")
19
20 tess = turtle.Turtle()     # Create tess and set some attributes
21 tess.color("blue", "red")
22 tess.pensize(3)
23
24 xs = [48,117,200,240,160,260,220]
25
26 for a in xs:
27     draw_bar(tess, a)
28
29 wn.mainloop()
```

It produces the following, which is more satisfying:



Filled bar chart

Mmm. Perhaps the bars should not be joined to each other at the bottom. We'll need to pick up the pen while making the gap between the bars. We'll leave that (and a few more tweaks) as exercises for you!

## 5.13. Glossary

### block

A group of consecutive statements with the same indentation.

### body

The block of statements in a compound statement that follows the header.

### Boolean algebra

Some rules for rearranging and reasoning about Boolean expressions.

### Boolean expression

An expression that is either true or false.

### Boolean value

There are exactly two Boolean values: `True` and `False`. Boolean values result when a Boolean expression is evaluated by the Python interpreter. They have type `bool`.

### branch

One of the possible paths of the flow of execution determined by conditional execution.

**chained conditional**

A conditional branch with more than two possible flows of execution. In Python chained conditionals are written with `if ... elif ... else` statements.

**comparison operator**

One of the six operators that compares two values: `==`, `!=`, `>`, `<`, `>=`, and `<=`.

**condition**

The Boolean expression in a conditional statement that determines which branch is executed.

**conditional statement**

A statement that controls the flow of execution depending on some condition. In Python the keywords `if`, `elif`, and `else` are used for conditional statements.

**logical operator**

One of the operators that combines Boolean expressions: `and`, `or`, and `not`.

**nesting**

One program structure within another, such as a conditional statement inside a branch of another conditional statement.

**prompt**

A visual cue that tells the user that the system is ready to accept input data.

**truth table**

A concise table of Boolean values that can describe the semantics of an operator.

**type conversion**

An explicit function call that takes a value of one type and computes a corresponding value of another type.

**wrapping code in a function**

The process of adding a function header and parameters to a sequence of program statements is often referred to as “wrapping the code in a function”. This process is very useful whenever the program statements in question are going to be used multiple times. It is even more useful when it allows the programmer to express their mental chunking, and how they’ve broken a complex problem into pieces.

## 5.14. Exercises

1. Assume the days of the week are numbered `0, 1, 2, 3, 4, 5, 6` from Sunday to Saturday. Write a function which is given the day number, and it returns the day name (a string).

2. You go on a wonderful holiday (perhaps to jail, if you don't like happy exercises) leaving on day number 3 (a Wednesday). You return home after 137 sleeps. Write a general version of the program which asks for the starting day number, and the length of your stay, and it will tell you the name of day of the week you will return on.
3. Give the logical opposites of these conditions

```

1   a > b
2   a >= b
3   a >= 18 and day == 3
4   a >= 18 and day != 3

```

4. What do these expressions evaluate to?

```

1   3 == 3
2   3 != 3
3   3 >= 4
4   not (3 < 4)

```

5. Complete this truth table:

p	q	r	(not (p and q)) or r
F	F	F	?
F	F	T	?
F	T	F	?
F	T	T	?
T	F	F	?
T	F	T	?
T	T	F	?
T	T	T	?

6. Write a function which is given an exam mark, and it returns a string — the grade for that mark — according to this scheme:

Mark	Grade
$\geq 75$	First
[70-75)	Upper Second
[60-70)	Second
[50-60)	Third
[45-50)	F1 Supp
[40-45)	F2
$< 40$	F3

The square and round brackets denote closed and open intervals. A closed interval includes the number, and open interval excludes it. So 39.99999 gets grade F3, but 40 gets grade F2. Assume

```

1   xs = [83, 75, 74.9, 70, 69.9, 65, 60, 59.9, 55, 50,
2           49.9, 45, 44.9, 40, 39.9, 2, 0]

```

Test your function by printing the mark and the grade for all the elements in this list.

7. Modify the turtle bar chart program so that the pen is up for the small gaps between each bar.
8. Modify the turtle bar chart program so that the bar for any value of 200 or more is filled with red, values between [100 and 200) are filled with yellow, and bars representing values less than 100 are filled with green.
9. In the turtle bar chart program, what do you expect to happen if one or more of the data values in the list is negative? Try it out. Change the program so that when it prints the text value for the negative bars, it puts the text below the bottom of the bar.
10. Write a function `find_hypot` which, given the length of two sides of a right-angled triangle, returns the length of the hypotenuse. (*Hint:  $x ** 0.5$  will return the square root.*)
11. Write a function `is_rightangled` which, given the length of three sides of a triangle, will determine whether the triangle is right-angled. Assume that the third argument to the function is always the longest side. It will return `True` if the triangle is right-angled, or `False` otherwise.

*Hint: Floating point arithmetic is not always exactly accurate, so it is not safe to test floating point numbers for equality. If a good programmer wants to know whether  $x$  is equal or close enough to  $y$ , they would probably code it up as:*

```

1  if abs(x-y) < 0.000001:    # If x is approximately equal to y
2  ...

```

12. Extend the above program so that the sides can be given to the function in any order.
13. If you're intrigued by why floating point arithmetic is sometimes inaccurate, on a piece of paper, divide 10 by 3 and write down the decimal result. You'll find it does not terminate, so you'll need an infinitely long sheet of paper. The representation of numbers in computer memory or on your calculator has similar problems: memory is finite, and some digits may have to be discarded. So small inaccuracies creep in. Try this script:

```

1 import math
2 a = math.sqrt(2.0)
3 print(a, a*a)
4 print(a*a == 2.0)

```

# Chapter 6: Fruitful functions

## 6.1. Return values

The built-in functions we have used, such as `abs`, `pow`, `int`, `max`, and `range`, have produced results. Calling each of these functions generates a value, which we usually assign to a variable or use as part of an expression.

```
1 biggest = max(3, 7, 2, 5)
2 x = abs(3 - 11) + 10
```

We also wrote our own function to return the final amount for a compound interest calculation.

In this chapter, we are going to write more functions that return values, which we will call *fruitful functions*, for want of a better name. The first example is `area`, which returns the area of a circle with the given radius:

```
1 def area(radius):
2     b = 3.14159 * radius**2
3     return b
```

We have seen the `return` statement before, but in a fruitful function the `return` statement includes a **return value**. This statement means: evaluate the `return` expression, and then return it immediately as the result (the fruit) of this function. The expression provided can be arbitrarily complicated, so we could have written this function like this:

```
1 def area(radius):
2     return 3.14159 * radius * radius
```

On the other hand, **temporary variables** like `b` above often make debugging easier.

Sometimes it is useful to have multiple `return` statements, one in each branch of a conditional. We have already seen the built-in `abs`, now we see how to write our own:

```
1 def absolute_value(x):
2     if x < 0:
3         return -x
4     else:
5         return x
```

Another way to write the above function is to leave out the `else` and just follow the `if` condition by the second `return` statement.

```
1 def absolute_value(x):
2     if x < 0:
3         return -x
4     return x
```

Think about this version and convince yourself it works the same as the first one.

Code that appears after a `return` statement, or any other place the flow of execution can never reach, is called **dead code**, or **unreachable code**.

In a fruitful function, it is a good idea to ensure that every possible path through the program hits a `return` statement. The following version of `absolute_value` fails to do this:

```
1 def bad_absolute_value(x):
2     if x < 0:
3         return -x
4     elif x > 0:
5         return x
```

This version is not correct because if `x` happens to be `0`, neither condition is true, and the function ends without hitting a `return` statement. In this case, the `return` value is a special value called `None`:

```
1 >>> print(bad_absolute_value(0))
2 None
```

All Python functions return `None` whenever they do not return another value.

It is also possible to use a `return` statement in the middle of a `for` loop, in which case control immediately returns from the function. Let us assume that we want a function which looks through a list of words. It should return the first 2-letter word. If there is not one, it should return the empty string:

```
1 def find_first_2_letter_word(xs):
2     for wd in xs:
3         if len(wd) == 2:
4             return wd
5     return ""

1 >>> find_first_2_letter_word(["This", "is", "a", "dead", "parrot"])
2 'is'
3 >>> find_first_2_letter_word(["I", "like", "cheese"])
4 ''
```

Single-step through this code and convince yourself that in the first test case that we've provided, the function returns while processing the second element in the list: it does not have to traverse the whole list.

## 6.2. Program development

At this point, you should be able to look at complete functions and tell what they do. Also, if you have been doing the exercises, you have written some small functions. As you write larger functions, you might start to have more difficulty, especially with runtime and semantic errors.

To deal with increasingly complex programs, we are going to suggest a technique 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 we 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:

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

### Distance formula

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 two points are the inputs, which we can represent using four parameters. The return value is the distance, which is a floating-point value.

Already we can write an outline of the function that captures our thinking so far:

```
1 def distance(x1, y1, x2, y2):
2     return 0.0
```

Obviously, this version of the function doesn't compute distances; it always returns zero. But it is syntactically correct, and it will run, which means that we can test it before we make it more complicated.

To test the new function, we call it with sample values:

```
1 >>> distance(1, 2, 4, 6)
2 0.0
```

We chose these values so that the horizontal distance equals 3 and the vertical distance equals 4; that way, the result is 5 (the hypotenuse of a 3-4-5 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 lines of code. After each incremental change, we test the function again. If an error occurs at any point, we know where it must be — in the last line we added.

A logical first step in the computation is to find the differences  $x_2 - x_1$  and  $y_2 - y_1$ . We will refer to those values using temporary variables named `dx` and `dy`.

```
1 def distance(x1, y1, x2, y2):
2     dx = x2 - x1
3     dy = y2 - y1
4     return 0.0
```

If we call the function with the arguments shown above, when the flow of execution gets to the return statement, `dx` should be 3 and `dy` should be 4. We can check that this is the case in PyScripter by putting the cursor on the return statement, and running the program to break execution when it gets to the cursor (using the F4 key). Then we inspect the variables `dx` and `dy` by hovering the mouse above them, to confirm that the function is getting the right parameters 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`:

```
1 def distance(x1, y1, x2, y2):
2     dx = x2 - x1
3     dy = y2 - y1
4     dsquared = dx*dx + dy*dy
5     return 0.0
```

Again, we could run the program at this stage and check the value of `dsquared` (which should be 25).

Finally, using the fractional exponent 0.5 to find the square root, we compute and return the result:

```
1 def distance(x1, y1, x2, y2):
2     dx = x2 - x1
3     dy = y2 - y1
4     dsquared = dx*dx + dy*dy
5     result = dsquared**0.5
6     return result
```

If that works correctly, you are done. Otherwise, you might want to inspect the value of `result` before the return statement.

When you start out, you might add only a line or two of code at a time. As you gain more experience, you might find yourself writing and debugging bigger conceptual chunks. Either way, stepping

through your code one line at a time and verifying that each step matches your expectations can save you a lot of debugging time. As you improve your programming skills you should find yourself managing bigger and bigger chunks: this is very similar to the way we learned to read letters, syllables, words, phrases, sentences, paragraphs, etc., or the way we learn to chunk music — from individual notes to chords, bars, phrases, and so on.

The key aspects of the process are:

1. Start with a working skeleton program and make small incremental changes. At any point, if there is an error, you will know exactly where it is.
2. Use temporary variables to refer to intermediate values so that you can easily inspect and check them.
3. Once the program is working, relax, sit back, and play around with your options. (There is interesting research that links “playfulness” to better understanding, better learning, more enjoyment, and a more positive mindset about what you can achieve — so spend some time fiddling around!) You might want to consolidate multiple statements into one bigger compound expression, or rename the variables you’ve used, or see if you can make the function shorter. A good guideline is to aim for making code as easy as possible for others to read.

Here is another version of the function. It makes use of a square root function that is in the `math` module (we’ll learn about modules shortly). Which do you prefer? Which looks “closer” to the Pythagorean formula we started out with?

```
1 import math
2
3 def distance(x1, y1, x2, y2):
4     return math.sqrt( (x2-x1)**2 + (y2-y1)**2 )
5
6
7 >>> distance(1, 2, 4, 6)
8 5.0
```

## 6.3. Debugging with `print`

Another powerful technique for debugging (an alternative to single-stepping and inspection of program variables), is to insert extra `print` functions in carefully selected places in your code. Then, by inspecting the output of the program, you can check whether the algorithm is doing what you expect it to. Be clear about the following, however:

- You must have a clear solution to the problem, and must know what should happen before you can debug a program. Work on *solving* the problem on a piece of paper (perhaps using a flowchart to record the steps you take) *before* you concern yourself with writing code. Writing

a program doesn't solve the problem — it simply *automates* the manual steps you would take. So first make sure you have a pen-and-paper manual solution that works. Programming then is about making those manual steps happen automatically.

- Do not write **chatterbox** functions. A chatterbox is a fruitful function that, in addition to its primary task, also asks the user for input, or prints output, when it would be more useful if it simply shut up and did its work quietly.

For example, we've seen built-in functions like `range`, `max` and `abs`. None of these would be useful building blocks for other programs if they prompted the user for input, or printed their results while they performed their tasks.

So a good tip is to avoid calling `print` and `input` functions inside fruitful functions, *unless the primary purpose of your function is to perform input and output*. The one exception to this rule might be to temporarily sprinkle some calls to `print` into your code to help debug and understand what is happening when the code runs, but these will then be removed once you get things working.

## 6.4. Composition

As you should expect by now, you can call one function from within another. This ability is called **composition**.

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. Fortunately, we've just written a function, `distance`, that does just that, so now all we have to do is use it:

```
1 radius = distance(xc, yc, xp, yp)
```

The second step is to find the area of a circle with that radius and return it. Again we will use one of our earlier functions:

```
1 result = area(radius)
2 return result
```

Wrapping that up in a function, we get:

```
1 def area2(xc, yc, xp, yp):
2     radius = distance(xc, yc, xp, yp)
3     result = area(radius)
4     return result
```

We called this function `area2` to distinguish it from the `area` function defined earlier.

The temporary variables `radius` and `result` are useful for development, debugging, and single-stepping through the code to inspect what is happening, but once the program is working, we can make it more concise by composing the function calls:

```
1 def area2(xc, yc, xp, yp):
2     return area(distance(xc, yc, xp, yp))
```

## 6.5. Boolean functions

Functions can return Boolean values, which is often convenient for hiding complicated tests inside functions. For example:

```
1 def is_divisible(x, y):
2     """ Test if x is exactly divisible by y """
3     if x % y == 0:
4         return True
5     else:
6         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 the `x` is or is not divisible by `y`.

We can make the function more concise by taking advantage of the fact that the condition of the `if` statement is itself a Boolean expression. We can return it directly, avoiding the `if` statement altogether:

```
1 def is_divisible(x, y):
2     return x % y == 0
```

This session shows the new function in action:

```
1 >>> is_divisible(6, 4)
2 False
3 >>> is_divisible(6, 3)
4 True
```

Boolean functions are often used in conditional statements:

```
1 if is_divisible(x, y):
2     ... # Do something ...
3 else:
4     ... # Do something else ...
```

It might be tempting to write something like:

```
1 if is_divisible(x, y) == True:
```

but the extra comparison is unnecessary.

## 6.6. Programming with style

Readability is very important to programmers, since in practice programs are read and modified far more often than they are written. But, like most rules, we occasionally break them. Most of the code examples in this book will be consistent with the *Python Enhancement Proposal 8* (PEP 8<sup>8</sup>), a style guide developed by the Python community.

We'll have more to say about style as our programs become more complex, but a few pointers will be helpful already:

- use 4 spaces (instead of tabs) for indentation
- limit line length to 78 characters
- when naming identifiers, use CamelCase for classes (we'll get to those) and lowercase\_with\_underscores for functions and variables
- place imports at the top of the file
- keep function definitions together
- use docstrings to document functions
- use two blank lines to separate function definitions from each other
- keep top level statements, including function calls, together at the bottom of the program

---

<sup>8</sup><http://www.python.org/dev/peps/pep-0008/>

## 6.7. Unit testing

It is a common best practice in software development to include automatic **unit testing** of source code. Unit testing provides a way to automatically verify that individual pieces of code, such as functions, are working properly. This makes it possible to change the implementation of a function at a later time and quickly test that it still does what it was intended to do.

Some years back organizations had the view that their valuable asset was the program code and documentation. Organizations will now spend a large portion of their software budgets on crafting (and preserving) their tests.

Unit testing also forces the programmer to think about the different cases that the function needs to handle. You also only have to type the tests once into the script, rather than having to keep entering the same test data over and over as you develop your code.

Extra code in your program which is there because it makes debugging or testing easier is called **scaffolding**.

A collection of tests for some code is called its **test suite**.

There are a few different ways to do unit testing in Python — but at this stage we’re going to ignore what the Python community usually does, and we’re going to start with two functions that we’ll write ourselves. We’ll use these for writing our unit tests.

Let’s start with the `absolute_value` function that we wrote earlier in this chapter. Recall that we wrote a few different versions, the last of which was incorrect, and had a bug. Would tests have caught this bug?

First we plan our tests. We’d like to know if the function returns the correct value when its argument is negative, or when its argument is positive, or when its argument is zero. When planning your tests, you’ll always want to think carefully about the “edge” cases — here, an argument of `0` to `absolute_value` is on the edge of where the function behaviour changes, and as we saw at the beginning of the chapter, it is an easy spot for the programmer to make a mistake! So it is a good case to include in our test suite.

We’re going to write a helper function for checking the results of one test. It takes a boolean argument and will either print a message telling us that the test passed, or it will print a message to inform us that the test failed. The first line of the body (after the function’s docstring) magically determines the line number in the script where the call was made from. This allows us to print the line number of the test, which will help when we want to identify which tests have passed or failed.

```
1 import sys
2
3 def test(did_pass):
4     """ Print the result of a test. """
5     linenum = sys._getframe(1).f_lineno    # Get the caller's line number.
6     if did_pass:
7         msg = "Test at line {0} ok.".format(linenum)
8     else:
9         msg = "Test at line {0} FAILED.".format(linenum)
10    print(msg)
```

There is also some slightly tricky string formatting using the `format` method which we will gloss over for the moment, and cover in detail in a future chapter. But with this function written, we can proceed to construct our test suite:

```
1 def test_suite():
2     """ Run the suite of tests for code in this module (this file).
3     """
4     test(absolute_value(17) == 17)
5     test(absolute_value(-17) == 17)
6     test(absolute_value(0) == 0)
7     test(absolute_value(3.14) == 3.14)
8     test(absolute_value(-3.14) == 3.14)
9
10 test_suite()      # Here is the call to run the tests
```

Here you'll see that we've constructed five tests in our test suite. We could run this against the first or second versions (the correct versions) of `absolute_value`, and we'd get output similar to the following:

```
1 Test at line 25 ok.
2 Test at line 26 ok.
3 Test at line 27 ok.
4 Test at line 28 ok.
5 Test at line 29 ok.
```

But let's say you change the function to an incorrect version like this:

```
1 def absolute_value(n):    # Buggy version
2     """ Compute the absolute value of n """
3     if n < 0:
4         return 1
5     elif n > 0:
6         return n
```

Can you find at least two mistakes in this code? Our test suite can! We get:

```
1 Test at line 25 ok.
2 Test at line 26 FAILED.
3 Test at line 27 FAILED.
4 Test at line 28 ok.
5 Test at line 29 FAILED.
```

These are three examples of *failing tests*.

There is a built-in Python statement called `assert` that does almost the same as our `test` function (except the program stops when the first assertion fails). You may want to read about it, and use it instead of our `test` function.

## 6.8. Glossary

### Boolean function

A function that returns a Boolean value. The only possible values of the `bool` type are `False` and `True`.

### chatterbox function

A function which interacts with the user (using `input` or `print`) when it should not. Silent functions that just convert their input arguments into their output results are usually the most useful ones.

### composition (of functions)

Calling one function from within the body of another, or using the `return` value of one function as an argument to the call of another.

### dead code

Part of a program that can never be executed, often because it appears after a `return` statement.

### fruitful function

A function that yields a `return` value instead of `None`.

### incremental development

A program development plan intended to simplify debugging by adding and testing only a small amount of code at a time.

**None**

A special Python value. One use in Python is that it is returned by functions that do not execute a `return` statement with a `return` argument.

**return value**

The value provided as the result of a function call.

**scaffolding**

Code that is used during program development to assist with development and debugging. The unit test code that we added in this chapter are examples of scaffolding.

**temporary variable**

A variable used to store an intermediate value in a complex calculation.

**test suite**

A collection of tests for some code you have written.

**unit testing**

An automatic procedure used to validate that individual units of code are working properly. Having a test suite is extremely useful when somebody modifies or extends the code: it provides a safety net against going backwards by putting new bugs into previously working code. The term *regression* testing is often used to capture this idea that we don't want to go backwards!

## 6.9. Exercises

All of the exercises below should be added to a single file. In that file, you should also add the `test` and `test_suite` scaffolding functions shown above, and then, as you work through the exercises, add the new tests to your test suite. (If you open the online version of the textbook, you can easily copy and paste the tests and the fragments of code into your Python editor.)

After completing each exercise, confirm that all the tests pass.

1. The four compass points can be abbreviated by single-letter strings as “N”, “E”, “S”, and “W”.

Write a function `turn_clockwise` that takes one of these four compass points as its parameter, and returns the next compass point in the clockwise direction. Here are some tests that should pass:

```
1  test(turn_clockwise("N") == "E")
2  test(turn_clockwise("W") == "N")
```

You might ask “What if the argument to the function is some other value?” For all other cases, the function should return the value `None`:

```

1 test(turn_clockwise(42) == None)
2 test(turn_clockwise("rubbish") == None)

```

2. Write a function `day_name` that converts an integer number 0 to 6 into the name of a day. Assume day 0 is “Sunday”. Once again, return `None` if the arguments to the function are not valid. Here are some tests that should pass:

```

1 test(day_name(3) == "Wednesday")
2 test(day_name(6) == "Saturday")
3 test(day_name(42) == None)

```

3. Write the inverse function `day_num` which is given a day name, and returns its number:

```

1 test(day_num("Friday") == 5)
2 test(day_num("Sunday") == 0)
3 test(day_num(day_name(3)) == 3)
4 test(day_name(day_num("Thursday")) == "Thursday")

```

Once again, if this function is given an invalid argument, it should return `None`:

```
1 test(day_num("Halloween") == None)
```

4. Write a function that helps answer questions like “Today is Wednesday. I leave on holiday in 19 days time. What day will that be?” So the function must take a day name and a delta argument — the number of days to add — and should return the resulting day name:

```

1 test(day_add("Monday", 4) == "Friday")
2 test(day_add("Tuesday", 0) == "Tuesday")
3 test(day_add("Tuesday", 14) == "Tuesday")
4 test(day_add("Sunday", 100) == "Tuesday")

```

*Hint: use the first two functions written above to help you write this one.*

5. Can your `day_add` function already work with negative deltas? For example, -1 would be yesterday, or -7 would be a week ago:

```

1 test(day_add("Sunday", -1) == "Saturday")
2 test(day_add("Sunday", -7) == "Sunday")
3 test(day_add("Tuesday", -100) == "Sunday")

```

If your function already works, explain why. If it does not work, make it work.

*Hint: Play with some cases of using the modulus function % (introduced at the beginning of the previous chapter). Specifically, explore what happens to  $x \% 7$  when  $x$  is negative.*

6. Write a function `days_in_month` which takes the name of a month, and returns the number of days in the month. Ignore leap years:

```

1 test(days_in_month("February") == 28)
2 test(days_in_month("December") == 31)

```

If the function is given invalid arguments, it should return `None`.

7. Write a function `to_secs` that converts hours, minutes and seconds to a total number of seconds.

Here are some tests that should pass:

```

1 test(to_secs(2, 30, 10) == 9010)
2 test(to_secs(2, 0, 0) == 7200)
3 test(to_secs(0, 2, 0) == 120)
4 test(to_secs(0, 0, 42) == 42)
5 test(to_secs(0, -10, 10) == -590)

```

8. Extend `to_secs` so that it can cope with real values as inputs. It should always return an integer number of seconds (truncated towards zero):

```

1 test(to_secs(2.5, 0, 10.71) == 9010)
2 test(to_secs(2.433, 0, 0) == 8758)

```

9. Write three functions that are the “inverses” of `to_secs`:

1. `hours_in` returns the whole integer number of hours represented by a total number of seconds.

2. `minutes_in` returns the whole integer number of left over minutes in a total number of seconds, once the hours have been taken out.

3. `seconds_in` returns the left over seconds represented by a total number of seconds.

You may assume that the total number of seconds passed to these functions is an integer. Here are some test cases:

```

1 test(hours_in(9010) == 2)
2 test(minutes_in(9010) == 30)
3 test(seconds_in(9010) == 10)

```

### **It won't always be obvious what is wanted ...**

In the third case above, the requirement seems quite ambiguous and fuzzy. But the test clarifies what we actually need to do.

Unit tests often have this secondary benefit of clarifying the specifications. If you write your own test suites, consider it part of the problem-solving process as you ask questions about what you really expect to happen, and whether you've considered all the possible cases.

Given our emphasis on *thinking like a computer scientist*, you might enjoy reading at least one reference about thinking, and about fun ideas like *fluid intelligence*, a key ingredient in problem solving. See, for example, <http://psychology.about.com/od/cognitivepsychology/a/fluid-crystal.htm>. Learning Computer Science requires a good mix of both fluid and crystallized kinds of intelligence.

10. Which of these tests fail? Explain why.

```

1 test(3 % 4 == 0)
2 test(3 % 4 == 3)
3 test(3 / 4 == 0)
4 test(3 // 4 == 0)
5 test(3+4 * 2 == 14)
6 test(4-2+2 == 0)
7 test(len("hello, world!") == 13)

```

11. Write a `compare` function that returns 1 if  $a > b$ , 0 if  $a == b$ , and -1 if  $a < b$

```

1 test(compare(5, 4) == 1)
2 test(compare(7, 7) == 0)
3 test(compare(2, 3) == -1)
4 test(compare(42, 1) == 1)

```

12. Write a function called `hypotenuse` that returns the length of the hypotenuse of a right triangle given the lengths of the two legs as parameters:

```

1 test(hypotenuse(3, 4) == 5.0)
2 test(hypotenuse(12, 5) == 13.0)
3 test(hypotenuse(24, 7) == 25.0)
4 test(hypotenuse(9, 12) == 15.0)

```

13. Write a function `slope(x1, y1, x2, y2)` that returns the slope of the line through the points  $(x_1, y_1)$  and  $(x_2, y_2)$ . Be sure your implementation of `slope` can pass the following tests:

```

1 test(slope(5, 3, 4, 2) == 1.0)
2 test(slope(1, 2, 3, 2) == 0.0)
3 test(slope(1, 2, 3, 3) == 0.5)
4 test(slope(2, 4, 1, 2) == 2.0)

```

Then use a call to `slope` in a new function named `intercept(x1, y1, x2, y2)` that returns the y-intercept of the line through the points  $(x_1, y_1)$  and  $(x_2, y_2)$

```

1 test(intercept(1, 6, 3, 12) == 3.0)
2 test(intercept(6, 1, 1, 6) == 7.0)
3 test(intercept(4, 6, 12, 8) == 5.0)

```

14. Write a function called `is_even(n)` that takes an integer as an argument and returns `True` if the argument is an **even number** and `False` if it is **odd**.

Add your own tests to the test suite.

15. Now write the function `is_odd(n)` that returns `True` when  $n$  is odd and `False` otherwise. Include unit tests for this function too.

Finally, modify it so that it uses a call to `is_even` to determine if its argument is an odd integer, and ensure that its test still pass.

16. Write a function `is_factor(f, n)` that passes these tests:

```
1 test(is_factor(3, 12))
2 test(not is_factor(5, 12))
3 test(is_factor(7, 14))
4 test(not is_factor(7, 15))
5 test(is_factor(1, 15))
6 test(is_factor(15, 15))
7 test(not is_factor(25, 15))
```

An important role of unit tests is that they can also act as unambiguous “specifications” of what is expected. These test cases answer the question “Do we treat 1 and 15 as factors of 15”?

17. Write `is_multiple` to satisfy these unit tests:

```
1 test(is_multiple(12, 3))
2 test(is_multiple(12, 4))
3 test(not is_multiple(12, 5))
4 test(is_multiple(12, 6))
5 test(not is_multiple(12, 7))
```

Can you find a way to use `is_factor` in your definition of `is_multiple`?

18. Write the function `f2c(t)` designed to return the integer value of the nearest degree Celsius for given temperature in Fahrenheit. (*hint: you may want to make use of the built-in function, `round`. Try printing `round.__doc__` in a Python shell or looking up help for the `round` function, and experimenting with it until you are comfortable with how it works.*)

```
1 test(f2c(212) == 100)      # Boiling point of water
2 test(f2c(32) == 0)         # Freezing point of water
3 test(f2c(-40) == -40)     # Wow, what an interesting case!
4 test(f2c(36) == 2)
5 test(f2c(37) == 3)
6 test(f2c(38) == 3)
7 test(f2c(39) == 4)
```

19. Now do the opposite: write the function `c2f` which converts Celsius to Fahrenheit:

```
1 test(c2f(0) == 32)
2 test(c2f(100) == 212)
3 test(c2f(-40) == -40)
4 test(c2f(12) == 54)
5 test(c2f(18) == 64)
6 test(c2f(-48) == -54)
```

# Chapter 7: Iteration

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.

Repeated execution of a set of statements is called **iteration**. Because iteration is so common, Python provides several language features to make it easier. We've already seen the `for` statement in chapter 3. This the the form of iteration you'll likely be using most often. But in this chapter we've going to look at the `while` statement — another way to have your program do iteration, useful in slightly different circumstances.

Before we do that, let's just review a few ideas...

## 7.1. Assignment

As we have mentioned previously, 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).

```
1 airtime_remaining = 15
2 print(airtime_remaining)
3 airtime_remaining = 7
4 print(airtime_remaining)
```

The output of this program is:

```
1 15
2 7
```

because the first time `airtime_remaining` is printed, its value is 15, and the second time, its value is 7.

It is especially important to distinguish between an assignment statement and a Boolean expression that tests for equality. Because Python uses the equal token (`=`) for assignment, it is tempting to interpret a statement like `a = b` as a Boolean test. Unlike mathematics, it is not! Remember that the Python token for the equality operator is `==`.

Note too that an equality test is symmetric, but assignment is not. For example, if `a == 7` then `7 == a`. But in Python, the statement `a = 7` is legal and `7 = a` is not.

In Python, an assignment statement can make two variables equal, but because further assignments can change either of them, they don't have to stay that way:

```
1 a = 5
2 b = a      # After executing this line, a and b are now equal
3 a = 3      # After executing this line, a and b are no longer equal
```

The third line changes the value of `a` but does not change the value of `b`, so they are no longer equal. (In some programming languages, a different symbol is used for assignment, such as `<-` or `:=`, to avoid confusion. Some people also think that variable was an unfortunate word to choose, and instead we should have called them assignables. Python chooses to follow common terminology and token usage, also found in languages like C, C++, Java, and C#, so we use the tokens `=` for assignment, `==` for equality, and we talk of variables.

## 7.2. Updating variables

When an assignment statement is executed, the right-hand side expression (i.e. the expression that comes after the assignment token) is evaluated first. This produces a value. Then the assignment is made, so that the variable on the left-hand side now refers to the new value.

One of the most common forms of assignment is an update, where the new value of the variable depends on its old value. Deduct 40 cents from my airtime balance, or add one run to the scoreboard.

```
1 n = 5
2 n = 3 * n + 1
```

Line 2 means *get the current value of n, multiply it by three and add one, and assign the answer to n, thus making n refer to the value*. So after executing the two lines above, `n` will point/refer to the integer 16.

If you try to get the value of a variable that has never been assigned to, you'll get an error:

```
1 >>> w = x + 1
2 Traceback (most recent call last):
3   File "<interactive input>", line 1, in
4     NameError: name 'x' is not defined
```

Before you can update a variable, you have to **initialize** it to some starting value, usually with a simple assignment:

```
1 runs_scored = 0
2 ...
3 runs_scored = runs_scored + 1
```

Line 3 — updating a variable by adding 1 to it — is very common. It is called an **increment** of the variable; subtracting 1 is called a **decrement**. Sometimes programmers also talk about bumping a variable, which means the same as incrementing it by 1.

## 7.3. The for loop revisited

Recall that the for loop processes each item in a list. Each item in turn is (re-)assigned to the loop variable, and the body of the loop is executed. We saw this example in an earlier chapter:

```
1 for f in ["Joe", "Zoe", "Brad", "Angelina", "Zuki", "Thandi", "Paris"]:
2     invitation = "Hi " + f + ". Please come to my party on Saturday!"
3     print(invitation)
```

Running through all the items in a list is called **traversing** the list, or **traversal**.

Let us write a function now to sum up all the elements in a list of numbers. Do this by hand first, and try to isolate exactly what steps you take. You'll find you need to keep some “running total” of the sum so far, either on a piece of paper, in your head, or in your calculator. Remembering things from one step to the next is precisely why we have variables in a program: so we'll need some variable to remember the “running total”. It should be initialized with a value of zero, and then we need to traverse the items in the list. For each item, we'll want to update the running total by adding the next number to it.

```
1 def mysum(xs):
2     """ Sum all the numbers in the list xs, and return the total. """
3     running_total = 0
4     for x in xs:
5         running_total = running_total + x
6     return running_total
7
8 # Add tests like these to your test suite ...
9 test(mysum([1, 2, 3, 4]) == 10)
10 test(mysum([1.25, 2.5, 1.75]) == 5.5)
11 test(mysum([1, -2, 3]) == 2)
12 test(mysum([]) == 0)
13 test(mysum(range(11)) == 55) # 11 is not included in the list.
```

## 7.4. The while statement

Here is a fragment of code that demonstrates the use of the while statement:

```

1 def sum_to(n):
2     """ Return the sum of 1+2+3 ... n """
3     ss = 0
4     v = 1
5     while v <= n:
6         ss = ss + v
7         v = v + 1
8     return ss
9
10 # For your test suite
11 test(sum_to(4) == 10)
12 test(sum_to(1000) == 500500)

```

You can almost read the `while` statement as if it were English. It means, while `v` is less than or equal to `n`, continue executing the body of the loop. Within the body, each time, increment `v`. When `v` passes `n`, return your accumulated sum.

More formally, here is precise flow of execution for a `while` statement:

- Evaluate the condition at line 5, yielding a value which is either False or True.
- If the value is False, exit the `while` statement and continue execution at the next statement (line 8 in this case).
- If the value is True, execute each of the statements in the body (lines 6 and 7) and then go back to the `while` statement at line 5.

The body consists of all of the statements indented below the `while` keyword.

Notice that if the loop condition is False the first time we get loop, the statements in the body of the loop are never executed.

The body of the loop should change the value of one or more variables so that eventually the condition becomes false 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 here, we can prove that the loop terminates because we know that the value of `n` is finite, and we can see that the value of `v` increments each time through the loop, so eventually it will have to exceed `n`. In other cases, it is not so easy, even impossible in some cases, to tell if the loop will ever terminate.

What you will notice here is that the `while` loop is more work for you — the programmer — than the equivalent `for` loop. When using a `while` loop one has to manage the loop variable yourself: give it an initial value, test for completion, and then make sure you change something in the body so that the loop terminates. By comparison, here is an equivalent function that uses `for` instead:

```

1 def sum_to(n):
2     """ Return the sum of 1+2+3 ... n """
3     ss = 0
4     for v in range(n+1):
5         ss = ss + v
6     return ss

```

Notice the slightly tricky call to the `range` function — we had to add one onto `n`, because `range` generates its list up to but excluding the value you give it. It would be easy to make a programming mistake and overlook this, but because we've made the investment of writing some unit tests, our test suite would have caught our error.

So why have two kinds of loop if `for` looks easier? This next example shows a case where we need the extra power that we get from the `while` loop.

## 7.5. The Collatz 3n + 1 sequence

Let's look at a simple sequence that has fascinated and foxed mathematicians for many years. They still cannot answer even quite simple questions about this.

The “computational rule” for creating the sequence is to start from some given `n`, and to generate the next term of the sequence from `n`, either by halving `n`, (whenever `n` is even), or else by multiplying it by three and adding 1. The sequence terminates when `n` reaches 1.

This Python function captures that algorithm:

```

1 def seq3np1(n):
2     """ Print the 3n+1 sequence from n,
3         terminating when it reaches 1.
4     """
5     while n != 1:
6         print(n, end=", ")
7         if n % 2 == 0:      # n is even
8             n = n // 2
9         else:                # n is odd
10            n = n * 3 + 1
11    print(n, end=".\\n")

```

Notice first that the `print` function on line 6 has an extra argument `end=" , "`. This tells the `print` function to follow the printed string with whatever the programmer chooses (in this case, a comma followed by a space), instead of ending the line. So each time something is printed in the loop, it is printed on the same output line, with the numbers separated by commas. The call to `print(n, end=".\\n")` at line 11 after the loop terminates will then print the final value of `n` followed by a period and a newline character. (You'll cover the `\n` (newline character) in the next chapter).

The condition for continuing with this loop is `n != 1`, so the loop will continue running until it reaches its termination condition, (i.e. `n == 1`).

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, the value of `n` is divided by 2 using integer division. If it is odd, the value is replaced by `n * 3 + 1`. Here are some examples:

```

1  >>> seq3np1(3)
2  3, 10, 5, 16, 8, 4, 2, 1.
3  >>> seq3np1(19)
4  19, 58, 29, 88, 44, 22, 11, 34, 17, 52, 26, 13,
5          40, 20, 10, 5, 16, 8, 4, 2, 1.
6  >>> seq3np1(21)
7  21, 64, 32, 16, 8, 4, 2, 1.
8  >>> seq3np1(16)
9  16, 8, 4, 2, 1.
10 >>>

```

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, then the value of `n` will be even each time through the loop until it reaches 1. The previous example ends with such a sequence, starting with 16.

See if you can find a small starting number that needs more than a hundred steps before it terminates.

Particular values aside, the interesting question was first posed by a German mathematician called Lothar Collatz: the *Collatz conjecture* (also known as the  *$3n + 1$  conjecture*), is that this sequence terminates for all positive values of `n`. So far, no one has been able to prove it or disprove it! (A conjecture is a statement that might be true, but nobody knows for sure.)

Think carefully about what would be needed for a proof or disproof of the conjecture “All positive integers will eventually converge to 1 using the Collatz rules”. With fast computers we have been able to test every integer up to very large values, and so far, they have all eventually ended up at 1. But who knows? Perhaps there is some as-yet untested number which does not reduce to 1.

You’ll notice that if you don’t stop when you reach 1, the sequence gets into its own cyclic loop: 1, 4, 2, 1, 4, 2, 1, 4 ... So one possibility is that there might be other cycles that we just haven’t found yet.

Wikipedia has an informative article about the Collatz conjecture. The sequence also goes under other names (Hailstone sequence, Wonderous numbers, etc.), and you’ll find out just how many integers have already been tested by computer, and found to converge!

### Choosing between for and while

Use a `for` loop if you know, before you start looping, the maximum number of times that you’ll need to execute the body. For example, if you’re traversing a list of elements, you know that the

maximum number of loop iterations you can possibly need is “all the elements in the list”. Or if you need to print the 12 times table, we know right away how many times the loop will need to run.

So any problem like “iterate this weather model for 1000 cycles”, or “search this list of words”, “find all prime numbers up to 10000” suggest that a `for` loop is best.

By contrast, if you are required to repeat some computation until some condition is met, and you cannot calculate in advance when (or if) this will happen, as we did in this  $3n + 1$  problem, you’ll need a `while` loop.

We call the first case **definite iteration** — we know ahead of time some definite bounds for what is needed. The latter case is called **indefinite iteration** — we’re not sure how many iterations we’ll need — we cannot even establish an upper bound!

## 7.6. Tracing a program

To write effective computer programs, and to build a good conceptual model of program execution, a programmer needs to develop the ability to **trace** the execution of a computer program. Tracing involves becoming the computer and following the flow of execution through a sample program run, recording the state of all variables and any output the program generates after each instruction is executed.

To understand this process, let’s trace the call to `seq3np1(3)` from the previous section. At the start of the trace, we have a variable, `n` (the parameter), with an initial value of 3. Since 3 is not equal to 1, the `while` loop body is executed. `3` is printed and `3 % 2 == 0` is evaluated. Since it evaluates to `False`, the `else` branch is executed and `3 * 3 + 1` is evaluated and assigned to `n`.

To keep track of all this as you hand trace a program, make a column heading on a piece of paper for each variable created as the program runs and another one for output. Our trace so far would look something like this:

		output printed so far
1	<code>n</code>	
2	--	-----
3	<code>3</code>	<code>3,</code>
4	<code>10</code>	

Since `10 != 1` evaluates to `True`, the loop body is again executed, and `10` is printed. `10 % 2 == 0` is `True`, so the `if` branch is executed and `n` becomes 5. By the end of the trace we have:

```
1 n          output printed so far
2 --
3 3          3,
4 10         3, 10,
5 5          3, 10, 5,
6 16         3, 10, 5, 16,
7 8          3, 10, 5, 16, 8,
8 4          3, 10, 5, 16, 8, 4,
9 2          3, 10, 5, 16, 8, 4, 2,
10 1         3, 10, 5, 16, 8, 4, 2, 1.
```

Tracing can be a bit tedious and error prone (that's why we get computers to do this stuff in the first place!), but it is an essential skill for a programmer to have. From this trace we can learn a lot about the way our code works. We can observe that as soon as `n` becomes a power of 2, for example, the program will require  $\log_2(n)$  executions of the loop body to complete. We can also see that the final 1 will not be printed as output within the body of the loop, which is why we put the special `print` function at the end.

Tracing a program is, of course, related to single-stepping through your code and being able to inspect the variables. Using the computer to **single-step** for you is less error prone and more convenient. Also, as your programs get more complex, they might execute many millions of steps before they get to the code that you're really interested in, so manual tracing becomes impossible. Being able to set a **breakpoint** where you need one is far more powerful. So we strongly encourage you to invest time in learning using to use your programming environment to full effect.

There are also some great visualization tools becoming available to help you trace and understand small fragments of Python code. The one we recommend is at <http://pythontutor.com/>

We've cautioned against chatterbox functions, but used them here. As we learn a bit more Python, we'll be able to show you how to generate a list of values to hold the sequence, rather than having the function print them. Doing this would remove the need to have all these pesky print functions in the middle of our logic, and will make the function more useful.

## 7.7. Counting digits

The following function counts the number of decimal digits in a positive integer:

```
1 def num_digits(n):
2     count = 0
3     while n != 0:
4         count = count + 1
5         n = n // 10
6     return count
```

A call to `print(num_digits(710))` will print 3. Trace the execution of this function call (perhaps using the single step function in PyScripter, or the Python visualizer, or on some paper) to convince yourself that it works.

This function demonstrates an important pattern of computation called a **counter**. The variable `count` is initialized to 0 and then incremented each time the loop body is executed. When the loop exits, `count` contains the result — the total number of times the loop body was executed, which is the same as the number of digits.

If we wanted to only count digits that are either 0 or 5, adding a conditional before incrementing the counter will do the trick:

```
1 def num_zero_and_five_digits(n):
2     count = 0
3     while n > 0:
4         digit = n % 10
5         if digit == 0 or digit == 5:
6             count = count + 1
7         n = n // 10
8     return count
```

Confirm that `test(num_zero_and_five_digits(1055030250) == 7)` passes.

Notice, however, that `test(num_digits(0) == 1)` fails. Explain why. Do you think this is a bug in the code, or a bug in the specifications, or our expectations, or the tests?

## 7.8. Abbreviated assignment

Incrementing a variable is so common that Python provides an abbreviated syntax for it:

```
1 >>> count = 0
2 >>> count += 1
3 >>> count
4 1
5 >>> count += 1
6 >>> count
7 2
```

`count += 1` is an abbreviation for `count = count + 1`. We pronounce the operator as “plus-equals”. The increment value does not have to be 1:

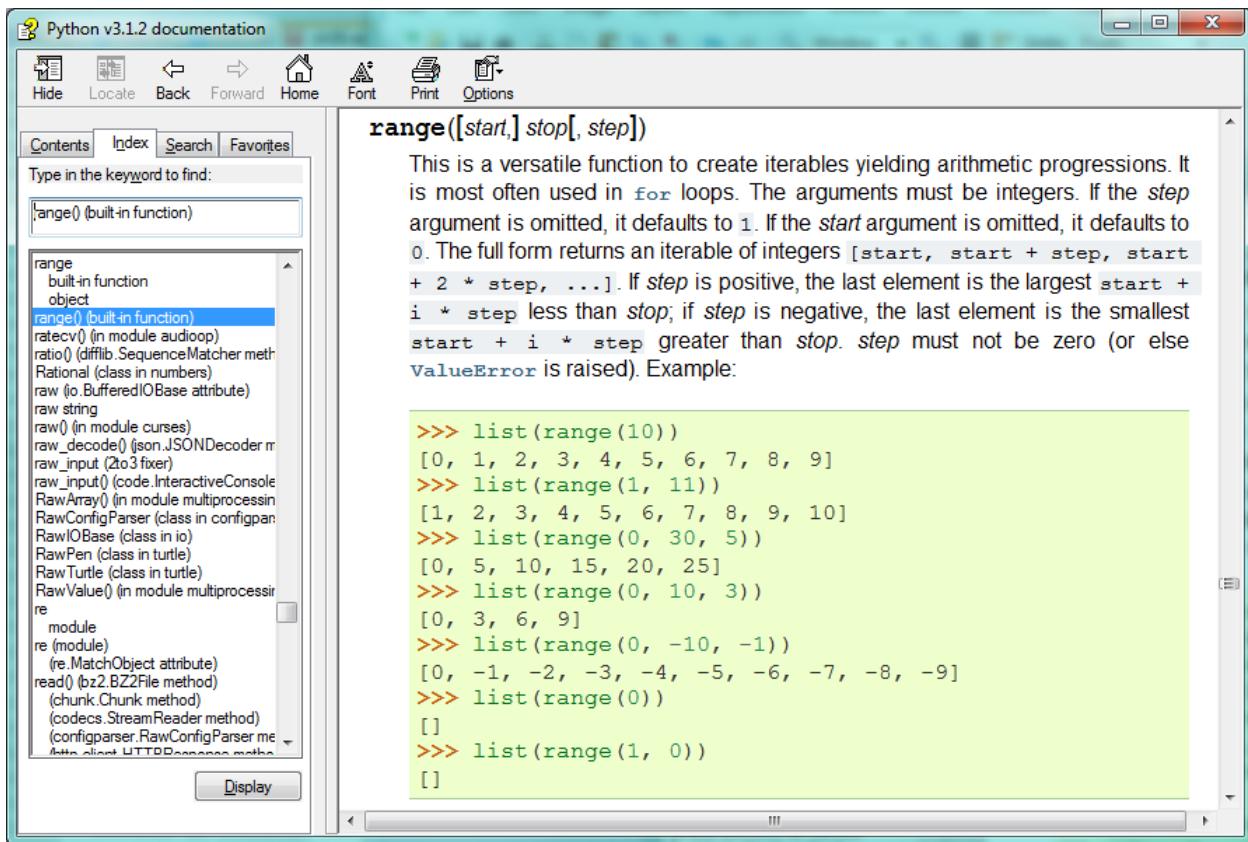
```
1 >>> n = 2
2 >>> n += 5
3 >>> n
4 7
```

There are similar abbreviations for `-=`, `*=`, `/=`, `//=` and `%=`:

```
1 >>> n = 2
2 >>> n *= 5
3 >>> n
4 10
5 >>> n -= 4
6 >>> n
7 6
8 >>> n //= 2
9 >>> n
10 3
11 >>> n %= 2
12 >>> n
13 1
```

## 7.9. Help and meta-notation

Python comes with extensive documentation for all its built-in functions, and its libraries. Different systems have different ways of accessing this help. In PyScripter, click on the Help menu item, and select Python Manuals. Then search for help on the built-in function `range`. You’ll get something like this:



Notice the square brackets in the description of the arguments. These are examples of **meta-notation** — notation that describes Python syntax, but is not part of it. The square brackets in this documentation mean that the argument is optional — the programmer can omit it. So what this first line of help tells us is that `range` must always have a `stop` argument, but it may have an optional `start` argument (which must be followed by a comma if it is present), and it can also have an optional `step` argument, preceded by a comma if it is present.

The examples from help show that `range` can have either 1, 2 or 3 arguments. The list can start at any starting value, and go up or down in increments other than 1. The documentation here also says that the arguments must be integers.

Other meta-notation you'll frequently encounter is the use of bold and italics. The bold means that these are tokens — keywords or symbols — typed into your Python code exactly as they are, whereas the italic terms stand for “something of this type”. So the syntax description

**for variable in list:**

means you can substitute any legal variable and any legal list when you write your Python code.

This (simplified) description of the `print` function, shows another example of meta-notation in which the ellipses (...) mean that you can have as many objects as you like (even zero), separated by commas:

**print([object, ...])**

Meta-notation gives us a concise and powerful way to describe the pattern of some syntax or feature.

## 7.10. Tables

One of the things loops are good for is generating tables. Before computers were readily available, people had to calculate logarithms, sines and cosines, and other mathematical functions by hand. To make that easier, mathematics books contained long tables listing the values of these functions. Creating the tables was slow and boring, and they tended to be full of errors.

When computers appeared on the scene, one of the initial reactions was, “*This is great! We can use the computers to generate the tables, so there will be no errors.*” That turned out to be true (mostly) but shortsighted. Soon thereafter, computers and calculators were so pervasive that the tables became obsolete.

Well, almost. For some operations, computers use tables of values to get an approximate answer and then perform computations to improve the approximation. In some cases, there have been errors in the underlying tables, most famously in the table the Intel Pentium processor chip used to perform floating-point division.

Although a log table is not as useful as it once was, it still makes a good example of iteration. The following program outputs a sequence of values in the left column and 2 raised to the power of that value in the right column:

```
1 for x in range(13):    # Generate numbers 0 to 12
2     print(x, "\t", 2**x)
```

The string "\t" represents a **tab character**. The backslash character in "\t" indicates the beginning of an **escape sequence**. Escape sequences are used to represent invisible characters like tabs and newlines. The sequence \n represents a **newline**.

An escape sequence can appear anywhere in a string; in this example, the tab escape sequence is the only thing in the string. How do you think you represent a backslash in a string?

As characters and strings are displayed on the screen, an invisible marker called the cursor keeps track of where the next character will go. After a `print` function, the cursor normally goes to the beginning of the next line.

The tab character shifts the cursor to the right until it reaches one of the tab stops. Tabs are useful for making columns of text line up, as in the output of the previous program:

```
1 0      1
2 1      2
3 2      4
4 3      8
5 4      16
6 5      32
7 6      64
8 7      128
9 8      256
10 9      512
11 10     1024
12 11     2048
13 12     4096
```

Because of the tab characters between the columns, the position of the second column does not depend on the number of digits in the first column.

## 7.11. Two-dimensional tables

A two-dimensional table is a table where you read the value at the intersection of a row and a column. A multiplication table is a good example. Let's say you want to print a multiplication table for the values from 1 to 6.

A good way to start is to write a loop that prints the multiples of 2, all on one line:

```
1 for i in range(1, 7):
2     print(2 * i, end="   ")
3 print()
```

Here we've used the `range` function, but made it start its sequence at 1. As the loop executes, the value of `i` changes from 1 to 6. When all the elements of the range have been assigned to `i`, the loop terminates. Each time through the loop, it displays the value of `2 * i`, followed by three spaces.

Again, the extra `end=" "` argument in the `print` function suppresses the newline, and uses three spaces instead. After the loop completes, the call to `print` at line 3 finishes the current line, and starts a new line.

The output of the program is:

```
1 2      4      6      8      10     12
```

So far, so good. The next step is to **encapsulate** and **generalize**.

## 7.12. Encapsulation and generalization

Encapsulation is the process of wrapping a piece of code in a function, allowing you to take advantage of all the things functions are good for. You have already seen some examples of encapsulation, including `is_divisible` in a previous chapter.

Generalization means taking something specific, such as printing the multiples of 2, and making it more general, such as printing the multiples of any integer.

This function encapsulates the previous loop and generalizes it to print multiples of `n`:

```
1 def print_multiples(n):
2     for i in range(1, 7):
3         print(n * i, end=" ")
4     print()
```

To encapsulate, all we had to do was add the first line, which declares the name of the function and the parameter list. To generalize, all we had to do was replace the value 2 with the parameter `n`.

If we call this function with the argument 2, we get the same output as before. With the argument 3, the output is:

```
1 3      6      9      12      15      18
```

With the argument 4, the output is:

```
1 4      8      12      16      20      24
```

By now you can probably guess how to print a multiplication table — by calling `print_multiples` repeatedly with different arguments. In fact, we can use another loop:

```
1 for i in range(1, 7):
2     print_multiples(i)
```

Notice how similar this loop is to the one inside `print_multiples`. All we did was replace the `print` function with a function call.

The output of this program is a multiplication table:

1	1	2	3	4	5	6
2	2	4	6	8	10	12
3	3	6	9	12	15	18
4	4	8	12	16	20	24
5	5	10	15	20	25	30
6	6	12	18	24	30	36

## 7.13. More encapsulation

To demonstrate encapsulation again, let's take the code from the last section and wrap it up in a function:

```
1 def print_mult_table():
2     for i in range(1, 7):
3         print_multiples(i)
```

This process is a common **development plan**. We develop code by writing lines of code outside any function, or typing them in to the interpreter. When we get the code working, we extract it and wrap it up in a function.

This development plan is particularly useful if you don't know how to divide the program into functions when you start writing. This approach lets you design as you go along.

## 7.14. Local variables

You might be wondering how we can use the same variable, `i`, in both `print_multiples` and `print_mult_table`. Doesn't it cause problems when one of the functions changes the value of the variable?

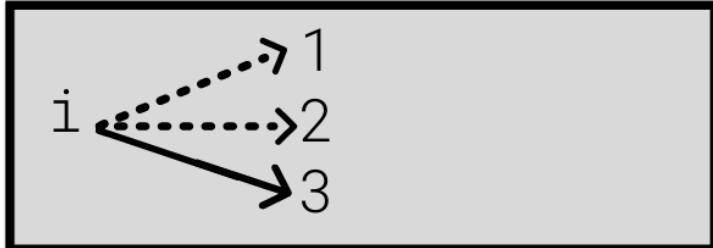
The answer is no, because the `i` in `print_multiples` and the `i` in `print_mult_table` are not the same variable.

Variables created inside a function definition are local; you can't access a local variable from outside its home function. That means you are free to have multiple variables with the same name as long as they are not in the same function.

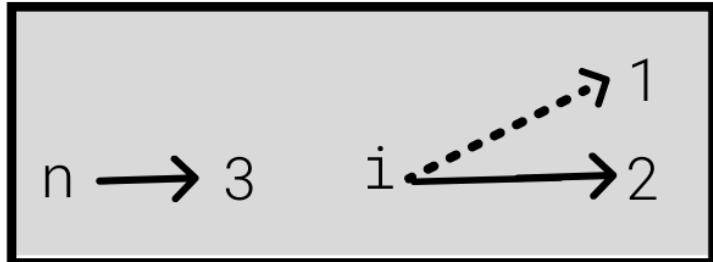
Python examines all the statements in a function — if any of them assign a value to a variable, that is the clue that Python uses to make the variable a local variable.

The stack diagram for this program shows that the two variables named `i` are not the same variable. They can refer to different values, and changing one does not affect the other.

```
print_mult_table
```



```
print_multiples
```



Stack 2 diagram

The value of *i* in `print_mult_table` goes from 1 to 6. In the diagram it happens to be 3. The next time through the loop it will be 4. Each time through the loop, `print_mult_table` calls `print_multiples` with the current value of *i* as an argument. That value gets assigned to the parameter *n*.

Inside `print_multiples`, the value of *i* goes from 1 to 6. In the diagram, it happens to be 2. Changing this variable has no effect on the value of *i* in `print_mult_table`.

It is common and perfectly legal to have different local variables with the same name. In particular, names like *i* and *j* are used frequently as loop variables. If you avoid using them in one function just because you used them somewhere else, you will probably make the program harder to read.

The visualizer at <http://pythontutor.com/> shows very clearly how the two variables *i* are distinct variables, and how they have independent values.

## 7.15. The `break` statement

The `break` statement is used to immediately leave the body of its loop. The next statement to be executed is the first one after the body:

```

1 for i in [12, 16, 17, 24, 29]:
2     if i % 2 == 1: # If the number is odd
3         break        # ... immediately exit the loop
4     print(i)
5 print("done")

```

This prints:

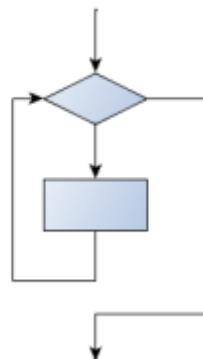
```

1 12
2 16
3 done

```

### The pre-test loop – standard loop behaviour

`for` and `while` loops do their tests at the start, before executing any part of the body. They're called **pre-test** loops, because the test happens before (pre) the body. `break` and `return` are our tools for adapting this standard behaviour.

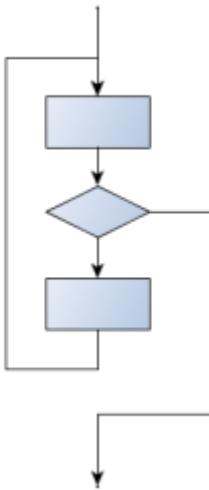


## 7.16. Other flavours of loops

Sometimes we'd like to have the **middle-test** loop with the exit test in the middle of the body, rather than at the beginning or at the end. Or a **post-test** loop that puts its exit test as the last thing in the body. Other languages have different syntax and keywords for these different flavours, but Python just uses a combination of `while` and `if` condition: `break` to get the job done.

A typical example is a problem where the user has to input numbers to be summed. To indicate that there are no more inputs, the user enters a special value, often the value `-1`, or the empty string. This needs a middle-exit loop pattern: input the next number, then test whether to exit, or else process the number:

The middle-test loop flowchart



```

1 total = 0
2 while True:
3     response = input("Enter the next number. (Leave blank to end)")
4     if response == "":
5         break
6     total += int(response)
7 print("The total of the numbers you entered is ", total)

```

Convince yourself that this fits the middle-exit loop flowchart: line 3 does some useful work, lines 4 and 5 can exit the loop, and if they don't line 6 does more useful work before the next iteration starts.

The `while bool-expr:` uses the Boolean expression to determine whether to iterate again. `True` is a trivial Boolean expression, so `while True:` means always do the loop body again. This is a language idiom — a convention that most programmers will recognize immediately. Since the expression on line 2 will never terminate the loop, (it is a dummy test) the programmer must arrange to break (or return) out of the loop body elsewhere, in some other way (i.e. in lines 4 and 5 in this sample). A clever compiler or interpreter will understand that line 2 is a fake test that must always succeed, so it won't even generate a test, and our flowchart never even put the diamond-shape dummy test box at the top of the loop!

Similarly, by just moving the `if condition: break` to the end of the loop body we create a pattern for a post-test loop. Post-test loops are used when you want to be sure that the loop body always executes at least once (because the first test only happens at the end of the execution of the first loop body). This is useful, for example, if we want to play an interactive game against the user — we always want to play at least one game:

```

1 while True:
2     play_the_game_once()
3     response = input("Play again? (yes or no)")
4     if response != "yes":
5         break
6 print("Goodbye!")

```

**Hint:** Think about where you want the exit test to happen.

Once you've recognized that you need a loop to repeat something, think about its terminating condition — when will I want to stop iterating? Then figure out whether you need to do the test before starting the first (and every other) iteration, or at the end of the first (and every other) iteration, or perhaps in the middle of each iteration. Interactive programs that require input from the user or read from files often need to exit their loops in the middle or at the end of an iteration, when it becomes clear that there is no more data to process, or the user doesn't want to play our game anymore.

## 7.17. An example

The following program implements a simple guessing game:

```

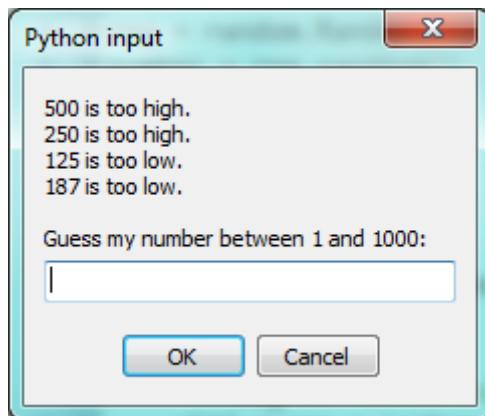
1 import random                  # We cover random numbers in the
2 rng = random.Random()          # modules chapter, so peek ahead.
3 number = rng.randrange(1, 1000) # Get random number between [1 and 1000).
4
5 guesses = 0
6 msg = ""
7
8 while True:
9     guess = int(input(msg + "\nGuess my number between 1 and 1000: "))
10    guesses += 1
11    if guess > number:
12        msg += str(guess) + " is too high.\n"
13    elif guess < number:
14        msg += str(guess) + " is too low.\n"
15    else:
16        break
17
18 input("\n\nGreat, you got it in {0} guesses!\n\n".format(guesses))

```

This program makes use of the mathematical law of **trichotomy** (given real numbers  $a$  and  $b$ , exactly one of these three must be true:  $a > b$ ,  $a < b$ , or  $a == b$ ).

At line 18 there is a call to the `input` function, but we don't do anything with the result, not even assign it to a variable. This is legal in Python. Here it has the effect of popping up the input dialog window and waiting for the user to respond before the program terminates. Programmers often use the trick of doing some extra input at the end of a script, just to keep the window open.

Also notice the use of the `msg` variable, initially an empty string, on lines 6, 12 and 14. Each time through the loop we extend the message being displayed: this allows us to display the program's feedback right at the same place as we're asking for the next guess.



## 7.18. The `continue` statement

This is a control flow statement that causes the program to immediately skip the processing of the rest of the body of the loop, for the current iteration. But the loop still carries on running for its remaining iterations:

```
1 for i in [12, 16, 17, 24, 29, 30]:  
2     if i % 2 == 1:      # If the number is odd  
3         continue        # Don't process it  
4     print(i)  
5 print("done")
```

This prints:

```
1 12  
2 16  
3 24  
4 30  
5 done
```

## 7.19. More generalization

As another example of generalization, imagine you wanted a program that would print a multiplication table of any size, not just the six-by-six table. You could add a parameter to `print_mult_table`:

```
1 def print_mult_table(high):
2     for i in range(1, high+1):
3         print_multiples(i)
```

We replaced the value 7 with the expression `high+1`. If we call `print_mult_table` with the argument 7, it displays:

```
1 1 2 3 4 5 6
2 2 4 6 8 10 12
3 3 6 9 12 15 18
4 4 8 12 16 20 24
5 5 10 15 20 25 30
6 6 12 18 24 30 36
7 7 14 21 28 35 42
```

This is fine, except that we probably want the table to be square — with the same number of rows and columns. To do that, we add another parameter to `print_multiples` to specify how many columns the table should have.

Just to be annoying, we call this parameter `high`, demonstrating that different functions can have parameters with the same name (just like local variables). Here's the whole program:

```
1 def print_multiples(n, high):
2     for i in range(1, high+1):
3         print(n * i, end="   ")
4     print()
5
6 def print_mult_table(high):
7     for i in range(1, high+1):
8         print_multiples(i, high)
```

Notice that when we added a new parameter, we had to change the first line of the function (the function heading), and we also had to change the place where the function is called in `print_mult_table`.

Now, when we call `print_mult_table(7)`:

1	1	2	3	4	5	6	7
2	2	4	6	8	10	12	14
3	3	6	9	12	15	18	21
4	4	8	12	16	20	24	28
5	5	10	15	20	25	30	35
6	6	12	18	24	30	36	42
7	7	14	21	28	35	42	49

When you generalize a function appropriately, you often get a program with capabilities you didn't plan. For example, you might notice that, because  $ab = ba$ , all the entries in the table appear twice. You could save ink by printing only half the table. To do that, you only have to change one line of `print_mult_table`. Change

```
1 print_multiples(i, high+1)
```

to

```
1 print_multiples(i, i+1)
```

and you get:

1	1						
2	2	4					
3	3	6	9				
4	4	8	12	16			
5	5	10	15	20	25		
6	6	12	18	24	30	36	
7	7	14	21	28	35	42	49

## 7.20. Functions

A few times now, we have mentioned all the things functions are good for. By now, you might be wondering what exactly those things are. Here are some of them:

1. Capturing your mental chunking. Breaking your complex tasks into sub-tasks, and giving the sub-tasks a meaningful name is a powerful mental technique. Look back at the example that illustrated the post-test loop: we assumed that we had a function called `play_the_game_once`. This chunking allowed us to put aside the details of the particular game — is it a card game, or noughts and crosses, or a role playing game — and simply focus on one isolated part of our program logic — letting the player choose whether they want to play again.

2. Dividing a long program into functions allows you to separate parts of the program, debug them in isolation, and then compose them into a whole.
3. Functions facilitate the use of iteration.
4. Well-designed functions are often useful for many programs. Once you write and debug one, you can reuse it.

## 7.21. Paired Data

We've already seen lists of names and lists of numbers in Python. We're going to peek ahead in the textbook a little, and show a more advanced way of representing our data. Making a pair of things in Python is as simple as putting them into parentheses, like this:

```
1 year_born = ("Paris Hilton", 1981)
```

We can put many pairs into a list of pairs:

```
1 celebs = [("Brad Pitt", 1963), ("Jack Nicholson", 1937),
2                               ("Justin Bieber", 1994)]
```

Here is a quick sample of things we can do with structured data like this. First, print all the celebs:

```
1 print(celebs)
2 print(len(celebs))
3 [("Brad Pitt", 1963), ("Jack Nicholson", 1937), ("Justin Bieber", 1994)]
4 3
```

Notice that the celebs list has just 3 elements, each of them pairs.

Now we print the names of those celebrities born before 1980:

```
1 for (nm, yr) in celebs:
2     if yr < 1980:
3         print(nm)
```

```
1 Brad Pitt
2 Jack Nicholson
```

This demonstrates something we have not seen yet in the `for` loop: instead of using a single loop control variable, we've used a pair of variable names, (`nm`, `yr`), instead. The loop is executed three times — once for each pair in the list, and on each iteration both the variables are assigned values from the pair of data that is being handled.

## 7.22. Nested Loops for Nested Data

Now we'll come up with an even more adventurous list of structured data. In this case, we have a list of students. Each student has a name which is paired up with another list of subjects that they are enrolled for:

```

1 students = [
2     ("John", ["CompSci", "Physics"]),
3     ("Vusi", ["Maths", "CompSci", "Stats"]),
4     ("Jess", ["CompSci", "Accounting", "Economics", "Management"]),
5     ("Sarah", ["InfSys", "Accounting", "Economics", "CommLaw"]),
6     ("Zuki", ["Sociology", "Economics", "Law", "Stats", "Music"])]
```

Here we've assigned a list of five elements to the variable `students`. Let's print out each student name, and the number of subjects they are enrolled for:

```

1 # Print all students with a count of their courses.
2 for (name, subjects) in students:
3     print(name, "takes", len(subjects), "courses")
```

Python agreeably responds with the following output:

```

1 John takes 2 courses
2 Vusi takes 3 courses
3 Jess takes 4 courses
4 Sarah takes 4 courses
5 Zuki takes 5 courses
```

Now we'd like to ask how many students are taking CompSci. This needs a counter, and for each student we need a second loop that tests each of the subjects in turn:

```

1 # Count how many students are taking CompSci
2 counter = 0
3 for (name, subjects) in students:
4     for s in subjects:           # A nested loop!
5         if s == "CompSci":
6             counter += 1
7
8 print("The number of students taking CompSci is", counter)
```

```
1 The number of students taking CompSci is 3
```

You should set up a list of your own data that interests you — perhaps a list of your CDs, each containing a list of song titles on the CD, or a list of movie titles, each with a list of movie stars who acted in the movie. You could then ask questions like “Which movies starred Angelina Jolie?”

## 7.23. Newton's method for finding square roots

Loops are often used in programs that compute numerical results by starting with an approximate answer and iteratively improving it.

For example, before we had calculators or computers, people needed to calculate square roots manually. Newton used a particularly good method (there is some evidence that this method was known many years before). Suppose that you want to know the square root of  $n$ . If you start with almost any approximation, you can compute a better approximation (closer to the actual answer) with the following formula:

```
1 better = (approx + n/approx)/2
```

Repeat this calculation a few times using your calculator. Can you see why each iteration brings your estimate a little closer? One of the amazing properties of this particular algorithm is how quickly it converges to an accurate answer — a great advantage for doing it manually.

By using a loop and repeating this formula until the better approximation gets close enough to the previous one, we can write a function for computing the square root. (In fact, this is how your calculator finds square roots — it may have a slightly different formula and method, but it is also based on repeatedly improving its guesses.)

This is an example of an indefinite iteration problem: we cannot predict in advance how many times we'll want to improve our guess — we just want to keep getting closer and closer. Our stopping condition for the loop will be when our old guess and our improved guess are “close enough” to each other.

Ideally, we'd like the old and new guess to be exactly equal to each other when we stop. But exact equality is a tricky notion in computer arithmetic when real numbers are involved. Because real numbers are not represented absolutely accurately (after all, a number like  $\pi$  or the square root of two has an infinite number of decimal places because it is irrational), we need to formulate the stopping test for the loop by asking “is  $a$  close enough to  $b$ ”? This stopping condition can be coded like this:

```
1 if abs(a-b) < 0.001: # Make this smaller for better accuracy
2     break
```

Notice that we take the absolute value of the difference between  $a$  and  $b$ !

This problem is also a good example of when a middle-exit loop is appropriate:

```
1 def sqrt(n):
2     approx = n/2.0      # Start with some or other guess at the answer
3     while True:
4         better = (approx + n/approx)/2.0
5         if abs(approx - better) < 0.001:
6             return better
7         approx = better
8
9 # Test cases
10 print(sqrt(25.0))
11 print(sqrt(49.0))
12 print(sqrt(81.0))
```

The output is:

```
1 5.00000000002
2 7.0
3 9.0
```

See if you can improve the approximations by changing the stopping condition. Also, step through the algorithm (perhaps by hand, using your calculator) to see how many iterations were needed before it achieved this level of accuracy for  $\sqrt{25}$ .

## 7.24. 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).

Some kinds of knowledge are not algorithmic. For example, learning dates from history or your multiplication tables involves memorization of specific solutions.

But the techniques you learned for addition with carrying, subtraction with borrowing, and long division are all algorithms. Or if you are an avid Sudoku puzzle solver, you might have some specific set of steps that you always follow.

One of the characteristics of algorithms is that they do not require any intelligence to carry out. They are mechanical processes in which each step follows from the last according to a simple set of rules. And they're designed to solve a general class or category of problems, not just a single problem.

Understanding that hard problems can be solved by step-by-step algorithmic processes (and having technology to execute these algorithms for us) is one of the major breakthroughs that has had enormous benefits. So while the execution of the algorithm may be boring and may require no intelligence, algorithmic or computational thinking — i.e. using algorithms and automation as the basis for approaching problems — is rapidly transforming our society. Some claim that this shift

towards algorithmic thinking and processes is going to have even more impact on our society than the invention of the printing press. And the process of designing algorithms is interesting, intellectually challenging, and a central part of what we call programming.

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 a step-by-step mechanical algorithm.

## 7.25. Glossary

### **algorithm**

A step-by-step process for solving a category of problems.

### **body**

The statements inside a loop.

### **breakpoint**

A place in your program code where program execution will pause (or break), allowing you to inspect the state of the program's variables, or single-step through individual statements, executing them one at a time.

### **bump**

Programmer slang. Synonym for increment.

### **continue statement**

A statement that causes the remainder of the current iteration of a loop to be skipped. The flow of execution goes back to the top of the loop, evaluates the condition, and if this is true the next iteration of the loop will begin.

### **counter**

A variable used to count something, usually initialized to zero and incremented in the body of a loop.

### **cursor**

An invisible marker that keeps track of where the next character will be printed.

### **decrement**

Decrease by 1.

### **definite iteration**

A loop where we have an upper bound on the number of times the body will be executed. Definite iteration is usually best coded as a `for` loop.

**development plan**

A process for developing a program. In this chapter, we demonstrated a style of development based on developing code to do simple, specific things and then encapsulating and generalizing.

**encapsulate**

To divide a large complex program into components (like functions) and isolate the components from each other (by using local variables, for example).

**escape sequence**

An escape character, \, followed by one or more printable characters used to designate a non printable character.

**generalize**

To replace something unnecessarily specific (like a constant value) with something appropriately general (like a variable or parameter). Generalization makes code more versatile, more likely to be reused, and sometimes even easier to write.

**increment**

Both as a noun and as a verb, increment means to increase by 1.

**infinite loop**

A loop in which the terminating condition is never satisfied.

**indefinite iteration**

A loop where we just need to keep going until some condition is met. A `while` statement is used for this case.

**initialization (of a variable)**

To initialize a variable is to give it an initial value. Since in Python, variables don't exist until they are assigned values, they are initialized when they are created. In other programming languages this is not the case, and variables can be created without being initialized, in which case they have either default or garbage values.

**iteration**

Repeated execution of a set of programming statements.

**loop**

The construct that allows us to repeatedly execute a statement or a group of statements until a terminating condition is satisfied.

**loop variable**

A variable used as part of the terminating condition of a loop.

**meta-notation**

Extra symbols or notation that helps describe other notation. Here we introduced square brackets, ellipses, italics, and bold as meta-notation to help describe optional, repeatable, substitutable and fixed parts of the Python syntax.

### **middle-test loop**

A loop that executes some of the body, then tests for the exit condition, and then may execute some more of the body. We don't have a special Python construct for this case, but can use `while` and `break` together.

### **nested loop**

A loop inside the body of another loop.

### **newline**

A special character that causes the cursor to move to the beginning of the next line.

### **post-test loop**

A loop that executes the body, then tests for the exit condition. We don't have a special Python construct for this, but can use `while` and `break` together.

### **pre-test loop**

A loop that tests before deciding whether to execute its body. `for` and `while` are both pre-test loops.

### **single-step**

A mode of interpreter execution where you are able to execute your program one step at a time, and inspect the consequences of that step. Useful for debugging and building your internal mental model of what is going on.

### **tab**

A special character that causes the cursor to move to the next tab stop on the current line.

### **trichotomy**

Given any real numbers  $a$  and  $b$ , exactly one of the following relations holds:  $a < b$ ,  $a > b$ , or  $a == b$ . Thus when you can establish that two of the relations are false, you can assume the remaining one is true.

### **trace**

To follow the flow of execution of a program by hand, recording the change of state of the variables and any output produced.

## **7.26. Exercises**

This chapter showed us how to sum a list of items, and how to count items. The counting example also had an `if` statement that let us only count some selected items. In the previous chapter we also

showed a function `find_first_2_letter_word` that allowed us an “early exit” from inside a loop by using `return` when some condition occurred. We now also have `break` to exit a loop but not the enclosing function, and `continue` to abandon the current iteration of the loop without ending the loop.

Composition of list traversal, summing, counting, testing conditions and early exit is a rich collection of building blocks that can be combined in powerful ways to create many functions that are all slightly different.

The first six questions are typical functions you should be able to write using only these building blocks.

1. Write a function to count how many odd numbers are in a list.
2. Sum up all the even numbers in a list.
3. Sum up all the negative numbers in a list.
4. Count how many words in a list have length 5.
5. Sum all the elements in a list up to but not including the first even number. (Write your unit tests. What if there is no even number?)
6. Count how many words occur in a list up to and including the first occurrence of the word “sam”. (Write your unit tests for this case too. What if “sam” does not occur?)
7. Add a print function to Newton’s `sqrt` function that prints out better each time it is calculated. Call your modified function with 25 as an argument and record the results.
8. Trace the execution of the last version of `print_mult_table` and figure out how it works.
9. Write a function `print_triangular_numbers(n)` that prints out the first n triangular numbers. A call to `print_triangular_numbers(5)` would produce the following output:

1	1	1
2	2	3
3	3	6
4	4	10
5	5	15

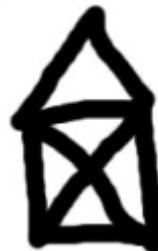
(*hint: use a web search to find out what a triangular number is.*)

10. Write a function, `is_prime`, which takes a single integer argument and returns True when the argument is a prime number and False otherwise. Add tests for cases like this:

```
1 test(is_prime(11))
2 test(not is_prime(35))
3 test(is_prime(19911121))
```

The last case could represent your birth date. Were you born on a prime day? In a class of 100 students, how many do you think would have prime birth dates?

11. Revisit the drunk pirate problem from the exercises in chapter 3. This time, the drunk pirate makes a turn, and then takes some steps forward, and repeats this. Our social science student now records pairs of data: the angle of each turn, and the number of steps taken after the turn. Her experimental data is  $[(160, 20), (-43, 10), (270, 8), (-43, 12)]$ . Use a turtle to draw the path taken by our drunk friend.
12. Many interesting shapes can be drawn by the turtle by giving a list of pairs like we did above, where the first item of the pair is the angle to turn, and the second item is the distance to move forward. Set up a list of pairs so that the turtle draws a house with a cross through the centre, as show here. This should be done without going over any of the lines / edges more than once, and without lifting your pen.



13. Not all shapes like the one above can be drawn without lifting your pen, or going over an edge more than once. Which of these can be drawn?



Now read Wikipedia's article ([http://en.wikipedia.org/wiki/Eulerian\\_path](http://en.wikipedia.org/wiki/Eulerian_path)) about Eulerian paths. Learn how to tell immediately by inspection whether it is possible to find a solution or not. If the path is possible, you'll also know where to put your pen to start drawing, and where you should end up!

14. What will `num_digits(0)` return? Modify it to return 1 for this case. Why does a call to `num_digits(-24)` result in an infinite loop? (hint:  $-1//10$  evaluates to -1) Modify `num_digits` so that it works correctly with any integer value. Add these tests:

```
1 test(num_digits(0) == 1)
2 test(num_digits(-12345) == 5)
```

15. Write a function `num_even_digits(n)` that counts the number of even digits in `n`. These tests should pass:

```

1 test(num_even_digits(123456) == 3)
2 test(num_even_digits(2468) == 4)
3 test(num_even_digits(1357) == 0)
4 test(num_even_digits(0) == 1)

```

16. Write a function `sum_of_squares(xs)` that computes the sum of the squares of the numbers in the list `xs`. For example, `sum_of_squares([2, 3, 4])` should return `4+9+16` which is 29:

```

1 test(sum_of_squares([2, 3, 4]) == 29)
2 test(sum_of_squares([]) == 0)
3 test(sum_of_squares([2, -3, 4]) == 29)

```

17. You and your friend are in a team to write a two-player game, human against computer, such as Tic-Tac-Toe / Noughts and Crosses. Your friend will write the logic to play one round of the game, while you will write the logic to allow many rounds of play, keep score, decide who plays, first, etc. The two of you negotiate on how the two parts of the program will fit together, and you come up with this simple scaffolding (which your friend will improve later):

```

1 # Your friend will complete this function
2 def play_once(human_plays_first):
3     """
4         Must play one round of the game. If the parameter
5             is True, the human gets to play first, else the
6                 computer gets to play first. When the round ends,
7                     the return value of the function is one of
8                         -1 (human wins), 0 (game drawn), 1 (computer wins).
9     """
10    # This is all dummy scaffolding code right at the moment...
11    import random                      # See Modules chapter ...
12    rng = random.Random()
13    # Pick a random result between -1 and 1.
14    result = rng.randrange(-1,2)
15    print("Human plays first={0}, winner={1} "
16          .format(human_plays_first, result))
17    return result

```

- Write the main program which repeatedly calls this function to play the game, and after each round it announces the outcome as “I win!”, “You win!”, or “Game drawn!”. It then asks the player “Do you want to play again?” and either plays again, or says “Goodbye”, and terminates.
- Keep score of how many wins each player has had, and how many draws there have been. After each round of play, also announce the scores.
- Add logic so that the players take turns to play first.
- Compute the percentage of wins for the human, out of all games played. Also announce this at the end of each round.

- e. Draw a flowchart of your logic.

# Chapter 8: Strings

## 8.1. A compound data type

So far we have seen built-in types like `int`, `float`, `bool`, `str` and we've seen lists and pairs. Strings, lists, and pairs are qualitatively different from the others because they are made up of smaller pieces. In the case of strings, they're made up of smaller strings each containing one character

Types that comprise smaller pieces are called **compound data types**. Depending on what we are doing, we may want to treat a compound data type as a single thing, or we may want to access its parts. This ambiguity is useful.

## 8.2. Working with strings as single things

We previously saw that each turtle instance has its own attributes and a number of methods that can be applied to the instance. For example, we could set the turtle's color, and we wrote `tess.turn(90)`.

Just like a turtle, a string is also an object. So each string instance has its own attributes and methods.

For example:

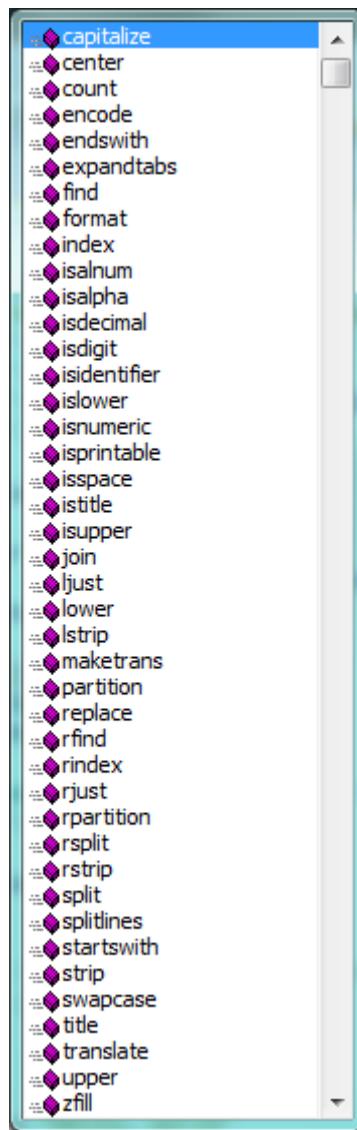
```
1 >>> ss = "Hello, World!"  
2 >>> tt = ss.upper()  
3 >>> tt  
4 'HELLO, WORLD!'
```

`upper` is a method that can be invoked on any string object to create a new string, in which all the characters are in uppercase. (The original string `ss` remains unchanged.)

There are also methods such as `lower`, `capitalize`, and `swapcase` that do other interesting stuff.

To learn what methods are available, you can consult the Help documentation, look for string methods, and read the documentation. Or, if you're a bit lazier, simply type the following into a PyScripter script:

When you type the period to select one of the methods of `ss`, PyScripter will pop up a selection window showing all the methods (there are around 70 of them — thank goodness we'll only use a few of those!) that could be used on your string.



When you type the name of the method, some further help about its parameter and return type, and its docstring, will be displayed. This is a good example of a tool — PyScripter — using the meta-information — the docstrings — provided by the module programmers.

```
greet = "Hello, World"
xx= greet.swapcase()
print(xx)
```

\*\* No/Unknown parameters \*\*  
S.swapcase() -> str  
  
Return a copy of S with uppercase characters converted to lowercase  
and vice versa.

## 8.3. Working with the parts of a string

The **indexing operator** (Python uses square brackets to enclose the index) selects a single character substring from a string:

```
1 >>> fruit = "banana"  
2 >>> m = fruit[1]  
3 >>> print(m)
```

The expression `fruit[1]` selects character number 1 from `fruit`, and creates a new string containing just this one character. The variable `m` refers to the result. When we display `m`, we could get a surprise:

```
1 a
```

Computer scientists always start counting from zero! The letter at subscript position zero of "banana" is `b`. So at position [1] we have the letter `a`.

If we want to access the zero-eth letter of a string, we just place 0, or any expression that evaluates to 0, inbetween the brackets:

```
1 >>> m = fruit[0]  
2 >>> print(m)  
3 b
```

The expression in brackets is called an **index**. An index specifies a member of an ordered collection, in this case the collection of characters in the string. The index *indicates* which one you want, hence the name. It can be any integer expression.

We can use `enumerate` to visualize the indices:

```
1 >>> fruit = "banana"  
2 >>> list(enumerate(fruit))  
3 [(0, 'b'), (1, 'a'), (2, 'n'), (3, 'a'), (4, 'n'), (5, 'a')]
```

Do not worry about `enumerate` at this point, we will see more of it in the chapter on lists.

Note that indexing returns a *string* — Python has no special type for a single character. It is just a string of length 1.

We've also seen lists previously. The same indexing notation works to extract elements from a list:

```
1 >>> prime_nums = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31]
2 >>> prime_nums[4]
3 11
4 >>> friends = ["Joe", "Zoe", "Brad", "Angelina", "Zuki", "Thandi", "Paris"]
5 >>> friends[3]
6 'Angelina'
```

## 8.4. Length

The `len` function, when applied to a string, returns the number of characters in a string:

```
1 >>> fruit = "banana"
2 >>> len(fruit)
3 6
```

To get the last letter of a string, you might be tempted to try something like this:

That won't work. It causes the runtime error `IndexError: string index out of range`. The reason is that there is no character at index position 6 in "banana". Because we start counting at zero, the six indexes are numbered 0 to 5. To get the last character, we have to subtract 1 from the length of `fruit`:

Alternatively, we 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.

As you might have guessed, indexing with a negative index also works like this for lists.

We won't use negative indexes in the rest of these notes — not many computer languages use this idiom, and you'll probably be better off avoiding it. But there is plenty of Python code out on the Internet that will use this trick, so it is best to know that it exists.

## 8.5. Traversal and the `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 encode a traversal is with a `while` statement:

This loop traverses the string and displays each letter on a line by itself. The loop condition is `ix < len(fruit)`, so when `ix` is equal to the length of the string, the condition is false, and the body of the loop is not executed. The last character accessed is the one with the index `len(fruit)-1`, which is the last character in the string.

But we've previously seen how the `for` loop can easily iterate over the elements in a list and it can do so for strings as well:

Each time through the loop, the next character in the string is assigned to the variable `c`. The loop continues until no characters are left. Here we can see the expressive power the `for` loop gives us compared to the `while` loop when traversing a string.

The following example shows how to use concatenation and a `for` loop to generate an abecedarian series. Abecedarian refers to a series or list in which the elements appear in alphabetical order. For example, 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:

The output of this program is:

```
1 Jack
2 Kack
3 Lack
4 Mack
5 Nack
6 Oack
7 Pack
8 Qack
```

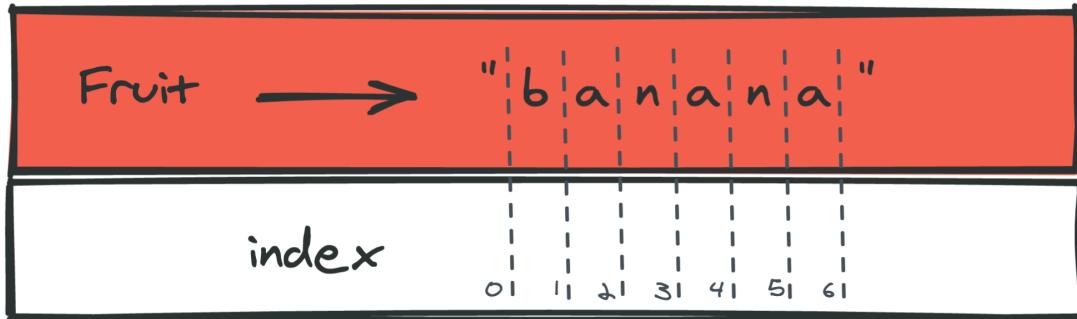
Of course, that's not quite right because Ouack and Quack are misspelled. You'll fix this as an exercise below.

## 8.6. Slices

A *substring* of a string is obtained by taking a **slice**. Similarly, we can slice a list to refer to some sublist of the items in the list:

```
1 >>> s = "Pirates of the Caribbean"
2 >>> print(s[0:7])
3 Pirates
4 >>> print(s[11:14])
5 the
6 >>> print(s[15:24])
7 Caribbean
8 >>> friends = ["Joe", "Zoe", "Brad", "Angelina", "Zuki", "Thandi", "Paris"]
9 >>> print(friends[2:4])
10 ['Brad', 'Angelina']
```

The operator `[n:m]` returns the part of the string from the `n`'th character to the `m`'th character, including the first but excluding the last. This behavior makes sense if you imagine the indices pointing *between* the characters, as in the following diagram:



If you imagine this as a piece of paper, the slice operator `[n:m]` copies out the part of the paper between the `n` and `m` positions. Provided `m` and `n` are both within the bounds of the string, your result will be of length  $(m-n)$ .

Three tricks are added to this: if you omit the first index (before the colon), the slice starts at the beginning of the string (or list). If you omit the second index, the slice extends to the end of the string (or list). Similarly, if you provide value for `n` that is bigger than the length of the string (or list), the slice will take all the values up to the end. (It won't give an "out of range" error like the normal indexing operation does.) Thus:

```

1 >>> fruit = "banana"
2 >>> fruit[:3]
3 'ban'
4 >>> fruit[3:]
5 'ana'
6 >>> fruit[3:999]
7 'ana'
```

What do you think `s[:]` means? What about `friends[4:]`?

## 8.7. String comparison

The comparison operators work on strings. To see if two strings are equal:

Other comparison operations are useful for putting words in lexicographical order:

This is similar to the alphabetical order you would use with a dictionary, except that all the uppercase letters come before all the lowercase letters. As a result:

```
1 Your word, Zebra, 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. A more difficult problem is making the program realize that zebras are not fruit.

## 8.8. 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:

Instead of producing the output Jello, world!, this code produces the runtime error `TypeError: 'str' object does not support item assignment.`

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:

The solution here is to concatenate a new first letter onto a slice of `greeting`. This operation has no effect on the original string.

## 8.9. The `in` and `not in` operators

The `in` operator tests for membership. When both of the arguments to `in` are strings, `in` checks whether the left argument is a substring of the right argument.

```
1 >>> "p" in "apple"
2 True
3 >>> "i" in "apple"
4 False
5 >>> "ap" in "apple"
6 True
7 >>> "pa" in "apple"
8 False
```

Note that a string is a substring of itself, and the empty string is a substring of any other string. (Also note that computer scientists like to think about these edge cases quite carefully!)

```

1 >>> "a" in "a"
2 True
3 >>> "apple" in "apple"
4 True
5 >>> "" in "a"
6 True
7 >>> "" in "apple"
8 True

```

The `not in` operator returns the logical opposite results of `in`:

```

1 >>> "x" not in "apple"
2 True

```

Combining the `in` operator with string concatenation using `+`, we can write a function that removes all the vowels from a string:

```

1 def remove_vowels(s):
2     vowels = "aeiouAEIOU"
3     s_sans_vowels = ""
4     for x in s:
5         if x not in vowels:
6             s_sans_vowels += x
7     return s_sans_vowels
8
9 test(remove_vowels("compsci") == "cmpsci")
10 test(remove_vowels("aAbEefIijOopUus") == "bfjps")

```

## 8.10. A find function

What does the following function do?

```

1 def find(strng, ch):
2     """
3         Find and return the index of ch in strng.
4         Return -1 if ch does not occur in strng.
5     """
6     ix = 0
7     while ix < len(strng):
8         if strng[ix] == ch:
9             return ix

```

```
10     ix += 1
11     return -1
12
13 test(find("CompSci", "p") == 3)
14 test(find("CompSci", "C") == 0)
15 test(find("CompSci", "i") == 6)
16 test(find("CompSci", "x") == -1)
```

In a sense, `find` is the opposite of the indexing 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 another example where we see a `return` statement inside a loop. If `strng[ix] == ch`, the function returns immediately, breaking out of the loop prematurely.

If the character doesn't appear in the string, then the program exits the loop normally and returns `-1`.

This pattern of computation is sometimes called a **eureka traversal** or **short-circuit evaluation**, because as soon as we find what we are looking for, we can cry “Eureka!”, take the short-circuit, and stop looking.

## 8.11. Looping and counting

The following program counts the number of times the letter `a` appears in a string, and is another example of the counter pattern introduced in counting.

```
1 def count_a(text):
2     count = 0
3     for c in text:
4         if c == "a":
5             count += 1
6     return(count)
7
8 test(count_a("banana") == 3)
```

## 8.12. Optional parameters

To find the locations of the second or third occurrence of a character in a string, we can modify the `find` function, adding a third parameter for the starting position in the search string:

```

1 def find2(strng, ch, start):
2     ix = start
3     while ix < len(strng):
4         if strng[ix] == ch:
5             return ix
6         ix += 1
7     return -1
8
9 test(find2("banana", "a", 2) == 3)

```

The call `find2("banana", "a", 2)` now returns 3, the index of the first occurrence of “a” in “banana” starting the search at index 2. What does `find2("banana", "n", 3)` return? If you said, 4, there is a good chance you understand how `find2` works.

Better still, we can combine `find` and `find2` using an **optional parameter**:

```

1 def find(strng, ch, start=0):
2     ix = start
3     while ix < len(strng):
4         if strng[ix] == ch:
5             return ix
6         ix += 1
7     return -1

```

When a function has an optional parameter, the caller may provide a matching argument. If the third argument is provided to `find`, it gets assigned to `start`. But if the caller leaves the argument out, then `start` is given a default value indicated by the assignment `start=0` in the function definition.

So the call `find("banana", "a", 2)` to this version of `find` behaves just like `find2`, while in the call `find("banana", "a")`, `start` will be set to the **default value** of 0.

Adding another optional parameter to `find` makes it search from a starting position, up to but not including the end position:

```

1 def find(strng, ch, start=0, end=None):
2     ix = start
3     if end is None:
4         end = len(strng)
5     while ix < end:
6         if strng[ix] == ch:
7             return ix
8         ix += 1
9     return -1

```

The optional value for `end` is interesting: we give it a default value `None` if the caller does not supply any argument. In the body of the function we test what `end` is, and if the caller did not supply any argument, we reassign `end` to be the length of the string. If the caller has supplied an argument for `end`, however, the caller's value will be used in the loop.

The semantics of `start` and `end` in this function are precisely the same as they are in the `range` function.

Here are some test cases that should pass:

```
1 ss = "Python strings have some interesting methods."
2 test(find(ss, "s") == 7)
3 test(find(ss, "s", 7) == 7)
4 test(find(ss, "s", 8) == 13)
5 test(find(ss, "s", 8, 13) == -1)
6 test(find(ss, ".") == len(ss)-1)
```

## 8.13. The built-in `find` method

Now that we've done all this work to write a powerful `find` function, we can reveal that strings already have their own built-in `find` method. It can do everything that our code can do, and more!

The built-in `find` method is more general than our version. It can find substrings, not just single characters:

```
1 >>> "banana".find("nan")
2
3 >>> "banana".find("na", 3)
4 4
```

Usually we'd prefer to use the methods that Python provides rather than reinvent our own equivalents. But many of the built-in functions and methods make good teaching exercises, and the underlying techniques you learn are your building blocks to becoming a proficient programmer.

## 8.14. The `split` method

One of the most useful methods on strings is the `split` method: it splits a single multi-word string into a list of individual words, removing all the whitespace between them. (Whitespace means any tabs, newlines, or spaces.) This allows us to read input as a single string, and split it into words.

```

1 >>> ss = "Well I never did said Alice"
2 >>> wds = ss.split()
3 >>> wds
4 ['Well', 'I', 'never', 'did', 'said', 'Alice']

```

## 8.15. Cleaning up your strings

We'll often work with strings that contain punctuation, or tab and newline characters, especially, as we'll see in a future chapter, when we read our text from files or from the Internet. But if we're writing a program, say, to count word frequencies or check the spelling of each word, we'd prefer to strip off these unwanted characters.

We'll show just one example of how to strip punctuation from a string. Remember that strings are immutable, so we cannot change the string with the punctuation — we need to traverse the original string and create a new string, omitting any punctuation:

```

1 punctuation = !"#$%&'()*+,-./:;=>?@[\\]^_`{|}~"
2
3 def remove_punctuation(s):
4     s_sans_punct = ""
5     for letter in s:
6         if letter not in punctuation:
7             s_sans_punct += letter
8     return s_sans_punct

```

Setting up that first assignment is messy and error-prone. Fortunately, the Python `string` module already does it for us. So we will make a slight improvement to this program — we'll import the `string` module and use its definition:

```

1 import string
2
3 def remove_punctuation(s):
4     s_without_punct = ""
5     for letter in s:
6         if letter not in string.punctuation:
7             s_without_punct += letter
8     return s_without_punct
9
10 test(remove_punctuation('"Well, I never did!', said Alice.')) ==
11                 "Well I never did said Alice")
12 test(remove_punctuation("Are you very, very, sure?")) ==
13                 "Are you very very sure")

```

Composing together this function and the `split` method from the previous section makes a useful combination — we'll clean out the punctuation, and `split` will clean out the newlines and tabs while turning the string into a list of words:

```
1 my_story = """
2 Pythons are constrictors, which means that they will 'squeeze' the life
3 out of their prey. They coil themselves around their prey and with
4 each breath the creature takes the snake will squeeze a little tighter
5 until they stop breathing completely. Once the heart stops the prey
6 is swallowed whole. The entire animal is digested in the snake's
7 stomach except for fur or feathers. What do you think happens to the fur,
8 feathers, beaks, and eggshells? The 'extra stuff' gets passed out as ---
9 you guessed it --- snake POOP! """
10
11 wds = remove_punctuation(my_story).split()
12 print(wds)
```

The output:

```
1 ['Pythons', 'are', 'constrictors', ..., 'it', 'snake', 'POOP']
```

There are other useful string methods, but this book isn't intended to be a reference manual. On the other hand, the *Python Library Reference* is. Along with a wealth of other documentation, it is available at the [Python website](https://www.python.org/)<sup>9</sup>.

## 8.16. The string format method

The easiest and most powerful way to format a string in Python 3 is to use the `format` method. To see how this works, let's start with a few examples:

```
1 s1 = "His name is {0}!".format("Arthur")
2 print(s1)
3
4 name = "Alice"
5 age = 10
6 s2 = "I am {1} and I am {0} years old.".format(age, name)
7 print(s2)
8
9 n1 = 4
10 n2 = 5
11 s3 = "2**10 = {0} and {1} * {2} = {3:f}".format(2**10, n1, n2, n1 * n2)
12 print(s3)
```

---

<sup>9</sup><https://www.python.org/>

Running the script produces:

```
1 His name is Arthur!
2 I am Alice and I am 10 years old.
3 2**10 = 1024 and 4 * 5 = 20.000000
```

The template string contains place holders, `... {0} ... {1} ... {2} ...` etc. The `format` method substitutes its arguments into the place holders. The numbers in the place holders are indexes that determine which argument gets substituted — make sure you understand line 6 above!

But there's more! Each of the replacement fields can also contain a **format specification** — it is always introduced by the `:` symbol (Line 11 above uses one.) This modifies how the substitutions are made into the template, and can control things like:

- whether the field is aligned to the left `<`, center `^`, or right `>`
- the width allocated to the field within the result string (a number like `10`)
- the type of conversion (we'll initially only force conversion to float, `f`, as we did in line 11 of the code above, or perhaps we'll ask integer numbers to be converted to hexadecimal using `x`)
- if the type conversion is a float, you can also specify how many decimal places are wanted (typically, `.2f` is useful for working with currencies to two decimal places.)

Let's do a few simple and common examples that should be enough for most needs. If you need to do anything more esoteric, use help and read all the powerful, gory details.

```
1 n1 = "Paris"
2 n2 = "Whitney"
3 n3 = "Hilton"
4
5 print("Pi to three decimal places is {0:.3f}".format(3.1415926))
6 print("123456789 123456789 123456789 123456789 123456789 123456789")
7 print("|||{0:<15}|||{1:^15}|||{2:>15}|||Born in {3}|||"
8     .format(n1,n2,n3,1981))
9 print("The decimal value {0} converts to hex value {0:x}"
10     .format(123456))
```

This script produces the output:

```
1 Pi to three decimal places is 3.142
2 123456789 123456789 123456789 123456789 123456789 123456789
3 |||Paris|||Whitney|||Hilton|||Born in 1981|||
4 The decimal value 123456 converts to hex value 1e240
```

You can have multiple placeholders indexing the same argument, or perhaps even have extra arguments that are not referenced at all:

```
1 letter = """
2 Dear {0} {2}.
3 {0}, I have an interesting money-making proposition for you!
4 If you deposit $10 million into my bank account, I can
5 double your money ...
6 """
7
8 print(letter.format("Paris", "Whitney", "Hilton"))
9 print(letter.format("Bill", "Henry", "Gates"))
```

This produces the following:

```
1 Dear Paris Hilton.
2 Paris, I have an interesting money-making proposition for you!
3 If you deposit $10 million into my bank account, I can
4 double your money ...
5
6
7 Dear Bill Gates.
8 Bill, I have an interesting money-making proposition for you!
9 If you deposit $10 million into my bank account I can
10 double your money ...
```

As you might expect, you'll get an index error if your placeholders refer to arguments that you do not provide:

```
1 >>> "hello {3}".format("Dave")
2 Traceback (most recent call last):
3   File "<interactive input>", line 1, in <module>
4     IndexError: tuple index out of range
```

The following example illustrates the real utility of string formatting. First, we'll try to print a table without using string formatting:

```
1 print("i\ti**2\ti**3\ti**5\ti**10\ti**20")
2 for i in range(1, 11):
3     print(i, "\t", i**2, "\t", i**3, "\t", i**5, "\t",
4           i**10, "\t", i**20)
```

This program prints out a table of various powers of the numbers from 1 to 10. (This assumes that the tab width is 8. You might see something even worse than this if you tab width is set to 4.) In its current form it relies on the tab character (\t) to align the columns of values, but this breaks down when the values in the table get larger than the tab width:

```

1 i      i**2    i**3    i**5    i**10   i**20
2 1      1       1       1       1       1
3 2      4       8       32      1024    1048576
4 3      9       27      243     59049   3486784401
5 4      16      64      1024    1048576   1099511627776
6 5      25      125     3125    9765625   95367431640625
7 6      36      216     7776    60466176   3656158440062976
8 7      49      343     16807   282475249   79792266297612001
9 8      64      512     32768   1073741824   1152921504606846976
10 9     81      729     59049   3486784401   12157665459056928801
11 10    100     1000    100000  10000000000000000000000000000000

```

One possible solution would be to change the tab width, but the first column already has more space than it needs. The best solution would be to set the width of each column independently. As you may have guessed by now, string formatting provides a much nicer solution. We can also right-justify each field:

```

1 layout = "{0:>4}{1:>6}{2:>6}{3:>8}{4:>13}{5:>24}"
2
3 print(layout.format("i", "i**2", "i**3", "i**5", "i**10", "i**20"))
4 for i in range(1, 11):
5     print(layout.format(i, i**2, i**3, i**5, i**10, i**20))

```

Running this version produces the following (much more satisfying) output:

```

1 i  i**2  i**3    i**5    i**10           i**20
2 1  1    1     1     1               1
3 2  4    8     32    1024            1048576
4 3  9    27    243   59049           3486784401
5 4  16   64    1024  1048576        1099511627776
6 5  25   125   3125  9765625        95367431640625
7 6  36   216   7776  60466176        3656158440062976
8 7  49   343   16807 282475249        79792266297612001
9 8  64   512   32768 1073741824        1152921504606846976
10 9  81   729   59049 3486784401        12157665459056928801
11 10 100  1000  100000 10000000000000000000000000000000

```

## 8.17. Summary

This chapter introduced a lot of new ideas. The following summary may prove helpful in remembering what you learned.

**indexing ([])**

Access a single character in a string using its position (starting from 0). Example: "This"[2] evaluates to "i".

**length function (len)**

Returns the number of characters in a string. Example: len("happy") evaluates to 5.

**for loop traversal (for)**

*Traversing* a string means accessing each character in the string, one at a time. For example, the following for loop:

```
1 for ch in "Example":  
2     ...
```

executes the body of the loop 7 times with different values of ch each time.

**slicing ([:])**

A *slice* is a substring of a string. Example: 'bananas and cream'[3:6] evaluates to ana (so does 'bananas and cream'[1:4]).

**string comparison (>, <, >=, <=, ==, !=)**

The six common comparison operators work with strings, evaluating according to lexicographical order. Examples:

"apple" < "banana" evaluates to True. "Zeta" < "Appricot" evaluates to False. "Zebra" <= "aardvark" evaluates to True because all upper case letters precede lower case letters.

**in and not in operator (in, not in)**

The in operator tests for membership. In the case of strings, it tests whether one string is contained inside another string. Examples: "heck" in "I'll be checking for you." evaluates to True. "cheese" in "I'll be checking for you." evaluates to False.

## 8.18. Glossary

**compound data type**

A data type in which the values are made up of components, or elements, that are themselves values.

**default value**

The value given to an optional parameter if no argument for it is provided in the function call.

**docstring**

A string constant on the first line of a function or module definition (and as we will see later, in class and method definitions as well).

Docstrings provide a convenient way to associate documentation with code. Docstrings are also used by programming tools to provide interactive help.

**dot notation**

Use of the **dot operator**, `.`, to access methods and attributes of an object.

**immutable data value**

A data value which cannot be modified. Assignments to elements or slices (sub-parts) of immutable values cause a runtime error.

**index**

A variable or value used to select a member of an ordered collection, such as a character from a string, or an element from a list.

**mutable data value**

A data value which can be modified. The types of all mutable values are compound types. Lists and dictionaries are mutable; strings and tuples are not.

**optional parameter**

A parameter written in a function header with an assignment to a default value which it will receive if no corresponding argument is given for it in the function call.

**short-circuit evaluation**

A style of programming that shortcuts extra work as soon as the outcome is known with certainty. In this chapter our `find` function returned as soon as it found what it was looking for; it didn't traverse all the rest of the items in the string.

**slice**

A part of a string (substring) specified by a range of indices. More generally, a subsequence of any sequence type in Python can be created using the slice operator (`sequence[start:stop]`).

**traverse**

To iterate through the elements of a collection, performing a similar operation on each.

**whitespace**

Any of the characters that move the cursor without printing visible characters. The constant `string.whitespace` contains all the white-space characters.

## 8.19. Exercises

We suggest you create a single file containing the test scaffolding from our previous chapters, and put all functions that require tests into that file.

1. What is the result of each of the following:

```
1 >>> "Python"[1]
2 >>> "Strings are sequences of characters." [5]
3 >>> len("wonderful")
4 >>> "Mystery"[:4]
5 >>> "p" in "Pineapple"
6 >>> "apple" in "Pineapple"
7 >>> "pear" not in "Pineapple"
8 >>> "apple" > "pineapple"
9 >>> "pineapple" < "Peach"
```

2. Modify:

```
1 prefixes = "JKLMNOPQ"
2 suffix = "ack"
3
4 for letter in prefixes:
5     print(letter + suffix)
```

so that Ouack and Quack are spelled correctly.

3. Encapsulate

```
1 fruit = "banana"
2 count = 0
3 for char in fruit:
4     if char == "a":
5         count += 1
6 print(count)
```

in a function named `count_letters`, and generalize it so that it accepts the string and the letter as arguments. Make the function return the number of characters, rather than print the answer. The caller should do the printing.

4. Now rewrite the `count_letters` function so that instead of traversing the string, it repeatedly calls the `find` method, with the optional third parameter to locate new occurrences of the letter being counted.
5. Assign to a variable in your program a triple-quoted string that contains your favourite paragraph of text — perhaps a poem, a speech, instructions to bake a cake, some inspirational verses, etc.

Write a function which removes all punctuation from the string, breaks the string into a list of words, and counts the number of words in your text that contain the letter “e”. Your program should print an analysis of the text like this:

1 Your text contains 243 words, of which 109 (44.8%) contain an "e".

6. Print a neat looking multiplication table like this:

```
1      1  2  3  4  5  6  7  8  9  10 11 12
2      :-----.
3      1:    1  2  3  4  5  6  7  8  9  10 11 12
4      2:    2  4  6  8 10 12 14 16 18 20 22 24
5      3:    3  6  9 12 15 18 21 24 27 30 33 36
6      4:    4  8 12 16 20 24 28 32 36 40 44 48
7      5:    5 10 15 20 25 30 35 40 45 50 55 60
8      6:    6 12 18 24 30 36 42 48 54 60 66 72
9      7:    7 14 21 28 35 42 49 56 63 70 77 84
10     8:    8 16 24 32 40 48 56 64 72 80 88 96
11     9:    9 18 27 36 45 54 63 72 81 90 99 108
12    10:   10 20 30 40 50 60 70 80 90 100 110 120
13    11:   11 22 33 44 55 66 77 88 99 110 121 132
14    12:   12 24 36 48 60 72 84 96 108 120 132 144
```

7. Write a function that reverses its string argument, and satisfies these tests:

```
1 test(reverse("happy") == "yppah")
2 test(reverse("Python") == "nohtyP")
3 test(reverse("") == "")
4 test(reverse("a") == "a")
```

8. Write a function that mirrors its argument:

```
1 test(mirror("good") == "gooddoog")
2 test(mirror("Python") == "PythonnohtyP")
3 test(mirror("") == "")
4 test(mirror("a") == "aa")
```

9. Write a function that removes all occurrences of a given letter from a string:

```
1 test(remove_letter("a", "apple") == "pple")
2 test(remove_letter("a", "banana") == "bnn")
3 test(remove_letter("z", "banana") == "banana")
4 test(remove_letter("i", "Mississippi") == "Msssspp")
5 test(remove_letter("b", "") = "")
6 test(remove_letter("b", "c") = "c")
```

10. Write a function that recognizes palindromes. (Hint: use your `reverse` function to make this easy!):

```
1 test(is_palindrome("abba"))
2 test(not is_palindrome("abab"))
3 test(is_palindrome("tenet"))
4 test(not is_palindrome("banana"))
5 test(is_palindrome("straw warts"))
6 test(is_palindrome("a"))
7 # test(is_palindrome(""))      # Is an empty string a palindrome?
```

11. Write a function that counts how many times a substring occurs in a string:

```
1 test(count("is", "Mississippi") == 2)
2 test(count("an", "banana") == 2)
3 test(count("ana", "banana") == 2)
4 test(count("nana", "banana") == 1)
5 test(count("nanan", "banana") == 0)
6 test(count("aaa", "aaaaaa") == 4)
```

12. Write a function that removes the first occurrence of a string from another string:

```
1 test(remove("an", "banana") == "bana")
2 test(remove("cyc", "bicycle") == "bile")
3 test(remove("iss", "Mississippi") == "Mississippi")
4 test(remove("eggs", "bicycle") == "bicycle")
```

13. Write a function that removes all occurrences of a string from another string:

```
1 test(remove_all("an", "banana") == "ba")
2 test(remove_all("cyc", "bicycle") == "bile")
3 test(remove_all("iss", "Mississippi") == "Mippi")
4 test(remove_all("eggs", "bicycle") == "bicycle")
```

# Chapter 9: Tuples

## 9.1. Tuples are used for grouping data

We saw earlier that we could group together pairs of values by surrounding with parentheses. Recall this example:

```
1 >>> year_born = ("Paris Hilton", 1981)
```

This is an example of a **data structure** - a mechanism for grouping and organizing data to make it easier to use.

The pair is an example of a **tuple**. Generalizing this, a tuple can be used to group any number of items into a single compound value. Syntactically, a tuple is a comma-separated sequence of values. Although it is not necessary, it is conventional to enclose tuples in parentheses:

```
1 >>> julia = ("Julia", "Roberts", 1967, "Duplicity", 2009, "Actress", "Atlanta, Georg\\
2 ia")
```

Tuples are useful for representing what other languages often call *records* -some related information that belongs together, like your student record. There is no description of what each of these fields means, but we can guess. A tuple lets us "chunk" together related information and use it as a single thing.

Tuples support the same sequence operations as strings. The index operator selects an element from a tuple.

```
1 >>> julia[2]
2 1967
```

But if we try to use item assignment to modify one of the elements of the tuple, we get an error:

```
1 >>> julia[0] = "X"
2 TypeError: 'tuple' object does not support item assignment
```

So like strings, tuples are immutable. Once Python has created a tuple in memory, it cannot be changed.

Of course, even if we can't modify the elements of a tuple, we can always make the `julia` variable reference a new tuple holding different information. To construct the new tuple, it is convenient that we can slice parts of the old tuple and join up the bits to make the new tuple. So if `julia` has a new recent film, we could change her variable to reference a new tuple that used some information from the old one:

```
1 >>> julia = julia[:3] + ("Eat Pray Love", 2010) + julia[5:]  
2 >>> julia  
3 ("Julia", "Roberts", 1967, "Eat Pray Love", 2010, "Actress", "Atlanta, Georgia")
```

To create a tuple with a single element (but you're probably not likely to do that too often), we have to include the final comma, because without the final comma, Python treats the (5) below as an integer in parentheses:

```
1 >>> tup = (5,)  
2 >>> type(tup)  
3 <class 'tuple'>  
4 >>> x = (5)  
5 >>> type(x)  
6 <class 'int'>
```

## 9.2. Tuple assignment

Python has a very powerful **tuple assignment** feature that allows a tuple of variables on the left of an assignment to be assigned values from a tuple on the right of the assignment. (We already saw this used for pairs, but it generalizes.)

```
1 (name, surname, b_year, movie, m_year, profession, b_place) = julia
```

This does the equivalent of seven assignment statements, all on one easy line. One requirement is that the number of variables on the left must match the number of elements in the tuple.

One way to think of tuple assignment is as tuple packing/unpacking.

In tuple packing, the values on the left are 'packed' together in a tuple:

```
1 >>> b = ("Bob", 19, "CS")      # tuple packing
```

In tuple unpacking, the values in a tuple on the right are 'unpacked' into the variables/names on the left:

```

1 >>> b = ("Bob", 19, "CS")
2 >>> (name, age, studies) = b      # tuple unpacking
3 >>> name
4 'Bob'
5 >>> age
6 19
7 >>> studies
8 'CS'
```

Once in a while, it is useful to swap the values of two variables. With conventional assignment statements, we have to use a temporary variable. For example, to swap `a` and `b`:

```

1 temp = a
2 a = b
3 b = temp
```

Tuple assignment solves this problem neatly:

```
1 (a, b) = (b, a)
```

The left side is a tuple of variables; the right side is a tuple of values. Each value is assigned to its respective variable. All the expressions on the right side are evaluated before any of the assignments. This feature makes tuple assignment quite versatile.

Naturally, the number of variables on the left and the number of values on the right have to be the same:

```

1 >>> (a, b, c, d) = (1, 2, 3)
2 ValueError: need more than 3 values to unpack
```

## 9.3. Tuples as return values

Functions can always only return a single value, but by making that value a tuple, we can effectively group together as many values as we like, and return them together. This is very useful - we often want to know some batsman's highest and lowest score, or we want to find the mean and the standard deviation, or we want to know the year, the month, and the day, or if we're doing some ecological modelling we may want to know the number of rabbits and the number of wolves on an island at a given time.

For example, we could write a function that returns both the area and the circumference of a circle of radius `r`:

```

1 def f(r):
2     """ Return (circumference, area) of a circle of radius r """
3     c = 2 * math.pi * r
4     a = math.pi * r * r
5     return (c, a)

```

## 9.4. Composability of Data Structures

We saw in an earlier chapter that we could make a list of pairs, and we had an example where one of the items in the tuple was itself a list:

```

1 students = [
2     ("John", ["CompSci", "Physics"]),
3     ("Vusi", ["Maths", "CompSci", "Stats"]),
4     ("Jess", ["CompSci", "Accounting", "Economics", "Management"]),
5     ("Sarah", ["InfSys", "Accounting", "Economics", "CommLaw"]),
6     ("Zuki", ["Sociology", "Economics", "Law", "Stats", "Music"])]

```

Tuples items can themselves be other tuples. For example, we could improve the information about our movie stars to hold the full date of birth rather than just the year, and we could have a list of some of her movies and dates that they were made, and so on:

```

1 julia_more_info = ( ("Julia", "Roberts"), (8, "October", 1967),
2                     "Actress", ("Atlanta", "Georgia"),
3                     [ ("Duplicity", 2009),
4                       ("Notting Hill", 1999),
5                       ("Pretty Woman", 1990),
6                       ("Erin Brockovich", 2000),
7                       ("Eat Pray Love", 2010),
8                       ("Mona Lisa Smile", 2003),
9                       ("Oceans Twelve", 2004) ])

```

Notice in this case that the tuple has just five elements - but each of those in turn can be another tuple, a list, a string, or any other kind of Python value. This property is known as being **heterogeneous**, meaning that it can be composed of elements of different types.

## 9.5. Glossary

### **data structure**

An organization of data for the purpose of making it easier to use.

**immutable data value**

A data value which cannot be modified. Assignments to elements or slices (sub-parts) of immutable values cause a runtime error.

**mutable data value**

A data value which can be modified. The types of all mutable values are compound types. Lists and dictionaries are mutable; strings and tuples are not.

**tuple**

An immutable data value that contains related elements. Tuples are used to group together related data, such as a person's name, their age, and their gender.

**tuple assignment**

An assignment to all of the elements in a tuple using a single assignment statement. Tuple assignment occurs *simultaneously* rather than in sequence, making it useful for swapping values.

## 9.6. Exercises

1. We've said nothing in this chapter about whether you can pass tuples as arguments to a function. Construct a small Python example to test whether this is possible, and write up your findings.
2. Is a pair a generalization of a tuple, or is a tuple a generalization of a pair?
3. Is a pair a kind of tuple, or is a tuple a kind of pair?

# Chapter 10: Event handling

## 10.1. Event-driven programming

Most programs and devices like a cellphone respond to *events* – things that happen. For example, you might move your mouse, and the computer responds. Or you click a button, and the program does something interesting. In this chapter we'll touch very briefly on how event-driven programming works.

## 10.2. Keypress events

Here's a program with some new features. Copy it into your workspace, run it. When the turtle window opens, press the arrow keys and make tess move about!

```
1 import turtle
2
3 turtle.setup(400,500)                      # Determine the window size
4 wn = turtle.Screen()                         # Get a reference to the window
5 wn.title("Handling keypresses!")            # Change the window title
6 wn.bgcolor("lightgreen")                     # Set the background color
7 tess = turtle.Turtle()                        # Create our favorite turtle
8
9 # The next four functions are our "event handlers".
10 def h1():
11     tess.forward(30)
12
13 def h2():
14     tess.left(45)
15
16 def h3():
17     tess.right(45)
18
19 def h4():
20     wn.bye()                                # Close down the turtle window
21
22 # These lines "wire up" keypresses to the handlers we've defined.
23 wn.onkey(h1, "Up")
```

```

24 wn.onkey(h2, "Left")
25 wn.onkey(h3, "Right")
26 wn.onkey(h4, "q")
27
28 # Now we need to tell the window to start listening for events,
29 # If any of the keys that we're monitoring is pressed, its
30 # handler will be called.
31 wn.listen()
32 wn.mainloop()

```

Here are some points to note:

- We need the call to the window's `listen` method at line 31, otherwise it won't notice our keypresses.
- We named our handler functions `h1`, `h2` and so on, but we can choose better names. The handlers can be arbitrarily complex functions that call other functions, etc.
- Pressing the `q` key on the keyboard calls function `h4` (because we bound the `q` key to `h4` on line 26). While executing `h4`, the window's `bye` method (line 20) closes the turtle window, which causes the window's `mainloop` call (line 31) to end its execution. Since we did not write any more statements after line 32, this means that our program has completed everything, so it too will terminate.
- We can refer to keys on the keyboard by their character code (as we did in line 26), or by their symbolic names. Some of the symbolic names to try are Cancel (the Break key), BackSpace, Tab, Return(the Enter key), Shift\_L (any Shift key), Control\_L (any Control key), Alt\_L (any Alt key), Pause, Caps\_Lock, Escape, Prior (Page Up), Next (Page Down), End, Home, Left, Up, Right, Down, Print, Insert, Delete, F1, F2, F3, F4, F5, F6, F7, F8, F9, F10, F11, F12, Num\_Lock, and Scroll\_Lock.

### 10.3. Mouse events

A mouse event is a bit different from a keypress event because its handler needs two parameters to receive `x,y` coordinate information telling us where the mouse was when the event occurred.

```

1 import turtle
2
3 turtle.setup(400,500)
4 wn = turtle.Screen()
5 wn.title("How to handle mouse clicks on the window!")
6 wn.bgcolor("lightgreen")
7
8 tess = turtle.Turtle()
9 tess.color("purple")
10 tess.pensize(3)
11 tess.shape("circle")
12

```

```
13 def h1(x, y):
14     tess.goto(x, y)
15
16 wn.onclick(h1) # Wire up a click on the window.
17 wn.mainloop()
```

There is a new turtle method used at line 14 – this allows us to move the turtle to an *absolute* coordinate position. (Most of the examples that we've seen so far move the turtle *relative* to where it currently is). So what this program does is move the turtle (and draw a line) to wherever the mouse is clicked. Try it out!

If we add this line before line 14, we'll learn a useful debugging trick too:

```
1 wn.title("Got click at coords {0}, {1}".format(x, y))
```

Because we can easily change the text in the window's title bar, it is a useful place to display occasional debugging or status information. (Of course, this is not the real purpose of the window title!)

But there is more!

Not only can the window receive mouse events: individual turtles can also have their own handlers for mouse clicks. The turtle that “receives” the click event will be the one under the mouse. So we'll create two turtles. Each will bind a handler to its own `onclick` event. And the two handlers can do different things for their turtles.

```
1 import turtle
2
3 turtle.setup(400,500)           # Determine the window size
4 wn = turtle.Screen()            # Get a reference to the window
5 wn.title("Handling mouse clicks!") # Change the window title
6 wn.bgcolor("lightgreen")         # Set the background color
7 tess = turtle.Turtle()           # Create two turtles
8 tess.color("purple")
9 alex = turtle.Turtle()          # Move them apart
10 alex.color("blue")
11 alex.forward(100)
12
13 def handler_for_tess(x, y):
14     wn.title("Tess clicked at {0}, {1}".format(x, y))
15     tess.left(42)
16     tess.forward(30)
17
18 def handler_for_alex(x, y):
```

```
19     wn.title("Alex clicked at {0}, {1}".format(x, y))
20     alex.right(84)
21     alex.forward(50)
22
23 tess.onclick(handler_for_tess)
24 alex.onclick(handler_for_alex)
25
26 wn.mainloop()
```

Run this, click on the turtles, see what happens!

## 10.4. Automatic events from a timer

Alarm clocks, kitchen timers, and thermonuclear bombs in James Bond movies are set to create an “automatic” event after a certain interval. The turtle module in Python has a timer that can cause an event when its time is up.

```
1 import turtle
2
3 turtle.setup(400,500)
4 wn = turtle.Screen()
5 wn.title("Using a timer")
6 wn.bgcolor("lightgreen")
7
8 tess = turtle.Turtle()
9 tess.color("purple")
10 tess.pensize(3)
11
12 def h1():
13     tess.forward(100)
14     tess.left(56)
15
16 wn.ontimer(h1, 2000)
17 wn.mainloop()
```

On line 16 the timer is started and set to explode in 2000 milliseconds (2 seconds). When the event does occur, the handler is called, and tess springs into action.

Unfortunately, when one sets a timer, it only goes off once. So a common idiom, or style, is to restart the timer inside the handler. In this way the timer will keep on giving new events. Try this program:

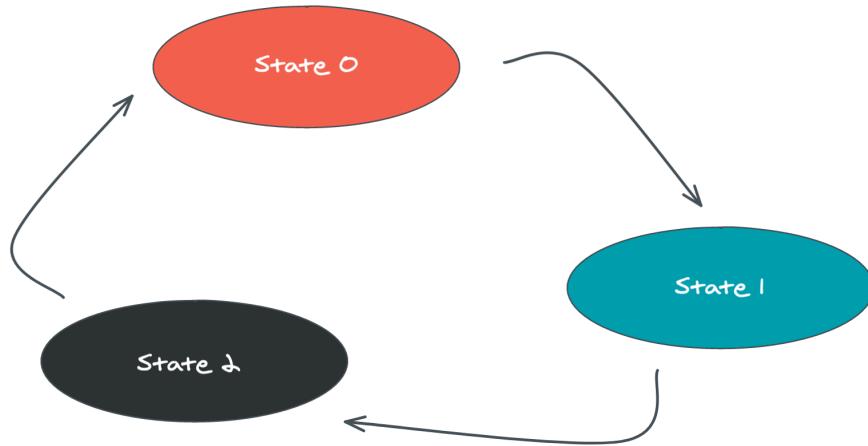
```
1 import turtle
2
3 turtle.setup(400,500)
4 wn = turtle.Screen()
5 wn.title("Using a timer to get events!")
6 wn.bgcolor("lightgreen")
7
8 tess = turtle.Turtle()
9 tess.color("purple")
10
11 def h1():
12     tess.forward(100)
13     tess.left(56)
14     wn.ontimer(h1, 60)
15
16 h1()
17 wn.mainloop()
```

## 10.5. An example: state machines

A state machine is a system that can be in one of a few different states. We draw a state diagram to represent the machine, where each state is drawn as a circle or an ellipse. Certain events occur which cause the system to leave one state and transition into a different state. These state transitions are usually drawn as an arrow on the diagram.

This idea is not new: when first turning on a cellphone, it goes into a state which we could call “Awaiting PIN”. When the correct PIN is entered, it transitions into a different state – say “Ready”. Then we could lock the phone, and it would enter a “Locked” state, and so on.

A simple state machine that we encounter often is a traffic light. Here is a state diagram which shows that the machine continually cycles through three different states, which we’ve numbered 0, 1 and 2.



We're going to build a program that uses a turtle to simulate the traffic lights. There are three lessons here. The first shows off some different ways to use our turtles. The second demonstrates how we would program a state machine in Python, by using a variable to keep track of the current state, and a number of different `if` statements to inspect the current state, and take the actions as we change to a different state. The third lesson is to use events from the keyboard to trigger the state changes.

Copy and run this program. Make sure you understand what each line does, consulting the documentation as you need to.

```
1 import turtle      # Tess becomes a traffic light.  
2  
3 turtle.setup(400,500)  
4 wn = turtle.Screen()  
5 wn.title("Tess becomes a traffic light!")  
6 wn.bgcolor("lightgreen")  
7 tess = turtle.Turtle()  
8  
9  
10 def draw_housing():  
11     """ Draw a nice housing to hold the traffic lights """  
12     tess.pensize(3)  
13     tess.color("black", "darkgrey")  
14     tess.begin_fill()
```

```
15     tess.forward(80)
16     tess.left(90)
17     tess.forward(200)
18     tess.circle(40, 180)
19     tess.forward(200)
20     tess.left(90)
21     tess.end_fill()
22
23
24 draw_housing()
25
26 tess.penup()
27 # Position tess onto the place where the green light should be
28 tess.forward(40)
29 tess.left(90)
30 tess.forward(50)
31 # Turn tess into a big green circle
32 tess.shape("circle")
33 tess.shapesize(3)
34 tess.fillcolor("green")
35
36 # A traffic light is a kind of state machine with three states,
37 # Green, Orange, Red. We number these states 0, 1, 2
38 # When the machine changes state, we change tess' position and
39 # her fillcolor.
40
41 # This variable holds the current state of the machine
42 state_num = 0
43
44
45 def advance_state_machine():
46     global state_num
47     if state_num == 0:      # Transition from state 0 to state 1
48         tess.forward(70)
49         tess.fillcolor("orange")
50         state_num = 1
51     elif state_num == 1:    # Transition from state 1 to state 2
52         tess.forward(70)
53         tess.fillcolor("red")
54         state_num = 2
55     else:                  # Transition from state 2 to state 0
56         tess.back(140)
57         tess.fillcolor("green")
```

```
58     state_num = 0
59
60 # Bind the event handler to the space key.
61 wn.onkey(advance_state_machine, "space")
62
63 wn.listen()                      # Listen for events
64 wn.mainloop()
```

The new Python statement is at line 46. The `global` keyword tells Python not to create a new local variable for `state_num` (in spite of the fact that the function assigns to this variable at lines 50, 54, and 58). Instead, in this function, `state_num` always refers to the variable that was created at line 42.

What the code in `advance_state_machine` does is advance from whatever the current state is, to the next state. On the state change we move tess to her new position, change her color, and, of course, we assign to `state_num` the number of the new state we've just entered.

Each time the space bar is pressed, the event handler causes the traffic light machine to move to its new state.

## 10.6. Glossary

### **bind**

We bind a function (or associate it) with an event, meaning that when the event occurs, the function is called to handle it.

### **event**

Something that happens “outside” the normal control flow of our program, usually from some user action. Typical events are mouse operations and keypresses. We’ve also seen that a timer can be primed to create an event.

### **handler**

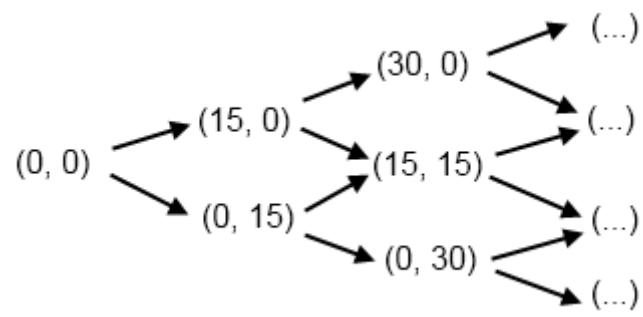
A function that is called in response to an event.

## 10.7. Exercises

1. Add some new key bindings to the first sample program:

- Pressing keys R, G or B should change tess’s color to Red, Green or Blue.
- Pressing keys + or - should increase or decrease the width of tess’s pen. Ensure that the pen size stays between 1 and 20 (inclusive).
- Handle some other keys to change some attributes of tess, or attributes of the window, or to give her new behaviour that can be controlled from the keyboard.

2. Change the traffic light program so that changes occur automatically, driven by a timer.
3. In an earlier chapter we saw two turtle methods, `hideturtle` and `showturtle` that can hide or show a turtle. This suggests that we could take a different approach to the traffic lights program. Add to your program above as follows: draw a second housing for another set of traffic lights. Create three separate turtles to represent each of the green, orange and red lights, and position them appropriately within your new housing. As your state changes occur, just make one of the three turtles visible at any time. Once you've made the changes, sit back and ponder some deep thoughts: you've now got two different ways to use turtles to simulate the traffic lights, and both seem to work. Is one approach somehow preferable to the other? Which one more closely resembles reality – i.e. the traffic lights in your town?
4. Now that you've got a traffic light program with different turtles for each light, perhaps the visibility / invisibility trick wasn't such a great idea. If we watch traffic lights, they turn on and off – but when they're off they are still there, perhaps just a darker color. Modify the program now so that the lights don't disappear: they are either on, or off. But when they're off, they're still visible.
5. Your traffic light controller program has been patented, and you're about to become seriously rich. But your new client needs a change. They want four states in their state machine: Green, then Green and Orange together, then Orange only, and then Red. Additionally, they want different times spent in each state. The machine should spend 3 seconds in the Green state, followed by one second in the Green+Orange state, then one second in the Orange state, and then 2 seconds in the Red state. Change the logic in the state machine.
6. If you don't know how tennis scoring works, ask a friend or consult Wikipedia. A single game in tennis between player A and player B always has a score. We want to think about the "state of the score" as a state machine. The game starts in state  $(0, 0)$ , meaning neither player has any score yet. We'll assume the first element in this pair is the score for player A. If player A wins the first point, the score becomes  $(15, 0)$ . If B wins the first point, the state becomes  $(0, 15)$ . Below are the first few states and transitions for a state diagram. In this diagram, each state has two possible outcomes (A wins the next point, or B does), and the uppermost arrow is always the transition that happens when A wins the point. Complete the diagram, showing all transitions and all states. (Hint: there are twenty states, if you include the deuce state, the advantage states, and the "A wins" and "B wins" states in your diagram.)



# Chapter 11: Lists

A **list** is an ordered collection of values. The values that make up a list are called its **elements**, or its **items**. We will use the term **element** or **item** to mean the same thing. Lists are similar to strings, which are ordered collections of characters, except that the elements of a list can be of any type. Lists and strings — and other collections that maintain the order of their items — are called **sequences**.

## 11.1. List values

There are several ways to create a new list; the simplest is to enclose the elements in square brackets ([ and ]):

```
1 ps = [10, 20, 30, 40]
2 qs = ["spam", "bungee", "swallow"]
```

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 (amazingly) another list:

```
1 zs = ["hello", 2.0, 5, [10, 20]]
```

A list within another list is said to be **nested**.

Finally, a list with no elements is called an **empty list**, and is denoted [].

We have already seen that we can assign list values to variables or pass lists as parameters to functions:

```
1 >>> vocabulary = ["apple", "cheese", "dog"]
2 >>> numbers = [17, 123]
3 >>> an_empty_list = []
4 >>> print(vocabulary, numbers, an_empty_list)
5 ["apple", "cheese", "dog"] [17, 123] []
```

## 11.2. Accessing elements

The syntax for accessing the elements of a list is the same as the syntax for accessing the characters of a string — the index operator: [] (not to be confused with an empty list). The expression inside the brackets specifies the index. Remember that the indices start at 0:

```
1 >>> numbers[0]
2 17
```

Any expression evaluating to an integer can be used as an index:

```
1 >>> numbers[9-8]
2 5
3 >>> numbers[1.0]
4 Traceback (most recent call last):
5   File "<interactive input>", line 1, in <module>
6 TypeError: list indices must be integers, not float
```

If you try to access or assign to an element that does not exist, you get a runtime error:

```
1 >>> numbers[2]
2 Traceback (most recent call last):
3   File "<interactive input>", line 1, in <module>
4 IndexError: list index out of range
```

It is common to use a loop variable as a list index.

```
1 horsemen = ["war", "famine", "pestilence", "death"]
2
3 for i in [0, 1, 2, 3]:
4     print(horsemen[i])
```

Each time through the loop, the variable `i` is used as an index into the list, printing the `i`'th element. This pattern of computation is called a **list traversal**.

The above sample doesn't need or use the index `i` for anything besides getting the items from the list, so this more direct version — where the `for` loop gets the items — might be preferred:

```
1 horsemen = ["war", "famine", "pestilence", "death"]
2
3 for h in horsemen:
4     print(h)
```

## 11.3. List length

The function `len` returns the length of a list, which is equal to the number of its elements. If you are going to use an integer index to access the list, it is a good idea to use this value as the upper bound of a loop instead of a constant. That way, if the size of the list changes, you won't have to go through the program changing all the loops; they will work correctly for any size list:

```

1 horsemen = ["war", "famine", "pestilence", "death"]
2
3 for i in range(len(horsemen)):
4     print(horsemen[i])

```

The last time the body of the loop is executed, `i` is `len(horsemen) - 1`, which is the index of the last element. (But the version without the index looks even better now!)

Although a list can contain another list, the nested list still counts as a single element in its parent list. The length of this list is 4:

```

1 >>> len(["car makers", 1, ["Ford", "Toyota", "BMW"], [1, 2, 3]])
2 4

```

## 11.4. List membership

`in` and `not in` are Boolean operators that test membership in a sequence. We used them previously with strings, but they also work with lists and other sequences:

```

1 >>> horsemen = ["war", "famine", "pestilence", "death"]
2 >>> "pestilence" in horsemen
3 True
4 >>> "debauchery" in horsemen
5 False
6 >>> "debauchery" not in horsemen
7 True

```

Using this produces a more elegant version of the nested loop program we previously used to count the number of students doing Computer Science in the section `nested_data`:

```

1 students = [
2     ("John", ["CompSci", "Physics"]),
3     ("Vusi", ["Maths", "CompSci", "Stats"]),
4     ("Jess", ["CompSci", "Accounting", "Economics", "Management"]),
5     ("Sarah", ["InfSys", "Accounting", "Economics", "CommLaw"]),
6     ("Zuki", ["Sociology", "Economics", "Law", "Stats", "Music"])]
7
8 # Count how many students are taking CompSci
9 counter = 0
10 for (name, subjects) in students:
11     if "CompSci" in subjects:
12         counter += 1
13
14 print("The number of students taking CompSci is", counter)

```

## 11.5. List operations

The + operator concatenates lists:

```
1 >>> a = [1, 2, 3]
2 >>> b = [4, 5, 6]
3 >>> c = a + b
4 >>> c
5 [1, 2, 3, 4, 5, 6]
```

Similarly, the \* operator repeats a list a given number of times:

```
1 >>> [0] * 4
2 [0, 0, 0, 0]
3 >>> [1, 2, 3] * 3
4 [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.

## 11.6. List slices

The slice operations we saw previously with strings let us work with sublists:

```
1 >>> a_list = ["a", "b", "c", "d", "e", "f"]
2 >>> a_list[1:3]
3 ['b', 'c']
4 >>> a_list[:4]
5 ['a', 'b', 'c', 'd']
6 >>> a_list[3:]
7 ['d', 'e', 'f']
8 >>> a_list[:]
9 ['a', 'b', 'c', 'd', 'e', 'f']
```

## 11.7. Lists are mutable

Unlike strings, lists are **mutable**, which means we can change their elements. Using the index operator on the left side of an assignment, we can update one of the elements:

```

1 >>> fruit = ["banana", "apple", "quince"]
2 >>> fruit[0] = "pear"
3 >>> fruit[2] = "orange"
4 >>> fruit
5 ['pear', 'apple', 'orange']

```

The bracket operator applied to a list can appear anywhere in an expression. When it appears on the left side of an assignment, it changes one of the elements in the list, so the first element of `fruit` has been changed from "banana" to "pear", and the last from "quince" to "orange". An assignment to an element of a list is called **item assignment**. Item assignment does not work for strings:

```

1 >>> my_string = "TEST"
2 >>> my_string[2] = "X"
3 Traceback (most recent call last):
4   File "<interactive input>", line 1, in <module>
5 TypeError: 'str' object does not support item assignment

```

but it does for lists:

```

1 >>> my_list = ["T", "E", "S", "T"]
2 >>> my_list[2] = "X"
3 >>> my_list
4 ['T', 'E', 'X', 'T']

```

With the slice operator we can update a whole sublist at once:

```

1 >>> a_list = ["a", "b", "c", "d", "e", "f"]
2 >>> a_list[1:3] = ["x", "y"]
3 >>> a_list
4 ['a', 'x', 'y', 'd', 'e', 'f']

```

We can also remove elements from a list by assigning an empty list to them:

```

1 >>> a_list = ["a", "b", "c", "d", "e", "f"]
2 >>> a_list[1:3] = []
3 >>> a_list
4 ['a', 'd', 'e', 'f']

```

And we can add elements to a list by squeezing them into an empty slice at the desired location:

```
1 >>> a_list = ["a", "d", "f"]
2 >>> a_list[1:1] = ["b", "c"]
3 >>> a_list
4 ['a', 'b', 'c', 'd', 'f']
5 >>> a_list[4:4] = ["e"]
6 >>> a_list
7 ['a', 'b', 'c', 'd', 'e', 'f']
```

## 11.8. List deletion

Using slices to delete list elements can be error-prone. Python provides an alternative that is more readable. The `del` statement removes an element from a list:

```
1 >>> a = ["one", "two", "three"]
2 >>> del a[1]
3 >>> a
4 ['one', 'three']
```

As you might expect, `del` causes a runtime error if the index is out of range.

You can also use `del` with a slice to delete a sublist:

```
1 >>> a_list = ["a", "b", "c", "d", "e", "f"]
2 >>> del a_list[1:5]
3 >>> a_list
4 ['a', 'f']
```

As usual, the sublist selected by slice contains all the elements up to, but not including, the second index.

## 11.9. Objects and references

After we execute these assignment statements

```
1 a = "banana"
2 b = "banana"
```

we know that `a` and `b` will refer to a string object with the letters "banana". But we don't know yet whether they point to the *same* string object.

There are two possible ways the Python interpreter could arrange its memory:



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.

We can test whether two names refer to the same object using the `is` operator:

```
1 >>> a is b
2 True
```

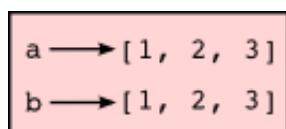
This tells us that both `a` and `b` refer to the same object, and that it is the second of the two state snapshots that accurately describes the relationship.

Since strings are *immutable*, Python optimizes resources by making two names that refer to the same string value refer to the same object.

This is not the case with lists:

```
1 >>> a = [1, 2, 3]
2 >>> b = [1, 2, 3]
3 >>> a == b
4 True
5 >>> a is b
6 False
```

The state snapshot here looks like this:



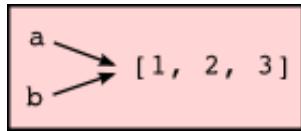
`a` and `b` have the same value but do not refer to the same object.

## 11.10. Aliasing

Since variables refer to objects, if we assign one variable to another, both variables refer to the same object:

```
1 >>> a = [1, 2, 3]
2 >>> b = a
3 >>> a is b
4 True
```

In this case, the state snapshot looks like this:



Because the same list has two different names, `a` and `b`, we say that it is **aliased**. Changes made with one alias affect the other:

```
1 >>> b[0] = 5
2 >>> a
3 [5, 2, 3]
```

Although this behavior can be useful, it is sometimes unexpected or undesirable. In general, it is safer to avoid aliasing when you are working with mutable objects (i.e. lists at this point in our textbook, but we'll meet more mutable objects as we cover classes and objects, dictionaries and sets). Of course, for immutable objects (i.e. strings, tuples), there's no problem — it is just not possible to change something and get a surprise when you access an alias name. That's why Python is free to alias strings (and any other immutable kinds of data) when it sees an opportunity to economize.

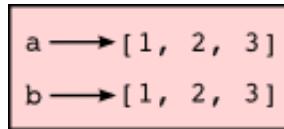
## 11.11. Cloning lists

If we want to modify a list and also keep a copy of the original, we need to be able to make a copy of the list itself, not just the reference. This process is sometimes called **cloning**, to avoid the ambiguity of the word `copy`.

The easiest way to clone a list is to use the slice operator:

```
1 >>> a = [1, 2, 3]
2 >>> b = a[:]
3 >>> b
4 [1, 2, 3]
```

Taking any slice of `a` creates a new list. In this case the slice happens to consist of the whole list. So now the relationship is like this:



Now we are free to make changes to `b` without worrying that we'll inadvertently be changing `a`:

```
1 >>> b[0] = 5
2 >>> a
3 [1, 2, 3]
```

## 11.12. Lists and for loops

The `for` loop also works with lists, as we've already seen. The generalized syntax of a `for` loop is:

```
1 for VARIABLE in LIST:
2     BODY
```

So, as we've seen

```
1 friends = ["Joe", "Zoe", "Brad", "Angelina", "Zuki", "Thandi", "Paris"]
2 for friend in friends:
3     print(friend)
```

It almost reads like English: For (every) friend in (the list of) friends, print (the name of the) friend.

Any list expression can be used in a `for` loop:

```
1 for number in range(20):
2     if number % 3 == 0:
3         print(number)
4
5 for fruit in ["banana", "apple", "quince"]:
6     print("I like to eat " + fruit + "s!")
```

The first example prints all the multiples of 3 between 0 and 19. The second example expresses enthusiasm for various fruits.

Since lists are mutable, we often want to traverse a list, changing each of its elements. The following squares all the numbers in the list `xs`:

```

1 xs = [1, 2, 3, 4, 5]
2
3 for i in range(len(xs)):
4     xs[i] = xs[i]**2

```

Take a moment to think about `range(len(xs))` until you understand how it works.

In this example we are interested in both the *value* of an item, (we want to square that value), and its *index* (so that we can assign the new value to that position). This pattern is common enough that Python

provides a nicer way to implement it:

```

1 xs = [1, 2, 3, 4, 5]
2
3 for (i, val) in enumerate(xs):
4     xs[i] = val**2

```

`enumerate` generates pairs of both (index, value) during the list traversal. Try this next example to see more clearly how `enumerate` works:

```

1 for (i, v) in enumerate(["banana", "apple", "pear", "lemon"]):
2     print(i, v)

```

```

1 0 banana
2 1 apple
3 2 pear
4 3 lemon

```

## 11.13. List parameters

Passing a list as an argument actually passes a reference to the list, not a copy or clone of the list. So parameter passing creates an alias for you: the caller has one variable referencing the list, and the called function has an alias, but there is only one underlying list object. For example, the function below takes a list as an argument and multiplies each element in the list by 2:

```

1 def double_stuff(a_list):
2     """ Overwrite each element in a_list with double its value. """
3     for (idx, val) in enumerate(a_list):
4         a_list[idx] = 2 * val

```

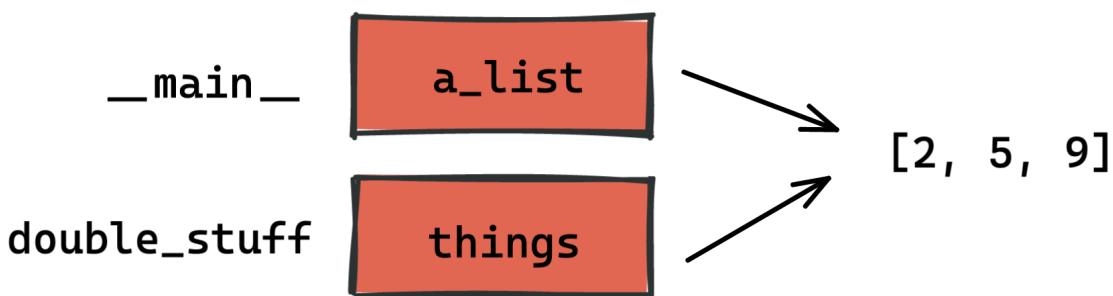
If we add the following onto our script:

```
1 things = [2, 5, 9]
2 double_stuff(things)
3 print(things)
```

When we run it we'll get:

```
1 [4, 10, 18]
```

In the function above, the parameter `a_list` and the variable `things` are aliases for the same object. So before any changes to the elements in the list, the state snapshot looks like this:



Since the list object is shared by two frames, we drew it between them.

If a function modifies the items of a list parameter, the caller sees the change.

Use the Python visualizer!

We've already mentioned the Python visualizer at <http://pythontutor.com>. It is a very useful tool for building a good understanding of references, aliases, assignments, and passing arguments to functions. Pay special attention to cases where you clone a list or have two separate lists, and cases where there is only one underlying list, but more than one variable is aliased to reference the list.

## 11.14. List methods

The dot operator can also be used to access built-in methods of list objects. We'll start with the most useful method for adding something onto the end of an existing list:

```
1 >>> mylist = []
2 >>> mylist.append(5)
3 >>> mylist.append(27)
4 >>> mylist.append(3)
5 >>> mylist.append(12)
6 >>> mylist
7 [5, 27, 3, 12]
```

`append` is a list method which adds the argument passed to it to the end of the list. We'll use it heavily when we're creating new lists. Continuing with this example, we show several other list methods:

```
1 >>> mylist.insert(1, 12) # Insert 12 at pos 1, shift other items up
2 >>> mylist
3 [5, 12, 27, 3, 12]
4 >>> mylist.count(12)      # How many times is 12 in mylist?
5 2
6 >>> mylist.extend([5, 9, 5, 11]) # Put whole list onto end of mylist
7 >>> mylist
8 [5, 12, 27, 3, 12, 5, 9, 5, 11]
9 >>> mylist.index(9)          # Find index of first 9 in mylist
10 6
11 >>> mylist.reverse()
12 >>> mylist
13 [11, 5, 9, 5, 12, 3, 27, 12, 5]
14 >>> mylist.sort()
15 >>> mylist
16 [3, 5, 5, 9, 11, 12, 12, 27]
17 >>> mylist.remove(12)        # Remove the first 12 in the list
18 >>> mylist
19 [3, 5, 5, 9, 11, 12, 27]
```

Experiment and play with the list methods shown here, and read their documentation until you feel confident that you understand how they work.

## 11.15. Pure functions and modifiers

Functions which take lists as arguments and change them during execution are called **modifiers** and the changes they make are called **side effects**.

A **pure function** does not produce side effects. It communicates with the calling program only through parameters, which it does not modify, and a return value. Here is `double_stuff` written as a pure function:

```

1 def double_stuff(a_list):
2     """ Return a new list which contains
3         doubles of the elements in a_list.
4     """
5     new_list = []
6     for value in a_list:
7         new_elem = 2 * value
8         new_list.append(new_elem)
9
10    return new_list

```

This version of `double_stuff` does not change its arguments:

```

1 >>> things = [2, 5, 9]
2 >>> xs = double_stuff(things)
3 >>> things
4 [2, 5, 9]
5 >>> xs
6 [4, 10, 18]

```

An early rule we saw for assignment said “first evaluate the right hand side, then assign the resulting value to the variable”. So it is quite safe to assign the function result to the same variable that was passed to the function:

```

1 >>> things = [2, 5, 9]
2 >>> things = double_stuff(things)
3 >>> things
4 [4, 10, 18]

```

Which style is better?

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. Nevertheless, modifiers are convenient at times, and in some cases, functional programs are less efficient.

In general, we recommend that you write pure functions whenever it is reasonable to do so and resort to modifiers only if there is a compelling advantage. This approach might be called a *functional programming style*.

## 11.16. Functions that produce lists

The pure version of `double_stuff` above made use of an important **pattern** for your toolbox. Whenever you need to write a function that creates and returns a list, the pattern is usually:

```

1 initialize a result variable to be an empty list
2 loop
3     create a new element
4     append it to result
5 return the result

```

Let us show another use of this pattern. Assume you already have a function `is_prime(x)` that can test if `x` is prime. Write a function to return a list of all prime numbers less than `n`:

```

1 def primes_lessthan(n):
2     """ Return a list of all prime numbers less than n. """
3     result = []
4     for i in range(2, n):
5         if is_prime(i):
6             result.append(i)
7     return result

```

## 11.17. Strings and lists

Two of the most useful methods on strings involve conversion to and from lists of substrings. The `split` method (which we've already seen) breaks a string into a list of words. By default, any number of whitespace characters is considered a word boundary:

```

1 >>> song = "The rain in Spain..."
2 >>> wds = song.split()
3 >>> wds
4 ['The', 'rain', 'in', 'Spain...']

```

An optional argument called a **delimiter** can be used to specify which string to use as the boundary marker between substrings. The following example uses the string `ai` as the delimiter:

```

1 >>> song.split("ai")
2 ['The r', 'n in Sp', 'n...']

```

Notice that the delimiter doesn't appear in the result.

The inverse of the `split` method is `join`. You choose a desired **separator** string, (often called the *glue*) and join the list with the glue between each of the elements:

```

1 >>> glue = ";" 
2 >>> s = glue.join(wds) 
3 >>> s 
4 'The;rain;in;Spain...' 
```

The list that you glue together (`wds` in this example) is not modified. Also, as these next examples show, you can use empty glue or multi-character strings as glue:

```

1 >>> " --- ".join(wds) 
2 'The --- rain --- in --- Spain...' 
3 >>> "" .join(wds) 
4 'TheraininSpain...' 
```

## 11.18. list and range

Python has a built-in type conversion function called `list` that tries to turn whatever you give it into a list.

```

1 >>> xs = list("Crunchy Frog") 
2 >>> xs 
3 ['C', 'r', 'u', 'n', 'c', 'h', 'y', ' ', 'F', 'r', 'o', 'g'] 
4 >>> "" .join(xs) 
5 'Crunchy Frog' 
```

One particular feature of `range` is that it doesn't instantly compute all its values: it "puts off" the computation, and does it on demand, or "lazily". We'll say that it gives a **promise** to produce the values

when they are needed. This is very convenient if your computation short-circuits a search and returns early, as in this case:

```

1 def f(n): 
2     """ Find the first positive integer between 101 and less 
3         than n that is divisible by 21 
4     """ 
5     for i in range(101, n): 
6         if (i % 21 == 0): 
7             return i 
8 
9 
10 test(f(110) == 105) 
11 test(f(1000000000) == 105) 
```

In the second test, if `range` were to eagerly go about building a list with all those elements, you would soon exhaust your computer's available memory and crash the program. But it is cleverer than that! This computation works just fine, because the `range` object is just a promise to produce the elements if and when they are needed. Once the condition in the `if` becomes true, no further elements are generated, and the function returns. (Note: Before Python 3, `range` was not lazy. If you use an earlier versions of Python, YMMV!)

### YMMV: Your Mileage May Vary

The acronym YMMV stands for *your mileage may vary*. American car advertisements often quoted fuel consumption figures for cars, e.g. that they would get 28 miles per gallon. But this always had to be accompanied by legal small-print warning the reader that they might not get the same. The term YMMV is now used idiomatically to mean “your results may differ”, e.g. *The battery life on this phone is 3 days, but YMMV.*

You'll sometimes find the lazy `range` wrapped in a call to `list`. This forces Python to turn the lazy promise into an actual list:

```
1 >>> range(10)          # Create a lazy promise
2 range(0, 10)
3 >>> list(range(10))    # Call in the promise, to produce a list.
4 [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

## 11.19. Nested lists

A nested list is a list that appears as an element in another list. In this list, the element with index 3 is a nested list:

```
1 >>> nested = ["hello", 2.0, 5, [10, 20]]
```

If we output the element at index 3, we get:

```
1 >>> print(nested[3])
2 [10, 20]
```

To extract an element from the nested list, we can proceed in two steps:

```
1 >>> elem = nested[3]
2 >>> elem[0]
3 10
```

Or we can combine them:

```

1 >>> nested[3][1]
2 20

```

Bracket operators evaluate from left to right, so this expression gets the 3'th element of `nested` and extracts the 1'th element from it.

## 11.20. Matrices

Nested lists are often used to represent matrices. For example, the matrix:

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

might be represented as:

```

1 >>> mx = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]

```

`mx` is a list with three elements, where each element is a row of the matrix. We can select an entire row from the matrix in the usual way:

```

1 >>> mx[1]
2 [4, 5, 6]

```

Or we can extract a single element from the matrix using the double-index form:

```

1 >>> mx[1][2]
2 6

```

The first index selects the row, and the second index selects the column. Although this way of representing matrices is common, it is not the only possibility. A small variation is to use a list of columns instead of a list of rows. Later we will see a more radical alternative using a dictionary.

## 11.21. Glossary

### aliases

Multiple variables that contain references to the same object.

### clone

To create a new object that has the same value as an existing object.

Copying a reference to an object creates an alias but doesn't clone the object.

**delimiter**

A character or string used to indicate where a string should be split.

**element**

One of the values in a list (or other sequence). The bracket operator selects elements of a list. Also called *item*.

**immutable data value**

A data value which cannot be modified. Assignments to elements or slices (sub-parts) of immutable values cause a runtime error.

**index**

An integer value that indicates the position of an item in a list.

Indexes start from 0.

**item**

See *element*.

**list**

A collection of values, each in a fixed position within the list. Like other types `str`, `int`, `float`, etc. there is also a `list` type-converter function that tries to turn whatever argument you give it into a list.

**list traversal**

The sequential accessing of each element in a list.

**modifier**

A function which changes its arguments inside the function body. Only mutable types can be changed by modifiers.

**mutable data value**

A data value which can be modified. The types of all mutable values are compound types. Lists and dictionaries are mutable; strings and tuples are not.

**nested list**

A list that is an element of another list.

**object**

A thing to which a variable can refer.

**pattern**

A sequence of statements, or a style of coding something that has general applicability in a number of different situations. Part of becoming a mature Computer Scientist is to learn and establish the

patterns and algorithms that form your toolkit. Patterns often correspond to your “mental chunking”.

**promise**

An object that promises to do some work or deliver some values if they’re eventually needed, but it lazily puts off doing the work immediately. Calling `range` produces a promise.

**pure function**

A function which has no side effects. Pure functions only make changes to the calling program through their return values.

**sequence**

Any of the data types that consist of an ordered collection of elements, with each element identified by an index.

**side effect**

A change in the state of a program made by calling a function. Side effects can only be produced by modifiers.

**step size**

The interval between successive elements of a linear sequence. The third (and optional argument) to the `range` function is called the step size.

If not specified, it defaults to 1.

## 11.22. Exercises

1. What is the Python interpreter’s response to the following?

```
1 >>> list(range(10, 0, -2))
```

The three arguments to the `range` function are *start*, *stop*, and *step*, respectively. In this example, *start* is greater than *stop*. What happens if *start* < *stop* and *step* < 0? Write a rule for the relationships among *start*, *stop*, and *step*.

2. Consider this fragment of code:

```
1 import turtle  
2  
3 tess = turtle.Turtle()  
4 alex = tess  
5 alex.color("hotpink")
```

Does this fragment create one or two turtle instances? Does setting the color of `alex` also change the color of `tess`? Explain in detail.

3. Draw a state snapshot for `a` and `b` before and after the third line of the following Python code is executed:

```
1 a = [1, 2, 3]
2 b = a[:]
3 b[0] = 5
```

4. What will be the output of the following program?

```
1 this = ["I", "am", "not", "a", "crook"]
2 that = ["I", "am", "not", "a", "crook"]
3 print("Test 1: {}".format(this is that))
4 that = this
5 print("Test 2: {}".format(this is that))
```

Provide a *detailed* explanation of the results.

5. Lists can be used to represent mathematical *vectors*. In this exercise and several that follow you will

write functions to perform standard operations on vectors. Create a script named `vectors.py` and write Python code to pass the tests in each case.

Write a function `add_vectors(u, v)` that takes two lists of numbers of the same length, and returns a new list containing the sums of the corresponding elements of each:

```
1 test(add_vectors([1, 1], [1, 1]) == [2, 2])
2 test(add_vectors([1, 2], [1, 4]) == [2, 6])
3 test(add_vectors([1, 2, 1], [1, 4, 3]) == [2, 6, 4])
```

6. Write a function `scalar_mult(s, v)` that takes a number, `s`, and a list, `v` and returns the scalar multiple<sup>10</sup> of `v` by `s`:

---

<sup>10</sup>[http://en.wikipedia.org/wiki/Scalar\\_multiple](http://en.wikipedia.org/wiki/Scalar_multiple)

```

1 test(scalar_mult(5, [1, 2]) == [5, 10])
2 test(scalar_mult(3, [1, 0, -1]) == [3, 0, -3])
3 test(scalar_mult(7, [3, 0, 5, 11, 2]) == [21, 0, 35, 77, 14])

```

7. Write a function `dot_product(u, v)` that takes two lists of numbers of the same length, and returns the

sum of the products of the corresponding elements of each (the [dot\\_product<sup>11</sup>](#)).

```

1 test(dot_product([1, 1], [1, 1]) == 2)
2 test(dot_product([1, 2], [1, 4]) == 9)
3 test(dot_product([1, 2, 1], [1, 4, 3]) == 12)

```

8. **Extra challenge for the mathematically inclined:** Write a function `cross_product(u, v)` that takes two

lists of numbers of length 3 and returns their [cross product<sup>12</sup>](#). You should write your own tests.

9. Describe the relationship between " ".join(song.split()) and song in the fragment of code below. Are

they the same for all strings assigned to song? When would they be different?

```
1 song = "The rain in Spain..."
```

10. Write a function `replace(s, old, new)` that replaces all occurrences of `old` with `new` in a string `s`:

```

1 test(replace("Mississippi", "i", "I") == "MIssIssIppI")
2
3 s = "I love spom! Spom is my favorite food. Spom, spom, yum!"
4 test(replace(s, "om", "am") ==
5     "I love spam! Spam is my favorite food. Spam, spam, yum!")
6
7 test(replace(s, "o", "a") ==
8     "I lave spam! Spam is my favarite faad. Spam, spam, yum!")

```

---

<sup>11</sup>[http://en.wikipedia.org/wiki/Dot\\_product](http://en.wikipedia.org/wiki/Dot_product)

<sup>12</sup>[http://en.wikipedia.org/wiki/Cross\\_product](http://en.wikipedia.org/wiki/Cross_product)

**Hint:** use the `split` and `join` methods.

11. Suppose you want to swap around the values in two variables. You decide to factor this out into a

reusable function, and write this code:

```
1 def swap(x, y):      # Incorrect version
2     print("before swap statement: x:", x, "y:", y)
3     (x, y) = (y, x)
4     print("after swap statement: x:", x, "y:", y)
5
6 a = ["This", "is", "fun"]
7 b = [2,3,4]
8 print("before swap function call: a:", a, "b:", b)
9 swap(a, b)
10 print("after swap function call: a:", a, "b:", b)
```

Run this program and describe the results. Oops! So it didn't do what you intended! Explain why not. Using a Python visualizer like the one at <http://pythontutor.com> may help you build a good conceptual model of what is going on. What will be the values of `a` and `b` after the call to `swap`?

# Chapter 12: Modules

A **module** is a file containing Python definitions and statements intended for use in other Python programs. There are many Python modules that come with Python as part of the **standard library**. We have seen at least two of these already, the `turtle` module and the `string` module.

We have also shown you how to access help. The help system contains a listing of all the standard modules that are available with Python. Play with help!

## 12.1. Random numbers

We often want to use random numbers in programs, here are a few typical uses:

- To play a game of chance where the computer needs to throw some dice, pick a number, or flip a coin
- To shuffle a deck of playing cards randomly,
- To allow/make an enemy spaceship appear at a random location and start shooting at the player
- To simulate possible rainfall when we make a computerized model for estimating the environmental impact  
of building a dam,
- For encrypting banking sessions on the Internet.

Python provides a module `random` that helps with tasks like this. You can look it up using help, but here are the key things we'll do with it:

```
1 import random
2
3 # Create a black box object that generates random numbers
4 rng = random.Random()
5
6 dice_throw = rng.randrange(1,7)    # Return an int, one of 1,2,3,4,5,6
7 delay_in_seconds = rng.random() * 5.0
```

The `randrange` method call generates an integer between its lower and upper argument, using the same semantics as `range` — so the lower bound is included, but the upper bound is excluded. All the values have

an equal probability of occurring (i.e. the results are *uniformly distributed*). Like `range`, `randrange` can also take an optional step argument. So let's assume we needed a random odd number less than 100, we could say:

```
1 r_odd = rng.randrange(1, 100, 2)
```

Other methods can also generate other distributions e.g. a bell-shaped, or “normal” distribution might be more appropriate for estimating seasonal rainfall, or the concentration of a compound in the body after taking a dose of medicine.

The `random` method returns a floating point number in the interval  $[0.0, 1.0]$  — the square bracket means “closed interval on the left” and the round parenthesis means “open interval on the right”. In other words, 0.0 is possible, but all returned numbers will be strictly less than 1.0. It is usual to *scale* the results after calling this method, to get them into an interval suitable for your application. In the case shown here, we’ve converted the result of the method call to a number in the interval  $[0.0, 5.0]$ . Once more, these are uniformly distributed numbers — numbers close to 0 are just as likely to occur as numbers close to 0.5, or numbers close to 1.0.

This example shows how to shuffle a list. (`shuffle` cannot work directly with a lazy promise, so notice that we had to convert the range object using the `list` type converter first.)

```
1 cards = list(range(52)) # Generate ints [0 .. 51]
2                               # representing a pack of cards.
3 rng.shuffle(cards)        # Shuffle the pack
```

### 12.1.1. Repeatability and Testing

Random number generators are based on a **deterministic** algorithm — repeatable and predictable. So they’re called **pseudo-random** generators — they are not genuinely random. They start with a *seed* value. Each time you ask for another random number, you’ll get one based on the current seed attribute, and the state of the seed (which is one of the attributes of the generator) will be updated.

For debugging and for writing unit tests, it is convenient to have repeatability — programs that do the same thing every time they are run. We can arrange this by forcing the random number generator to be initialized with a known seed every time. (Often this is only wanted during testing — playing a game of cards where the shuffled deck was always in the same order as last time you played would get boring very rapidly!)

```
1 drng = random.Random(123) # Create generator with known starting state
```

This alternative way of creating a random number generator gives an explicit seed value to the object. Without this argument, the system probably uses something based on the time. So grabbing some random numbers from `drng` today will give you precisely the same random sequence as it will tomorrow!

### 12.1.2. Picking balls from bags, throwing dice, shuffling a pack of cards

Here is an example to generate a list containing n random ints between a lower and an upper bound:

```

1 import random
2
3 def make_random_ints(num, lower_bound, upper_bound):
4     """
5         Generate a list containing num random ints between lower_bound
6         and upper_bound. upper_bound is an open bound.
7     """
8     rng = random.Random() # Create a random number generator
9     result = []
10    for i in range(num):
11        result.append(rng.randrange(lower_bound, upper_bound))
12    return result
13
14
15 >>> make_random_ints(5, 1, 13) # Pick 5 random month numbers
16 [8, 1, 8, 5, 6]

```

Notice that we got a duplicate in the result. Often this is wanted, e.g. if we throw a die five times, we would expect some duplicates.

But what if you don't want duplicates? If you wanted 5 distinct months, then this algorithm is wrong. In this case a good algorithm is to generate the list of possibilities, shuffle it, and slice off the number of elements you want:

```

1 xs = list(range(1,13)) # Make list 1..12 (there are no duplicates)
2 rng = random.Random() # Make a random number generator
3 rng.shuffle(xs) # Shuffle the list
4 result = xs[:5] # Take the first five elements

```

In statistics courses, the first case — allowing duplicates — is usually described as pulling balls out of a bag *with replacement* — you put the drawn ball back in each time, so it can occur again. The latter case, with no duplicates, is usually described as pulling balls out of the bag *without replacement*. Once the ball is drawn, it doesn't go back to be drawn again. TV lotto games work like this.

The second “shuffle and slice” algorithm would not be so great if you only wanted a few elements, but from a very large domain. Suppose I wanted five numbers between one and ten million, without duplicates. Generating a list of ten million items, shuffling it, and then slicing off the first five would be a performance disaster! So let us have another try:

```
1 import random
2
3 def make_random_ints_no_dups(num, lower_bound, upper_bound):
4     """
5         Generate a list containing num random ints between
6         lower_bound and upper_bound. upper_bound is an open bound.
7         The result list cannot contain duplicates.
8     """
9     result = []
10    rng = random.Random()
11    for i in range(num):
12        while True:
13            candidate = rng.randrange(lower_bound, upper_bound)
14            if candidate not in result:
15                break
16            result.append(candidate)
17    return result
18
19 xs = make_random_ints_no_dups(5, 1, 10000000)
20 print(xs)
```

This agreeably produces 5 random numbers, without duplicates:

```
1 [3344629, 1735163, 9433892, 1081511, 4923270]
```

Even this function has its pitfalls. Can you spot what is going to happen in this case?

```
1 xs = make_random_ints_no_dups(10, 1, 6)
```

## 12.2. The `time` module

As we start to work with more sophisticated algorithms and bigger programs, a natural concern is “*is our code efficient?*” One way to experiment is to time how long various operations take. The `time` module has a function called `process_time` that is recommended for this purpose. Whenever `process_time` is called, it returns a floating point number representing how many seconds have elapsed since your program started running.

The way to use it is to call `process_time`, assign the result to a variable, say `t0`, just before you start executing the code you want to measure. Then after execution, call `process_time` again, (this time we’ll save

the result in variable `t1`). The difference `t1 - t0` is the time elapsed, and is a measure of how fast your program is running.

Let's try a small example. Python has a built-in `sum` function that can sum the elements in a list. We can also write our own. How do we think they would compare for speed? We'll try to do the summation of a list

[0, 1, 2 ...] in both cases, and compare the results:

```
1 import time
2
3 def do_my_sum(xs):
4     sum = 0
5     for v in xs:
6         sum += v
7     return sum
8
9 sz = 10000000          # Lets have 10 million elements in the list
10 testdata = range(sz)
11
12 t0 = time.process_time()
13 my_result = do_my_sum(testdata)
14 t1 = time.process_time()
15 print("my_result = {0} (time taken = {1:.4f} seconds)"
16       .format(my_result, t1-t0))
17
18 t2 = time.process_time()
19 their_result = sum(testdata)
20 t3 = time.process_time()
21 print("their_result = {0} (time taken = {1:.4f} seconds)"
22       .format(their_result, t3-t2))
```

On a reasonably modest laptop, we get these results:

```
1 my_sum      = 49999995000000 (time taken = 1.5567 seconds)
2 their_sum = 49999995000000 (time taken = 0.9897 seconds)
```

So our function runs about 57% slower than the built-in one. Generating and summing up ten million elements in under a second is not too shabby!

## 12.3. The `math` module

The `math` module contains the kinds of mathematical functions you'd typically find on your calculator (`sin`, `cos`, `sqrt`, `asin`, `log`, `log10`) and some mathematical constants like `pi` and `e`:

```
1 >>> import math
2 >>> math.pi           # Constant pi
3 3.141592653589793
4 >>> math.e            # Constant natural log base
5 2.718281828459045
6 >>> math.sqrt(2.0)    # Square root function
7 1.4142135623730951
8 >>> math.radians(90)   # Convert 90 degrees to radians
9 1.5707963267948966
10 >>> math.sin(math.radians(90)) # Find sin of 90 degrees
11 1.0
12 >>> math.asin(1.0) * 2      # Double the arcsin of 1.0 to get pi
13 3.141592653589793
```

Like almost all other programming languages, angles are expressed in *radians* rather than degrees. There are two functions `radians` and `degrees` to convert between these two popular ways of measuring angles.

Notice another difference between this module and our use of `random` and `turtle`: in `random` and `turtle` we create objects and we call methods on the object. This is because objects have *state* — a turtle has a color, a position, a heading, etc., and every random number generator has a seed value that determines its next result.

Mathematical functions are “pure” and don’t have any state — calculating the square root of 2.0 doesn’t depend on any kind of state or history about what happened in the past. So the functions are not methods of an object — they are simply functions that are grouped together in a module called `math`.

## 12.4. Creating your own modules

All we need to do to create our own modules is to save our script as a file with a `.py` extension. Suppose, for example, this script is saved as a file named `seqtools.py`:

```
1 def remove_at(pos, seq):
2     return seq[:pos] + seq[pos+1:]
```

We can now use our module, both in scripts we write, or in the interactive Python interpreter. To do so, we must first `import` the module.

```
1 >>> import seqtools
2 >>> s = "A string!"
3 >>> seqtools.remove_at(4, s)
4 'A sting!'
```

We do not include the .py file extension when importing. Python expects the file names of Python modules to end in .py, so the file extension is not included in the **import statement**.

The use of modules makes it possible to break up very large programs into manageable sized parts, and to keep related parts together.

## 12.5. Namespaces

A **namespace** is a collection of identifiers that belong to a module, or to a function, (and as we will see soon, in classes too). Generally, we like a namespace to hold “related” things, e.g. all the math functions, or all the typical things we’d do with random numbers.

Each module has its own namespace, so we can use the same identifier name in multiple modules without causing an identification problem.

```
1 # Module1.py
2
3 question = "What is the meaning of Life, the Universe, and Everything?"
4 answer = 42

1 # Module2.py
2
3 question = "What is your quest?"
4 answer = "To seek the holy grail."
```

We can now import both modules and access question and answer in each:

```
1 import module1
2 import module2
3
4 print(module1.question)
5 print(module2.question)
6 print(module1.answer)
7 print(module2.answer)
```

will output the following:

```
1 What is the meaning of Life, the Universe, and Everything?
2 What is your quest?
3 42
4 To seek the holy grail.
```

Functions also have their own namespaces:

```
1 def f():
2     n = 7
3     print("printing n inside of f:", n)
4
5 def g():
6     n = 42
7     print("printing n inside of g:", n)
8
9 n = 11
10 print("printing n before calling f:", n)
11 f()
12 print("printing n after calling f:", n)
13 g()
14 print("printing n after calling g:", n)
```

Running this program produces the following output:

```
1 printing n before calling f: 11
2 printing n inside of f: 7
3 printing n after calling f: 11
4 printing n inside of g: 42
5 printing n after calling g: 11
```

The three n's here do not collide since they are each in a different namespace — they are three names for three different variables, just like there might be three different instances of people, all called "Bruce".

Namespaces permit several programmers to work on the same project without having naming collisions.

How are namespaces, files and modules related?

Python has a convenient and simplifying one-to-one mapping, one module per file, giving rise to one namespace. Also, Python takes the module name from the file name, and this becomes the name of the namespace.

`math.py` is a filename, the module is called `math`, and its namespace is `math`. So in Python the concepts are more or less interchangeable.

But you will encounter other languages (e.g. C#), that allow one module to span multiple files, or one file to have multiple namespaces, or many files to all share the same namespace. So the name of the file doesn't need to be the same as the namespace.

So a good idea is to try to keep the concepts distinct in your mind.

Files and directories organize *where* things are stored in our computer. On the other hand, namespaces and modules are a programming concept: they help us organize how we want to group related functions and attributes. They are not about “where” to store things, and should not have to coincide with the file and directory structures.

So in Python, if you rename the file `math.py`, its module name also changes, your `import` statements would need to change, and your code that refers to functions or attributes inside that namespace would also need to change.

In other languages this is not necessarily the case. So don't blur the concepts, just because Python blurs them!

## 12.6. Scope and lookup rules

The **scope** of an identifier is the region of program code in which the identifier can be accessed, or used.

There are three important scopes in Python:

- **Local scope** refers to identifiers declared within a function. These identifiers are kept in the namespace that belongs to the function, and each function has its own namespace.
- **Global scope** refers to all the identifiers declared within the current module, or file.
- **Built-in scope** refers to all the identifiers built into Python — those like `range` and `min` that can be used without having to import anything, and are (almost) always available.

Python (like most other computer languages) uses precedence rules: the same name could occur in more than one of these scopes, but the innermost, or local scope, will always take precedence over the global scope, and the global scope always gets used in preference to the built-in scope. Let's start with a simple example:

```
1 def range(n):  
2     return 123*n  
3  
4 print(range(10))
```

What gets printed? We've defined our own function called `range`, so there is now a potential ambiguity. When we use `range`, do we mean our own one, or the built-in one? Using the scope

lookup rules determines this: our own `range` function, not the built-in one, is called, because our function `range` is in the global namespace, which takes precedence over the built-in names.

So although names like `range` and `min` are built-in, they can be “hidden” from your use if you choose to define your own variables or functions that reuse those names. (It is a confusing practice to redefine built-in names — so to be a good programmer you need to understand the scope rules and understand that you can do nasty things that will cause confusion, and then you avoid doing them!)

Now, a slightly more complex example:

```
1 n = 10
2 m = 3
3 def f(n):
4     m = 7
5     return 2*n+m
6
7 print(f(5), n, m)
```

This prints 17 10 3. The reason is that the two variables `m` and `n` in lines 1 and 2 are outside the function in the global namespace. Inside the function, new variables called `n` and `m` are created *just for the duration of the execution of f*. These are created in the local namespace of function `f`. Within the body of `f`, the scope lookup rules determine that we use the local variables `m` and `n`. By contrast, after we’ve returned from `f`, the `n` and `m` arguments to the `print` function refer to the original variables on lines 1 and 2, and these have not been changed in any way by executing function `f`.

Notice too that the `def` puts name `f` into the global namespace here. So it can be called on line 7.

What is the scope of the variable `n` on line 1? Its scope — the region in which it is visible — is lines 1, 2, 6, 7. It is hidden from view in lines 3, 4, 5 because of the local variable `n`.

## 12.7. Attributes and the dot operator

Variables defined inside a module are called **attributes** of the module. We’ve seen that objects have attributes too: for example, most objects have a `__doc__` attribute, some functions have a `__annotations__` attribute. Attributes are accessed using the **dot operator** (`.`). The `question` attribute of `module1` and `module2` is accessed using `module1.question` and `module2.question`.

Modules contain functions as well as attributes, and the dot operator is used to access them in the same way. `seqtools.remove_at` refers to the `remove_at` function in the `seqtools` module.

When we use a dotted name, we often refer to it as a **fully qualified name**, because we’re saying exactly which `question` attribute we mean.

## 12.8. Three import statement variants

Here are three different ways to import names into the current namespace, and to use them:

```
1 import math
2 x = math.sqrt(10)
```

Here just the single identifier `math` is added to the current namespace. If you want to access one of the functions in the module, you need to use the dot notation to get to it.

Here is a different arrangement:

```
1 from math import cos, sin, sqrt
2 x = sqrt(10)
```

The names are added directly to the current namespace, and can be used without qualification. The name `math` is not itself imported, so trying to use the qualified form `math.sqrt` would give an error.

Then we have a convenient shorthand:

```
1 from math import *      # Import all the identifiers from math,
2                      # adding them to the current namespace.
3 x = sqrt(10)          # Use them without qualification.
```

Of these three, the first method is generally preferred, even though it means a little more typing each time. Although, we can make things shorter by importing a module under a different name:

```
1 >>> import math as m
2 >>> m.pi
3 3.141592653589793
```

But hey, with nice editors that do auto-completion, and fast fingers, that's a small price!

Finally, observe this case:

```
1 def area(radius):
2     import math
3     return math.pi * radius * radius
4
5 x = math.sqrt(10)      # This gives an error
```

Here we imported `math`, but we imported it into the local namespace of `area`. So the name is usable within the function body, but not in the enclosing script, because it is not in the global namespace.

## 12.9. Turn your unit tester into a module

Near the end of Chapter 6 (Fruitful functions) we introduced unit testing, and our own test function, and you've had to copy this into each module for which you wrote tests. Now we can put that definition into a module of its own, say `unit_tester.py`, and simply use one line in each new script instead:

```
1 from unit_tester import test
```

## 12.10. Glossary

### attribute

A variable defined inside a module (or class or instance – as we will see later). Module attributes are accessed by using the **dot operator** (.)).

### dot operator

The dot operator (.) permits access to attributes and functions of a module (or attributes and methods of a class or instance – as we have seen elsewhere).

### fully qualified name

A name that is prefixed by some namespace identifier and the dot operator, or by an instance object, e.g. `math.sqrt` or `tess.forward(10)`.

### import statement

A statement which makes the objects contained in a module available for use within another module. There are two forms for the import statement. Using hypothetical modules named `mymod1` and `mymod2` each containing functions `f1` and `f2`, and variables `v1` and `v2`, examples of these two forms include:

```
1 import mymod1
2 from mymod2 import f1, f2, v1, v2
```

The second form brings the imported objects into the namespace of the importing module, while the first form preserves a separate namespace for the imported module, requiring `mymod1.v1` to access the `v1` variable from that module.

### method

Function-like attribute of an object. Methods are *invoked* (called) on an object using the dot operator. For example:

```
1 >>> s = "this is a string."
2 >>> s.upper()
3 'THIS IS A STRING.'
4 >>>
```

We say that the method, `upper` is invoked on the string, `s`. `s` is implicitly the first argument to `upper`.

### module

A file containing Python definitions and statements intended for use in other Python programs. The contents of a module are made available to the other program by using the `import` statement.

### namespace

A syntactic container providing a context for names so that the same name can reside in different namespaces without ambiguity. In Python, modules, classes, functions and methods all form namespaces.

### naming collision

A situation in which two or more names in a given namespace cannot be unambiguously resolved. Using

```
1 import string
```

instead of

```
1 from string import *
```

prevents naming collisions.

### standard library

A library is a collection of software used as tools in the development of other software. The standard library of a programming language is the set of such tools that are distributed with the core programming language. Python comes with an extensive standard library.

## 12.11. Exercises

1. Open help for the `calendar` module.
1. Try the following:

```
1 import calendar
2 cal = calendar.TextCalendar()      # Create an instance
3 cal.pryear(2012)                 # What happens here?
```

2. Observe that the week starts on Monday. An adventurous CompSci student believes that it is better

mental chunking to have his week start on Thursday, because then there are only two working days to the weekend, and every week has a break in the middle. Read the documentation for `TextCalendar`, and see how you can help him print a calendar that suits his needs.

3. Find a function to print just the month in which your birthday occurs this year.  
4. Try this:

```
1 d = calendar.LocaleTextCalendar(6, "SPANISH")
2 d.pryear(2012)
3
4 Try a few other languages, including one that doesn't work, and see what happens.
```

5. Experiment with `calendar.isleap`. What does it expect as an argument? What does it return as a result? What kind of a function is this?

Make detailed notes about what you learned from these exercises.

2. Open help for the `math` module.
1. How many functions are in the `math` module?
  2. What does `math.ceil` do? What about `math.floor`? (*hint*: both `floor` and `ceil` expect floating point arguments.)
  3. Describe how we have been computing the same value as `math.sqrt` without using the `math` module.
  4. What are the two data constants in the `math` module?

Record detailed notes of your investigation in this exercise.

3. Investigate the `copy` module. What does `deepcopy` do? In which exercises from last chapter would `deepcopy` have come in handy?
4. Create a module named `mymodule1.py`. Add attributes `myage` set to your current age, and `year` set to

the current year. Create another module named `mymodule2.py`. Add attributes `myage` set to 0, and `year` set to the year you were born. Now create a file named `namespace_test.py`. Import both of

the modules above and write the following statement:

```
1     print( (mymodule2.myage - mymodule1.myage) ==
2             (mymodule2.year - mymodule1.year) )
```

When you will run `namespace_test.py` you will see either `True` or `False` as output depending on whether or not you've already had your birthday this year.

What this example illustrates is that out different modules can both have attributes named `myage` and `year`. Because they're in different namespaces, they don't clash with one another. When we write `namespace_test.py`, we fully qualify exactly which variable `year` or `myage` we are referring to.

5. Add the following statement to `mymodule1.py`, `mymodule2.py`, and `namespace_test.py` from the previous

exercise:

```
1     print("My name is", __name__)
```

Run `namespace_test.py`. What happens? Why? Now add the following to the bottom of `mymodule1.py`:

```
1     if __name__ == "__main__":
2         print("This won't run if I'm imported.")
```

Run `mymodule1.py` and `namespace_test.py` again. In which case do you see the new print statement?

6. In a Python shell / interactive interpreter, try the following:

```
1     >>> import this
```

What does Tim Peters have to say about namespaces?

7. Give the Python interpreter's response to each of the following from a continuous interpreter session:

```

1      >>> s = "If we took the bones out, it wouldn't be crunchy, would it?"
2      >>> s.split()
3      >>> type(s.split())
4      >>> s.split("o")
5      >>> s.split("i")
6      >>> "0".join(s.split("o"))

```

Be sure you understand why you get each result. Then apply what you have learned to fill in the body of the function below using the `split` and `join` methods of `str` objects:

```

1  def myreplace(old, new, s):
2      """ Replace all occurrences of old with new in s. """
3      ...
4
5
6      test(myreplace("", ";", "this, that, and some other thing") ==
7              "this; that; and some other thing")
8      test(myreplace(" ", "**",
9                      "Words will now      be    separated by stars.") ==
10             "Words**will**now**be**separated**by**stars.")

```

Your solution should pass the tests.

8. Create a module named `wordtools.py` with our test scaffolding in place.

Now add functions to these tests pass:

```

1  test(cleanword("what?") == "what")
2  test(cleanword("'now!') == "now")
3  test(cleanword("?+='w-o-r-d!',@$()'") == "word")
4
5  test(has_dashdash("distance--but")))
6  test(not has_dashdash("several"))
7  test(has_dashdash("spoke--"))
8  test(has_dashdash("distance--but")))
9  test(not has_dashdash("-yo-yo-"))
10
11 test(extract_words("Now is the time!  'Now', is the time? Yes, now.") == ['now', 'is' \
12 , 'the', 'time', 'now', 'is', 'the', 'time', 'yes', 'now'])
13 test(extract_words("she tried to curtsey as she spoke--fancy") == ['she', 'tried', 'to' \
14 , 'curtsey', 'as', 'she', 'spoke', 'fancy'])

```

```
15 test(wordcount("now", ["now", "is", "time", "is", "now", "is", "is"])) == 2)
16 test(wordcount("is", ["now", "is", "time", "is", "now", "the", "is"])) == 3)
17 test(wordcount("time", ["now", "is", "time", "is", "now", "is", "is"])) == 1)
18 test(wordcount("frog", ["now", "is", "time", "is", "now", "is", "is"])) == 0)
19
20 test(wordset(["now", "is", "time", "is", "now", "is", "is"])) == ["is", "now", "time"\n21 ])
22 test(wordset(["I", "a", "a", "is", "a", "is", "I", "am"])) == ["I", "a", "am", "is"])
23 test(wordset(["or", "a", "am", "is", "are", "be", "but", "am"])) == ["a", "am", "are"\n24 , "be", "but", "is", "or"])
25
26 test(longestword(["a", "apple", "pear", "grape"])) == 5)
27 test(longestword(["a", "am", "I", "be"])) == 2)
28 test(longestword(["this", "supercalifragilisticexpialidocious"])) == 34)
29 test(longestword([])) == 0)
```

Save this module so you can use the tools it contains in future programs.

# Chapter 13: Files

## 13.1. About files

While a program is running, its data is stored in *random access memory* (RAM). RAM is fast and inexpensive, but it is also **volatile**, which means that when the program ends, or the computer shuts down, data in

RAM disappears. To make data available the next time the computer is turned on and the program is started, it has to be written to a **non-volatile** storage medium, such a hard drive, usb drive, or CD-RW.

Data on non-volatile storage media is stored in named locations on the media called **files**. By reading and writing files, programs can save information between program runs.

Working with files is a lot like working with a notebook. To use a notebook, it has to be opened. When done, it has to be closed. While the notebook is open, it can either be read from or written to. In either case, the notebook holder knows where they are. They can read the whole notebook in its natural order or they can skip around.

All of this applies to files as well. To open a file, we specify its name and indicate whether we want to read or write.

## 13.2. Writing our first file

Let's begin with a simple program that writes three lines of text into a file:

```
1 myfile = open("test.txt", "w")
2 myfile.write("My first file written from Python\n")
3 myfile.write("-----\n")
4 myfile.write("Hello, world!\n")
5 myfile.close()
```

Opening a file creates what we call a **file handle**. In this example, the variable `myfile` refers to the new handle object. Our program calls methods on the handle, and this makes changes to the actual file which is usually located on our disk.

On line 1, the `open` function takes two arguments. The first is the name of the file, and the second is the **mode**. Mode "`w`" means that we are opening the file for writing.

With mode "w", if there is no file named `test.txt` on the disk, it will be created. If there already is one, it will be replaced by the file we are writing.

To put data in the file we invoke the `write` method on the handle, shown in lines 2, 3 and 4 above. In bigger programs, lines 2–4 will usually be replaced by a loop that writes many more lines into the file.

Closing the file handle (line 5) tells the system that we are done writing and makes the disk file available for reading by other programs (or by our own program).

### A handle is somewhat like a TV remote control

We're all familiar with a remote control for a TV. We perform operations on the remote control — switch channels, change the volume, etc. But the real action happens on the TV. So, by simple analogy, we'd call the remote control our handle to the underlying TV.

Sometimes we want to emphasize the difference — the file handle is not the same as the file, and the remote control is not the same as the TV. But at other times we prefer to treat them as a single mental chunk, or abstraction, and we'll just say “close the file”, or “flip the TV channel”.

## 13.3. Reading a file line-at-a-time

Now that the file exists on our disk, we can open it, this time for reading, and read all the lines in the file, one at a time. This time, the mode argument is "r" for reading:

```
1 mynewhandle = open("test.txt", "r")
2 while True:                                # Keep reading forever
3     theline = mynewhandle.readline()          # Try to read next line
4     if len(theline) == 0:                    # If there are no more lines
5         break                                 #      leave the loop
6
7     # Now process the line we've just read
8     print(theline, end="")
9
10 mynewhandle.close()
```

This is a handy pattern for our toolbox. In bigger programs, we'd squeeze more extensive logic into the body of the loop at line 8 —for example, if each line of the file contained the name and email address of one of our friends, perhaps we'd split the line into some pieces and call a function to send the friend a party invitation.

On line 8 we suppress the newline character that `print` usually appends to our strings. Why? This is because the string already has its own newline: the `readline` method in line 3 returns everything up to *and including* the newline character. This also explains the end-of-file detection logic: when

there are no more lines to be read from the file, `readline` returns an empty string — one that does not even have a newline at the end, hence its length is 0.

**Fail first ...**

In our sample case here, we have three lines in the file, yet we enter the loop *four* times. In Python, you only learn that the file has no more lines by failure to read another line. In some other programming languages (e.g. Pascal), things are different: there you read three lines, but you have what is called *look ahead* — after reading the third line you already know that there are no more lines in the file. You're not even allowed to try to read the fourth line.

So the templates for working line-at-a-time in Pascal and Python are subtly different!

When you transfer your Python skills to your next computer language, be sure to ask how you'll know when the file has ended: is the style in the language “try, and after you fail you'll know”, or is it “look ahead”?

If we try to open a file that doesn't exist, we get an error:

```
1 >>> mynewhandle = open("wharrah.txt", "r")
2 IOError: [Errno 2] No such file or directory: "wharrah.txt"
```

## 13.4. Turning a file into a list of lines

It is often useful to fetch data from a disk file and turn it into a list of lines. Suppose we have a file containing our friends and their email addresses, one per line in the file. But we'd like the lines sorted into alphabetical order. A good plan is to read everything into a list of lines, then sort the list, and then write the sorted list back to another file:

```
1 f = open("friends.txt", "r")
2 xs = f.readlines()
3 f.close()
4
5 xs.sort()
6
7 g = open("sortedfriends.txt", "w")
8 for v in xs:
9     g.write(v)
10 g.close()
```

The `readlines` method in line 2 reads all the lines and returns a list of the strings.

We could have used the template from the previous section to read each line one-at-a-time, and to build up the list ourselves, but it is a lot easier to use the method that the Python implementors gave us!

## 13.5. Reading the whole file at once

Another way of working with text files is to read the complete contents of the file into a string, and then to use our string-processing skills to work with the contents.

We'd normally use this method of processing files if we were not interested in the line structure of the file. For example, we've seen the `split` method on strings which can break a string into words. So here is how we might count the number of words in a file:

```
1 f = open("somefile.txt")
2 content = f.read()
3 f.close()
4
5 words = content.split()
6 print("There are {} words in the file.".format(len(words)))
```

Notice here that we left out the "r" mode in line 1. By default, if we don't supply the mode, Python opens the file for reading.

Your file paths may need to be explicitly named.

In the above example, we're assuming that the file `somefile.txt` is in the same directory as your Python source code. If this is not the case, you may need to provide a full or a relative path to the file. On Windows, a full path could look like "`C:\\temp\\somefile.txt`", while on a Unix system the full path could be "`/home/jimmy/somefile.txt`".

We'll return to this later in this chapter.

## 13.6. Working with binary files

Files that hold photographs, videos, zip files, executable programs, etc. are called **binary** files: they're not organized into lines, and cannot be opened with a normal text editor. Python works just as easily with binary files, but when we read from the file we're going to get bytes back rather than a string. Here we'll copy one binary file to another:

```
1 f = open("somefile.zip", "rb")
2 g = open("thecopy.zip", "wb")
3
4 while True:
5     buf = f.read(1024)
6     if len(buf) == 0:
7         break
8     g.write(buf)
9
10 f.close()
11 g.close()
```

There are a few new things here. In lines 1 and 2 we added a "b" to the mode to tell Python that the files are binary rather than text files. In line 5, we see `read` can take an argument which tells it how many bytes to attempt to read from the file. Here we chose to read and write up to 1024 bytes on each iteration of the loop. When we get back an empty buffer from our attempt to read, we know we can break out of the loop and close both the files.

If we set a breakpoint at line 6, (or `print type(buf)` there) we'll see that the type of `buf` is `bytes`. We don't do any detailed work with `bytes` objects in this textbook.

## 13.7. An example

Many useful line-processing programs will read a text file line-at-a-time and do some minor processing as they write the lines to an output file. They might number the lines in the output file, or insert extra blank lines after every 60 lines to make it convenient for printing on sheets of paper, or extract some specific columns only from each line in the source file, or only print lines that contain a specific substring. We call this kind of program a **filter**.

Here is a filter that copies one file to another, omitting any lines that begin with #:

```
1 def filter(oldfile, newfile):
2     infile = open(oldfile, "r")
3     outfile = open(newfile, "w")
4     while True:
5         text = infile.readline()
6         if len(text) == 0:
7             break
8         if text[0] == "#":
9             continue
10        # Put any more processing logic here
11        outfile.write(text)
```

```
13
14     infile.close()
15     outfile.close()
```

The `continue` statement at line 9 skips over the remaining lines in the current iteration of the loop, but the loop will still iterate. This style looks a bit contrived here, but it is often useful to say “*get the lines we’re not concerned with out of the way early, so that we have cleaner more focused logic in the meaty part of the loop that might be written around line 11.*”

Thus, if `text` is the empty string, the loop exits. If the first character of `text` is a hash mark, the flow of execution goes to the top of the loop, ready to start processing the next line. Only if both conditions fail do we fall through to do the processing at line 11, in this example, writing the line into the new file.

Let’s consider one more case: suppose our original file contained empty lines. At line 6 above, would this program find the first empty line in the file, and terminate immediately? No! Recall that `readline` always includes the newline character in the string it returns. It is only when we try to read *beyond* the end of the file that we get back the empty string of length 0.

## 13.8. Directories

Files on non-volatile storage media are organized by a set of rules known as a **file system**. File systems are made up of files and **directories**, which are containers for both files and other directories.

When we create a new file by opening it and writing, the new file goes in the current directory (wherever we were when we ran the program). Similarly, when we open a file for reading, Python looks for it in the current directory.

If we want to open a file somewhere else, we have to specify the **path** to the file, which is the name of the directory (or folder) where the file is located:

```
1 >>> wordsfile = open("/usr/share/dict/words", "r")
2 >>> wordlist = wordsfile.readlines()
3 >>> print(wordlist[:6])
4 [ '\n', 'A\n', "A's\n", 'AOL\n', "AOL's\n", 'Aachen\n' ]
```

This (Unix) example opens a file named `words` that resides in a directory named `dict`, which resides in `share`, which resides in `usr`, which resides in the top-level directory of the system, called `/`. It then reads in each line into a list using `readlines`, and prints out the first 5 elements from that list.

A Windows path might be “`c:/temp/words.txt`” or “`c:\\temp\\\\words.txt`”. Because backslashes are used to escape things like newlines and tabs, we need to write two backslashes in a literal string to get one! So the length of these two strings is the same!

We cannot use / or \ as part of a filename; they are reserved as a **delimiter** between directory and filenames.

The file /usr/share/dict/words should exist on Unix-based systems, and contains a list of words in alphabetical order.

## 13.9. What about fetching something from the web?

The Python libraries are pretty messy in places. But here is a very simple example that copies the contents at some web URL to a local file.

```
1 import urllib.request  
2  
3 url = "https://www.ietf.org/rfc/rfc793.txt"  
4 destination_filename = "rfc793.txt"  
5  
6 urllib.request.urlretrieve(url, destination_filename)
```

The `urlretrieve` function — just one call — could be used to download any kind of content from the Internet.

We'll need to get a few things right before this works:

- The resource we're trying to fetch must exist! Check this using a browser.
- We'll need permission to write to the destination filename, and the file will be created in the “current directory” - i.e. the same folder that the Python program is saved in.
- If we are behind a proxy server that requires authentication, (as some students are), this may require some more special handling to work around our proxy. Use a local resource for the purpose of this demonstration!

Here is a slightly different example. Rather than save the web resource to our local disk, we read it directly into a string, and return it:

```
1 import urllib.request  
2  
3 def retrieve_page(url):  
4     """ Retrieve the contents of a web page.  
5         The contents is converted to a string before returning it.  
6     """  
7     my_socket = urllib.request.urlopen(url)  
8     dta = str(my_socket.read())  
9     my_socket.close()
```

```
10     return dta
11
12 the_text = retrieve_page("https://www.ietf.org/rfc/rfc793.txt")
13 print(the_text)
```

Opening the remote url returns what we call a **socket**. This is a handle to our end of the connection between our program and the remote web server. We can call read, write, and close methods on the socket object in much the same way as we can work with a file handle.

## 13.10. Glossary

### **delimiter**

A sequence of one or more characters used to specify the boundary between separate parts of text.

### **directory**

A named collection of files, also called a folder. Directories can contain files and other directories, which are referred to as *subdirectories* of the directory that contains them.

### **file**

A named entity, usually stored on a hard drive, floppy disk, or CD-ROM, that contains a stream of characters.

### **file system**

A method for naming, accessing, and organizing files and the data they contain.

### **handle**

An object in our program that is connected to an underlying resource (e.g. a file). The file handle lets our program manipulate/read/write/close the actual file that is on our disk.

### **mode**

A distinct method of operation within a computer program. Files in Python can be opened in one of four modes: read ("r"), write ("w"), append ("a"), and read and write ("+").

### **non-volatile memory**

Memory that can maintain its state without power. Hard drives, flash drives, and rewritable compact disks (CD-RW) are each examples of non-volatile memory.

### **path**

A sequence of directory names that specifies the exact location of a file.

**text file**

A file that contains printable characters organized into lines separated by newline characters.

**socket**

One end of a connection allowing one to read and write information to or from another computer.

**volatile memory**

Memory which requires an electrical current to maintain state. The *main memory* or RAM of a computer is volatile. Information stored in RAM is lost when the computer is turned off.

## 13.11. Exercises

1. Write a program that reads a file and writes out a new file with the lines in reversed order (i.e. the first line in the old file becomes the last one in the new file.)
2. Write a program that reads a file and prints only those lines that contain the substring `snake`.
3. Write a program that reads a text file and produces an output file which is a copy of the file, except the first five columns of each line contain a four digit line number, followed by a space. Start numbering the first line in the output file at 1. Ensure that every line number is formatted to the same width in the output file. Use one of your Python programs as test data for this exercise: your output should be a printed and numbered listing of the Python program.
4. Write a program that undoes the numbering of the previous exercise: it should read a file with numbered lines and produce another file without line numbers.

# Chapter 14: List Algorithms

This chapter is a bit different from what we've done so far: rather than introduce more new Python syntax and features, we're going to focus on the program development process, and some algorithms that work with lists.

As in all parts of this book, our expectation is that you, the reader, will copy our code into your Python environment, play and experiment, and work along with us.

Part of this chapter works with the book *Alice in Wonderland* and a *vocabulary file*. Download these two files to your local machine at the following links. [https://learnpythontherightway.com/\\_downloads/alice\\_in\\_wonderland.txt<sup>13</sup>](https://learnpythontherightway.com/_downloads/alice_in_wonderland.txt<sup>13</sup>) and [https://learnpythontherightway.com/\\_downloads/vocab.txt<sup>14</sup>](https://learnpythontherightway.com/_downloads/vocab.txt<sup>14</sup>).

## 14.1. Test-driven development

Early in our Fruitful functions chapter we introduced the idea of *incremental development*, where we added

small fragments of code to slowly build up the whole, so that we could easily find problems early. Later in that same chapter we introduced unit testing and gave code for our testing framework so that we could capture, in code, appropriate tests for the functions we were writing.

**Test-driven development (TDD)** is a software development practice which takes these practices one step further. The key idea is that automated tests should be written *first*. This technique is called

*test-driven* because — if we are to believe the extremists — non-testing code should only be written when there is a failing test to make pass.

We can still retain our mode of working in small incremental steps, but now we'll define and express those steps in terms of a sequence of increasingly sophisticated unit tests that demand more from our code at each stage.

We'll turn our attention to some standard algorithms that process lists now, but as we proceed through this chapter we'll attempt to do so in the spirit envisaged by TDD.

## 14.2. The linear search algorithm

We'd like to know the index where a specific item occurs within in a list of items. Specifically, we'll return the index of the item if it is found, or we'll return -1 if the item doesn't occur in the list. Let

---

<sup>13</sup>[https://learnpythontherightway.com/\\_downloads/alice\\_in\\_wonderland.txt](https://learnpythontherightway.com/_downloads/alice_in_wonderland.txt)

<sup>14</sup>[https://learnpythontherightway.com/\\_downloads/vocab.txt](https://learnpythontherightway.com/_downloads/vocab.txt)

us

start with some tests:

```

1 friends = ["Joe", "Zoe", "Brad", "Angelina", "Zuki", "Thandi", "Paris"]
2 test(search_linear(friends, "Zoe") == 1)
3 test(search_linear(friends, "Joe") == 0)
4 test(search_linear(friends, "Paris") == 6)
5 test(search_linear(friends, "Bill") == -1)

```

Motivated by the fact that our tests don't even run, let alone pass, we now write the function:

```

1 def search_linear(xs, target):
2     """ Find and return the index of target in sequence xs """
3     for (i, v) in enumerate(xs):
4         if v == target:
5             return i
6     return -1

```

There are a some points to learn here: We've seen a similar algorithm in section 8.10 when we searched for a character in a string. There we used a `while` loop, here we've used a `for` loop, coupled with `enumerate` to extract the `(i, v)` pair on each iteration. There are other variants — for example, we could have used `range` and made the loop run only over the indexes, or we could have used the idiom of returning `None` when the item was not found in the list. But the essential similarity in all these variations is that we test every item in the list in turn, from first to last, using the pattern of the short-circuit *eureka*

*traversal* that we introduced earlier —that we return from the function as soon as we find the target that we're looking for.

Searching all items of a sequence from first to last is called a **linear search**. Each time we check whether `v == target` we'll call it a **probe**. We like to count probes as a measure of how efficient our algorithm is, and this will be a good enough indication of how long our algorithm will take to execute.

Linear searching is characterized by the fact that the number of probes needed to find some target depends directly on the length of the list. So if the list becomes ten times bigger, we can expect to wait ten times longer when searching for things. Notice too, that if we're searching for a target that is not present in the list, we'll have to go all the way to the end before we can return the negative value. So this case needs  $N$  probes, where  $N$  is the length of the list. However, if we're searching for a target that does exist in the list, we could be lucky and find it immediately in position 0, or we might have to look further, perhaps even all the way to the last item. On average, when the target is present, we're going to need to go about halfway through the list, or  $N/2$  probes.

We say that this search has **linear performance** (linear meaning straight line) because, if we were to measure the average search times for different sizes of lists ( $N$ ), and then plot a graph of time-to-search against  $N$ , we'd get a more-or-less straight line graph.

Analysis like this is pretty meaningless for small lists — the computer is quick enough not to bother if the list only has a handful of items. So generally, we’re interested in the **scalability** of our algorithms — how do they perform if we throw bigger problems at them. Would this search be a sensible one to use if we had a million or ten million items (perhaps the catalog of books in your local library) in our list? What happens for really large datasets, e.g. how does Google search so brilliantly well?

## 14.3. A more realistic problem

As children learn to read, there are expectations that their vocabulary will grow. So a child of age 14 is expected to know more words than a child of age 8. When prescribing reading books for a grade, an important question might be “*which words in this book are not in the expected vocabulary at this level?*”

Let us assume we can read a vocabulary of words into our program, and read the text of a book, and split it into words. Let us write some tests for what we need to do next. Test data can usually be very small, even if we intend to finally use our program for larger cases:

```

1 vocab = ["apple", "boy", "dog", "down",
           "fell", "girl", "grass", "the", "tree"]
2 book_words = "the apple fell from the tree to the grass".split()
3 test(find_unknown_words(vocab, book_words) == ["from", "to"])
4 test(find_unknown_words([], book_words) == book_words)
5 test(find_unknown_words(vocab, ["the", "boy", "fell"]) == [])

```

Notice we were a bit lazy, and used `split` to create our list of words —it is easier than typing out the list, and very convenient if you want to input a sentence into the program and turn it into a list of words.

We now need to implement the function for which we’ve written tests, and we’ll make use of our linear search. The basic strategy is to run through each of the words in the book, look it up in the vocabulary, and if it is not in the vocabulary, save it into a new resulting list which we return from the function:

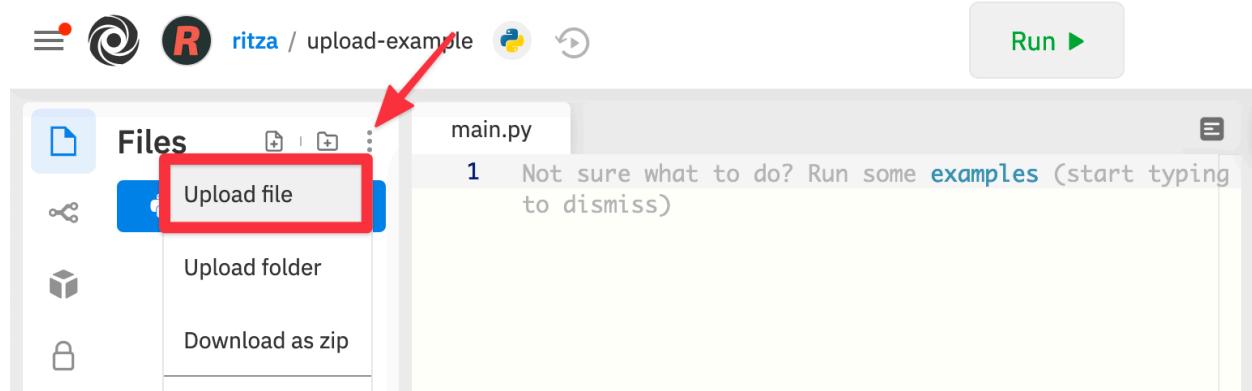
```

1 def find_unknown_words(vocab, wds):
2     """ Return a list of words in wds that do not occur in vocab """
3     result = []
4     for w in wds:
5         if (search_linear(vocab, w) < 0):
6             result.append(w)
7     return result

```

We can happily report now that the tests all pass.

Now let us look at the scalability. We have more realistic vocabulary in the text file that could be downloaded at the beginning of this chapter. Upload the `vocab.txt` file to a new repl so that you can access it from your code.



Now let us read in the file (as a single string) and split it into a list of words. For convenience, we'll create a function to do this for us, and test it on the vocab file.

```
1 def load_words_from_file(filename):
2     """ Read words from filename, return list of words. """
3     f = open(filename, "r")
4     file_content = f.read()
5     f.close()
6     wds = file_content.split()
7     return wds
8
9 bigger_vocab = load_words_from_file("vocab.txt")
10 print("There are {0} words in the vocab, starting with\n {1} "
11       .format(len(bigger_vocab), bigger_vocab[:6]))
```

Python responds with:

```
1 There are 19469 words in the vocab, starting with
2 ['a', 'aback', 'abacus', 'abandon', 'abandoned', 'abandonment']
```

So we've got a more sensible size vocabulary. Now let us load up a book, once again we'll use the one we downloaded at the beginning of this chapter. Loading a book is much like loading words from a file, but

we're going to do a little extra black magic. Books are full of punctuation, and have mixtures of lowercase and uppercase letters. We need to clean up the contents of the book. This will involve removing punctuation, and converting everything to the same case (lowercase, because our vocabulary is all in lowercase). So we'll want a more sophisticated way of converting text to words.

```

1 test(text_to_words("My name is Earl!")) == ["my", "name", "is", "earl"])
2 test(text_to_words('"Well, I never!', said Alice.')) ==
3                         ["well", "i", "never", "said", "alice"])

```

There is a powerful `translate` method available for strings. The idea is that one sets up desired substitutions — for every character, we can give a corresponding replacement character. The `translate` method will apply these replacements throughout the whole string. So here we go:

```

1 def text_to_words(the_text):
2     """ return a list of words with all punctuation removed,
3         and all in lowercase.
4     """
5
6     my_substitutions = the_text.maketrans(
7         # If you find any of these
8         "ABCDEFGHIJKLMNPQRSTUVWXYZ0123456789!\"#$%&()*+, -./: ;<=>?@[ ]^_`{|}~'\\",
9         # Replace them by these
10        "abcdefghijklmnopqrstuvwxyz")
11
12    # Translate the text now.
13    cleaned_text = the_text.translate(my_substitutions)
14    wds = cleaned_text.split()
15    return wds

```

The translation turns all uppercase characters into lowercase, and all punctuation characters and digits into spaces. Then, of course, `split` will get rid of the spaces as it breaks the text into a list of words.

The tests pass.

Now we're ready to read in our book:

```

1 def get_words_in_book(filename):
2     """ Read a book from filename, and return a list of its words. """
3     f = open(filename, "r")
4     content = f.read()
5     f.close()
6     wds = text_to_words(content)
7     return wds
8
9 book_words = get_words_in_book("alice_in_wonderland.txt")
10 print("There are {0} words in the book, the first 100 are\n{1}".
11       format(len(book_words), book_words[:100]))

```

Python prints the following (all on one line, we've cheated a bit for the textbook):

```

1 There are 27336 words in the book, the first 100 are
2 ['alice', 's', 'adventures', 'in', 'wonderland', 'lewis', 'carroll',
3   'chapter', 'i', 'down', 'the', 'rabbit', 'hole', 'alice', 'was',
4   'beginning', 'to', 'get', 'very', 'tired', 'of', 'sitting', 'by',
5   'her', 'sister', 'on', 'the', 'bank', 'and', 'of', 'having',
6   'nothing', 'to', 'do', 'once', 'or', 'twice', 'she', 'had',
7   'peeped', 'into', 'the', 'book', 'her', 'sister', 'was', 'reading',
8   'but', 'it', 'had', 'no', 'pictures', 'or', 'conversations', 'in',
9   'it', 'and', 'what', 'is', 'the', 'use', 'of', 'a', 'book',
10  'thought', 'alice', 'without', 'pictures', 'or', 'conversation',
11  'so', 'she', 'was', 'considering', 'in', 'her', 'own', 'mind',
12  'as', 'well', 'as', 'she', 'could', 'for', 'the', 'hot', 'day',
13  'made', 'her', 'feel', 'very', 'sleepy', 'and', 'stupid',
14  'whether', 'the', 'pleasure', 'of', 'making', 'a']

```

Well now we have all the pieces ready. Let us see what words in this book are not in the vocabulary:

```
1 >>> missing_words = find_unknown_words(bigger_vocab, book_words)
```

We wait a considerable time now, something like a minute, before Python finally works its way through this, and prints a list of 3398 words in the book that are not in the vocabulary. Mmm... This is not particularly scalable. For a vocabulary that is twenty times larger (you'll often find school dictionaries with 300 000 words, for example), and longer books, this is going to be slow. So let us make some timing measurements while we think about how we can improve this in the next section.

```

1 import time
2
3 t0 = time.process_time()
4 missing_words = find_unknown_words(bigger_vocab, book_words)
5 t1 = time.process_time()
6 print("There are {0} unknown words.".format(len(missing_words)))
7 print("That took {0:.4f} seconds.".format(t1-t0))

```

We get the results and some timing that we can refer back to later:

```

1 There are 3398 unknown words.
2 That took 49.8014 seconds.

```

## 14.4. Binary Search

If you think about what we've just done, it is not how we work in real life. If you were given a vocabulary and asked to tell if some word was present, you'd probably start in the middle. You can do this because the

vocabulary is ordered — so you can probe some word in the middle, and immediately realize that your target was before (or perhaps after) the one you had probed. Applying this principle repeatedly leads us to a

very much better algorithm for searching in a list of items that are already ordered. (Note that if the items are not ordered, you have little choice other than to look through all of them. But, if we know the items are in order, we can improve our searching technique).

Lets start with some tests. Remember, the list needs to be sorted:

```
1 xs = [2,3,5,7,11,13,17,23,29,31,37,43,47,53]
2 test(search_binary(xs, 20) == -1)
3 test(search_binary(xs, 99) == -1)
4 test(search_binary(xs, 1) == -1)
5 for (i, v) in enumerate(xs):
6     test(search_binary(xs, v) == i)
```

Even our test cases are interesting this time: notice that we start with items not in the list and look at boundary conditions — in the middle of the list, less than all items in the list, bigger than the biggest.

Then we use a loop to use every list item as a target, and to confirm that our binary search returns the corresponding index of that item in the list.

It is useful to think about having a *region-of-interest* (ROI) within the list being searched. This ROI will be the portion of the list in which it is still possible that our target might be found. Our algorithm will start with the ROI set to all the items in the list. On the first probe in the middle of the ROI, there are three possible outcomes: either we find the target, or we learn that we can discard the top half

of the ROI, or we learn that we can discard the bottom half of the ROI. And we keep doing this repeatedly, until we find our target, or until we end up with no more items in our region of interest. We can code this as follows:

```

1 def search_binary(xs, target):
2     """ Find and return the index of key in sequence xs """
3     lb = 0
4     ub = len(xs)
5     while True:
6         if lb == ub:    # If region of interest (ROI) becomes empty
7             return -1
8
9         # Next probe should be in the middle of the ROI
10        mid_index = (lb + ub) // 2
11
12        # Fetch the item at that position
13        item_at_mid = xs[mid_index]
14
15        print("ROI[{0}:{1}](size={2}), probed='{3}', target='{4}'"
16              .format(lb, ub, ub-lb, item_at_mid, target))
17
18        # How does the probed item compare to the target?
19        if item_at_mid == target:
20            return mid_index      # Found it!
21        if item_at_mid < target:
22            lb = mid_index + 1    # Use upper half of ROI next time
23        else:
24            ub = mid_index      # Use lower half of ROI next time

```

The region of interest is represented by two variables, a lower bound `lb` and an upper bound `ub`. It is important to be precise about what values these indexes have. We'll make `lb` hold the index of the first

item in the ROI, and make `ub` hold the index just *beyond* the last item of interest. So these semantics are similar to a Python slice semantics: the region of interest is exactly the slice `xs[1b:ub]`. (The algorithm never actually takes any array slices!)

With this code in place, our tests pass. Great. Now if we substitute a call to this search algorithm instead of calling the `search_linear` in `find_unknown_words`, can we improve our performance? Let's do that, and again run this test:

```

1 t0 = time.process_time()
2 missing_words = find_unknown_words(bigger_vocab, book_words)
3 t1 = time.process_time()
4 print("There are {0} unknown words.".format(len(missing_words)))
5 print("That took {0:.4f} seconds.".format(t1-t0))

```

What a spectacular difference! More than 200 times faster!

- 1 There are 3398 unknown words.
- 2 That took 0.2262 seconds.

Why is this binary search so much faster than the linear search? If we uncomment the print statement on lines 15 and 16, we'll get a trace of the probes done during a search. Let's go ahead, and try that:

```

1  >>> search_binary(bigger_vocab, "magic")
2  ROI[0:19469](size=19469), probed='known', target='magic'
3  ROI[9735:19469](size=9734), probed='retailer', target='magic'
4  ROI[9735:14602](size=4867), probed='overthrow', target='magic'
5  ROI[9735:12168](size=2433), probed='mission', target='magic'
6  ROI[9735:10951](size=1216), probed='magnificent', target='magic'
7  ROI[9735:10343](size=608), probed='liken', target='magic'
8  ROI[10040:10343](size=303), probed='looks', target='magic'
9  ROI[10192:10343](size=151), probed='lump', target='magic'
10 ROI[10268:10343](size=75), probed='machete', target='magic'
11 ROI[10306:10343](size=37), probed='mafia', target='magic'
12 ROI[10325:10343](size=18), probed='magnanimous', target='magic'
13 ROI[10325:10334](size=9), probed='magical', target='magic'
14 ROI[10325:10329](size=4), probed='maggot', target='magic'
15 ROI[10328:10329](size=1), probed='magic', target='magic'
16 10328

```

Here we see that finding the target word “magic” needed just 14 probes before it was found at index 10328. The important thing is that each probe halves (with some truncation) the remaining region of interest. By

contrast, the linear search would have needed 10329 probes to find the same target word.

The word *binary* means *two*. Binary search gets its name from the fact that each probe splits the list into two pieces and discards the one half from the region of interest.

The beauty of the algorithm is that we could double the size of the vocabulary, and it would only need one more probe! And after another doubling, just another one probe. So as the vocabulary gets bigger, this algorithm's performance becomes even more impressive.

Can we put a formula to this? If our list size is  $N$ , what is the biggest number of probes  $k$  we could need? The maths is a bit easier if we turn the question around: how big a list  $N$  could we deal with, given that we

were only allowed to make  $k$  probes?

With 1 probe, we can only search a list of size 1. With two probes we could cope with lists up to size 3 - (test the middle item with the first probe, then test either the left or right sublist with the second probe). With one more probe, we could cope with 7 items (the middle item, and two sublists of size 3). With four probes, we can search 15 items, and 5 probes lets us search up to 31 items. So the general relationship is given by the formula

```
1 N = 2 ** k - 1
```

where  $k$  is the number of probes we're allowed to make, and  $N$  is the maximum size of the list that can be searched in that many probes. This function is *exponential* in  $k$  (because  $k$  occurs in the exponent part).

If we wanted to turn the formula around and solve for  $k$  in terms of  $N$ , we need to move the constant 1 to the other side, and take a  $\log$  (base 2) on each side. (The  $\log$  is the inverse of an exponent.) So the formula

for  $k$  in terms of  $N$  is now:

$$k = \lceil \log_2(N + 1) \rceil$$

The square-only-on-top brackets are called *ceiling brackets*: this means that you must round the number up to the next whole integer.

Let us try this on a calculator, or in Python, which is the mother of all calculators: suppose I have 1000 elements to be searched, what is the maximum number of probes I'll need? (There is a pesky  $+1$  in the formula, so let us not forget to add it on...):

```
1 >>> from math import log
2 >>> log(1000 + 1, 2)
3 9.967226258835993
```

Telling us that we'll need 9.96 probes maximum, to search 1000 items is not quite what we want. We forgot to take the ceiling. The `ceil` function in the `math` module does exactly this. So more accurately, now:

```
1 >>> from math import log, ceil
2 >>> ceil(log(1000 + 1, 2))
3 10
4 >>> ceil(log(1000000 + 1, 2))
5 20
6 >>> ceil(log(1000000000 + 1, 2))
7 30
```

This tells us that searching 1000 items needs 10 probes. (Well technically, with 10 probes we can search exactly 1023 items, but the easy and useful stuff to remember here is that "1000 items needs 10 probes, a million needs 20 probes, and a billion items only needs 30 probes").

You will rarely encounter algorithms that scale to large datasets as beautifully as binary search does!

## 14.5. Removing adjacent duplicates from a list

We often want to get the unique elements in a list, i.e. produce a new list in which each different element occurs just once. Consider our case of looking for words in Alice in Wonderland that are not in our vocabulary. We had a report that there are 3398 such words, but there are duplicates in that list. In fact, the word “alice” occurs 398 times in the book, and it is not in our vocabulary! How should we remove these duplicates?

A good approach is to sort the list, then remove all adjacent duplicates. Let us start with removing adjacent duplicates

```

1 test(remove_adjacent_dups([1,2,3,3,3,3,5,6,9,9]) == [1,2,3,5,6,9])
2 test(remove_adjacent_dups([]) == [])
3 test(remove_adjacent_dups(["a", "big", "big", "bite", "dog"]) ==
4                 ["a", "big", "bite", "dog"])

```

The algorithm is easy and efficient. We simply have to remember the most recent item that was inserted into the result, and avoid inserting it again:

```

1 def remove_adjacent_dups(xs):
2     """ Return a new list in which all adjacent
3         duplicates from xs have been removed.
4     """
5     result = []
6     most_recent_elem = None
7     for e in xs:
8         if e != most_recent_elem:
9             result.append(e)
10            most_recent_elem = e
11
12    return result

```

The amount of work done in this algorithm is linear — each item in `xs` causes the loop to execute exactly once, and there are no nested loops. So doubling the number of elements in `xs` should cause this function to run twice as long: the relationship between the size of the list and the time to run will be graphed as a straight (linear) line.

Let us go back now to our analysis of Alice in Wonderland. Before checking the words in the book against the vocabulary, we’ll sort those words into order, and eliminate duplicates. So our new code looks like this:

```
1 all_words = get_words_in_book("alice_in_wonderland.txt")
2 all_words.sort()
3 book_words = remove_adjacent_dups(all_words)
4 print("There are {0} words in the book. Only {1} are unique.".
5         format(len(all_words), len(book_words)))
6 print("The first 100 words are\n{0}".
7         format(book_words[:100]))
```

Almost magically, we get the following output:

```
1 There are 27336 words in the book. Only 2570 are unique.
2 The first 100 words are
3 ['_i_', 'a', 'abide', 'able', 'about', 'above', 'absence', 'absurd',
4  'acceptance', 'accident', 'accidentally', 'account', 'accounting',
5  'accounts', 'accusation', 'accustomed', 'ache', 'across', 'act',
6  'actually', 'ada', 'added', 'adding', 'addressed', 'addressing',
7  'adjourn', 'adoption', 'advance', 'advantage', 'adventures',
8  'advice', 'advisable', 'advise', 'affair', 'affectionately',
9  'afford', 'afore', 'afraid', 'after', 'afterwards', 'again',
10 'against', 'age', 'ago', 'agony', 'agree', 'ah', 'ahem', 'air',
11 'airs', 'alarm', 'alarmed', 'alas', 'alice', 'alive', 'all',
12 'allow', 'almost', 'alone', 'along', 'aloud', 'already', 'also',
13 'altered', 'alternately', 'altogether', 'always', 'am', 'ambition',
14 'among', 'an', 'ancient', 'and', 'anger', 'angrily', 'angry',
15 'animal', 'animals', 'ann', 'annoy', 'annoyed', 'another',
16 'answer', 'answered', 'answers', 'antipathies', 'anxious',
17 'anxiously', 'any', 'anything', 'anywhere', 'appealed', 'appear',
18 'appearance', 'appeared', 'appearing', 'applause', 'apple',
19 'apples', 'arch']
```

Lewis Carroll was able to write a classic piece of literature using only 2570 different words!

## 14.6. Merging sorted lists

Suppose we have two sorted lists. Devise an algorithm to merge them together into a single sorted list.

A simple but inefficient algorithm could be to simply append the two lists together, and sort the result:

```

1 newlist = (xs + ys)
2 newlist.sort()

```

But this doesn't take advantage of the fact that the two lists are already sorted, and is going to have poor scalability and performance for very large lists.

Lets get some tests together first:

```

1 xs = [1,3,5,7,9,11,13,15,17,19]
2 ys = [4,8,12,16,20,24]
3 zs = xs+ys
4 zs.sort()
5 test(merge(xs, []) == xs)
6 test(merge([], ys) == ys)
7 test(merge([], []) == [])
8 test(merge(xs, ys) == zs)
9 test(merge([1,2,3], [3,4,5]) == [1,2,3,3,4,5])
10 test(merge(["a", "big", "cat"], ["big", "bite", "dog"]) ==
11           ["a", "big", "big", "bite", "cat", "dog"])

```

Here is our merge algorithm:

```

1 def merge(xs, ys):
2     """ merge sorted lists xs and ys. Return a sorted result """
3     result = []
4     xi = 0
5     yi = 0
6
7     while True:
8         if xi >= len(xs):          # If xs list is finished,
9             result.extend(ys[yi:]) # Add remaining items from ys
10            return result        # And we're done.
11
12         if yi >= len(ys):          # Same again, but swap roles
13             result.extend(xs[xi:])
14            return result
15
16         # Both lists still have items, copy smaller item to result.
17         if xs[xi] <= ys[yi]:
18             result.append(xs[xi])
19             xi += 1
20         else:
21             result.append(ys[yi])
22             yi += 1

```

The algorithm works as follows: we create a result list, and keep two indexes, one into each list (lines 3-5). On each iteration of the loop, whichever list item is smaller is copied to the result list, and that list's index is advanced. As soon as either index reaches the end of its list, we copy all the remaining items from the other list into the result, which we return.

## 14.7. Alice in Wonderland, again!

Underlying the algorithm for merging sorted lists is a deep pattern of computation that is widely reusable. The pattern essence is “*Run through the lists always processing the smallest remaining items from each, with these cases to consider:*”

- What should we do when either list has no more items?
- What should we do if the smallest items from each list are equal to each other?
- What should we do if the smallest item in the first list is smaller than the smallest one in the second list?
- What should we do in the remaining case?

Lets assume we have two sorted lists. Exercise your algorithmic skills by adapting the merging algorithm pattern for each of these cases:

- Return only those items that are present in both lists.
- Return only those items that are present in the first list, but not in the second.
- Return only those items that are present in the second list, but not in the first.
- Return items that are present in either the first or the second list.
- Return items from the first list that are not eliminated by a matching element in the second list. In this case, an item in the second list “knocks out” just one matching item in the first list. This operation is sometimes called *bagdiff*. For example `bagdiff([5,7,11,11,11,12,13], [7,8,11])` would return `[5,11,11,12,13]`

In the previous section we sorted the words from the book, and eliminated duplicates. Our vocabulary is also sorted. So third case above — find all items in the second list that are not in the first list, would be another way to implement `find_unknown_words`. Instead of searching for every word in the dictionary (either by linear or binary search), why not use a variant of the merge to return the words that occur in the book, but not in the vocabulary.

```

1 def find_unknowns_merge_pattern(vocab, wds):
2     """ Both the vocab and wds must be sorted.  Return a new
3         list of words from wds that do not occur in vocab.
4     """
5
6     result = []
7     xi = 0
8     yi = 0
9
10    while True:
11        if xi >= len(vocab):
12            result.extend(wds[yi:])
13            return result
14
15        if yi >= len(wds):
16            return result
17
18        if vocab[xi] == wds[yi]: # Good, word exists in vocab
19            yi += 1
20
21        elif vocab[xi] < wds[yi]: # Move past this vocab word,
22            xi += 1
23
24        else:                  # Got word that is not in vocab
25            result.append(wds[yi])
26            yi += 1

```

Now we put it all together:

Even more stunning performance here:

```

1 There are 828 unknown words.
2 That took 0.0410 seconds.

```

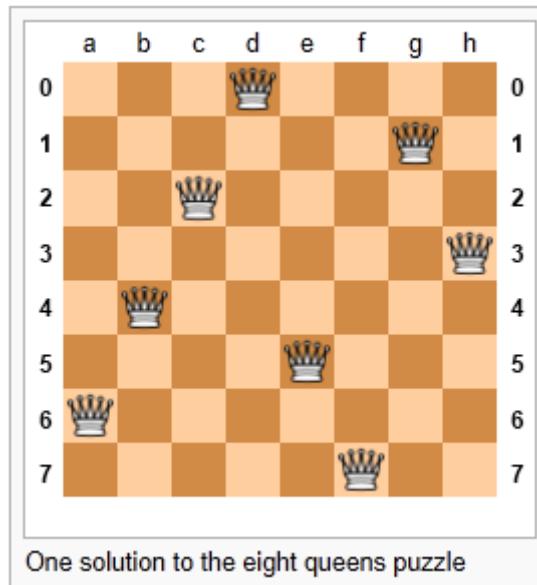
Let's review what we've done. We started with a word-by-word linear lookup in the vocabulary that ran in about 50 seconds. We implemented a clever binary search, and got that down to 0.22 seconds, more than 200

times faster. But then we did something even better: we sorted the words from the book, eliminated duplicates, and used a merging pattern to find words from the book that were not in the dictionary. This was about five times faster than even the binary lookup algorithm. At the end of the chapter our algorithm is more than a 1000 times faster than our first attempt!

That is what we can call a good day at the office!

## 14.8. Eight Queens puzzle, part 1

As told by Wikipedia, “*The eight queens puzzle is the problem of placing eight chess queens on an 8x8 chessboard so that no two queens attack each other. Thus, a solution requires that no two queens share the same row, column, or diagonal.*”



Please try this yourself, and find a few more solutions by hand.

We’d like to write a program to find solutions to this puzzle. In fact, the puzzle generalizes to placing N queens on an NxN board, so we’re going to think about the general case, not just the 8x8 case. Perhaps we can find solutions for 12 queens on a 12x12 board, or 20 queens on a 20x20 board.

How do we approach a complex problem like this? A good starting point is to think about our *data structures* – how exactly do we plan to represent the state of the chessboard and its queens in our program? Once we have some handle on what our puzzle is going to look like in memory, we can begin to think about the functions and logic we’ll need to solve the puzzle, i.e. how do we put another queen onto the board

somewhere, and to check whether it clashes with any of the queens already on the board.

The steps of finding a good representation, and then finding a good algorithm to operate on the data cannot always be done independently of each other. As you think about the operations you require, you may want

to change or reorganize the data somewhat to make it easier to do the operations you need.

This relationship between algorithms and data was elegantly expressed in the title of a book *Algorithms + Data Structures = Programs*, written by one of the pioneers in Computer Science,

Niklaus Wirth, the inventor  
of Pascal.

Let's brainstorm some ideas about how a chessboard and queens could be represented in memory.

- A two dimensional matrix (a list of 8 lists, each containing 8 squares) is one possibility. At each square of the board would like to know whether it contains a queen or not — just two possible states for each square — so perhaps each element in the lists could be True or False, or, more simply, 0 or 1.

Our state for the solution above could then have this data representation:

```

1      bd1 = [[0,0,0,1,0,0,0,0],
2                  [0,0,0,0,0,0,1,0],
3                  [0,0,1,0,0,0,0,0],
4                  [0,0,0,0,0,0,0,1],
5                  [0,1,0,0,0,0,0,0],
6                  [0,0,0,0,1,0,0,0],
7                  [1,0,0,0,0,0,0,0],
8                  [0,0,0,0,0,1,0,0]]
```

You should also be able to see how the empty board would be represented, and you should start to imagine what operations or changes you'd need to make to the data to place another queen somewhere on the board.

- Another idea might be to keep a list of coordinates of where the queens are. Using the notation in the illustration, for example, we could represent the state of that solution as:

```
1      bd2 = [ "a6", "b4", "c2", "d0", "e5", "f7", "g1", "h3" ]
```

- We could make other tweaks to this — perhaps each element in this list should rather be a tuple, with integer coordinates for both axes. And being good computer scientists, we'd probably start numbering each axis from 0 instead of at 1. Now our representation could be:

```
1      bd3 = [(0,6), (1,4), (2,2), (3,0), (4,5), (5,7), (6,1), (7,3)]
```

- Looking at this representation, we can't help but notice that the first coordinates are  $0, 1, 2, 3, 4, 5, 6, 7$  and they correspond exactly to the index position of the pairs in the list. So we could discard them, and come up with this really compact alternative representation of the solution:

```
1      bd4 = [6, 4, 2, 0, 5, 7, 1, 3]
```

This will be what we'll use, let's see where that takes us.

Let us now take some grand insight into the problem. Do you think it is a coincidence that there are no repeated numbers in the solution? The solution  $[6, 4, 2, 0, 5, 7, 1, 3]$  contains the numbers  $0, 1, 2, 3, 4, 5, 6, 7$ , but none are duplicated! Could other solutions contain duplicate numbers, or not?

A little thinking should convince you that there can never be duplicate numbers in a solution: the numbers represent the row on which the queen is placed, and because we are never permitted to put two queens in the same row, no solution will ever have duplicate row numbers in it.

Our key insight

*In our representation, any solution to the N queens problem must therefore be a permutation of the numbers  $[0 .. N-1]$ .*

Note that not all permutations are solutions. For example,  $[0, 1, 2, 3, 4, 5, 6, 7]$  has all queens on the same diagonal.

Wow, we seem to be making progress on this problem merely by thinking, rather than coding!

Our algorithm should start taking shape now. We can start with the list  $[0..N-1]$ , generate various permutations of that list, and check each permutation to see if it has any clashes (queens that are on the same diagonal). If it has no clashes, it is a solution, and we can print it.

Let us be precise and clear on this issue: if we only use permutations of the rows, and we're using our compact representation, no queens can clash on either rows or columns, and we don't even have to concern ourselves with those cases. So the only clashes we need to test for are clashes on the diagonals.

It sounds like a useful function will be one that can test if two queens share a diagonal. Each queen is on some  $(x,y)$  position. So does the queen at  $(5,2)$  share a diagonal with the one at  $(2,0)$ ? Does  $(5,2)$  clash with  $(3,0)$ ?

```

1 test(not share_diagonal(5,2,2,0))
2 test(share_diagonal(5,2,3,0))
3 test(share_diagonal(5,2,4,3))
4 test(share_diagonal(5,2,4,1))

```

A little geometry will help us here. A diagonal has a slope of either 1 or -1. The question we really want to ask is *is their distance between them the same in the x and the y direction?* If it is, they share a diagonal. Because diagonals can be to the left or right, it will make sense for this program to use the absolute distance in each direction:

```

1 def share_diagonal(x0, y0, x1, y1):
2     """ Is (x0, y0) on a shared diagonal with (x1, y1)? """
3     dy = abs(y1 - y0)          # Calc the absolute y distance
4     dx = abs(x1 - x0)          # CXalc the absolute x distance
5     return dx == dy            # They clash if dx == dy

```

If you copy the code and run it, you'll be happy to learn that the tests pass!

Now let's consider how we construct a solution by hand. We'll put a queen somewhere in the first column, then place one in the second column, only if it does not clash with the one already on the board. And

then we'll put a third one on, checking it against the two queens already to its left. When we consider the queen on column 6, we'll need to check for clashes against those in all the columns to its left, i.e. in columns 0,1,2,3,4,5.

So the next building block is a function that, given a partially completed puzzle, can check whether the queen at column c clashes with any of the queens to its left, at columns 0,1,2,..c-1:

```

1 # Solutions cases that should not have any clashes
2 test(not col_clashes([6,4,2,0,5], 4))
3 test(not col_clashes([6,4,2,0,5,7,1,3], 7))
4
5 # More test cases that should mostly clash
6 test(col_clashes([0,1], 1))
7 test(col_clashes([5,6], 1))
8 test(col_clashes([6,5], 1))
9 test(col_clashes([0,6,4,3], 3))
10 test(col_clashes([5,0,7], 2))
11 test(not col_clashes([2,0,1,3], 1))
12 test(col_clashes([2,0,1,3], 2))

```

Here is our function that makes them all pass:

```

1 def col_clashes(bs, c):
2     """ Return True if the queen at column c clashes
3         with any queen to its left.
4     """
5     for i in range(c):      # Look at all columns to the left of c
6         if share_diagonal(i, bs[i], c, bs[c]):
7             return True
8
9     return False           # No clashes - col c has a safe placement.

```

Finally, we're going to give our program one of our permutations — i.e. all queens placed somewhere, one on each row, one on each column. But does the permutation have any diagonal clashes?

```

1 test(not has_clashes([6,4,2,0,5,7,1,3])) # Solution from above
2 test(has_clashes([4,6,2,0,5,7,1,3]))      # Swap rows of first two
3 test(has_clashes([0,1,2,3]))              # Try small 4x4 board
4 test(not has_clashes([2,0,3,1]))          # Solution to 4x4 case

```

And the code to make the tests pass:

```

1 def has_clashes(the_board):
2     """ Determine whether we have any queens clashing on the diagonals.
3         We're assuming here that the_board is a permutation of column
4         numbers, so we're not explicitly checking row or column clashes.
5     """
6     for col in range(1,len(the_board)):
7         if col_clashes(the_board, col):
8             return True
9     return False

```

Summary of what we've done so far: we now have a powerful function called `has_clashes` that can tell if a configuration is a solution to the queens puzzle. Let's get on now with generating lots of permutations and finding solutions!

## 14.9. Eight Queens puzzle, part 2

This is the fun, easy part. We could try to find all permutations of  $[0,1,2,3,4,5,6,7]$  — that might be algorithmically challenging, and would be a *brute force* way of tackling the problem. We just try everything, and find all possible solutions.

Of course we know there are  $N!$  permutations of  $N$  things, so we can get an early idea of how long it would take to search all of them for all solutions. Not too long at all, actually  $-8!$  is only 40320 different

cases to check out. This is vastly better than starting with 64 places to put eight queens. If you do the sums for how many ways can you choose 8 of the 64 squares for your queens, the formula (called  $N$  choose  $k$ )

where you're choosing  $k=8$  squares of the available  $N=64$ ) yields a whopping 4426165368, obtained from  $(64! / (8! \times 56!))$ .

So our earlier key insight — that we only need to consider permutations — has reduced what we call the *problem space* from about 4.4 billion cases to just 40320!

We're not even going to explore all those, however. When we introduced the random number module, we learned that it had a `shuffle` method that randomly permuted a list of items. So we're going to write a “random”

algorithm to find solutions to the  $N$  queens problem. We'll begin with the permutation  $[0,1,2,3,4,5,6,7]$  and we'll repeatedly shuffle the list, and test each to see if it works! Along the way we'll count how many attempts we need before we find each solution, and we'll find 10 solutions (we could hit the same solution more than once, because `shuffle` is random!):

```
1 def main():
2     import random
3     rng = random.Random()    # Instantiate a generator
4
5     bd = list(range(8))      # Generate the initial permutation
6     num_found = 0
7     tries = 0
8     while num_found < 10:
9         rng.shuffle(bd)
10        tries += 1
11        if not has_clashes(bd):
12            print("Found solution {0} in {1} tries.".format(bd, tries))
13            tries = 0
14            num_found += 1
15
16 main()
```

Almost magically, and at great speed, we get this:

```
1 Found solution [3, 6, 2, 7, 1, 4, 0, 5] in 693 tries.  
2 Found solution [5, 7, 1, 3, 0, 6, 4, 2] in 82 tries.  
3 Found solution [3, 0, 4, 7, 1, 6, 2, 5] in 747 tries.  
4 Found solution [1, 6, 4, 7, 0, 3, 5, 2] in 428 tries.  
5 Found solution [6, 1, 3, 0, 7, 4, 2, 5] in 376 tries.  
6 Found solution [3, 0, 4, 7, 5, 2, 6, 1] in 204 tries.  
7 Found solution [4, 1, 7, 0, 3, 6, 2, 5] in 98 tries.  
8 Found solution [3, 5, 0, 4, 1, 7, 2, 6] in 64 tries.  
9 Found solution [5, 1, 6, 0, 3, 7, 4, 2] in 177 tries.  
10 Found solution [1, 6, 2, 5, 7, 4, 0, 3] in 478 tries.
```

Here is an interesting fact. On an 8x8 board, there are known to be 92 different solutions to this puzzle. We are randomly picking one of 40320 possible permutations of our representation. So our chances of picking a solution on each try are  $92/40320$ . Put another way, on average we'll need  $40320/92$  tries — about 438.26 — before we stumble across a solution. The number of tries we printed looks like our experimental data agrees quite nicely with our theory!

Save this code for later.

In the chapter on PyGame we plan to write a module to draw the board with its queens, and integrate that module with this code.

## 14.10. Glossary

### **binary search**

A famous algorithm that searches for a target in a sorted list. Each probe in the list allows us to discard half the remaining items, so the algorithm is very efficient.

### **linear**

Relating to a straight line. Here, we talk about graphing how the time taken by an algorithm depends on the size of the data it is processing. Linear algorithms have straight-line graphs that can describe this relationship.

### **linear search**

A search that probes each item in a list or sequence, from first, until it finds what it is looking for. It is used for searching for a target in unordered lists of items.

### **Merge algorithm**

An efficient algorithm that merges two already sorted lists, to produce a sorted list result. The merge algorithm is really a pattern of

computation that can be adapted and reused for various other scenarios, such as finding words that are in a book, but not in a vocabulary.

### probe

Each time we take a look when searching for an item is called a probe. In our chapter on `<span class="title-ref">Iteration</span>` we also played a guessing game where the computer tried to guess the user's secret number. Each of those tries would also be called a probe.

### test-driven development (TDD)

A software development practice which arrives at a desired feature through a series of small, iterative steps motivated by automated tests which are *written first* that express increasing refinements of the desired feature. (see the Wikipedia article on [Test-driven development<sup>15</sup>](#) for more information.)

## 14.11. Exercises

1. The section in this chapter called Alice in Wonderland, again started with the observation that the merge

algorithm uses a pattern that can be reused in other situations. Adapt the merge algorithm to write each of these functions, as was suggested there:

1. Return only those items that are present in both lists.
  2. Return only those items that are present in the first list, but not in the second.
  3. Return only those items that are present in the second list, but not in the first.
  4. Return items that are present in either the first or the second list.
  5. Return items from the first list that are not eliminated by a matching element in the second list. In this case, an item in the second list "knocks out" just one matching item in the first list. This operation is sometimes called *bagdiff*. For example `bagdiff([5,7,11,11,11,12,13], [7,8,11])` would return `[5,11,11,12,13]`
2. Modify the queens program to solve some boards of size 4, 12, and 16. What is the maximum size puzzle you

---

<sup>15</sup>[http://en.wikipedia.org/wiki/Test\\_driven\\_development](http://en.wikipedia.org/wiki/Test_driven_development)

can usually solve in under a minute?

3. Adapt the queens program so that we keep a list of solutions that have already printed, so that we don't

print the same solution more than once.

4. Chess boards are symmetric: if we have a solution to the queens problem, its mirror solution — either

flipping the board on the X or in the Y axis, is also a solution. And giving the board a 90 degree, 180 degree, or 270 degree rotation is also a solution. In some sense, solutions that are just mirror images or rotations of other solutions — in the same family — are less interesting than the unique “core cases”. Of the 92 solutions for the 8 queens problem, there are only 12 unique families if you take rotations and mirror images into account. Wikipedia has some fascinating stuff about this.

1. Write a function to mirror a solution in the Y axis,

2. Write a function to mirror a solution in the X axis,

3. Write a function to rotate a solution by 90 degrees anti-clockwise, and use this to provide 180 and 270 degree rotations too.

4. Write a function which is given a solution, and it generates the family of symmetries for that solution. For example, the symmetries of [0, 4, 7, 5, 2, 6, 1, 3] are :

```
[[0,4,7,5,2,6,1,3],[7,1,3,0,6,4,2,5],  
[4,6,1,5,2,0,3,7],[2,5,3,1,7,4,6,0],  
[3,1,6,2,5,7,4,0],[0,6,4,7,1,3,5,2],  
[7,3,0,2,5,1,6,4],[5,2,4,6,0,3,1,7]]
```

5. Now adapt the queens program so it won't list solutions that are in the same family. It only prints solutions from unique families.

5. Every week a computer scientist buys four lotto tickets. She always chooses the same prime numbers, with

the hope that if she ever hits the jackpot, she will be able to go onto TV and Facebook and tell everyone her secret. This will suddenly create widespread public interest in prime numbers, and will be the trigger event that ushers in a new age of enlightenment. She represents her weekly tickets in

Python as a list of lists:

```
my_tickets = [ [ 7, 17, 37, 19, 23, 43],  
[ 7, 2, 13, 41, 31, 43],  
[ 2, 5, 7, 11, 13, 17],  
[13, 17, 37, 19, 23, 43] ]
```

Complete these exercises.

1. Each lotto draw takes six random balls, numbered from 1 to 49.

Write a function to return a lotto draw.

2. Write a function that compares a single ticket and a draw, and returns the number of correct picks on that ticket:

```
test(lotto_match([42,4,7,11,1,13], [2,5,7,11,13,17]) == 3)
```

3. Write a function that takes a list of tickets and a draw, and returns a list telling how many picks were correct on each ticket:

```
test(lotto_matches([42,4,7,11,1,13], my_tickets) == [1,2,3,1])
```

4. Write a function that takes a list of integers, and returns the number of primes in the list:

```
test(primes_in([42, 4, 7, 11, 1, 13]) == 3)
```

5. Write a function to discover whether the computer scientist has missed any prime numbers in her selection of the four tickets. Return a list of all primes that she has missed:

```
test(prime_misses(my_tickets) == [3, 29, 47])
```

6. Write a function that repeatedly makes a new draw, and compares the draw to the four tickets.

1. Count how many draws are needed until one of the computer scientist's tickets has at least 3 correct picks. Try the experiment twenty times, and average out the number of draws needed.
2. How many draws are needed, on average, before she gets at least 4 picks correct?
3. How many draws are needed, on average, before she gets at least 5 correct? (Hint: this might take a while. It would be nice if you could print some dots, like a progress bar, to show when each of the 20 experiments has completed.)

Notice that we have difficulty constructing test cases here, because our random numbers are not deterministic. Automated testing only really works if you already know what the answer should be!

6. Read *Alice in Wonderland*. You can read the plain text version we have with this textbook, or if you

have e-book reader software on your PC, or a Kindle, iPhone, Android, etc. you'll be able to find a suitable version for your device at <http://www.gutenberg.org>. They also have html and pdf versions, with pictures, and thousands of other classic books!

# Chapter 15: Classes and Objects — the Basics

## 15.1. Object-oriented programming

Python is an **object-oriented programming** language, which means that it provides features that support **object-oriented programming**<sup>16</sup> (OOP).

Object-oriented programming has its roots in the 1960s, but it wasn't until the mid 1980s that it became the main **programming paradigm**<sup>17</sup> used in the creation of new software. It was developed as a way to handle the rapidly increasing size and complexity of software systems, and to make it easier to modify these large and complex systems over time.

Up to now, most of the programs we have been writing use a **procedural programming**<sup>18</sup> paradigm. In procedural programming the focus is on writing functions or *procedures* which operate on data. In object-oriented programming the focus is on the creation of **objects** which contain both data and functionality together. (We have seen turtle objects, string objects,

and random number generators, to name a few places where we've already worked with objects.)

Usually, each object definition corresponds to some object or concept in the real world, and the functions that operate on that object correspond to the ways real-world objects interact.

## 15.2. User-defined compound data types

We've already seen classes like `str`, `int`, `float` and `Turtle`. We are now ready to create our own user-defined class: the `Point`.

Consider the concept of a mathematical point. In two dimensions, a point is two numbers (coordinates) that are treated collectively as a single object. 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.

Some of the typical operations that one associates with points might be calculating the distance of a point from the origin, or from another point, or finding a midpoint of two points, or asking if a

---

<sup>16</sup>[http://en.wikipedia.org/wiki/Object-oriented\\_programming](http://en.wikipedia.org/wiki/Object-oriented_programming)

<sup>17</sup>[http://en.wikipedia.org/wiki/Programming\\_paradigm](http://en.wikipedia.org/wiki/Programming_paradigm)

<sup>18</sup>[http://en.wikipedia.org/wiki/Procedural\\_programming](http://en.wikipedia.org/wiki/Procedural_programming)

point falls

within a given rectangle or circle. We'll shortly see how we can organize these together with the data.

A natural way to represent a point in Python is with two numeric values. The question, then, is how to group these two values into a compound object. The quick and dirty solution is to use a tuple, and for some applications that might be a good choice.

An alternative is to define a new **class**. This approach involves a bit more effort, but it has advantages that will be apparent soon. We'll want our points to each have an **x** and a **y** attribute, so our first class definition looks like this:

```
1 class Point:  
2     """ Point class represents and manipulates x,y coords. """  
3  
4     def __init__(self):  
5         """ Create a new point at the origin """  
6         self.x = 0  
7         self.y = 0
```

Class definitions can appear anywhere in a program, but they are usually near the beginning (after the `import` statements). Some programmers and languages prefer to put every class in a module of its own — we won't do that here. The syntax rules for a class definition are the same as for other compound statements. There is a header which begins with the keyword, `class`, followed by the name of the class, and ending with a colon. Indentation levels tell us where the class ends.

If the first line after the class header is a string, it becomes the docstring of the class, and will be recognized by various tools. (This is also the way docstrings work in functions.)

Every class should have a method with the special name `__init__`. This **initializer method** is automatically called whenever a new instance of `Point` is created. It gives the programmer the opportunity to set up the attributes required within the new instance by giving them their initial state/values. The `self` parameter (we could choose any other name, but `self` is the convention) is automatically set to reference the newly created object that needs to be initialized.

So let's use our new `Point` class now:

```
1 p = Point()          # Instantiate an object of type Point  
2 q = Point()          # Make a second point  
3  
4 print(p.x, p.y, q.x, q.y) # Each point object has its own x and y
```

This program prints:

```
1 0 0 0 0
```

because during the initialization of the objects, we created two attributes called `x` and `y` for each, and gave them both the value 0.

This should look familiar — we've used classes before to create more than one object:

```
1 from turtle import Turtle
2
3 tess = Turtle()      # Instantiate objects of type Turtle
4 alex = Turtle()
```

The variables `p` and `q` are assigned references to two new `Point` objects. A function like `Turtle` or `Point` that creates a new object instance is called a **constructor**, and every class automatically provides a constructor function which is named the same as the class.

It may be helpful to think of a class as a *factory* for making objects. The class itself isn't an instance of a point, but it contains the machinery to make point instances. Every time we call the constructor, we're asking the factory to make us a new object. As the object comes off the production line, its initialization method is executed to get the object properly set up with its factory default settings.

The combined process of “make me a new object” and “get its settings initialized to the factory default settings” is called **instantiation**.

## 15.3. Attributes

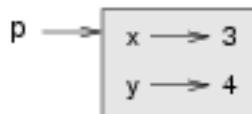
Like real world objects, object instances have both attributes and methods.

We can modify the attributes in an instance using dot notation:

```
1 >>> p.x = 3
2 >>> p.y = 4
```

Both modules and instances create their own namespaces, and the syntax for accessing names contained in each, called **attributes**, is the same. In this case the attribute we are selecting is a data item from an instance.

The following state diagram shows the result of these assignments:



The variable `p` refers to a `Point` object, which contains two attributes. Each attribute refers to a number.

We can access the value of an attribute using the same syntax:

```

1 >>> print(p.y)
2 4
3 >>> x = p.x
4 >>> print(x)
5 3

```

The expression `p.x` means, “Go to the object `p` refers to and get the value of `x`”. In this case, we assign that value to a variable named `x`. There is no conflict between the variable `x` (in the global namespace here) and the attribute `x` (in the namespace belonging to the instance). The purpose of dot notation is to fully qualify which variable we are referring to unambiguously.

We can use dot notation as part of any expression, so the following statements are legal:

```

1 print("x={0}, y={1})".format(p.x, p.y)
2 distance_squared_from_origin = p.x * p.x + p.y * p.y

```

The first line outputs `(x=3, y=4)`. The second line calculates the value 25.

## 15.4. Improving our initializer

To create a point at position (7, 6) currently needs three lines of code:

```

1 p = Point()
2 p.x = 7
3 p.y = 6

```

We can make our class constructor more general by placing extra parameters into the `__init__`-method, as shown in this example:

```

1 class Point:
2     """ Point class represents and manipulates x,y coords. """
3
4     def __init__(self, x=0, y=0):
5         """ Create a new point at x, y """
6         self.x = x
7         self.y = y
8
9 # Other statements outside the class continue below here.

```

The `x` and `y` parameters here are both optional. If the caller does not supply arguments, they’ll get the default values of 0. Here is our improved class in action:

```

1 >>> p = Point(4, 2)
2 >>> q = Point(6, 3)
3 >>> r = Point()          # r represents the origin (0, 0)
4 >>> print(p.x, q.y, r.x)
5 4 3 0

```

### Technically speaking ...

If we are really fussy, we would argue that the `__init__` method's docstring is inaccurate. `__init__` doesn't *create* the object (i.e. set aside memory for it), — it just initializes the object to its factory-default settings after its creation.

But tools like PyScripter understand that instantiation — creation and initialization — happen together, and they choose to display the *initializer's* docstring as the tooltip to guide the programmer that calls the class constructor.

So we're writing the docstring so that it makes the most sense when it pops up to help the programmer who is using our `Point` class:

The screenshot shows a code editor window with Python code. A tooltip is displayed over the line `q = Point(  
 x=0, y=0  
 Create a new point at x, y`. The tooltip contains the docstring for the `__init__` method, which is `""" Create a new point at x, y """`.

```

1
2 class Point:
3     """ Point class represents and manipulates x,y coords. """
4
5     def __init__(self, x=0, y=0):
6         """ Create a new point at x, y """
7         self.x = x
8         self.y = y
9
10 # Other statements outside the class continue below here.
11
12 q = Point(
    x=0, y=0
    Create a new point at x, y

```

## 15.5. Adding other methods to our class

The key advantage of using a class like `Point` rather than a simple tuple `(6, 7)` now becomes apparent. We can add methods to the `Point` class that are sensible operations for points, but which may not be

appropriate for other tuples like `(25, 12)` which might represent, say, a day and a month, e.g. Christmas day. So being able to calculate the distance from the origin is sensible for points, but not for `(day, month)` data. For `(day, month)` data, we'd like different operations, perhaps to find what day of the week it will fall on in 2020.

Creating a class like `Point` brings an exceptional amount of “organizational power” to our programs,

and to our thinking. We can group together the sensible operations, and the kinds of data they apply to, and each instance of the class can have its own state.

A **method** behaves like a function but it is invoked on a specific instance, e.g. `tess.right(90)`. Like a data attribute, methods are accessed using dot notation.

Let's add another method, `distance_from_origin`, to see better how methods work:

```
1  class Point:  
2      """ Create a new Point, at coordinates x, y """  
3  
4      def __init__(self, x=0, y=0):  
5          """ Create a new point at x, y """  
6          self.x = x  
7          self.y = y  
8  
9      def distance_from_origin(self):  
10         """ Compute my distance from the origin """  
11         return ((self.x ** 2) + (self.y ** 2)) ** 0.5
```

Let's create a few point instances, look at their attributes, and call our new method on them: (We must run our program first, to make our `Point` class available to the interpreter.)

```
1  >>> p = Point(3, 4)  
2  >>> p.x  
3  3  
4  >>> p.y  
5  4  
6  >>> p.distance_from_origin()  
7  5.0  
8  >>> q = Point(5, 12)  
9  >>> q.x  
10 5  
11 >>> q.y  
12 12  
13 >>> q.distance_from_origin()  
14 13.0  
15 >>> r = Point()  
16 >>> r.x  
17 0  
18 >>> r.y  
19 0  
20 >>> r.distance_from_origin()  
21 0.0
```

When defining a method, the first parameter refers to the instance being manipulated. As already noted, it is customary to name this parameter `self`.

Notice that the caller of `distance_from_origin` does not explicitly supply an argument to match the `self` parameter — this is done for us, behind our back.

## 15.6. Instances as arguments and parameters

We can pass an object as an argument in the usual way. We've already seen this in some of the turtle examples, where we passed the turtle to some function like `draw_bar` in the chapter titled Conditionals, so that the function could control and use whatever turtle instance we passed to it.

Be aware that our variable only holds a reference to an object, so passing `tess` into a function creates an alias: both the caller and the called function now have a reference, but there is only one turtle!

Here is a simple function involving our new `Point` objects:

```
1 def print_point(pt):
2     print("{0}, {1}".format(pt.x, pt.y))
```

`print_point` takes a point as an argument and formats the output in whichever way we choose. If we call `print_point(p)` with point `p` as defined previously, the output is `(3, 4)`.

## 15.7. Converting an instance to a string

Most object-oriented programmers probably would not do what we've just done in `print_point`. When we're working with classes and objects, a preferred alternative is to add a new method to the class. And we don't

like chatterbox methods that call `print`. A better approach is to have a method so that every instance can produce a string representation of itself. Let's initially call it `to_string`:

```
1 class Point:
2     # ...
3
4     def to_string(self):
5         return "{0}, {1}".format(self.x, self.y)
```

Now we can say:

```
1 >>> p = Point(3, 4)
2 >>> print(p.to_string())
3 (3, 4)
```

But don't we already have a `str` type converter that can turn our object into a string? Yes! And doesn't `print` automatically use this when printing things? Yes again! But these automatic mechanisms do not yet do exactly what we want:

```
1 >>> str(p)
2 '<__main__.Point object at 0x01F9AA10>'
3 >>> print(p)
4 '<__main__.Point object at 0x01F9AA10>'
```

Python has a clever trick up its sleeve to fix this. If we call our new method `__str__` instead of `to_string`, the Python interpreter will use our code whenever it needs to convert a `Point` to a string. Let's re-do this again, now:

```
1 class Point:
2     # ...
3
4     def __str__(self):      # All we have done is renamed the method
5         return "{0}, {1}".format(self.x, self.y)
```

and now things are looking great!

```
1 >>> str(p)      # Python now uses the __str__ method that we wrote.
2 (3, 4)
3 >>> print(p)
4 (3, 4)
```

## 15.8. Instances as return values

Functions and methods can return instances. For example, given two `Point` objects, find their midpoint. First we'll write this as a regular function:

```
1 def midpoint(p1, p2):
2     """ Return the midpoint of points p1 and p2 """
3     mx = (p1.x + p2.x)/2
4     my = (p1.y + p2.y)/2
5     return Point(mx, my)
```

The function creates and returns a new `Point` object:

```
1 >>> p = Point(3, 4)
2 >>> q = Point(5, 12)
3 >>> r = midpoint(p, q)
4 >>> r
5 (4.0, 8.0)
```

Now let us do this as a method instead. Suppose we have a point object, and wish to find the midpoint halfway between it and some other target point:

```
1 class Point:
2     # ...
3
4     def halfway(self, target):
5         """ Return the halfway point between myself and the target """
6         mx = (self.x + target.x)/2
7         my = (self.y + target.y)/2
8         return Point(mx, my)
```

This method is identical to the function, aside from some renaming. Its usage might be like this:

```
1 >>> p = Point(3, 4)
2 >>> q = Point(5, 12)
3 >>> r = p.halfway(q)
4 >>> r
5 (4.0, 8.0)
```

While this example assigns each point to a variable, this need not be done. Just as function calls are composable, method calls and object instantiation are also composable, leading to this alternative that uses no variables:

```
1 >>> print(Point(3, 4).halfway(Point(5, 12)))
2 (4.0, 8.0)
```

## 15.9. A change of perspective

The original syntax for a function call, `print_time(current_time)`, 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 considered the active agents. An invocation like `current_time.print_time()` says “Hey *current\_time!* Please print yourself!”

In our early introduction to turtles, we used an object-oriented style, so that we said `tess.forward(100)`, which asks the turtle to move itself forward by the given number of steps.

This change in perspective might be more polite, but it may not initially be obvious that it is useful. But sometimes shifting responsibility from the functions onto the objects makes it possible to write more versatile functions, and makes it easier to maintain and reuse code.

The most important advantage of the object-oriented style is that it fits our mental chunking and real-life experience more accurately. In real life our `cook` method is part of our microwave oven — we don’t

have a `cook` function sitting in the corner of the kitchen, into which we pass the microwave! Similarly, we use the cellphone’s own methods to send an sms, or to change its state to silent. The functionality of

real-world objects tends to be tightly bound up inside the objects themselves. OOP allows us to accurately mirror this when we organize our programs.

## 15.10. Objects can have state

Objects are most useful when we also need to keep some state that is updated from time to time. Consider a turtle object. Its state consists of things like its position, its heading, its color, and its shape. A

method like `left(90)` updates the turtle’s heading, `forward` changes its position, and so on.

For a bank account object, a main component of the state would be the current balance, and perhaps a log of all transactions. The methods would allow us to query the current balance, deposit new funds, or make

a payment. Making a payment would include an amount, and a description, so that this could be added to the transaction log. We’d also want a method to show the transaction log.

## 15.11. Glossary

**attribute**

One of the named data items that makes up an instance.

**class**

A user-defined compound type. A class can also be thought of as a template for the objects that are instances of it. (The iPhone is a class. By December 2010, estimates are that 50 million instances had been sold!)

**constructor**

Every class has a “factory”, called by the same name as the class, for making new instances. If the class has an *initializer method*, this method is used to get the attributes (i.e. the state) of the new object properly set up.

**initializer method**

A special method in Python (called `__init__`) that is invoked automatically to set a newly created object’s attributes to their initial (factory-default) state.

**instance**

An object whose type is of some class. Instance and object are used interchangeably.

**instantiate**

To create an instance of a class, and to run its initializer.

**method**

A function that is defined inside a class definition and is invoked on instances of that class.

**object**

A compound data type that is often used to model a thing or concept in the real world. It bundles together the data and the operations that are relevant for that kind of data. Instance and object are used interchangeably.

**object-oriented programming**

A powerful style of programming in which data and the operations that manipulate it are organized into objects.

**object-oriented language**

A language that provides features, such as user-defined classes and inheritance, that facilitate object-oriented programming.

## 15.12. Exercises

1. Rewrite the `distance` function from the chapter titled *Fruitful functions* so that it takes two `Points` as parameters instead of four numbers.
2. Add a method `reflect_x` to `Point` which returns a new `Point`, one which is the reflection of the point about the x-axis. For example, `Point(3, 5).reflect_x()` is `(3, -5)`
3. Add a method `slope_from_origin` which returns the slope of the line joining the origin to the point. For example:

```
1 >>> Point(4, 10).slope_from_origin()  
2 2.5
```

What cases will cause this method to fail?

4. The equation of a straight line is “ $y = ax + b$ ”, (or perhaps “ $y = mx + c$ ”). The coefficients `a` and `b` completely describe the line. Write a method in the `Point` class so that if a point instance is given another point, it will compute the equation of the straight line joining the two points. It must return the two coefficients as a tuple of two values. For example, :

```
1 >>> print(Point(4, 11).get_line_to(Point(6, 15)))  
2 >>> (2, 3)
```

This tells us that the equation of the line joining the two points is “ $y = 2x + 3$ ”. When will this method fail?

5. Given four points that fall on the circumference of a circle, find the midpoint of the circle. When will this function fail?

*Hint:* You *must* know how to solve the geometry problem *before* you think of going anywhere near programming. You cannot program a solution to a problem if you don't understand what you want the computer to do!

6. Create a new class, SMS\_store. The class will instantiate SMS\_store objects, similar to an inbox or outbox on a cellphone:

```
1 my_inbox = SMS_store()
```

This store can hold multiple SMS messages (i.e. its internal state will just be a list of messages). Each message will be represented as a tuple:

```
1 (has Been_viewed, from_number, time_arrived, text_of_SMS)
```

The inbox object should provide these methods:

```
1 my_inbox.add_new_arrival(from_number, time_arrived, text_of_SMS)
2     # Makes new SMS tuple, inserts it after other messages
3     # in the store. When creating this message, its
4     # has Been_viewed status is set False.
5
6 my_inbox.message_count()
7     # Returns the number of sms messages in my_inbox
8
9 my_inbox.get_unread_indexes()
10    # Returns list of indexes of all not-yet-viewed SMS messages
11
12 my_inbox.get_message(i)
13    # Return (from_number, time_arrived, text_of_sms) for message[i]
14    # Also change its state to "has been viewed".
15    # If there is no message at position i, return None
16
17 my_inbox.delete(i)      # Delete the message at index i
18 my_inbox.clear()        # Delete all messages from inbox
```

Write the class, create a message store object, write tests for these methods, and implement the methods.

# Chapter 16: Classes and Objects — Digging a little deeper

## 16.1. Rectangles

Let's say that we want a class to represent a rectangle which is located somewhere in the XY plane. The question is, what information do we have to provide in order to specify such a rectangle? To keep things simple, assume that the rectangle is oriented either vertically or horizontally, never at an angle.

There are a few possibilities: we could specify the center of the rectangle (two coordinates) and its size (width and height); or we could specify one of the corners and the size; or we could specify two opposing corners. A conventional choice is to specify the upper-left corner of the rectangle, and the size.

Again, we'll define a new class, and provide it with an initializer and a string converter method:

```
1  class Rectangle:
2      """ A class to manufacture rectangle objects """
3
4      def __init__(self, posn, w, h):
5          """ Initialize rectangle at posn, with width w, height h """
6          self.corner = posn
7          self.width = w
8          self.height = h
9
10     def __str__(self):
11         return "{0}, {1}, {2})".format(self.corner, self.width, self.height)
12
13
14 box = Rectangle(Point(0, 0), 100, 200)
15 bomb = Rectangle(Point(100, 80), 5, 10)    # In my video game
16 print("box: ", box)
17 print("bomb: ", bomb)
```

To specify the upper-left corner, we have embedded a `Point` object (as we used it in the previous chapter) within our new `Rectangle` object! We create two new `Rectangle` objects, and then print them producing:

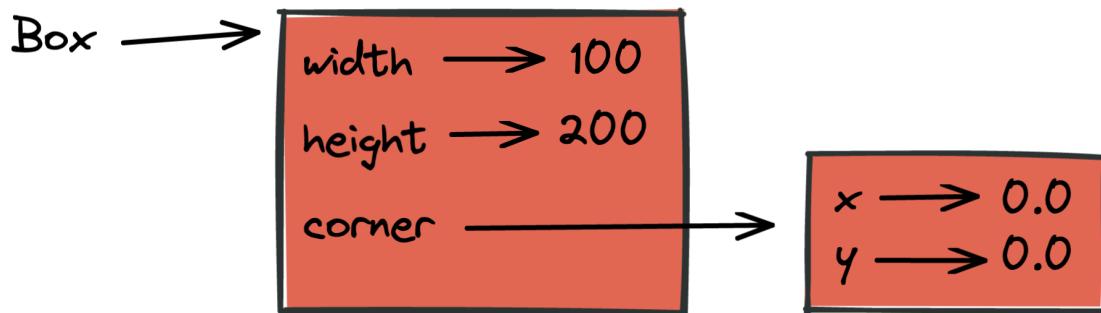
```

1 box: ((0, 0), 100, 200)
2 bomb: ((100, 80), 5, 10)

```

The dot operator composes. The expression `box.corner.x` means, “Go to the object that `box` refers to and select its attribute named `corner`, then go to that object and select its attribute named `x`”.

The figure shows the state of this object:



## 16.2. Objects are mutable

We can change the state of an object by making an assignment to one of its attributes. For example, to grow the size of a rectangle without changing its position, we could modify the values of `width` and `height`:

```

1 box.width += 50
2 box.height += 100

```

Of course, we'd probably like to provide a method to encapsulate this inside the class. We will also provide another method to move the position of the rectangle elsewhere:

```

1 class Rectangle:
2     # ...
3
4     def grow(self, delta_width, delta_height):
5         """ Grow (or shrink) this object by the deltas """
6         self.width += delta_width
7         self.height += delta_height
8
9     def move(self, dx, dy):

```

```
10     """ Move this object by the deltas """
11     self.corner.x += dx
12     self.corner.y += dy
```

Let us try this:

```
1 >>> r = Rectangle(Point(10,5), 100, 50)
2 >>> print(r)
3 ((10, 5), 100, 50)
4 >>> r.grow(25, -10)
5 >>> print(r)
6 ((10, 5), 125, 40)
7 >>> r.move(-10, 10)
8 print(r)
9 ((0, 15), 125, 40)
```

## 16.3. Sameness

The meaning of the word “same” seems perfectly clear until we give it some thought, and then we realize there is more to it than we initially expected.

For example, if we say, “Alice and Bob have the same car”, we mean that her car and his are the same make and model, but that they are two different cars. If we say, “Alice and Bob have the same mother”, we mean that her mother and his are the same person.

When we talk about objects, there is a similar ambiguity. For example, if two `Points` are the same, does that mean they contain the same data (coordinates) or that they are actually the same object?

We’ve already seen the `is` operator in the chapter on lists, where we talked about aliases: it allows us to find out if two references refer to the same object:

```
1 >>> p1 = Point(3, 4)
2 >>> p2 = Point(3, 4)
3 >>> p1 is p2
4 False
```

Even though `p1` and `p2` contain the same coordinates, they are not the same object. If we assign `p1` to `p3`, then the two variables are aliases of the same object:

```

1 >>> p3 = p1
2 >>> p1 is p3
3 True
```

This type of equality is called **shallow equality** because it compares only the references, not the contents of the objects.

To compare the contents of the objects — **deep equality** —we can write a function called `same_coordinates`:

```

1 def same_coordinates(p1, p2):
2     return (p1.x == p2.x) and (p1.y == p2.y)
```

Now if we create two different objects that contain the same data, we can use `same_point` to find out if they represent points with the same coordinates.

```

1 >>> p1 = Point(3, 4)
2 >>> p2 = Point(3, 4)
3 >>> same_coordinates(p1, p2)
4 True
```

Of course, if the two variables refer to the same object, they have both shallow and deep equality.

### Beware of `==`

“When I use a word,” Humpty Dumpty said, in a rather scornful tone, “it means just what I choose it to mean — neither more nor less.” *Alice in Wonderland*

Python has a powerful feature that allows a designer of a class to decide what an operation like `==` or `<` should mean. (We’ve just shown how we can control how our own objects are converted to strings, so we’ve already made a start!) We’ll cover more detail later. But sometimes the implementors will attach shallow equality semantics, and sometimes deep equality, as shown in this little experiment:

```

1 p = Point(4, 2)
2 s = Point(4, 2)
3 print("== on Points returns", p == s)
4 # By default, == on Point objects does a shallow equality test
5
6 a = [2,3]
7 b = [2,3]
8 print("== on lists returns", a == b)
9 # But by default, == does a deep equality test on lists
```

This outputs:

```
1 == on Points returns False
2 == on lists returns True
```

So we conclude that even though the two lists (or tuples, etc.) are distinct objects with different memory addresses, for lists the `==` operator tests for deep equality, while in the case of points it makes a shallow test.

## 16.4. Copying

Aliasing can make a program difficult to read because changes made 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:

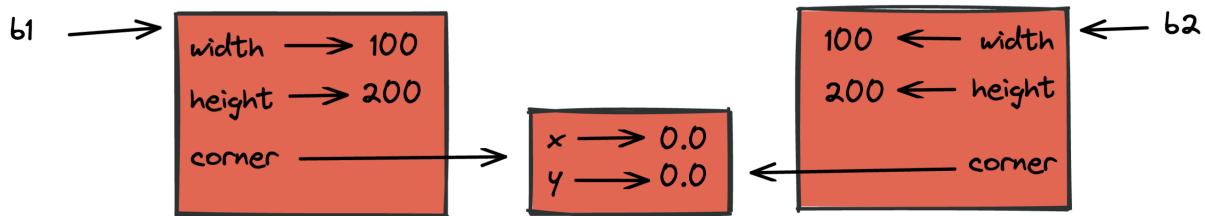
```
1 >>> import copy
2 >>> p1 = Point(3, 4)
3 >>> p2 = copy.copy(p1)
4 >>> p1 is p2
5 False
6 >>> same_coordinates(p1, p2)
7 True
```

Once we import the `copy` module, we can use the `copy` function to make a new `Point`. `p1` and `p2` are not the same point, but they contain the same data.

To copy a simple object like a `Point`, which doesn't contain any embedded objects, `copy` is sufficient. This is called **shallow copying**.

For something like a `Rectangle`, which contains a reference to a `Point`, `copy` doesn't do quite the right thing. It copies the reference to the `Point` object, so both the old `Rectangle` and the new one refer to a single `Point`.

If we create a box, `b1`, in the usual way and then make a copy, `b2`, using `copy`, the resulting state diagram looks like this:



This is almost certainly not what we want. In this case, invoking `grow` on one of the `Rectangle` objects would not affect the other, but invoking `move` on either would affect both! This behavior is confusing and error-prone. The shallow copy has created an alias to the `Point` that represents the corner.

Fortunately, the `copy` module contains a function named `deepcopy` that copies not only the object but also any embedded objects. It won't be surprising to learn that this operation is called a **deep copy**.

```
1 >>> b2 = copy.deepcopy(b1)
```

Now `b1` and `b2` are completely separate objects.

## 16.5. Glossary

### **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.

### **deep equality**

Equality of values, or two references that point to objects that have the same value.

### **shallow copy**

To copy the contents of an object, including any references to embedded objects; implemented by the `copy` function in the `copy` module.

### **shallow equality**

Equality of references, or two references that point to the same object.

## 16.6. Exercises

1. Add a method `area` to the `Rectangle` class that returns the area of any instance:

```
1 r = Rectangle(Point(0, 0), 10, 5)
2 test(r.area() == 50)
```

2. Write a `perimeter` method in the `Rectangle` class so that we can find the perimeter of any rectangle instance:

```
1 r = Rectangle(Point(0, 0), 10, 5)
2 test(r.perimeter() == 30)
```

3. Write a `flip` method in the `Rectangle` class that swaps the width and the height of any rectangle instance:

```
1 r = Rectangle(Point(100, 50), 10, 5)
2 test(r.width == 10 and r.height == 5)
3 r.flip()
4 test(r.width == 5 and r.height == 10)
```

4. Write a new method in the `Rectangle` class to test if a `Point` falls within the rectangle. For this exercise, assume that a rectangle at  $(0,0)$  with width 10 and height 5 has *open* upper bounds on the width and height, i.e. it stretches in the x direction from  $[0 \text{ to } 10)$ , where 0 is included but 10 is excluded, and from  $[0 \text{ to } 5)$  in the y direction. So it does not contain the point  $(10,2)$ . These tests should pass:

```
1 r = Rectangle(Point(0, 0), 10, 5)
2 test(r.contains(Point(0, 0)))
3 test(r.contains(Point(3, 3)))
4 test(not r.contains(Point(3, 7)))
5 test(not r.contains(Point(3, 5)))
6 test(r.contains(Point(3, 4.99999)))
7 test(not r.contains(Point(-3, -3)))
```

5. In games, we often put a rectangular “bounding box” around our sprites. (A sprite is an object that can

move about in the game, as we will see shortly.) We can then do *collision detection* between, say, bombs and spaceships, by comparing whether their rectangles overlap anywhere.

Write a function to determine whether two rectangles collide. *Hint: this might be quite a tough exercise! Think carefully about all the cases before you code.*

# Chapter 17: PyGame

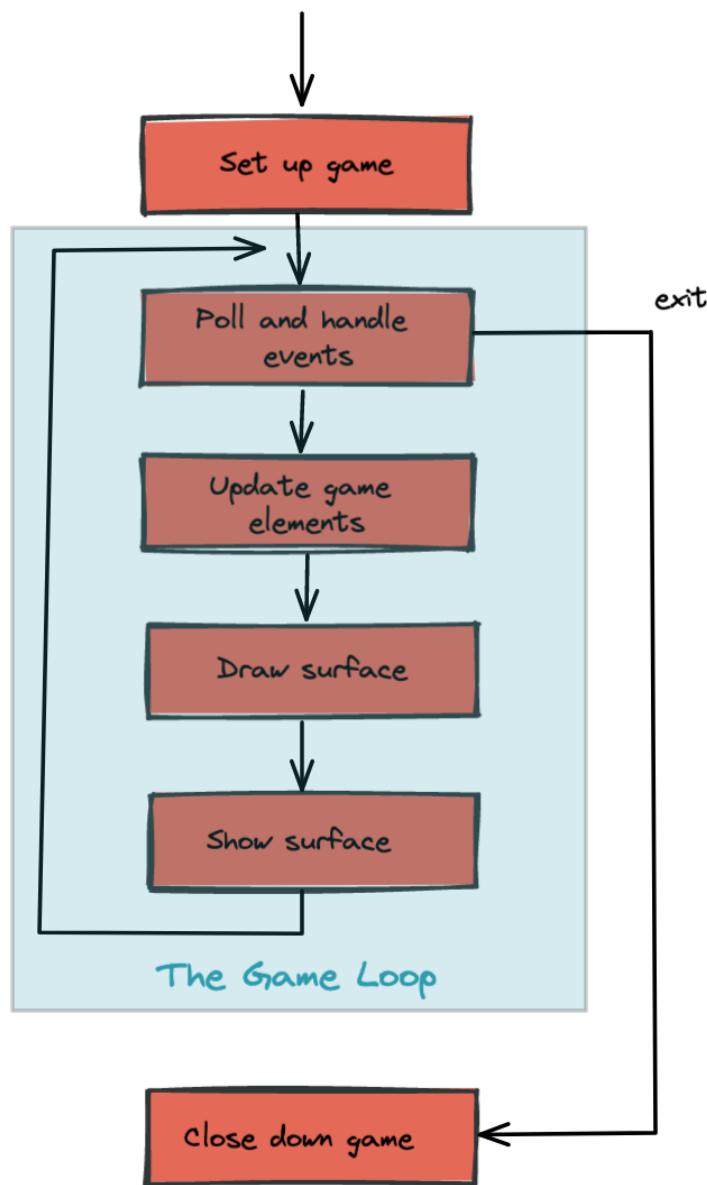
PyGame is a package that is not part of the standard Python distribution, so if you do not already have it installed (i.e. `import pygame` fails), download and install a suitable version from <http://pygame.org/download.shtml>. These notes are based on PyGame 1.9.1, the most recent version at the time of writing.

PyGame comes with a substantial set of tutorials, examples, and help, so there is ample opportunity to stretch yourself on the code. You may need to look around a bit to find these resources, though: if you've

installed PyGame on a Windows machine, for example, they'll end up in a folder like `C:\Python31\Lib\site-packages\pygame\` where you will find directories for *docs* and *examples*.

## 17.1. The game loop

The structure of the games we'll consider always follows this fixed pattern:

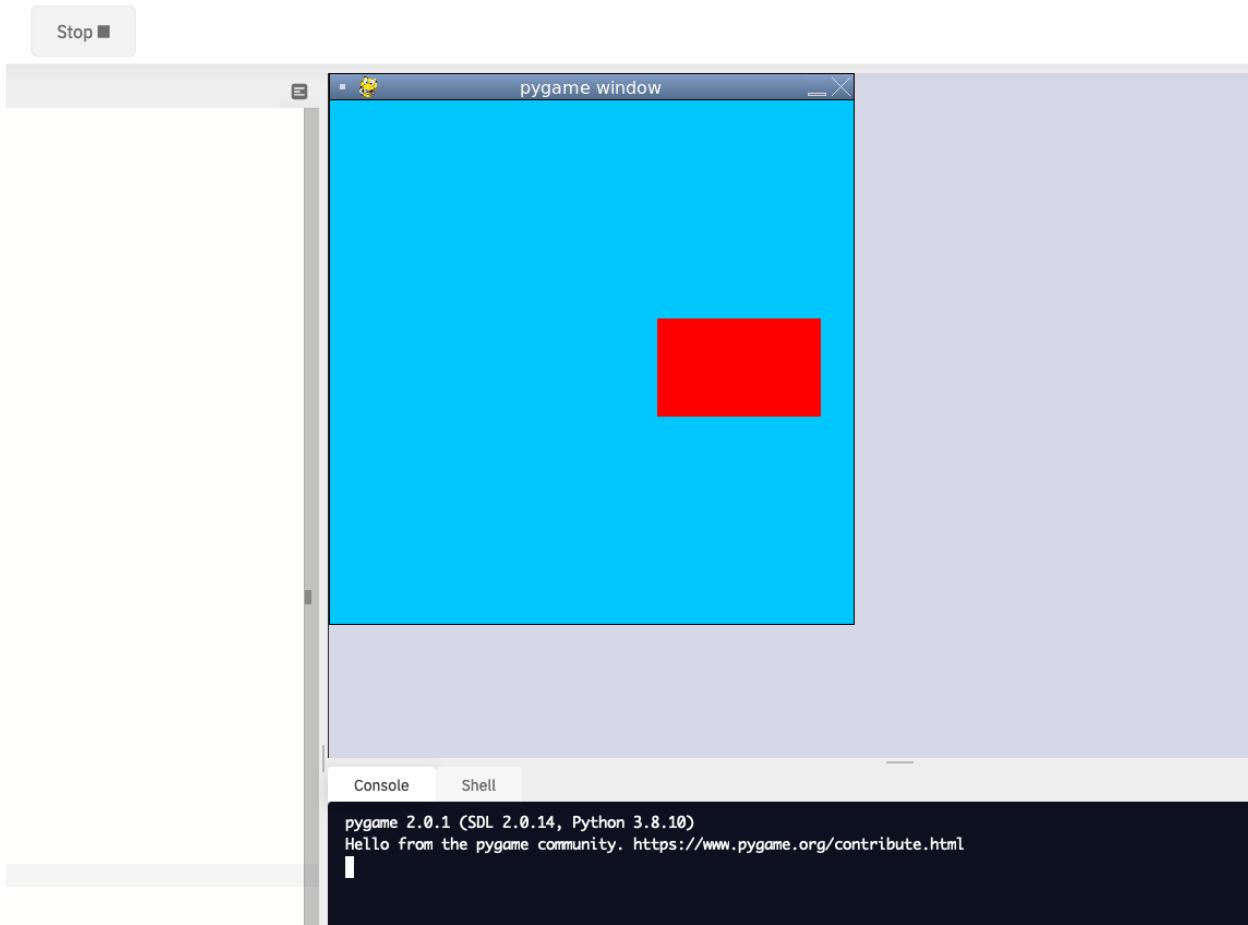


In every game, in the *setup* section we'll create a window, load and prepare some content, and then enter the game loop. The game loop continuously does four main things:

- it **polls** for events — i.e. asks the system whether events have occurred — and responds appropriately,
- it updates whatever internal data structures or objects need changing,
- it draws the current state of the game into a (non-visible) surface,
- it puts the just-drawn surface on display.

```
1 import pygame
2
3 def main():
4     """ Set up the game and run the main game loop """
5     pygame.init()      # Prepare the pygame module for use
6     surface_sz = 480   # Desired physical surface size, in pixels.
7
8     # Create surface of (width, height), and its window.
9     main_surface = pygame.display.set_mode((surface_sz, surface_sz))
10
11    # Set up some data to describe a small rectangle and its color
12    small_rect = (300, 200, 150, 90)
13    some_color = (255, 0, 0)          # A color is a mix of (Red, Green, Blue)
14
15    while True:
16        ev = pygame.event.poll()      # Look for any event
17        if ev.type == pygame.QUIT:    # Window close button clicked?
18            break                     # ... leave game loop
19
20        # Update your game objects and data structures here...
21
22        # We draw everything from scratch on each frame.
23        # So first fill everything with the background color
24        main_surface.fill((0, 200, 255))
25
26        # Overpaint a smaller rectangle on the main surface
27        main_surface.fill(some_color, small_rect)
28
29        # Now the surface is ready, tell pygame to display it!
30        pygame.display.flip()
31
32    pygame.quit()      # Once we leave the loop, close the window.
33
34 main()
```

This program pops up a window which stays there until we close it:



PyGame does all its drawing onto rectangular *surfaces*. After initializing PyGame at line 5, we create a window holding our main surface. The main loop of the game extends from line 15 to 30, with the following key bits of logic:

- First (line 16) we poll to fetch the next event that might be ready for us. This step will always be followed by some conditional statements that will determine whether any event that we're interested in has happened. Polling for the event consumes it, as far as PyGame is concerned, so we only get one chance to fetch and use each event. On line 17 we test whether the type of the event is the predefined constant called `pygame.QUIT`. This is the event that we'll see when the user clicks the close button on the PyGame window. In response to this event, we leave the loop.
- Once we've left the loop, the code at line 32 closes window, and we'll return from function `main`. Your program could go on to do other things, or reinitialize `pygame` and create another window, but it will usually just end too.
- There are different kinds of events — key presses, mouse motion,

mouse clicks, joystick movement, and so on. It is usual that we test and handle all these cases with new code squeezed in before line 19. The general idea is “handle events first, then worry about the other stuff”.

- At line 20 we’d update objects or data — for example, if we wanted to vary the color, position, or size of the rectangle we’re about to draw, we’d re-assign `some_color`, and `small_rect` here.
- A modern way to write games (now that we have fast computers and fast graphics cards) is to redraw everything from scratch on every iteration of the game loop. So the first thing we do at line 24 is fill the entire surface with a background color. The `fill` method of a surface takes two arguments — the color to use for filling, and the rectangle to be filled. But the second argument is optional, and if it is left out the entire surface is filled.

- In line 27 we fill a second rectangle, this time using `some_color`.

The placement and size of the rectangle are given by the tuple `small_rect`, a 4-element tuple (`x`, `y`, `width`, `height`).

- It is important to understand that the origin of the PyGame’s surface is at the top left corner (unlike the turtle module that puts its origin in the middle of the screen). So, if you wanted the rectangle closer to the top of the window, you need to make its `y` coordinate smaller.

- If your graphics display hardware tries to read from memory at the same time as the program is writing to that memory, they will interfere with each other, causing video noise and flicker. To get around this, PyGame keeps two buffers in the main surface — the *back buffer* that the program draws to, while the *front buffer* is being shown to the user. Each time the program has fully prepared its back buffer, it flips the back/front role of the two buffers. So the drawing on lines 24 and 27 does not change what is seen on the screen until we `flip` the buffers, on line 30.

## 17.2. Displaying images and text

To draw an image on the main surface, we load the image, say a beach ball, into its own new surface. The main surface has a `blit` method that copies pixels from the beach ball surface into its own surface. When we call `blit`, we can specify where the beach ball should be placed on the main surface. The term `blit` is widely used in computer graphics, and means *to make a fast copy of pixels from one area of memory to another*.

So in the setup section, before we enter the game loop, we'd load the image, like this:

```
1 ball = pygame.image.load("ball.png")
```

and after line 28 in the program above, we'd add this code to display our image at position (100,120):

```
1 main_surface.blit(ball, (50, 70))
```

To display text, we need do do three things. Before we enter the game loop, we instantiate a font object:

```
1 # Instantiate 16 point Courier font to draw text.  
2 my_font = pygame.font.SysFont("Courier", 16)
```

and after line 28, again, we use the font's render method to create a new surface containing the pixels of the drawn text, and then, as in the case for images, we blit our new surface onto the main surface. Notice

that render takes two extra parameters — the second tells it whether to carefully smooth edges of the text while drawing (this process is called *anti-aliasing*), and the second is the color that we want the

text text be. Here we've used (0,0,0) which is black:

```
1 the_text = my_font.render("Hello, world!", True, (0,0,0))  
2 main_surface.blit(the_text, (10, 10))
```

We'll demonstrate these two new features by counting the frames — the iterations of the game loop — and keeping some timing information. On each frame, we'll display the frame count, and the frame rate. We will

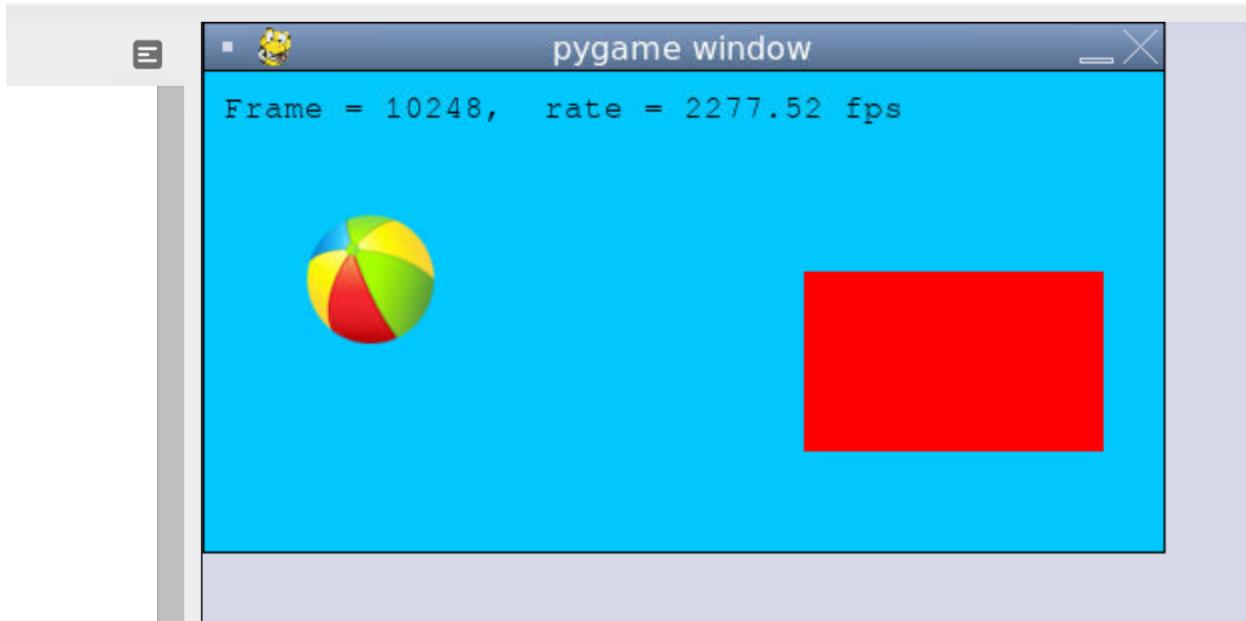
only update the frame rate after every 500 frames, when we'll look at the timing interval and can do the calculations.

First download an image of a beach ball. You can find one at [https://learnpythontherightway.com/\\_downloads/ball.png](https://learnpythontherightway.com/_downloads/ball.png). Upload it to your repl by using the “Upload file” menu. Now you can run the following code.

```
1 import pygame
2 import time
3
4 def main():
5
6     pygame.init()      # Prepare the PyGame module for use
7     main_surface = pygame.display.set_mode((480, 240))
8
9     # Load an image to draw. Substitute your own.
10    # PyGame handles gif, jpg, png, etc. image types.
11    ball = pygame.image.load("ball.png")
12
13    # Create a font for rendering text
14    my_font = pygame.font.SysFont("Courier", 16)
15
16    frame_count = 0
17    frame_rate = 0
18    t0 = time.process_time()
19
20    while True:
21
22        # Look for an event from keyboard, mouse, joystick, etc.
23        ev = pygame.event.poll()
24        if ev.type == pygame.QUIT:      # Window close button clicked?
25            break                      # Leave game loop
26
27        # Do other bits of logic for the game here
28        frame_count += 1
29        if frame_count % 500 == 0:
30            t1 = time.process_time()
31            frame_rate = 500 / (t1-t0)
32            t0 = t1
33
34        # Completely redraw the surface, starting with background
35        main_surface.fill((0, 200, 255))
36
37        # Put a red rectangle somewhere on the surface
38        main_surface.fill((255,0,0), (300, 100, 150, 90))
39
40        # Copy our image to the surface, at this (x,y) posn
41        main_surface.blit(ball, (50, 70))
42
43        # Make a new surface with an image of the text
```

```
44     the_text = my_font.render("Frame = {0}, rate = {1:.2f} fps"
45         .format(frame_count, frame_rate), True, (0,0,0))
46     # Copy the text surface to the main surface
47     main_surface.blit(the_text, (10, 10))
48
49     # Now that everything is drawn, put it on display!
50     pygame.display.flip()
51
52     pygame.quit()
53
54
55 main()
```

The frame rate is close to ridiculous — a lot faster than one's eye can process frames. (Commercial video games usually plan their action for 60 frames per second (fps).) Of course, our rate will drop once we start doing something a little more strenuous inside our game loop.



### 17.3. Drawing a board for the N queens puzzle

We previously solved our N queens puzzle. For the 8x8 board, one of the solutions was the list [6, 4, 2, 0, 5, 7, 1, 3]. Let's use that solution as testdata, and now use PyGame to draw that chessboard with its queens.

We'll create a new module for the drawing code, called `draw_queens.py`. When we have our test case(s) working, we can go back to our solver, import this new module, and add a call to our new function to draw a board each time a solution is discovered.

We begin with a background of black and red squares for the board. Perhaps we could create an image that we could load and draw, but that approach would need different background images for different size boards. Just drawing our own red and black rectangles of the appropriate size sounds like much more fun!

```
1 def draw_board(the_board):
2     """ Draw a chess board with queens, from the_board. """
3
4     pygame.init()
5     colors = [(255,0,0), (0,0,0)]      # Set up colors [red, black]
6
7     n = len(the_board)              # This is an NxN chess board.
8     surface_sz = 480                # Proposed physical surface size. \
9
10    sq_sz = surface_sz // n        # sq_sz is length of a square.
11    surface_sz = n * sq_sz         # Adjust to exactly fit n squares.
12
13    # Create the surface of (width, height), and its window.
14    surface = pygame.display.set_mode((surface_sz, surface_sz))
```

Here we precompute `sq_sz`, the integer size that each square will be, so that we can fit the squares nicely into the available window. So if we'd like the board to be 480x480, and we're drawing an 8x8 chessboard,

then each square will need to have a size of 60 units. But we notice that a 7x7 board cannot fit nicely into 480 — we're going to get some ugly border that our squares don't fill exactly. So we recompute the

surface size to exactly fit our squares before we create the window.

Now let's draw the squares, in the game loop. We'll need a nested loop: the outer loop will run over the rows of the chessboard, the inner loop over the columns:

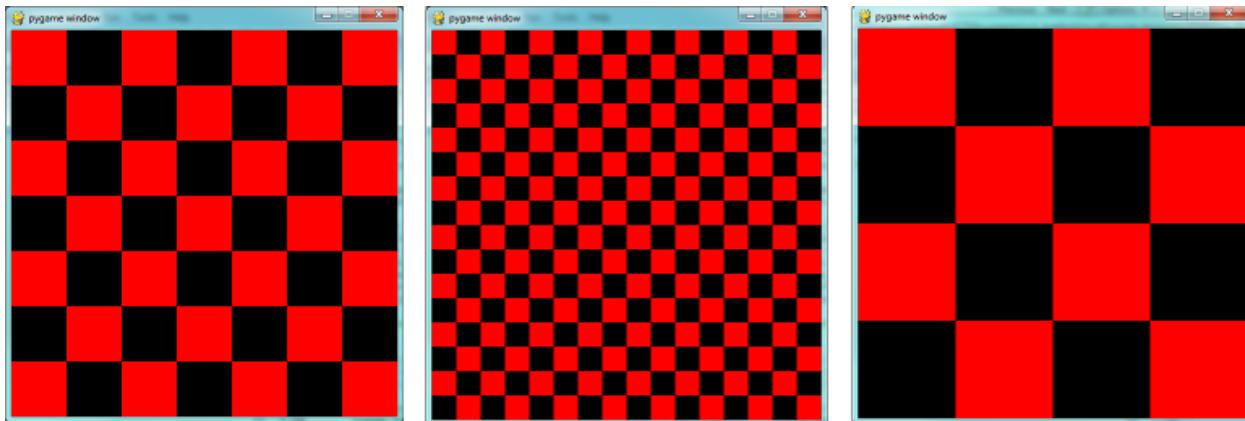
```

1 # Draw a fresh background (a blank chess board)
2 for row in range(n):           # Draw each row of the board.
3     c_indx = row % 2           # Change starting color on each row
4     for col in range(n):       # Run through cols drawing squares
5         the_square = (col*sq_sz, row*sq_sz, sq_sz, sq_sz)
6         surface.fill(colors[c_indx], the_square)
7         # now flip the color index for the next square
8         c_indx = (c_indx + 1) % 2

```

There are two important ideas in this code: firstly, we compute the rectangle to be filled from the `row` and `col` loop variables, multiplying them by the size of the square to get their position. And, of course, each square is a fixed width and height. So `the_square` represents the rectangle to be filled on the current iteration of the loop. The second idea is that we have to alternate colors on every square. In the earlier setup code we created a list containing two colors, here we manipulate `c_indx` (which will always either have the value 0 or 1) to start each row on a color that is different from the previous row's starting color, and to switch colors each time a square is filled.

This (together with the other fragments not shown to flip the surface onto the display) leads to the pleasing backgrounds like this, for different size boards:



Now, on to drawing the queens! Recall that our solution  $[6, 4, 2, 0, 5, 7, 1, 3]$  means that in column 0 of the board we want a queen at row 6, at column 1 we want a queen at row 4, and so on. So we need a loop running over each queen:

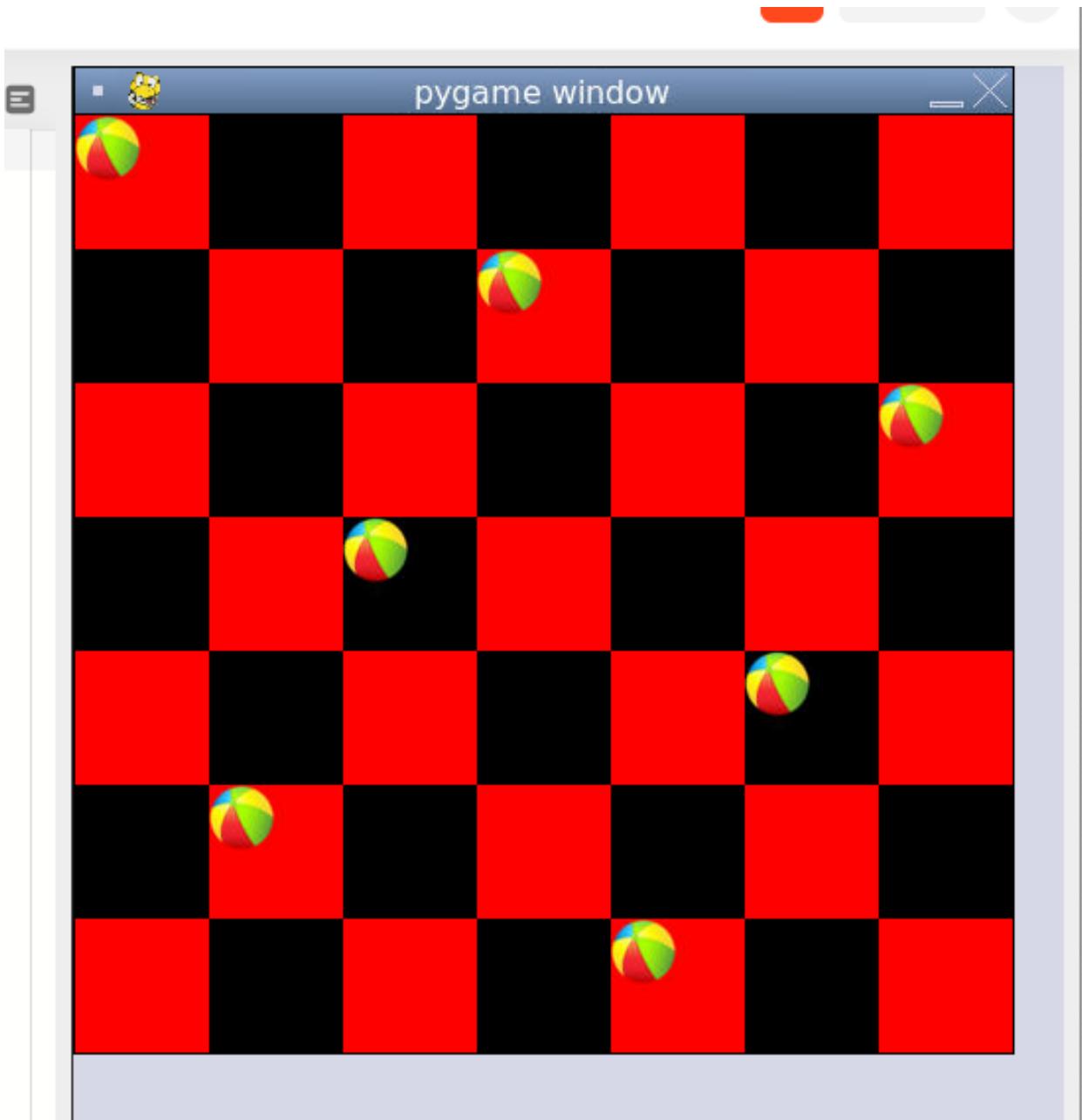
```

1 for (col, row) in enumerate(the_board):
2     # draw a queen at col, row...

```

In this chapter we already have a beach ball image, so we'll use that for our queens. In the setup code before our game loop, we load the ball image (as we did before), and in the body of the loop, we add the line:

```
1 surface.blit(ball, (col * sq_sz, row * sq_sz))
```



We're getting there, but those queens need to be centred in their squares! Our problem arises from the fact that both the ball and the rectangle have their upper left corner as their reference points. If we're going to centre this ball in the square, we need to give it an extra offset in both the x and y direction. (Since the ball is round and the square is square, the offset in the two directions will be the same, so we'll just compute a single offset value, and use it in both directions.)

The offset we need is half the (size of the square less the size of the ball). So we'll precompute this

in the game's setup section, after we've loaded the ball and determined the square size:

```
1 ball_offset = (sq_sz - ball.get_width()) // 2
```

Now we touch up the drawing code for the ball and we're done:

```
1 surface.blit(ball, (col * sq_sz + ball_offset, row * q_sz + ball_offset))
```

We might just want to think about what would happen if the ball was bigger than the square. In that case, `ball_offset` would become negative. So it would still be centered in the square - it would just spill over the boundaries, or perhaps obscure the square entirely!

Here is the complete program:

```
1 import pygame
2
3 def draw_board(the_board):
4     """ Draw a chess board with queens, as determined by the the_board. """
5
6     pygame.init()
7     colors = [(255,0,0), (0,0,0)]      # Set up colors [red, black]
8
9     n = len(the_board)                # This is an NxN chess board.
10    surface_sz = 480                 # Proposed physical surface size.
11
12    sq_sz = surface_sz // n          # sq_sz is length of a square.
13    surface_sz = n * sq_sz           # Adjust to exactly fit n squares.
14
15    # Create the surface of (width, height), and its window.
16    surface = pygame.display.set_mode((surface_sz, surface_sz))
17
18    ball = pygame.image.load("ball.png")
19
20    # Use an extra offset to centre the ball in its square.
21    # If the square is too small, offset becomes negative,
22    # but it will still be centered :-)
23    ball_offset = (sq_sz-ball.get_width()) // 2
24
25    while True:
26
27        # Look for an event from keyboard, mouse, etc.
28        ev = pygame.event.poll()
29        if ev.type == pygame.QUIT:
```

```
30     break;
31
32     # Draw a fresh background (a blank chess board)
33     for row in range(n):          # Draw each row of the board.
34         c_indx = row % 2          # Alternate starting color
35         for col in range(n):      # Run through cols drawing squares
36             the_square = (col*sq_sz, row*sq_sz, sq_sz, sq_sz)
37             surface.fill(colors[c_indx], the_square)
38             # Now flip the color index for the next square
39             c_indx = (c_indx + 1) % 2
40
41     # Now that squares are drawn, draw the queens.
42     for (col, row) in enumerate(the_board):
43         surface.blit(ball,
44                         (col*sq_sz+ball_offset, row*sq_sz+ball_offset))
45
46     pygame.display.flip()
47
48
49     pygame.quit()
50
51 if __name__ == "__main__":
52     draw_board([0, 5, 3, 1, 6, 4, 2])    # 7 x 7 to test window size
53     draw_board([6, 4, 2, 0, 5, 7, 1, 3])
54     draw_board([9, 6, 0, 3, 10, 7, 2, 4, 12, 8, 11, 5, 1])  # 13 x 13
55     draw_board([11, 4, 8, 12, 2, 7, 3, 15, 0, 14, 10, 6, 13, 1, 5, 9])
```

There is one more thing worth reviewing here. The conditional statement on line 50 tests whether the name of the currently executing program is `__main__`. This allows us to distinguish whether this module is being

run as a main program, or whether it has been imported elsewhere, and used as a module. If we run this module in Python, the test cases in lines 51-54 will be executed. However, if we import this module into another program (i.e. our N queens solver from earlier) the condition at line 50 will be false, and the statements on lines 51-54 won't run.

In Chapter 14, (14.9. Eight Queens puzzle, part 2) our main program looked like this:

```
1 def main():
2
3     bd = list(range(8))      # Generate the initial permutation
4     num_found = 0
5     tries = 0
6     while num_found < 10:
7         random.shuffle(bd)
8         tries += 1
9         if not has_clashes(bd):
10             print("Found solution {0} in {1} tries.".format(bd, tries))
11             tries = 0
12             num_found += 1
13
14 main()
```

Now we just need two changes. At the top of that program, we import the module that we've been working on here (assume we called it `draw_queens`). (You'll have to ensure that the two modules are saved in the same folder.) Then after line 10 here we add a call to draw the solution that we've just discovered:

```
1 draw_queens.draw_board(bd)
```

And that gives a very satisfying combination of program that can search for solutions to the N queens problem, and when it finds each, it pops up the board showing the solution.

## 17.4. Sprites

A sprite is an object that can move about in a game, and has internal behaviour and state of its own. For example, a spaceship would be a sprite, the player would be a sprite, and bullets and bombs would all be sprites.

Object oriented programming (OOP) is ideally suited to a situation like this: each object can have its own attributes and internal state, and a couple of methods. Let's have some fun with our N queens board. Instead

of placing the queen in her final position, we'd like to drop her in from the top of the board, and let her fall into position, perhaps bouncing along the way.

The first encapsulation we need is to turn each of our queens into an object. We'll keep a list of all the active sprites (i.e. a list of queen objects), and arrange two new things in our game loop:

- After handling events, but before drawing, call an `update` method

on every sprite. This will give each sprite a chance to modify its internal state in some way — perhaps change its image, or change its position, or rotate itself, or make itself grow a bit bigger or a bit smaller.

- Once all the sprites have updated themselves, the game loop can begin drawing - first the background, and then call a `draw` method on each sprite in turn, and delegate (hand off) the task of drawing to the object itself. This is in line with the OOP idea that we don't say "Hey, `draw`, show this queen!", but we prefer to say "Hey, `queen`, draw yourself!".

We start with a simple object, no movement or animation yet, just scaffolding, to see how to fit all the pieces together:

```
1 class QueenSprite:
2
3     def __init__(self, img, target_posn):
4         """ Create and initialize a queen for this
5             target position on the board
6             """
7         self.image = img
8         self.target_posn = target_posn
9         self.posn = target_posn
10
11    def update(self):
12        return          # Do nothing for the moment.
13
14    def draw(self, target_surface):
15        target_surface.blit(self.image, self.posn)
```

We've given the sprite three attributes: an image to be drawn, a target position, and a current position. If we're going to move the sprite about, the current position may need to be different from the target, which is where we want the queen finally to end up. In this code at this time we've done nothing in the `update` method, and our `draw` method (which can probably remain this simple in future) simply draws itself at its current position on the surface that is provided by the caller.

With its class definition in place, we now instantiate our N queens, put them into a list of sprites, and arrange for the game loop to call the `update` and `draw` methods on each frame. The new bits of code, and the revised game loop look like this:

```
1 all_sprites = []      # Keep a list of all sprites in the game
2
3 # Create a sprite object for each queen, and populate our list.
4 for (col, row) in enumerate(the_board):
5     a_queen = QueenSprite(ball,
6                           (col*sq_sz+ball_offset, row*sq_sz+ball_offset))
7     all_sprites.append(a_queen)
8
9 while True:
10    # Look for an event from keyboard, mouse, etc.
11    ev = pygame.event.poll()
12    if ev.type == pygame.QUIT:
13        break;
14
15    # Ask every sprite to update itself.
16    for sprite in all_sprites:
17        sprite.update()
18
19    # Draw a fresh background (a blank chess board)
20    # ... same as before ...
21
22    # Ask every sprite to draw itself.
23    for sprite in all_sprites:
24        sprite.draw(surface)
25
26    pygame.display.flip()
```

This works just like it did before, but our extra work in making objects for the queens has prepared the way for some more ambitious extensions.

Let us begin with a falling queen object. At any instant, it will have a velocity i.e. a speed, in a certain direction. (We are only working with movement in the y direction, but use your imagination!) So in the

object's update method, we want to change its current position by its velocity. If our N queens board is floating in space, velocity would stay constant, but hey, here on Earth we have gravity! Gravity changes

the velocity on each time interval, so we'll want a ball that speeds up as it falls further. Gravity will be constant for all queens, so we won't keep it in the instances — we'll just make it a variable in our module. We'll make one other change too: we will start every queen at the top of the board, so that it can fall towards its target position. With these changes, we now get the following:

```
1 gravity = 0.0001
2
3 class QueenSprite:
4
5     def __init__(self, img, target_posn):
6         self.image = img
7         self.target_posn = target_posn
8         (x, y) = target_posn
9         self.posn = (x, 0)      # Start ball at top of its column
10        self.y_velocity = 0     # with zero initial velocity
11
12    def update(self):
13        self.y_velocity += gravity      # Gravity changes velocity
14        (x, y) = self.posn
15        new_y_pos = y + self.y_velocity # Velocity moves the ball
16        self.posn = (x, new_y_pos)      # to this new position.
17
18    def draw(self, target_surface):    # Same as before.
19        target_surface.blit(self.image, self.posn)
```

Making these changes gives us a new chessboard in which each queen starts at the top of its column, and speeds up, until it drops off the bottom of the board and disappears forever. A good start — we have movement!

The next step is to get the ball to bounce when it reaches its own target position. It is pretty easy to bounce something — you just change the sign of its velocity, and it will move at the same speed in the opposite direction. Of course, if it is travelling up towards the top of the board it will be slowed down by gravity. (Gravity always sucks down!) And you'll find it bounces all the way up to where it began

from, reaches zero velocity, and starts falling all over again. So we'll have bouncing balls that never settle.

A realistic way to settle the object is to lose some energy (probably to friction) each time it bounces — so instead of simply reversing the sign of the velocity, we multiply it by some fractional factor — say

-0.65. This means the ball only retains 65% of its energy on each bounce, so it will, as in real life, stop bouncing after a short while, and settle on its “ground”.

The only changes are in the update method, which now looks like this:

```
1 def update(self):
2     self.y_velocity += gravity
3     (x, y) = self.posn
4     new_y_pos = y + self.y_velocity
5     (target_x, target_y) = self.target_posn      # Unpack the position
6     dist_to_go = target_y - new_y_pos            # How far to our floor?
7
8     if dist_to_go < 0:                          # Are we under floor?
9         self.y_velocity = -0.65 * self.y_velocity # Bounce
10        new_y_pos = target_y + dist_to_go        # Move back above floor
11
12    self.posn = (x, new_y_pos)                  # Set our new position.
```

Heh, heh, heh! We're not going to show animated screenshots, so copy the code into your Python environment and see for yourself.

## 17.5. Events

The only kind of event we've handled so far has been the QUIT event. But we can also detect keydown and keyup events, mouse motion, and mousebutton down or up events. Consult the PyGame documentation and follow the link to Event.

When your program polls for and receives an event object from PyGame, its event type will determine what secondary information is available. Each event object carries a *dictionary* (which you may only cover in due course in these notes). The dictionary holds certain *keys* and *values* that make sense for the type of event.

For example, if the type of event is MOUSEMOTION, we'll be able to find the mouse position and information about the state of the mouse buttons in the dictionary attached to the event. Similarly, if the event is

KEYDOWN, we can learn from the dictionary which key went down, and whether any modifier keys (shift, control, alt, etc.) are also down. You also get events when the game window becomes active (i.e. gets focus) or loses focus.

The event object with type NOEVENT is returned if there are no events waiting. Events can be printed, allowing you to experiment and play around. So dropping these lines of code into the game loop directly after polling for any event is quite informative:

```
1 if ev.type != pygame.NOEVENT:    # Only print if it is interesting!
2     print(ev)
```

With this in place, hit the space bar and the escape key, and watch the events you get. Click your three mouse buttons. Move your mouse over the window. (This causes a vast cascade of events, so you may also need to filter those out of the printing.) You'll get output that looks something like this:

```
1 <Event(17-VideoExpose {})>
2 <Event(1-ActiveEvent {'state': 1, 'gain': 0})>
3 <Event(2-KeyDown {'scancode': 57, 'key': 32, 'unicode': ' ', 'mod': 0})>
4 <Event(3-KeyUp {'scancode': 57, 'key': 32, 'mod': 0})>
5 <Event(2-KeyDown {'scancode': 1, 'key': 27, 'unicode': '\x1b', 'mod': 0})>
6 <Event(3-KeyUp {'scancode': 1, 'key': 27, 'mod': 0})>
7 ...
8 <Event(4-MouseMotion {'buttons': (0, 0, 0), 'pos': (323, 194), 'rel': (-3, -1)})>
9 <Event(4-MouseMotion {'buttons': (0, 0, 0), 'pos': (322, 193), 'rel': (-1, -1)})>
10 <Event(4-MouseMotion {'buttons': (0, 0, 0), 'pos': (321, 192), 'rel': (-1, -1)})>
11 <Event(4-MouseMotion {'buttons': (0, 0, 0), 'pos': (319, 192), 'rel': (-2, 0)})>
12 <Event(5-MouseButtonDown {'button': 1, 'pos': (319, 192)})>
13 <Event(6-MouseButtonUp {'button': 1, 'pos': (319, 192)})>
14 <Event(4-MouseMotion {'buttons': (0, 0, 0), 'pos': (319, 191), 'rel': (0, -1)})>
15 <Event(5-MouseButtonDown {'button': 2, 'pos': (319, 191)})>
16 <Event(5-MouseButtonUp {'button': 5, 'pos': (319, 191)})>
17 <Event(6-MouseButtonUp {'button': 5, 'pos': (319, 191)})>
18 <Event(6-MouseButtonUp {'button': 2, 'pos': (319, 191)})>
19 <Event(5-MouseButtonDown {'button': 3, 'pos': (319, 191)})>
20 <Event(6-MouseButtonUp {'button': 3, 'pos': (319, 191)})>
21 ...
22 <Event(1-ActiveEvent {'state': 1, 'gain': 0})>
23 <Event(12-Quit {})>
```

So let us now make these changes to the code near the top of our game loop:

```
1 while True:  
2  
3     # Look for an event from keyboard, mouse, etc.  
4     ev = pygame.event.poll()  
5     if ev.type == pygame.QUIT:  
6         break;  
7     if ev.type == pygame.KEYDOWN:  
8         key = ev.dict["key"]  
9         if key == 27:                      # On Escape key ...  
10            break                         # leave the game loop.  
11         if key == ord("r"):  
12             colors[0] = (255, 0, 0)        # Change to red + black.  
13         elif key == ord("g"):  
14             colors[0] = (0, 255, 0)        # Change to green + black.  
15         elif key == ord("b"):  
16             colors[0] = (0, 0, 255)        # Change to blue + black.  
17  
18     if ev.type == pygame.MOUSEBUTTONDOWN: # Mouse gone down?  
19         posn_of_click = ev.dict["pos"]    # Get the coordinates.  
20         print(posn_of_click)           # Just print them.
```

Lines 7-16 show typical processing for a KEYDOWN event — if a key has gone down, we test which key it is, and take some action. With this in place, we have another way to quit our queens program —by hitting the escape key. Also, we can use keys to change the color of the board that is drawn.

Finally, at line 20, we respond (pretty lamely) to the mouse button going down.

As a final exercise in this section, we'll write a better response handler to mouse clicks. What we will do is figure out if the user has clicked the mouse on one of our sprites. If there is a sprite under the mouse when the click occurs, we'll send the click to the sprite and let it respond in some sensible way.

We'll begin with some code that finds out which sprite is under the clicked position, perhaps none! We add a method to the class, `contains_point`, which returns True if the point is within the rectangle of the sprite:

```

1 def contains_point(self, pt):
2     """ Return True if my sprite rectangle contains point pt """
3     (my_x, my_y) = self.posn
4     my_width = self.image.get_width()
5     my_height = self.image.get_height()
6     (x, y) = pt
7     return ( x >= my_x and x < my_x + my_width and
8             y >= my_y and y < my_y + my_height)

```

Now in the game loop, once we've seen the mouse event, we determine which queen, if any, should be told to respond to the event:

```

1 if ev.type == pygame.MOUSEBUTTONDOWN:
2     posn_of_click = ev.dict["pos"]
3     for sprite in all_sprites:
4         if sprite.contains_point(posn_of_click):
5             sprite.handle_click()
6             break

```

And the final thing is to write a new method called `handle_click` in the `QueenSprite` class. When a sprite is clicked, we'll just add some velocity in the up direction, i.e. kick it back into the air.

```

1 def handle_click(self):
2     self.y_velocity += -0.3    # Kick it up

```

With these changes we have a playable game! See if you can keep all the balls on the move, not allowing any one to settle!

## 17.6. A wave of animation

Many games have sprites that are animated: they crouch, jump and shoot. How do they do that?

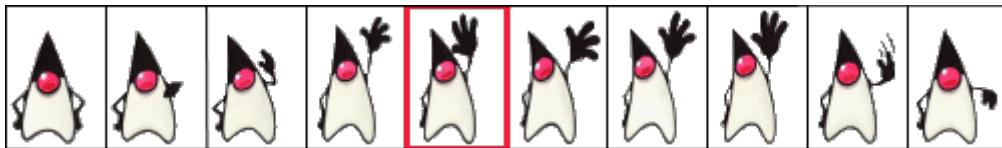
Consider this sequence of 10 images: if we display them in quick succession, Duke will wave at us. (Duke is a friendly visitor from the kingdom of Javaland.)



A compound image containing smaller *patches* which are intended for animation is called a **sprite sheet**. Download this sprite sheet by right-clicking in your browser and saving it in your working

directory  
with the name `duke_spritesheet.png`.

The sprite sheet has been quite carefully prepared: each of the 10 patches are spaced exactly 50 pixels apart. So, assuming we want to draw patch number 4 (numbering from 0), we want to draw only the rectangle  
that starts at x position 200, and is 50 pixels wide, within the sprite sheet. Here we've shown the patches and highlighted the patch we want to draw.



The `blit` method we've been using — for copying pixels from one surface to another — can copy a sub-rectangle of the source surface. So the grand idea here is that each time we draw Duke, we won't blit the  
whole sprite sheet. Instead we'll provide an extra rectangle argument that determines which portion of the sprite sheet will be blitted.

We're going to add new code in this section to our existing N queens drawing game. What we want is to put some instances of Duke on the chessboard somewhere. If the user clicks on one of them, we'll get him  
to respond by waving back, for one cycle of his animation.

But before we do that, we need another change. Up until now, our game loop has been running at really fast frame rates that are unpredictable. So we've chosen some *magic numbers* for gravity and for bouncing and  
kicking the ball on the basis of trial-and-error. If we're going to start animating more sprites, we need to tame our game loop to operate at a fixed, known frame rate. This will allow us to plan our animation better.

PyGame gives us the tools to do this in just two lines of code. In the setup section of the game, we instantiate a new `Clock` object:

```
1 my_clock = pygame.time.Clock()
```

and right at the bottom of the game loop, we call a method on this object that limits the frame rate to whatever we specify. So let's plan our game and animation for 60 frames per second, by adding this line at  
the bottom of our game loop:

```
1 my_clock.tick(60) # Waste time so that frame rate becomes 60 fps
```

You'll find that you have to go back and adjust the numbers for gravity and kicking the ball now, to match this much slower frame rate. When we plan an animation so that it only works sensibly at a

fixed frame rate, we say that we've *baked* the animation. In this case we're baking our animations for 60 frames per second.

To fit into the existing framework that we already have for our queens board, we want to create a DukeSprite class that has all the same methods as the QueenSprite class. Then we can add one or more Duke instances onto our list of `all_sprites`, and our existing game loop will then call methods of the Duke instance. Let us start with skeleton scaffolding for the new class:

```

1  class DukeSprite:
2
3      def __init__(self, img, target_posn):
4          self.image = img
5          self.posn = target_posn
6
7      def update(self):
8          return
9
10     def draw(self, target_surface):
11         return
12
13     def handle_click(self):
14         return
15
16     def contains_point(self, pt):
17         # Use code from QueenSprite here
18         return

```

The only changes we'll need to the existing game are all in the setup section. We load up the new sprite sheet and instantiate a couple of instances of Duke, at the positions we want on the chessboard. So before

entering the game loop, we add this code:

```

1  # Load the sprite sheet
2  duke_sprite_sheet = pygame.image.load("duke_spritesheet.png")
3
4  # Instantiate two duke instances, put them on the chessboard
5  duke1 = DukeSprite(duke_sprite_sheet,(sq_sz*2, 0))
6  duke2 = DukeSprite(duke_sprite_sheet,(sq_sz*5, sq_sz))
7
8  # Add them to the list of sprites which our game loop manages
9  all_sprites.append(duke1)
10 all_sprites.append(duke2)

```

Now the game loop will test if each instance has been clicked, will call the click handler for that instance. It will also call update and draw for all sprites. All the remaining changes we need to make will be made

in the methods of the DukeSprite class.

Let's begin with drawing one of the patches. We'll introduce a new attribute `curr_patch_num` into the class. It holds a value between 0 and 9, and determines which patch to draw. So the job of the `draw` method is to compute the sub-rectangle of the patch to be drawn, and to blit only that portion of the spritesheet:

```
1 def draw(self, target_surface):
2     patch_rect = (self.curr_patch_num * 50, 0,
3                   50, self.image.get_height())
4     target_surface.blit(self.image, self.posn, patch_rect)
```

Now on to getting the animation to work. We need to arrange logic in `update` so that if we're busy animating, we change the `curr_patch_num` every so often, and we also decide when to bring Duke back to his rest position, and stop the animation. An important issue is that the game loop frame rate —in our case 60 fps — is not the same as the *animation rate* — the rate at which we want to change Duke's animation patches. So we'll plan Duke wave's animation cycle for a duration of 1 second. In other words, we want to play out Duke's 10 animation patches over 60 calls to `update`. (This is how the baking of the animation takes place!) So we'll keep another animation frame counter in the class, which will be zero when we're not animating, and each call to `update` will increment the counter up to 59, and then back to 0. We can then divide that animation counter by 6, to set the `curr_patch_num` variable to select the patch we want to show.

```
1 def update(self):
2     if self.anim_frame_count > 0:
3         self.anim_frame_count = (self.anim_frame_count + 1) % 60
4         self.curr_patch_num = self.anim_frame_count // 6
```

Notice that if `anim_frame_count` is zero, i.e. Duke is at rest, nothing happens here. But if we start the counter running, it will count up to 59 before settling back to zero. Notice also, that because `anim_frame_count` can only be a value between 0 and 59, the `curr_patch_num` will always stay between 0 and 9. Just what we require!

Now how do we trigger the animation, and start it running? On the mouse click.

```
1 def handle_click(self):
2     if self.anim_frame_count == 0:
3         self.anim_frame_count = 5
```

Two things of interest here. We only start the animation if Duke is at rest. Clicks on Duke while he is already waving get ignored. And when we do start the animation, we set the counter to 5 — this

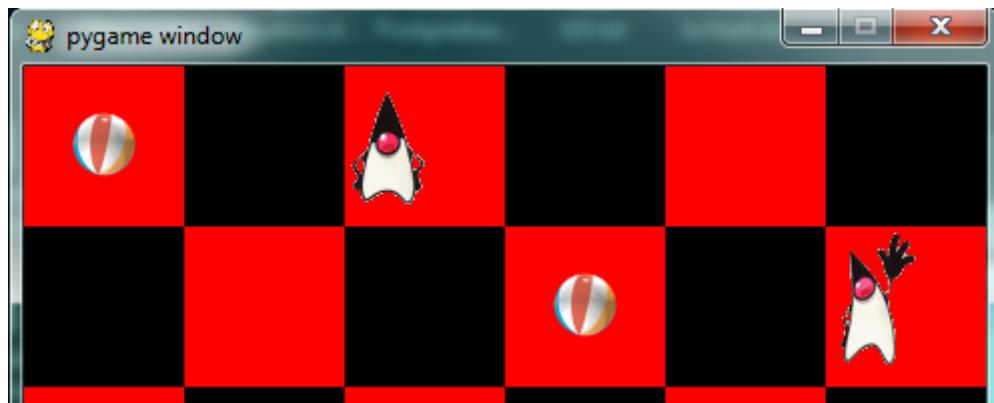
means that on

the very next call to update the counter becomes 6, and the image changes. If we had set the counter to 1, we would have needed to wait for 5 more calls to update before anything happened — a slight lag, but enough to make things feel sluggish.

The final touch-up is to initialize our two new attributes when we instantiate the class. Here is the code for the whole class now:

```
1  class DukeSprite:
2
3      def __init__(self, img, target_posn):
4          self.image = img
5          self.posn = target_posn
6          self.anim_frame_count = 0
7          self.curr_patch_num = 0
8
9      def update(self):
10         if self.anim_frame_count > 0:
11             self.anim_frame_count = (self.anim_frame_count + 1) % 60
12             self.curr_patch_num = self.anim_frame_count // 6
13
14     def draw(self, target_surface):
15         patch_rect = (self.curr_patch_num * 50, 0,
16                         50, self.image.get_height())
17         target_surface.blit(self.image, self.posn, patch_rect)
18
19     def contains_point(self, pt):
20         """ Return True if my sprite rectangle contains pt """
21         (my_x, my_y) = self.posn
22         my_width = self.image.get_width()
23         my_height = self.image.get_height()
24         (x, y) = pt
25         return (x >= my_x and x < my_x + my_width and
26                 y >= my_y and y < my_y + my_height)
27
28     def handle_click(self):
29         if self.anim_frame_count == 0:
30             self.anim_frame_count = 5
```

Now we have two extra Duke instances on our chessboard, and clicking on either causes that instance to wave.



## 17.7. Aliens - a case study

Find the example games with the PyGame package, (On a windows system, something like C:\Python3\Lib\site-packages\pygame\examples) and play the Aliens game. Then read the code, in an editor or Python environment that shows line numbers.

It does a number of much more advanced things than we do, and relies on the PyGame framework for more of its logic. Here are some of the points to notice:

- The frame rate is deliberately constrained near the bottom of the game loop at line 311. If we change that number we can make the game very slow or unplayably fast!
- There are different kinds of sprites: Explosions, Shots, Bombs, Aliens and a Player. Some of these have more than one image — by swapping the images, we get animation of the sprites, i.e. the Alien spacecraft lights change, and this is done at line 112.
- Different kinds of objects are referenced in different groups of sprites, and PyGame helps maintain these. This lets the program check for collisions between, say, the list of shots fired by the player, and the list of spaceships that are attacking. PyGame does a lot of the hard work for us.
- Unlike our game, objects in the Aliens game have a limited lifetime, and have to get killed. For example, if we shoot, a Shot object is created — if it reaches the top of the screen without exploding against anything, it has to be removed from the game. Lines 141-142 do this. Similarly, when a falling bomb gets close to the ground (line 156), it instantiates a new Explosion sprite, and the bomb kills itself.
- There are random timings that add to the fun — when to spawn the

next Alien, when an Alien drops the next bomb, etc.

- The game plays sounds too: a less-than-relaxing loop sound, plus sounds for the shots and explosions.

## 17.8. Reflections

Object oriented programming is a good organizational tool for software. In the examples in this chapter, we've started to use (and hopefully appreciate) these benefits. Here we had N queens each with its own

state, falling to its own floor level, bouncing, getting kicked, etc. We might have managed without the organizational power of objects — perhaps we could have kept lists of velocities for each queen, and lists

of target positions, and so on — our code would likely have been much more complicated, ugly, and a lot poorer!

## 17.9. Glossary

### **animation rate**

The rate at which we play back successive patches to create the illusion of movement. In the sample we considered in this chapter, we played Duke's 10 patches over the duration of one second. Not the same as the frame rate.

### **baked animation**

An animation that is designed to look good at a predetermined fixed frame rate. This reduces the amount of computation that needs to be done when the game is running. High-end commercial games usually bake their animations.

### **blit**

A verb used in computer graphics, meaning to make a fast copy of an image or pixels from a sub-rectangle of one image or surface to another surface or image.

### **frame rate**

The rate at which the game loop executes and updates the display.

### **game loop**

A loop that drives the logic of a game. It will usually poll for events, then update each of the objects in the game, then get everything drawn, and then put the newly drawn frame on display.

**pixel**

A single picture element, or dot, from which images are made.

**poll**

To ask whether something like a keypress or mouse movement has happened. Game loops usually poll to discover what events have occurred. This is different from event-driven programs like the ones seen in the chapter titled “Events”. In those cases, the button click or keypress event triggers the call of a handler function in your program, but this happens behind your back.

**sprite**

An active agent or element in a game, with its own state, position and behaviour.

**surface**

This is PyGame’s term for what the Turtle module calls a *canvas*. A surface is a rectangle of pixels used for displaying shapes and images.

## 17.10. Exercises

1. Have fun with Python, and with PyGame.
2. We deliberately left a bug in the code for animating Duke. If you click on one of the chessboard squares

to the right of Duke, he waves anyway. Why? Find a one-line fix for the bug.

3. Use your preferred search engine to search their image library for “sprite sheet playing cards”. Create a

list [0..51] to represent an encoding of the 52 cards in a deck. Shuffle the cards, slice off the top five as your hand in a poker deal. Display the hand you have been dealt.

4. So the Aliens game is in outer space, without gravity. Shots fly away forever, and bombs don’t speed up

when they fall. Add some gravity to the game. Decide if you’re going to allow your own shots to fall back on your head and kill you.

5. Those pesky Aliens seem to pass right through each other! Change the game so that they collide, and

destroy each other in a mighty explosion.

# Chapter 18: Recursion

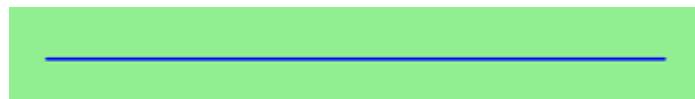
**Recursion** means “defining something in terms of itself” usually at some smaller scale, perhaps multiple times, to achieve your objective. For example, we might say “A human being is someone whose mother is a human being”, or “a directory is a structure that holds files and (smaller) directories”, or “a family tree starts with a couple who have children, each with their own family sub-trees”.

Programming languages generally support **recursion**, which means that, in order to solve a problem, functions can *call themselves* to solve smaller subproblems.

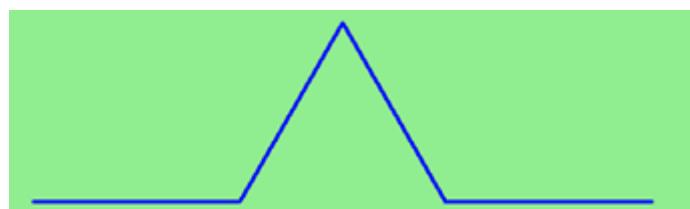
## 18.1. Drawing Fractals

For our purposes, a **fractal** is a drawing which also has *self-similar* structure, where it can be defined in terms of itself.

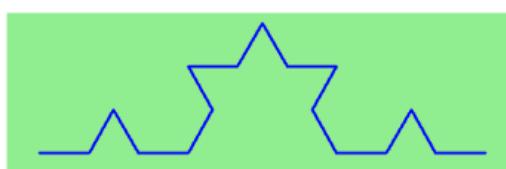
Let us start by looking at the famous Koch fractal. An order 0 Koch fractal is simply a straight line of a given size.



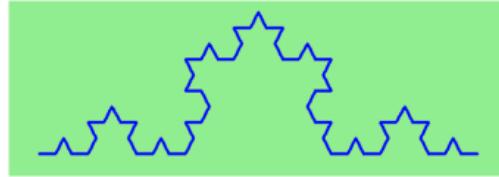
An order 1 Koch fractal is obtained like this: instead of drawing just one line, draw instead four smaller segments, in the pattern shown here:



Now what would happen if we repeated this Koch pattern again on each of the order 1 segments? We'd get this order 2 Koch fractal:



Repeating our pattern again gets us an order 3 Koch fractal:



Now let us think about it the other way around. To draw a Koch fractal of order 3, we can simply draw four order 2 Koch fractals. But each of these in turn needs four order 1 Koch fractals, and each of those in turn needs four order 0 fractals. Ultimately, the only drawing that will take place is at order 0. This is very simple to code up in Python:

```

1 def koch(t, order, size):
2     """
3         Make turtle t draw a Koch fractal of 'order' and 'size'.
4         Leave the turtle facing the same direction.
5     """
6
7     if order == 0:          # The base case is just a straight line
8         t.forward(size)
9     else:
10        koch(t, order-1, size/3)    # Go 1/3 of the way
11        t.left(60)
12        koch(t, order-1, size/3)
13        t.right(120)
14        koch(t, order-1, size/3)
15        t.left(60)
16        koch(t, order-1, size/3)
```

The key thing that is new here is that if order is not zero, `koch` calls itself recursively to get its job done.

Let's make a simple observation and tighten up this code. Remember that turning right by 120 is the same as turning left by -120. So with a bit of clever rearrangement, we can use a loop instead of lines 10-16:

```

1 def koch(t, order, size):
2     if order == 0:
3         t.forward(size)
4     else:
5         for angle in [60, -120, 60, 0]:
6             koch(t, order-1, size/3)
7             t.left(angle)

```

The final turn is 0 degrees — so it has no effect. But it has allowed us to find a pattern and reduce seven lines of code to three, which will make things easier for our next observations.

### Recursion, the high-level view

One way to think about this is to convince yourself that the function works correctly when you call it for an order 0 fractal. Then do a mental *leap of faith*, saying “*the fairy godmother* (or Python, if you can think of Python as your fairy godmother) *knows how to handle the recursive level 0 calls for me on lines 11, 13, 15, and 17, so I don’t need to think about that detail!*” All I need to focus on is how to draw an order 1 fractal *if I can assume the order 0 one is already working*.

You’re practicing *mental abstraction* — ignoring the subproblem while you solve the big problem.

If this mode of thinking works (and you should practice it!), then take it to the next level. Aha! now can I see that it will work when called for order 2 *under the assumption that it is already working for level 1*.

And, in general, if I can assume the order n-1 case works, can I just solve the level n problem?

Students of mathematics who have played with proofs of induction should see some very strong similarities here.

### Recursion, the low-level operational view

Another way of trying to understand recursion is to get rid of it! If we had separate functions to draw a level 3 fractal, a level 2 fractal, a level 1 fractal and a level 0 fractal, we could simplify the above code,

quite mechanically, to a situation where there was no longer any recursion, like this:

```

1 def koch_0(t, size):
2     t.forward(size)
3
4 def koch_1(t, size):
5     for angle in [60, -120, 60, 0]:
6         koch_0(t, size/3)
7         t.left(angle)
8
9 def koch_2(t, size):
10    for angle in [60, -120, 60, 0]:

```

```
11      koch_1(t, size/3)
12      t.left(angle)
13
14  def koch_3(t, size):
15      for angle in [60, -120, 60, 0]:
16          koch_2(t, size/3)
17          t.left(angle)
```

This trick of “unrolling” the recursion gives us an operational view of what happens. You can trace the program into `koch_3`, and from there, into `koch_2`, and then into `koch_1`, etc., all the way down the different layers of the recursion.

This might be a useful hint to build your understanding. The mental goal is, however, to be able to do the abstraction!

## 18.2. Recursive data structures

All of the Python data types we have seen can be grouped inside lists and tuples in a variety of ways. Lists and tuples can also be nested, providing many possibilities for organizing data. The organization of data for the purpose of making it easier to use is called a **data structure**.

It’s election time and we are helping to compute the votes as they come in. Votes arriving from individual wards, precincts, municipalities, counties, and states are sometimes reported as a sum total of votes and sometimes as a list of subtotals of votes. After considering how best to store the tallies, we decide to use a *nested number list*, which we define as follows:

A *nested number list* is a list whose elements are either:

1. numbers
2. nested number lists

Notice that the term, *nested number list* is used in its own definition. **Recursive definitions** like this are quite common in mathematics and computer science. They provide a concise and powerful way to describe **recursive data structures** that are partially composed of smaller and simpler instances of themselves. The definition is not circular, since at some point we will reach a list that does not have any lists as elements.

Now suppose our job is to write a function that will sum all of the values in a nested number list. Python has a built-in function which finds the sum of a sequence of numbers:

```
1  >>> sum([1, 2, 8])
2  11
```

For our *nested number list*, however, `sum` will not work:

```
1 >>> sum([1, 2, [11, 13], 8])
2 Traceback (most recent call last):
3   File "<interactive input>", line 1, in <module>
4     TypeError: unsupported operand type(s) for +: 'int' and 'list'
5 >>>
```

The problem is that the third element of this list, [11, 13], is itself a list, so it cannot just be added to 1, 2, and 8.

## 18.3. Processing recursive number lists

To sum all the numbers in our recursive nested number list we need to traverse the list, visiting each of the elements within its nested structure, adding any numeric elements to our sum, and *recursively repeating the summing process* with any elements which are themselves sub-lists.

Thanks to recursion, the Python code needed to sum the values of a nested number list is surprisingly short:

```
1 def r_sum(nested_num_list):
2     tot = 0
3     for element in nested_num_list:
4         if type(element) == type([]):
5             tot += r_sum(element)
6         else:
7             tot += element
8     return tot
```

The body of `r_sum` consists mainly of a `for` loop that traverses `nested_num_list`. If `element` is a numerical value (the `else` branch), it is simply added to `tot`. If `element` is a list, then `r_sum` is called again, with the element as an argument. The statement inside the function definition in which the function calls itself is known as the **recursive call**.

The example above has a **base case** (on line 7) which does not lead to a recursive call: the case where the element is not a (sub-) list. Without a base case, you'll have **infinite recursion**, and your program will not work.

Recursion is truly one of the most beautiful and elegant tools in computer science.

A slightly more complicated problem is finding the largest value in our nested number list:

```

1 def r_max(nxs):
2     """
3         Find the maximum in a recursive structure of lists
4         within other lists.
5         Precondition: No lists or sublists are empty.
6     """
7     largest = None
8     first_time = True
9     for e in nxs:
10         if type(e) == type([]):
11             val = r_max(e)
12         else:
13             val = e
14
15         if first_time or val > largest:
16             largest = val
17             first_time = False
18
19     return largest
20
21 test(r_max([2, 9, [1, 13], 8, 6]) == 13)
22 test(r_max([2, [[100, 7], 90], [1, 13], 8, 6]) == 100)
23 test(r_max([[13, 7], 90], 2, [1, 100], 8, 6]) == 100)
24 test(r_max(["joe", ["sam", "ben"]]) == "sam")

```

Tests are included to provide examples of `r_max` at work.

The added twist to this problem is finding a value for initializing `largest`. We can't just use `nxs[0]`, since that could be either a element or a list. To solve this problem (at every recursive call) we initialize a Boolean flag (at line 8). When we've found the value of interest, (at line 15) we check to see whether this is the initializing (first) value for `largest`, or a value that could potentially change `largest`.

Again here we have a base case at line 13. If we don't supply a base case, Python stops after reaching a maximum recursion depth and returns a runtime error. See how this happens, by running this little script which we will call ‘`infinite_recursion.py`’:

```

1 def recursion_depth(number):
2     print("{0}, ".format(number), end="")
3     recursion_depth(number + 1)
4
5 recursion_depth(0)

```

After watching the messages flash by, you will be presented with the end of a long traceback that ends with a message like the following:

```
1 RuntimeError: maximum recursion depth exceeded ...
```

We would certainly never want something like this to happen to a user of one of our programs, so in the next chapter we'll see how errors, any kinds of errors, are handled in Python.

## 18.4. Case study: Fibonacci numbers

The famous **Fibonacci sequence** 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 134, ... was devised by Fibonacci (1170-1250), who used this to model the breeding of (pairs) of rabbits. If, in generation 7 you had 21 pairs in total, of which 13 were adults, then next generation the adults will all have bred new children, and the previous children will have grown up to become adults. So in generation 8 you'll have  $13+21=34$ , of which 21 are adults.

This *model* to explain rabbit breeding made the simplifying assumption that rabbits never died. Scientists often make (unrealistic) simplifying assumptions and restrictions to make some headway with the problem.

If we number the terms of the sequence from 0, we can describe each term recursively as the sum of the previous two terms:

```
1 fib(0) = 0
2 fib(1) = 1
3 fib(n) = fib(n-1) + fib(n-2)  for n >= 2
```

This translates very directly into some Python:

```
1 def fib(n):
2     if n <= 1:
3         return n
4     t = fib(n-1) + fib(n-2)
5     return t
```

This is a particularly inefficient algorithm, and we'll show one way of fixing it when we learn about dictionaries:

```

1 import time
2 t0 = time.clock()
3 n = 35
4 result = fib(n)
5 t1 = time.clock()
6
7 print("fib({0}) = {1}, ({2:.2f} secs)".format(n, result, t1-t0))

```

We get the correct result, but an exploding amount of work! :

```
1 fib(35) = 9227465, (10.54 secs)
```

## 18.5. Example with recursive directories and files

The following program lists the contents of a directory and all its subdirectories.

```

1 import os
2
3 def get_dirlist(path):
4     """
5         Return a sorted list of all entries in path.
6         This returns just the names, not the full path to the names.
7     """
8     dirlist = os.listdir(path)
9     dirlist.sort()
10    return dirlist
11
12 def print_files(path, prefix = ""):
13     """
14         Print recursive listing of contents of path """
15     if prefix == "": # Detect outermost call, print a heading
16         print("Folder listing for", path)
17         prefix = "| "
18
19     dirlist = get_dirlist(path)
20     for f in dirlist:
21         print(prefix+f) # Print the line
22         fullname = os.path.join(path, f) # Turn name into full pathname
23         if os.path.isdir(fullname): # If a directory, recurse.
24             print_files(fullname, prefix + "| ")

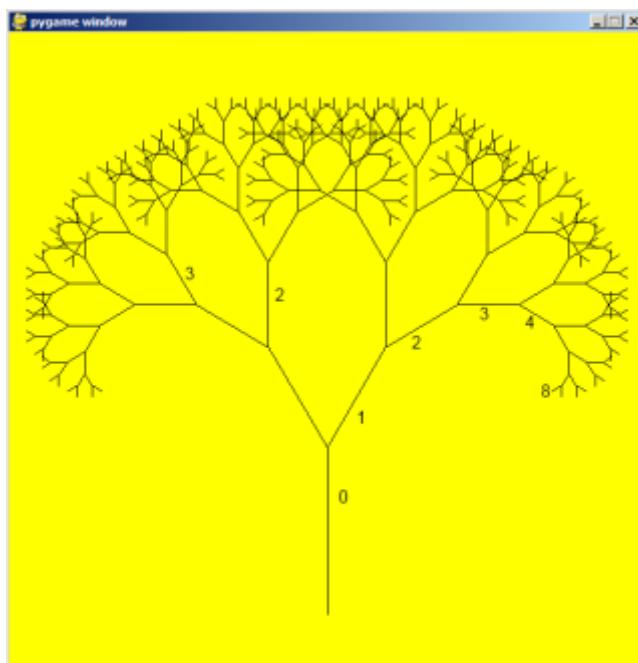
```

Calling the function `print_files` with some folder name will produce output similar to this:

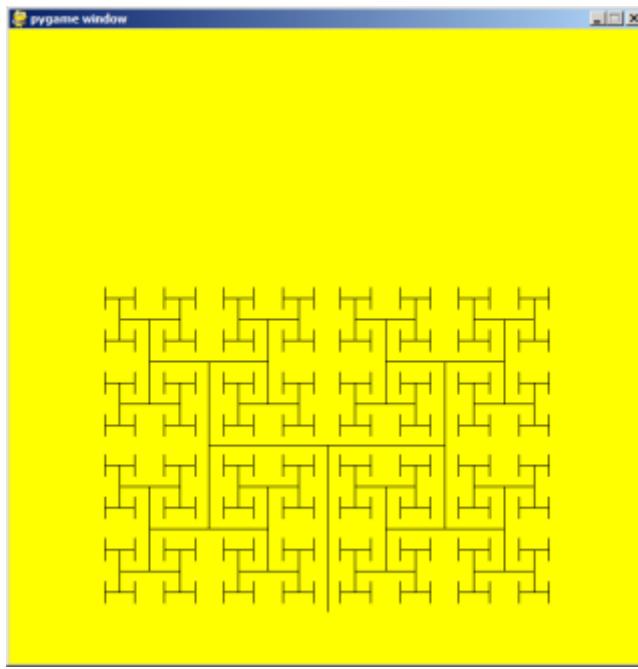
```
1 Folder listing for c:\python31\Lib\site-packages\pygame\examples
2 | __init__.py
3 | aacircle.py
4 | aliens.py
5 | arraydemo.py
6 | blend_fill.py
7 | blit.blends.py
8 | camera.py
9 | chimp.py
10 | cursors.py
11 | data
12 | | alien1.png
13 | | alien2.png
14 | | alien3.png
15 ...
```

## 18.6. An animated fractal, using PyGame

Here we have a tree fractal pattern of order 8. We've labelled some of the edges, showing the depth of the recursion at which each edge was drawn.



In the tree above, the angle of deviation from the trunk is 30 degrees. Varying that angle gives other interesting shapes, for example, with the angle at 90 degrees we get this:



An interesting animation occurs if we generate and draw trees very rapidly, each time varying the angle a little. Although the Turtle module can draw trees like this quite elegantly, we could struggle for good frame rates. So we'll use PyGame instead, with a few embellishments and observations. (Once again, we suggest you cut and paste this code into your Python environment.)

```
1 import pygame, math
2 pygame.init()           # prepare the pygame module for use
3
4 # Create a new surface and window.
5 surface_size = 1024
6 main_surface = pygame.display.set_mode((surface_size,surface_size))
7 my_clock = pygame.time.Clock()
8
9
10 def draw_tree(order, theta, sz, posn, heading, color=(0,0,0), depth=0):
11
12     trunk_ratio = 0.29      # How big is the trunk relative to whole tree?
13     trunk = sz * trunk_ratio # length of trunk
14     delta_x = trunk * math.cos(heading)
15     delta_y = trunk * math.sin(heading)
16     (u, v) = posn
17     newpos = (u + delta_x, v + delta_y)
18     pygame.draw.line(main_surface, color, posn, newpos)
19
20     if order > 0:    # Draw another layer of subtrees
```

```
21
22     # These next six lines are a simple hack to make the two major halves
23     # of the recursion different colors. Fiddle here to change colors
24     # at other depths, or when depth is even, or odd, etc.
25     if depth == 0:
26         color1 = (255, 0, 0)
27         color2 = (0, 0, 255)
28     else:
29         color1 = color
30         color2 = color
31
32     # make the recursive calls to draw the two subtrees
33     newsz = sz*(1 - trunk_ratio)
34     draw_tree(order-1, theta, newsz, newpos, heading-theta, color1, depth+1)
35     draw_tree(order-1, theta, newsz, newpos, heading+theta, color2, depth+1)
36
37
38 def gameloop():
39
40     theta = 0
41     while True:
42
43         # Handle events from keyboard, mouse, etc.
44         ev = pygame.event.poll()
45         if ev.type == pygame.QUIT:
46             break;
47
48         # Updates - change the angle
49         theta += 0.01
50
51         # Draw everything
52         main_surface.fill((255, 255, 0))
53         draw_tree(9, theta, surface_size*0.9, (surface_size//2, surface_size-50), -m\
54         ath.pi/2)
55
56         pygame.display.flip()
57         my_clock.tick(120)
58
59
60 gameloop()
61 pygame.quit()
```

- The `math` library works with angles in radians rather than degrees.

- Lines 14 and 15 uses some high school trigonometry. From the length of the desired line (`trunk`), and its desired angle, `cos` and `sin` help us to calculate the `x` and `y` distances we need to move.
- Lines 22-30 are unnecessary, except if we want a colorful tree.
- In the main game loop at line 49 we change the angle on every frame, and redraw the new tree.
- Line 18 shows that PyGame can also draw lines, and plenty more.

Check out the documentation. For example, drawing a small circle at each branch point of the tree can be accomplished by adding this line directly below line 18:

```
1     pygame.draw.circle(main_surface, color, (int(posn[0]), int(posn[1])), 3)
```

Another interesting effect — instructive too, if you wish to reinforce the idea of different instances of the function being called at different depths of recursion —is to create a list of colors, and let each recursive depth use a different color for drawing. (Use the depth of the recursion to index the list of colors.)

## 18.7. Glossary

### **base case**

A branch of the conditional statement in a recursive function that does not give rise to further recursive calls.

### **infinite recursion**

A function that calls itself recursively without ever reaching any base case. Eventually, infinite recursion causes a runtime error.

### **recursion**

The process of calling a function that is already executing.

### **recursive call**

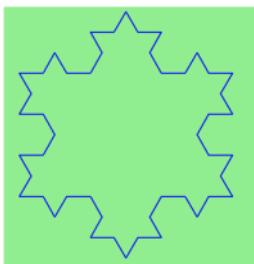
The statement that calls an already executing function. Recursion can also be indirect — function `f` can call `g` which calls `h`, and `h` could make a call back to `f`.

### recursive definition

A definition which defines something in terms of itself. To be useful it must include *base cases* which are not recursive. In this way it differs from a *circular definition*. Recursive definitions often provide an elegant way to express complex data structures, like a directory that can contain other directories, or a menu that can contain other menus.

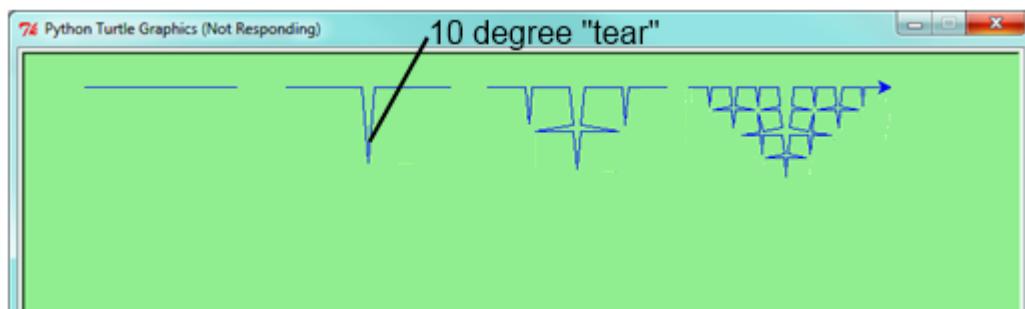
## 18.8. Exercises

1. Modify the Koch fractal program so that it draws a Koch snowflake, like this:



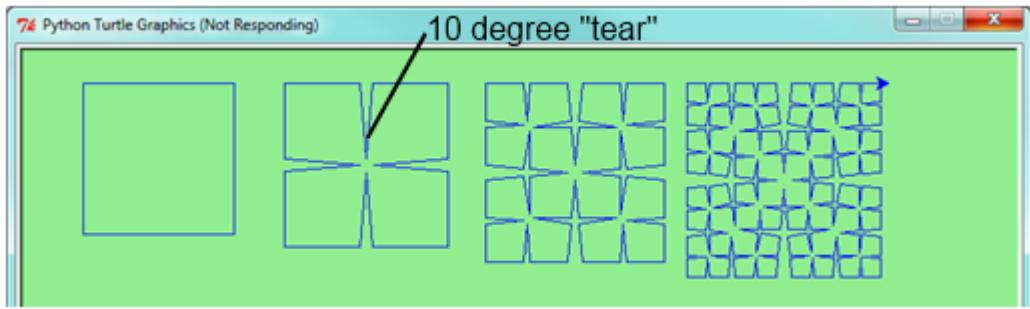
2. 1. Draw a Cesaro torn line fractal, of the order given by the user.

We show four different lines of orders 0,1,2,3. In this example, the angle of the tear is 10 degrees.



2. Four lines

make a square. Use the code in part a) to draw cesaro squares. Varying the angle gives interesting effects — experiment a bit, or perhaps let the user input the angle of the tear.

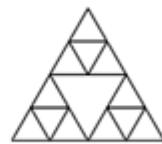
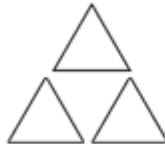


3. (For the

mathematically inclined). In the squares shown here, the higher-order drawings become a little larger. (Look at the bottom lines of each square - they're not aligned.) This is because we just halved the drawn part of the line for each recursive subproblem. So we've "grown" the overall square by the width of the tear(s). Can you solve the geometry problem so that the total size of the subproblem case (including the tear) remains exactly the same size as the original?

3. A Sierpinski triangle of order 0 is an equilateral triangle. An order 1 triangle can be drawn by drawing

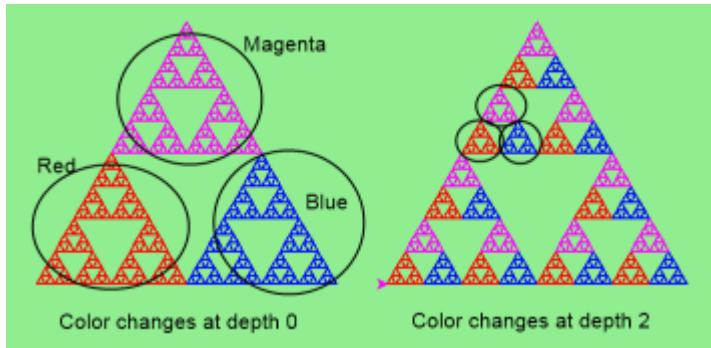
3 smaller triangles (shown slightly disconnected here, just to help our understanding). Higher order 2 and 3 triangles are also shown. Draw Sierpinski triangles of any order input by the user.



4.

Adapt the above program to change the color of its three sub-triangles at some depth of recursion. The

illustration below shows two cases: on the left, the color is changed at depth 0 (the outmost level of recursion), on the right, at depth 2. If the user supplies a negative depth, the color never changes. (Hint: add a new optional parameter `colorChangeDepth` (which defaults to -1), and make this one smaller on each recursive subcall. Then, in the section of code before you recurse, test whether the parameter is zero, and change color.)



5. Write a function, `recursive_min`, that returns the smallest value in a nested number list. Assume there are no empty lists or sublists:

```

1 test(recursive_min([2, 9, [1, 13], 8, 6]) == 1)
2 test(recursive_min([2, [[100, 1], 90], [10, 13], 8, 6]) == 1)
3 test(recursive_min([2, [[13, -7], 90], [1, 100], 8, 6]) == -7)
4 test(recursive_min([[[-13, 7], 90], 2, [1, 100], 8, 6]) == -13)

```

6. Write a function `count` that returns the number of occurrences of target in a nested list:

```

1 test(count(2, []), 0)
2 test(count(2, [2, 9, [2, 1, 13, 2], 8, [2, 6]]), 4)
3 test(count(7, [[9, [7, 1, 13, 2], 8], [7, 6]]), 2)
4 test(count(15, [[9, [7, 1, 13, 2], 8], [2, 6]]), 0)
5 test(count(5, [[5, [5, [1, 5], 5], 5], [5, 6]]), 6)
6 test(count("a",
7     [["this", "a", ["thing", "a"], "a"], "is"], ["a", "easy"])), 4)

```

7. Write a function `flatten` that returns a simple list containing all the values in a nested list:

```

1 test(flatten([2,9,[2,1,13,2],8,[2,6]]), [2,9,2,1,13,2,8,2,6])
2 test(flatten([[9,[7,1,13,2],8],[7,6]]), [9,7,1,13,2,8,7,6])
3 test(flatten([[9,[7,1,13,2],8],[2,6]]), [9,7,1,13,2,8,2,6])
4 test(flatten([[["this", "a", ["thing"], "a"], "is"], ["a", "easy"]]]),
5     [ "this", "a", "thing", "a", "is", "a", "easy"])
6 test(flatten([]), [])

```

8. Rewrite the fibonacci algorithm without using recursion. Can you find bigger terms of the sequence? Can you find `fib(200)`?

9. Use help to find out what `sys.getrecursionlimit()` and `sys.setrecursionlimit(n)` do. Create several

experiments similar to what was done in `infinite_recursion.py` to test your understanding of how these module functions work.

10. Write a program that walks a directory structure (as in the last section of this chapter), but instead of

printing filenames, it returns a list of all the full paths of files in the directory or the subdirectories. (Don't include directories in this list — just files.) For example, the output list might have elements like this:

```
[“C:Python31Lib\site-packages\pygame\docs\ref\mask.html”,  
 “C:Python31Lib\site-packages\pygame\docs\ref\midi.html”,  
 ...  
 “C:Python31Lib\site-packages\pygame\examples\aliens.py”,  
 ...  
 “C:Python31Lib\site-packages\pygame\examples\data\boom.wav”,  
 ... ]
```

11. Write a program named `litter.py` that creates an empty file named `trash.txt` in each subdirectory of a

directory tree given the root of the tree as an argument (or the current directory as a default). Now write a program named `cleanup.py` that removes all these files.

**Hint #1:** Use the program from the example in the last section of this chapter as a basis for these two recursive programs. Because you're going to destroy files on your disks, you better get this right, or you risk losing files you care about. So excellent advice is that initially you should fake the deletion of the files — just print the full path names of each file that you intend to delete. Once you're happy that your logic is correct, and you can see that you're not deleting the wrong things, you can replace the print statement with the real thing.

**Hint #2:** Look in the `os` module for a function that removes files.

# Chapter 19: Exceptions

## 19.1. Catching exceptions

Whenever a runtime error occurs, it creates an **exception** object. The program stops running at this point and Python prints out the traceback, which ends with a message describing the exception that occurred.

For example, dividing by zero creates an exception:

```
1 >>> print(55/0)
2 Traceback (most recent call last):
3   File "<interactive input>", line 1, in <module>
4 ZeroDivisionError: integer division or modulo by zero
```

So does accessing a non-existent list item:

```
1 >>> a = []
2 >>> print(a[5])
3 Traceback (most recent call last):
4   File "<interactive input>", line 1, in <module>
5 IndexError: list index out of range
```

Or trying to make an item assignment on a tuple:

```
1 >>> tup = ("a", "b", "d", "d")
2 >>> tup[2] = "c"
3 Traceback (most recent call last):
4   File "<interactive input>", line 1, in <module>
5 TypeError: 'tuple' object does not support item assignment
```

In each case, the error message on the last line has two parts: the type of error before the colon, and specifics about the error after the colon.

Sometimes we want to execute an operation that might cause an exception, but we don't want the program to stop. We can **handle the exception** using the `try` statement to “wrap” a region of code.

For example, we might prompt the user for the name of a file and then try to open it. If the file doesn't exist, we don't want the program to crash; we want to handle the exception:

```
1 filename = input("Enter a file name: ")
2 try:
3     f = open(filename, "r")
4 except:
5     print("There is no file named", filename)
```

The `try` statement has three separate clauses, or parts, introduced by the keywords `try` ... `except` ... `finally`. Either the `except` or the `finally` clauses can be omitted, so the above code considers the most common version of the `try` statement first.

The `try` statement executes and monitors the statements in the first block. If no exceptions occur, it skips the block under the `except` clause. If any exception occurs, it executes the statements in the `except` clause and then continues.

We could encapsulate this capability in a function: `exists` which takes a filename and returns true if the file exists, false if it doesn't:

```
1 def exists(filename):
2     try:
3         f = open(filename)
4         f.close()
5         return True
6     except:
7         return False
```

.. The `try` statement in this function was already introduced previously (the same code), so I thought it would be appropriate to add an `else` clause here.

pw: HI Victor - I looked at the `else:` and went against it! It is just a horrible language feature in my view. Not only do we complicate things by overload the keyword `else` (here, in the `if`, and in the `for` loop), but it adds no new expressive power over just doing the “didn't get an exception” inline. And the allowable combinations are hard to explain. You can omit the `except` clause if you have a `finally` clause. But you cannot have the `else` if you omit `except`, ... and so on. Too much risk for too little return, in my view.

### A template to test if a file exists, without using exceptions

The function we've just shown is not one we'd recommend. It opens and closes the file, which is semantically different from asking “does it exist?”. How? Firstly, it might update some timestamps on the file. Secondly, it might tell us that there is no such file if some other program already happens to have the file open, or if our permission settings don't allow us to open the file.

Python provides a module called `os.path` within the `os` module. It provides a number of useful functions to work with paths, files and directories, so you should check out the help.

```
1 import os  
2  
3 # This is the preferred way to check if a file exists.  
4 if os.path.isfile("c:/temp/testdata.txt"):
```

We can use multiple `except` clauses to handle different kinds of exceptions (see the [Errors and Exceptions<sup>19</sup>](#) lesson from Python creator Guido van Rossum's [Python Tutorial<sup>20</sup>](#) for a more complete discussion of exceptions). So the program could do one thing if the file does not exist, but do something else if the file was in use by another program.

## 19.2. Raising our own exceptions

Can our program deliberately cause its own exceptions? If our program detects an error condition, we can `raise` an exception. Here is an example that gets input from the user and checks that the number is non-negative:

Line 5 creates an exception object, in this case, a `ValueError` object, which encapsulates specific information about the error. Assume that in this case function A called B which called C which called D which called `get_age`. The `raise` statement on line 6 carries this object out as a kind of "return value", and immediately exits from `get_age()` to its caller D. Then D again exits to its caller C, and C exits to B and so on, each returning the exception object to their caller, until it encounters a `try ... except` that can handle the exception. We call this "unwinding the call stack".

`ValueError` is one of the built-in exception types which most closely matches the kind of error we want to raise. The complete listing of built-in exceptions can be found at the [Built-inExceptions<sup>21</sup>](#) section of the [Python Library Reference<sup>22</sup>](#), again by Python's creator, Guido van Rossum.

If the function that called `get_age` (or its caller, or their caller, ...) handles the error, then the program can carry on running; otherwise, Python prints the traceback and exits:

```
1 >>> get_age()  
2 Please enter your age: 42  
3 42  
4 >>> get_age()  
5 Please enter your age: -2  
6 Traceback (most recent call last):  
7   File "<interactive input>", line 1, in <module>  
8     File "learn_exceptions.py", line 4, in get_age  
9       raise ValueError("{0} is not a valid age".format(age))  
10 ValueError: -2 is not a valid age
```

<sup>19</sup><http://docs.python.org/py3k/tutorial/errors.html>

<sup>20</sup><http://docs.python.org/py3k/tutorial/index.html>

<sup>21</sup><http://docs.python.org/py3k/library/exceptions.html>

<sup>22</sup><http://docs.python.org/py3k/library/index.html>

The error message includes the exception type and the additional information that was provided when the exception object was first created.

It is often the case that lines 5 and 6 (creating the exception object, then raising the exception) are combined into a single statement, but there are really two different and independent things happening, so

perhaps it makes sense to keep the two steps separate when we first learn to work with exceptions. Here we show it all in a single statement:

```
1 raise ValueError("{0} is not a valid age".format(age))
```

## 19.3. Revisiting an earlier example

Using exception handling, we can now modify our `recursion_depth` example from the previous chapter so that it stops when it reaches the maximum recursion depth allowed:

```
1 def recursion_depth(number):
2     print("Recursion depth number", number)
3     try:
4         recursion_depth(number + 1)
5     except:
6         print("I cannot go any deeper into this wormhole.")
7
8 recursion_depth(0)
```

Run this version and observe the results.

## 19.4. The `finally` clause of the `try` statement

A common programming pattern is to grab a resource of some kind — e.g. we create a window for turtles to draw on, or we dial up a connection to our internet service provider, or we may open a file for writing. Then we perform some computation which may raise an exception, or may work without any problems.

Whatever happens, we want to “clean up” the resources we grabbed — e.g. close the window, disconnect our dial-up connection, or close the file. The `finally` clause of the `try` statement is the way to do just this. Consider this (somewhat contrived) example:

```
1 import turtle
2 import time
3
4 def show_poly():
5     try:
6         win = turtle.Screen()      # Grab/create a resource, e.g. a window
7         tess = turtle.Turtle()
8
9         # This dialog could be cancelled,
10        # or the conversion to int might fail, or n might be zero.
11        n = int(input("How many sides do you want in your polygon?"))
12        angle = 360 / n
13        for i in range(n):        # Draw the polygon
14            tess.forward(10)
15            tess.left(angle)
16            time.sleep(3)          # Make program wait a few seconds
17    finally:
18        win.bye()                # Close the turtle's window
19
20 show_poly()
21 show_poly()
22 show_poly()
```

In lines 20–22, `show_poly` is called three times. Each one creates a new window for its turtle, and draws a polygon with the number of sides input by the user. But what if the user enters a string that cannot be converted to an `int`? What if they close the dialog? We'll get an exception, *but even though we've had an exception, we still want to close the turtle's window*. Lines 17–18 does this for us. Whether we complete the statements in the `try` clause successfully or not, the `finally` block will always be executed.

Notice that the exception is still unhandled — only an `except` clause can handle an exception, so our program will still crash. But at least its turtle window will be closed before it crashes!

## 19.5. Glossary

```
1 exception
2     An error that occurs at runtime.
3
4 handle an exception
5     To prevent an exception from causing our program to crash, by wrapping
6     the block of code in a `try` ... `except` construct.
7
8 raise
9     To create a deliberate exception by using the `raise` statement.
```

## 19.6. Exercises

1. Write a function named `readposint` that uses the `input` dialog to prompt the user for a positive integer and then checks the input to confirm that it meets the requirements. It should be able to handle inputs that cannot be converted to `int`, as well as negative ints, and edge cases (e.g. when the user closes the dialog, or does not enter anything at all.)

# Chapter 20: Dictionaries

All of the compound data types we have studied in detail so far — strings, lists, and tuples — are sequence types, which use integers as indices to access the values they contain within them.

Dictionaries are yet another kind of compound type. They are Python’s built-in **mapping type**. They map **keys**, which can be any immutable type, to values, which can be any type (heterogeneous), just like the elements of a list or tuple. In other languages, they are called associative arrays since they associate a key with a value.

As an example, we will create a dictionary to translate English words into Spanish. For this dictionary, the keys are strings.

One way to create a dictionary is to start with the empty dictionary and add **key:value pairs**. The empty dictionary is denoted `{}`:

```
1 >>> eng2sp = {}  
2 >>> eng2sp["one"] = "uno"  
3 >>> eng2sp["two"] = "dos"
```

The first assignment creates a dictionary named `eng2sp`; the other assignments add new key:value pairs to the dictionary. We can print the current value of the dictionary in the usual way:

```
1 >>> print(eng2sp)  
2 {"two": "dos", "one": "uno"}
```

The key:value pairs of the dictionary are separated by commas. Each pair contains a key and a value separated by a colon.

## Hashing

The order of the pairs may not be what was expected. Python uses complex algorithms, designed for very fast access, to determine where the key:value pairs are stored in a dictionary. For our purposes we can think of this ordering as unpredictable.

You also might wonder why we use dictionaries at all when the same concept of mapping a key to a value could be implemented using a list of tuples:

```
1 >>> {"apples": 430, "bananas": 312, "oranges": 525, "pears": 217}
2 {'pears': 217, 'apples': 430, 'oranges': 525, 'bananas': 312}
3 >>> [('apples', 430), ('bananas', 312), ('oranges', 525), ('pears', 217)]
4 [('apples', 430), ('bananas', 312), ('oranges', 525), ('pears', 217)]
```

The reason dictionaries are very fast, implemented using a technique called hashing, which allows us to access a value very quickly. By contrast, the list of tuples implementation is slow. If we wanted to find a value associated with a key, we would have to iterate over every tuple, checking the 0th element. What if the key wasn't even in the list? We would have to get to the end of it to find out.

```
1 Another way to create a dictionary is to provide a list of key:value pairs using the\
2 same syntax as the previous output:
3
4 ````python
5 >>> eng2sp = {"one": "uno", "two": "dos", "three": "tres"}
```

It doesn't matter what order we write the pairs. The values in a dictionary are accessed with keys, not with indices, so there is no need to care about ordering.

Here is how we use a key to look up the corresponding value:

```
1 >>> print(eng2sp["two"])
2 'dos'
```

The key "two" yields the value "dos".

Lists, tuples, and strings have been called *sequences*, because their items occur in order. The dictionary is the first compound type that we've seen that is not a sequence, so we can't index or slice a dictionary.

## 20.1. Dictionary operations

The `del` statement removes a key:value pair from a dictionary. For example, the following dictionary contains the names of various fruits and the number of each fruit in stock:

```
1 >>> inventory = {"apples": 430, "bananas": 312, "oranges": 525, "pears": 217}
2 >>> print(inventory)
3 {'pears': 217, 'apples': 430, 'oranges': 525, 'bananas': 312}
```

If someone buys all of the pears, we can remove the entry from the dictionary:

```
1 >>> del inventory["pears"]
2 >>> print(inventory)
3 {'apples': 430, 'oranges': 525, 'bananas': 312}
```

Or if we're expecting more pears soon, we might just change the value associated with pears:

```
1 >>> inventory["pears"] = 0
2 >>> print(inventory)
3 {'pears': 0, 'apples': 430, 'oranges': 525, 'bananas': 312}
```

A new shipment of bananas arriving could be handled like this:

```
1 >>> inventory["bananas"] += 200
2 >>> print(inventory)
3 {'pears': 0, 'apples': 430, 'oranges': 525, 'bananas': 512}
```

The `len` function also works on dictionaries; it returns the number of key:value pairs:

```
1 >>> len(inventory)
2 4
```

## 20.2. Dictionary methods

Dictionaries have a number of useful built-in methods.

The `keys` method returns what Python 3 calls a `view` of its underlying keys. A view object has some similarities to the `range` object we saw earlier —it is a lazy promise, to deliver its elements when they're needed by the rest of the program. We can iterate over the view, or turn the view into a list like this:

```
1 for k in eng2sp.keys():    # The order of the k's is not defined
2     print("Got key", k, "which maps to value", eng2sp[k])
3
4 ks = list(eng2sp.keys())
5 print(ks)
```

This produces this output:

```
1 Got key three which maps to value tres
2 Got key two which maps to value dos
3 Got key one which maps to value uno
4 ['three', 'two', 'one']
```

It is so common to iterate over the keys in a dictionary that we can omit the `keys` method call in the `for` loop — iterating over a dictionary implicitly iterates over its keys:

```
1 for k in eng2sp:
2     print("Got key", k)
```

The `values` method is similar; it returns a view object which can be turned into a list:

```
1 >>> list(eng2sp.values())
2 ['tres', 'dos', 'uno']
```

The `items` method also returns a view, which promises a list of tuples — one tuple for each key:value pair:

```
1 >>> list(eng2sp.items())
2 [('three', 'tres'), ('two', 'dos'), ('one', 'uno')]
```

Tuples are often useful for getting both the key and the value at the same time while we are looping:

```
1 for (k,v) in eng2sp.items():
2     print("Got", k, "that maps to", v)
```

This produces:

```
1 Got three that maps to tres
2 Got two that maps to dos
3 Got one that maps to uno
```

The `in` and `not in` operators can test if a key is in the dictionary:

```
1 >>> "one" in eng2sp
2 True
3 >>> "six" in eng2sp
4 False
5 >>> "tres" in eng2sp      # Note that 'in' tests keys, not values.
6 False
```

This method can be very useful, since looking up a non-existent key in a dictionary causes a runtime error:

```
1 >>> eng2esp["dog"]
2 Traceback (most recent call last):
3 ...
4 KeyError: 'dog'
```

## 20.3. Aliasing and copying

As in the case of lists, because dictionaries are mutable, we need to be aware of aliasing. Whenever two variables refer to the same object, changes to one affect the other.

If we want to modify a dictionary and keep a copy of the original, use the `copy` method. For example, `opposites` is a dictionary that contains pairs of opposites:

```
1 >>> opposites = {"up": "down", "right": "wrong", "yes": "no"}
2 >>> alias = opposites
3 >>> copy = opposites.copy()  # Shallow copy
```

`alias` and `opposites` refer to the same object; `copy` refers to a fresh copy of the same dictionary. If we modify `alias`, `opposites` is also changed:

```
1 >>> alias["right"] = "left"
2 >>> opposites["right"]
3 'left'
```

If we modify `copy`, `opposites` is unchanged:

```
1 >>> copy["right"] = "privilege"
2 >>> opposites["right"]
3 'left'
```

## 20.4. Sparse matrices

We previously used a list of lists to represent a matrix. That is a good choice for a matrix with mostly nonzero values, but consider a [sparse matrix](#)<sup>23</sup> like this one:

$$\begin{bmatrix} 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 3 & 0 \end{bmatrix}$$

The list representation contains a lot of zeroes:

```
1 matrix = [[0, 0, 0, 1, 0],
2      [0, 0, 0, 0, 0],
3      [0, 2, 0, 0, 0],
4      [0, 0, 0, 0, 0],
5      [0, 0, 0, 3, 0]]
```

An alternative is to use a dictionary. For the keys, we can use tuples that contain the row and column numbers. Here is the dictionary representation of the same matrix:

```
1 >>> matrix = {(0, 3): 1, (2, 1): 2, (4, 3): 3}
```

We only need three key:value pairs, one for each nonzero element of the matrix. Each key is a tuple, and each value is an integer.

To access an element of the matrix, we could use the [] operator:

```
1 >>> matrix[(0, 3)]
2 1
```

Notice that the syntax for the dictionary representation is not the same as the syntax for the nested list representation. Instead of two integer indices, we use one index, which is a tuple of integers.

There is one problem. If we specify an element that is zero, we get an error, because there is no entry in the dictionary with that key:

```
1 >>> matrix[(1, 3)]
2 KeyError: (1, 3)
```

The `get` method solves this problem:

---

<sup>23</sup>[http://en.wikipedia.org/wiki/Sparse\\_matrix](http://en.wikipedia.org/wiki/Sparse_matrix)

```

1 >>> matrix.get((0, 3), 0)
2 1

```

The first argument is the key; the second argument is the value get should return if the key is not in the dictionary:

```

1 >>> matrix.get((1, 3), 0)
2 0

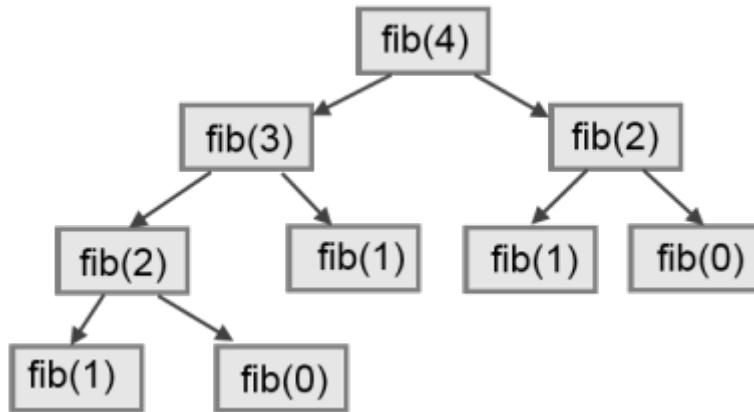
```

get definitely improves the semantics of accessing a sparse matrix. Shame about the syntax.

## 20.5. Memoization

If you played around with the fibo function from the chapter on recursion, you might have noticed that the bigger the argument you provide, the longer the function takes to run. Furthermore, the run time increases very quickly. On one of our machines, fib(20) finishes instantly, fib(30) takes about a second, and fib(40) takes roughly forever.

To understand why, consider this **call graph** for fib with  $n = 4$ :



A call graph shows some function frames (instances when the function has been invoked), with lines connecting each frame to the frames of the functions it calls. At the top of the graph, fib with  $n = 4$  calls fib with  $n = 3$  and  $n = 2$ . In turn, fib with  $n = 3$  calls fib with  $n = 2$  and  $n = 1$ . And so on.

Count how many times fib(0) and fib(1) are called. This is an inefficient solution to the problem, and it gets far worse as the argument gets bigger.

A good 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 an implementation of fib using memos:

```

1 alreadyknown = {0: 0, 1: 1}
2
3 def fib(n):
4     if n not in alreadyknown:
5         new_value = fib(n-1) + fib(n-2)
6         alreadyknown[n] = new_value
7     return alreadyknown[n]

```

The dictionary named `alreadyknown` keeps track of the Fibonacci numbers we already know. We start with only two pairs: 0 maps to 1; and 1 maps to 1.

Whenever `fib` is called, it checks the dictionary to determine if it contains the result. If it's there, the function can return immediately without making any more recursive calls. If not, it has to compute the new value. The new value is added to the dictionary before the function returns.

Using this version of `fib`, our machines can compute `fib(100)` in an eyeblink.

```

1 >>> fib(100)
2 354224848179261915075

```

## 20.6. Counting letters

In the exercises in Chapter 8 (Strings) we wrote a function that counted the number of occurrences of a letter in a string. A more general version of this problem is to form a frequency table of the letters in the string, that is, how many times each letter appears.

Such a frequency table might be useful for compressing a text file. Because different letters appear with different frequencies, we can compress a file by using shorter codes for common letters and longer codes for letters that appear less frequently.

Dictionaries provide an elegant way to generate a frequency table:

```

1 >>> letter_counts = {}
2 >>> for letter in "Mississippi":
3     ...     letter_counts[letter] = letter_counts.get(letter, 0) + 1
4 ...
5 >>> letter_counts
6 {'M': 1, 's': 4, 'p': 2, 'i': 4}

```

We start with an empty dictionary. For each letter in the string, we find the current count (possibly zero) and increment it. At the end, the dictionary contains pairs of letters and their frequencies.

It might be more appealing to display the frequency table in alphabetical order. We can do that with the `items` and `sort` methods:

```
1 >>> letter_items = list(letter_counts.items())
2 >>> letter_items.sort()
3 >>> print(letter_items)
4 [('M', 1), ('i', 4), ('p', 2), ('s', 4)]
```

Notice in the first line we had to call the type conversion function `list`. That turns the promise we get from `items` into a list, a step that is needed before we can use the list's `sort` method.

## 20.7. Glossary

```
1 call graph
2     A graph consisting of nodes which represent function frames (or
3     invocations), and directed edges (lines with arrows) showing which
4     frames gave rise to other frames.
5
6 dictionary
7     A collection of key:value pairs that maps from keys to values. The keys
8     can be any immutable value, and the associated value can be of any type.
9
10 immutable data value
11     A data value which cannot be modified. Assignments to elements or slices
12     (sub-parts) of immutable values cause a runtime error.
13
14 key
15     A data item that is *mapped to* a value in a dictionary. Keys are used
16     to look up values in a dictionary. Each key must be unique across the
17     dictionary.
18
19 key:value pair
20     One of the pairs of items in a dictionary. Values are looked up in a
21     dictionary by key.
22
23 mapping type
24     A mapping type is a data type comprised of a collection of keys and
25     associated values. Python's only built-in mapping type is the
26     dictionary. Dictionaries implement the [associative array](http://en.wikipedia.org\\
27 rg/wiki/Associative\_array) abstract data type.
28
29 memo
30     Temporary storage of precomputed values to avoid duplicating the same
31     computation.
```

```
33 mutable data value
34     A data value which can be modified. The types of all mutable values are
35     compound types. Lists and dictionaries are mutable; strings and tuples
36     are not.
```

## 20.8. Exercises

1. Write a program that reads a string and returns a table of the letters of the alphabet in alphabetical order which occur in the string together with the number of times each letter occurs. Case should be ignored. A sample output of the program when the user enters the data “ThiS is String with Upper and lower case Letters”, would look this this:

```
a 2
c 1
d 1
e 5
g 1
h 2
i 4
l 2
n 2
o 1
p 2
r 4
s 5
t 5
u 1
w 2
```

2. Give the Python interpreter’s response to each of the following from a continuous interpreter session:

```
1      >>> d = {"apples": 15, "bananas": 35, "grapes": 12}
2      >>> d["bananas"]

1      >>> d["oranges"] = 20
2      >>> len(d)
```

```
1      >>> "grapes" in d  
  
1      >>> d["pears"]  
  
1      >>> d.get("pears", 0)  
  
1      >>> fruits = list(d.keys())  
2      >>> fruits.sort()  
3      >>> print(fruits)  
  
1      >>> del d["apples"]  
2      >>> "apples" in d
```

Be sure you understand why you get each result. Then apply what you have learned to fill in the body of the function below:

```
1  def add_fruit(inventory, fruit, quantity=0):  
2      return  
3  
4      # Make these tests work...  
5      new_inventory = {}  
6      add_fruit(new_inventory, "strawberries", 10)  
7      test("strawberries" in new_inventory)  
8      test(new_inventory["strawberries"] == 10)  
9      add_fruit(new_inventory, "strawberries", 25)  
10     test(new_inventory["strawberries"] == 35)
```

3. Write a program called `alice_words.py` that creates a text file named `alice_words.txt` containing an

alphabetical listing of all the words, and the number of times each occurs, in the text version of Alice's Adventures in Wonderland. (You can obtain a free plain text version of the book, along with many others, from <http://www.gutenberg.org>.) The first 10 lines of your output file should look something like this:

```
Word Count  
=====
```

a 631

a-piece 1

abide 1

able 1

about 94

above 3

absence 1

absurd 2

How many times does the word `alice` occur in the book?

4. What is the longest word in Alice in Wonderland? How many characters does it have?

# Chapter 21: A Case Study: Indexing your files

We present a small case study that ties together modules, recursion, files, dictionaries and introduces simple serialization and deserialization.

In this chapter we're going to use a dictionary to help us find a file rapidly.

The case study has two components:

- A crawler program that scans the disk (or folder) and constructs and saves the dictionary to your disk.
- A query program that loads the dictionary and can rapidly answer user queries about where a file is located.

## 21.1. The Crawler

Near the end of the chapter on recursion we showed an example of how to recursively list all files under a given path of our filesystem.

We'll borrow (and change) that code somewhat to provide the skeleton of our crawler. It's function is to recursively traverse every file in a given path. (We'll figure out what to do with the file soon: for the moment we'll just print its short name, and its full path.)

```
1 # Crawler crawls the filesystem and builds a dictionary
2 import os
3
4 def crawl_files(path):
5     """ Recursively visit all files in path """
6
7     # Fetch all the entries in the current folder.
8     dirlist = os.listdir(path)
9     for f in dirlist:
10         # Turn each name into full pathname
11         fullname = os.path.join(path, f)
12
13         # If it is a directory, recurse.
```

```

14     if os.path.isdir(fullname):
15         crawl_files(fullname)
16     else: # Do something useful with the file
17         print("{0:30} {1}".format(f, fullname))
18
19 crawl_files("C:\\\\Python32")

```

We get output similar to this:

1	CherryPy-wininst.log	C:\Python32\CherryPy-wininst.log
2	bz2.pyd	C:\Python32\DLLs\bz2.pyd
3	py.ico	C:\Python32\DLLs\py.ico
4	pyc.ico	C:\Python32\DLLs\pyc.ico
5	pyexpat.pyd	C:\Python32\DLLs\pyexpat.pyd
6	python3.dll	C:\Python32\DLLs\python3.dll
7	select.pyd	C:\Python32\DLLs\select.pyd
8	sqlite3.dll	C:\Python32\DLLs\sqlite3.dll
9	tcl85.dll	C:\Python32\DLLs\tcl85.dll
10	tclpip85.dll	C:\Python32\DLLs\tclpip85.dll
11	tk85.dll	C:\Python32\DLLs\tk85.dll
12	...	

We'll adapt this now to store the short name and the full path of the file as a key:value pair in a dictionary. But first, two observations:

- We can have many instances of files with the same name (in different paths). For example, the name index.

html is quite common. But dictionary keys must be unique. Our solution is to map each key in our dictionary to a *list* of paths.

- File names are not case sensitive. (Well, not for Windows users!) So a good technique is to *normalize* the keys before storing them. Here we'll just ensure that all keys are converted to lowercase. Of course we'll do the same later when we write the query program.

We'll change the code above by setting up a global dictionary, initially empty: The statement `thedict = {}` inserted at line 3 will do this. Then instead of printing the information at line 17, we'll add the filename and path to the dictionary. The code will need to check whether the key already exists:

```

1 key = f.lower() # Normalize the filename
2 if key in thedict:
3     thedict[key].append(fullname)
4 else: # insert the key and a list of one pathname
5     thedict[key] = [fullname]

```

After running for a while the program terminates. We can interactively confirm that the dictionary seems to have been built correctly:

```

1 >>> len(thedict)
2 14861
3 >>> thedict["python.exe"]
4 ['C:\\Python32\\python.exe']
5 >>> thedict["logo.png"]
6 ['C:\\Python32\\Lib\\site-packages\\PyQt4\\doc\\html\\_static\\logo.png',
7 'C:\\Python32\\Lib\\site-packages\\PyQt4\\doc\\sphinx\\static\\logo.png',
8 'C:\\Python32\\Lib\\site-packages\\PyQt4\\examples\\demos\\textedit\\images\\logo.p\\
9 ng',
10 'C:\\Python32\\Lib\\site-packages\\sphinx-1.1.3-py3.2.egg\\sphinx\\themes\\scrolls\\
11 \\static\\logo.png']
12 >>>

```

It would be nice to add a progress bar while the crawler is running: a typical technique is to print dots to show progress. We'll introduce a count of how many files have been indexed (this can be a global variable), and after we've handled the current file, we'll add this code:

```

1 filecount += 1
2 if filecount % 100 == 0:
3     print(".", end="")
4     if filecount % 5000 == 0:
5         print()

```

As we complete each 100 files we print a dot. After every 50 dots we start a new line. You'll need to also create the global variable, initialize it to zero, and remember to declare the variable as global in the crawler.

The main calling code can now print some statistics for us. It becomes

```
1 crawl_files("C:\\Python32")
2 print() # End the last line of dots ...
3 print("Indexed {0} files, {1} entries in the dictionary."
4         .format(filecount, len(thedict)))
```

We'll now get something like

```
1 >>>
2 .....
3 .....
4 .....
5 .....
6 Indexed 18635 files, 14861 entries in the dictionary.
7 >>>
```

It is reassuring to look at the properties of the folder in our operating system, and note that it counts exactly the same number of files as our program does!

## 21.2. Saving the dictionary to disk

The dictionary we've built is an object. To save it we're going to turn it into a string, and write the string to a file on our disk. The string must be in a format that allows another program to unambiguously reconstruct another dictionary with the same key-value entries. The process of turning an object into a string representation is called **serialization**, and the inverse operation — reconstructing a new object from a string — is called **deserialization**.

There are a few ways to do this: some use binary formats, some use text formats, and the way different types of data are encoded differs. A popular, lightweight technique used extensively in web servers and web pages is to use JSON (JavaScript Object Notation) encoding.

Amazingly, we need just four new lines of code to save our dictionary to our disk:

```
1 import json
2
3 f = open("C:\\temp\\mydict.txt", "w")
4 json.dump(thedict, f)
5 f.close()
```

You can find the file on your disk and open it with a text editor to see what the JSON encoding looks like.

## 21.3. The Query Program

This needs to reconstruct the dictionary from the disk file, and then provide a lookup function:

```
1 import json
2
3 f = open("C:\\temp\\mydict.txt", "r")
4 dict = json.load(f)
5 f.close()
6 print("Loaded {} filenames for querying.".format(len(dict)))
7
8 def query(filename):
9     f = filename.lower()
10    if f not in dict:
11        print("No hits for {}".format(filename))
12    else:
13        print("{} is at {}".format(filename))
14        for p in dict[f]:
15            print("...", p)
```

And here is a sample run:

```
1 >>>
2 Loaded 14861 filenames for querying.
3 >>> query('python.exe')
4 python.exe is at
5 ... C:\Python32\python.exe
6 >>> query('java.exe')
7 No hits for java.exe
8 >>> query('INDEX.Html')
9 INDEX.Html is at
10 ... C:\Python32\Lib\site-packages\cherrypy\test\static\index.html
11 ... C:\Python32\Lib\site-packages\eric5\Documentation\Source\index.html
12 ... C:\Python32\Lib\site-packages\IPython\frontend\html\notebook\static\codemirror\m\
13 ode\css\index.html
14 ... C:\Python32\Lib\site-packages\IPython\frontend\html\notebook\static\codemirror\m\
15 ode\htmlmixed\index.html
16 ... C:\Python32\Lib\site-packages\IPython\frontend\html\notebook\static\codemirror\m\
17 ode\javascript\index.html
18 ... C:\Python32\Lib\site-packages\IPython\frontend\html\notebook\static\codemirror\m\
19 ode\markdown\index.html
20 ... C:\Python32\Lib\site-packages\IPython\frontend\html\notebook\static\codemirror\m\
21 ode\python\index.html
22 ... C:\Python32\Lib\site-packages\IPython\frontend\html\notebook\static\codemirror\m\
23 ode\rst\index.html
24 ... C:\Python32\Lib\site-packages\IPython\frontend\html\notebook\static\codemirror\m\
25 ode\xml\index.html
```

```

26 ... C:\Python32\Lib\site-packages\pygame\docs\index.html
27 ... C:\Python32\Lib\site-packages\pygame\docs\ref\index.html
28 ... C:\Python32\Lib\site-packages\PyQt4\doc\html\index.html
29 >>>

```

## 21.4. Compressing the serialized dictionary

The JSON file might get quite big. Gzip compression is available in Python, so let's take advantage of it...

When we saved the dictionary to disk we opened a text file for writing. We simply have to change that one line of the program (and import the correct modules), to create a gzip file instead of a normal text file. The replacement code is

```

1 import json, gzip, io
2
3 ## f = open("C:\\temp\\mydict.txt", "w")
4 f = io.TextIOWrapper(gzip.open("C:\\temp\\mydict.gz", mode="wb"))
5 json.dump(thedict, f)
6 f.close()

```

Magically, we now get a zipped file that is about 7 times smaller than the text version. (Compression/decompression like this is often done by web servers and browsers for significantly faster downloads.)

Now, of course, our query program needs to uncompress the data:

```

1 import json, gzip, io
2
3 ## f = open("C:\\temp\\mydict.txt", "r")
4 f = io.TextIOWrapper(gzip.open("C:\\temp\\mydict.gz", mode="r"))
5 dict = json.load(f)
6 f.close()
7 print("Loaded {} filenames for querying.".format(len(dict)))

```

Composability is the key...

In the earliest chapters of the book we talked about composability: the ability to join together or *compose* different fragments of code and functionality to build more powerful constructs.

This case study has shown an excellent example of this. Our JSON serializer and deserializer can link with our file mechanisms. The gzip compressor / decompressor can also present itself to our program as if it was just a specialized stream of data, as one might get from reading a file. The end result is a very elegant composition of powerful tools. Instead of requiring separate steps for serializing the dictionary to a string, compressing the string, writing the resulting bytes to a file, etc., the composability has let us do it all very easily!

## 21.5. Glossary

**deserialization**

Reconstruction an in-memory object from some external text representation

**gzip**

A lossless compression technique that reduces the storage size of data. (Lossless means you can recover the original data exactly.)

**JSON**

JavaScript Object Notation is a format for serializing and transporting objects, often used between web servers and web browsers that run JavaScript. Python contains a `json` module to provide this capability.

**serialization**

Turning an object into a string (or bytes) so that it can be sent over the internet, or saved in a file. The recipient can reconstruct a new object from the data.

# Chapter 22: Even more OOP

## 22.1. MyTime

As another example of a user-defined type, we'll define a class called `MyTime` that records the time of day. We'll provide an `__init__` method to ensure that every instance is created with appropriate attributes and initialization. The class definition looks like this:

```
1 class MyTime:  
2  
3     def __init__(self, hrs=0, mins=0, secs=0):  
4         """ Create a MyTime object initialized to hrs, mins, secs """  
5         self.hours = hrs  
6         self.minutes = mins  
7         self.seconds = secs
```

We can instantiate a new `MyTime` object:

```
1 tim1 = MyTime(11, 59, 30)
```

The state diagram for the object looks like this:



We'll leave it as an exercise for the readers to add a `__str__` method so that `MyTime` objects can print themselves decently.

## 22.2. Pure functions

In the next few sections, we'll write two versions of a function called `add_time`, which calculates the sum of two `MyTime` objects. They will demonstrate two kinds of functions: pure functions and modifiers.

The following is a rough version of `add_time`:

```
1 def add_time(t1, t2):
2     h = t1.hours + t2.hours
3     m = t1.minutes + t2.minutes
4     s = t1.seconds + t2.seconds
5     sum_t = MyTime(h, m, s)
6     return sum_t
```

The function creates a new `MyTime` object 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 parameters and it has no side effects, such as updating global variables, displaying a value, or getting user input.

Here is an example of how to use this function. We'll create two `MyTime` objects: `current_time`, which contains the current time; and `bread_time`, which contains the amount of time it takes for a breadmaker to make bread. Then we'll use `add_time` to figure out when the bread will be done.

```
1 >>> current_time = MyTime(9, 14, 30)
2 >>> bread_time = MyTime(3, 35, 0)
3 >>> done_time = add_time(current_time, bread_time)
4 >>> print(done_time)
5 12:49:30
```

The output of this program is `12:49:30`, which is correct. On the other hand, there are cases where the result is not correct. Can you think of one?

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 minutes column or the extra minutes into the hours column.

Here's a better version of the function:

```
1 def add_time(t1, t2):
2
3     h = t1.hours + t2.hours
4     m = t1.minutes + t2.minutes
5     s = t1.seconds + t2.seconds
6
7     if s >= 60:
8         s -= 60
9         m += 1
10
11    if m >= 60:
12        m -= 60
13        h += 1
14
```

```

15     sum_t = MyTime(h, m, s)
16     return sum_t

```

This function is starting to get bigger, and still doesn't work for all possible cases. Later we will suggest an alternative approach that yields better code.

## 22.3. Modifiers

There are times when it is useful for a function to modify one or more of the objects it gets as parameters. Usually, the caller keeps a reference to the objects it passes, so any changes the function makes are visible to the caller. Functions that work this way are called **modifiers**.

increment, which adds a given number of seconds to a `MyTime` object, would be written most naturally as a modifier. A rough draft of the function looks like this:

```

1 def increment(t, secs):
2     t.seconds += secs
3
4     if t.seconds >= 60:
5         t.seconds -= 60
6         t.minutes += 1
7
8     if t.minutes >= 60:
9         t.minutes -= 60
10        t.hours += 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 the parameter `seconds` is much greater than sixty? In that case, it is not enough to carry once; we have to keep doing it until `seconds` is less than sixty. One solution is to replace the `if` statements with `while` statements:

```

1 def increment(t, seconds):
2     t.seconds += seconds
3
4     while t.seconds >= 60:
5         t.seconds -= 60
6         t.minutes += 1
7
8     while t.minutes >= 60:
9         t.minutes -= 60
10        t.hours += 1

```

This function is now correct when `seconds` is not negative, and when `hours` does not exceed 23, but it is not a particularly good solution.

## 22.4. Converting increment to a method

Once again, OOP programmers would prefer to put functions that work with `MyTime` objects into the `MyTime` class, so let's convert `increment` to a method. To save space, we will leave out previously defined methods, but you should keep them in your version:

```
1 class MyTime:  
2     # Previous method definitions here...  
3  
4     def increment(self, seconds):  
5         self.seconds += seconds  
6  
7         while self.seconds >= 60:  
8             self.seconds -= 60  
9             self.minutes += 1  
10  
11        while self.minutes >= 60:  
12            self.minutes -= 60  
13            self.hours += 1
```

The transformation is purely mechanical — we move the definition into the class definition and (optionally) change the name of the first parameter to `self`, to fit with Python style conventions.

Now we can invoke `increment` using the syntax for invoking a method.

```
1 current_time.increment(500)
```

Again, the object on which the method is invoked gets assigned to the first parameter, `self`. The second parameter, `seconds` gets the value 500.

## 22.5. An “Aha!” insight

Often a high-level insight into the problem can make the programming much easier.

In this case, the insight is that a `MyTime` object is really a three-digit number in base 60! The second component is the ones column, the `minute` component is the sixties column, and the `hour` component 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 a `MyTime` object into a single number and take advantage of the fact that the computer knows how to do arithmetic with numbers. The following method is added to the `MyTime` class to convert any instance into a corresponding number of seconds:

```

1 class MyTime:
2     # ...
3
4     def to_seconds(self):
5         """ Return the number of seconds represented
6             by this instance
7         """
8
9         return self.hours * 3600 + self.minutes * 60 + self.seconds

```

Now, all we need is a way to convert from an integer back to a `MyTime` object. Supposing we have `tsecs` seconds, some integer division and mod operators can do this for us:

```

1 hrs = tsecs // 3600
2 leftoversecs = tsecs % 3600
3 mins = leftoversecs // 60
4 secs = leftoversecs % 60

```

You might have to think a bit to convince yourself that this technique to convert from one base to another is correct.

In OOP we're really trying to wrap together the data and the operations that apply to it. So we'd like to have this logic inside the `MyTime` class. A good solution is to rewrite the class initializer so that it can cope with initial values of seconds or minutes that are outside the **normalized** values. (A normalized time would be something like 3 hours 12 minutes and 20 seconds. The same time, but unnormalized could be 2 hours 70 minutes and 140 seconds.)

Let's rewrite a more powerful initializer for `MyTime`:

```

1 class MyTime:
2     # ...
3
4     def __init__(self, hrs=0, mins=0, secs=0):
5         """ Create a new MyTime object initialized to hrs, mins, secs.
6             The values of mins and secs may be outside the range 0-59,
7             but the resulting MyTime object will be normalized.
8         """
9
10        # Calculate total seconds to represent
11        totalsecs = hrs*3600 + mins*60 + secs
12        self.hours = totalsecs // 3600          # Split in h, m, s
13        leftoversecs = totalsecs % 3600
14        self.minutes = leftoversecs // 60
15        self.seconds = leftoversecs % 60

```

Now we can rewrite `add_time` like this:

```
1 def add_time(t1, t2):
2     secs = t1.to_seconds() + t2.to_seconds()
3     return MyTime(0, 0, secs)
```

This version is much shorter than the original, and it is much easier to demonstrate or reason that it is correct.

## 22.6. Generalization

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 times is better.

But if we have the insight to treat times as base 60 numbers and make the investment of writing the conversions, 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 `MyTime` objects 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 the programming easier, because there are fewer special cases and fewer opportunities for error.

Specialization versus Generalization

Computer Scientists are generally fond of specializing their types, while mathematicians often take the opposite approach, and generalize everything.

What do we mean by this?

If we ask a mathematician to solve a problem involving weekdays, days of the century, playing cards, time, or dominoes, their most likely response is to observe that all these objects can be represented by integers. Playing cards, for example, can be numbered from 0 to 51. Days within the century can be numbered. Mathematicians will say “*These things are enumerable — the elements can be uniquely numbered (and we can reverse this numbering to get back to the original concept). So let's number them, and confine our thinking to integers. Luckily, we have powerful techniques and a good understanding of integers, and so our abstractions — the way we tackle and simplify these problems — is to try to reduce them to problems about integers.*”

Computer Scientists tend to do the opposite. We will argue that there are many integer operations that are simply not meaningful for dominoes, or for days of the century. So we'll often define new specialized types, like `MyTime`, because we can restrict, control, and specialize the operations that are possible. Object-oriented programming is particularly popular because it gives us a good way to bundle methods and specialized data into a new type.

Both approaches are powerful problem-solving techniques. Often it may help to try to think about the problem from both points of view — “*What would happen if I tried to reduce everything to very few primitive types?*”, versus “*What would happen if this thing had its own specialized type?*”

## 22.7. Another example

The `after` function should compare two times, and tell us whether the first time is strictly after the second, e.g.

```

1 >>> t1 = MyTime(10, 55, 12)
2 >>> t2 = MyTime(10, 48, 22)
3 >>> after(t1, t2)           # Is t1 after t2?
4 True

```

This is slightly more complicated because it operates on two `MyTime` objects, not just one. But we'd prefer to write it as a method anyway — in this case, a method on the first argument:

```

1 class MyTime:
2     # Previous method definitions here...
3
4     def after(self, time2):
5         """ Return True if I am strictly greater than time2 """
6         if self.hours > time2.hours:
7             return True
8         if self.hours < time2.hours:
9             return False
10
11        if self.minutes > time2.minutes:
12            return True
13        if self.minutes < time2.minutes:
14            return False
15        if self.seconds > time2.seconds:
16            return True
17
18        return False

```

We invoke this method on one object and pass the other as an argument:

```

1 if current_time.after(done_time):
2     print("The bread will be done before it starts!")

```

We can almost read the invocation like English: If the current time is after the done time, then...

The logic of the `if` statements deserve special attention here. Lines 11-18 will only be reached if the two hour fields are the same. Similarly, the test at line 16 is only executed if both times have the same hours and the same minutes.

Could we make this easier by using our “Aha!” insight and extra work from earlier, and reducing both times to integers? Yes, with spectacular results!

```

1 class MyTime:
2     # Previous method definitions here...
3
4     def after(self, time2):
5         """ Return True if I am strictly greater than time2 """
6         return self.to_seconds() > time2.to_seconds()

```

This is a great way to code this: if we want to tell if the first time is after the second time, turn them both into integers and compare the integers.

## 22.8. Operator overloading

Some languages, including Python, make it possible to have different meanings for the same operator when applied to different types. For example, + in Python means quite different things for integers and for strings. This feature is called **operator overloading**.

It is especially useful when programmers can also overload the operators for their own user-defined types.

For example, to override the addition operator +, we can provide a method named `__add__`:

```

1 class MyTime:
2     # Previously defined methods here...
3
4     def __add__(self, other):
5         return MyTime(0, 0, self.to_seconds() + other.to_seconds())

```

As usual, the first parameter is the object on which the method is invoked. The second parameter is conveniently named `other` to distinguish it from `self`. To add two `MyTime` objects, we create and return a new `MyTime` object that contains their sum.

Now, when we apply the + operator to `MyTime` objects, Python invokes the `__add__` method that we have written:

```

1 >>> t1 = MyTime(1, 15, 42)
2 >>> t2 = MyTime(3, 50, 30)
3 >>> t3 = t1 + t2
4 >>> print(t3)
5 05:06:12

```

The expression `t1 + t2` is equivalent to `t1.__add__(t2)`, but obviously more elegant. As an exercise, add a method `__sub__(self, other)` that overloads the subtraction operator, and try it out.

For the next couple of exercises we'll go back to the `Point` class defined in our first chapter about objects, and overload some of its operators. Firstly, adding two points adds their respective (x, y) coordinates:

```

1 class Point:
2     # Previously defined methods here...
3
4     def __add__(self, other):
5         return Point(self.x + other.x, self.y + other.y)

```

There are several ways to override the behavior of the multiplication operator: by defining a method named `__mul__`, or `__rmul__`, or both.

If the left operand of `*` is a `Point`, Python invokes `__mul__`, which assumes that the other operand is also a `Point`. It computes the **dot product** of the two `Points`, defined according to the rules of linear algebra:

```

1 def __mul__(self, other):
2     return self.x * other.x + self.y * other.y

```

If the left operand of `*` is a primitive type and the right operand is a `Point`, Python invokes `__rmul__`, which performs **scalar multiplication**:

```

1 def __rmul__(self, other):
2     return Point(other * self.x, other * self.y)

```

The result is a new `Point` whose coordinates are a multiple of the original coordinates. If `other` is a type that cannot be multiplied by a floating-point number, then `__rmul__` will yield an error.

This example demonstrates both kinds of multiplication:

```

1 >>> p1 = Point(3, 4)
2 >>> p2 = Point(5, 7)
3 >>> print(p1 * p2)
4 43
5 >>> print(2 * p2)
6 (10, 14)

```

What happens if we try to evaluate `p2 * 2`? Since the first parameter is a `Point`, Python invokes `__mul__` with 2 as the second argument. Inside `__mul__`, the program tries to access the `x` coordinate of `other`, which fails because an integer has no attributes:

```

1 >>> print(p2 * 2)
2 AttributeError: 'int' object has no attribute 'x'

```

Unfortunately, the error message is a bit opaque. This example demonstrates some of the difficulties of object-oriented programming. Sometimes it is hard enough just to figure out what code is running.

## 22.9. Polymorphism

Most of the methods we have written only work for a specific type. When we create a new object, we write methods that operate on that type.

But there are certain operations that we will want to apply to many types, such as the arithmetic operations in the previous sections. If many types support the same set of operations, we can write functions that work on any of those types.

For example, the `multadd` operation (which is common in linear algebra) takes three parameters; it multiplies the first two and then adds the third. We can write it in Python like this:

```
1 def multadd (x, y, z):  
2     return x * y + z
```

This function will work for any values of `x` and `y` that can be multiplied and for any value of `z` that can be added to the product.

We can invoke it with numeric values:

```
1 >>> multadd (3, 2, 1)  
2 7
```

Or with `Point`s:

```
1 >>> p1 = Point(3, 4)  
2 >>> p2 = Point(5, 7)  
3 >>> print(multadd (2, p1, p2))  
4 (11, 15)  
5 >>> print(multadd (p1, p2, 1))  
6 44
```

In the first case, the `Point` is multiplied by a scalar and then added to another `Point`. In the second case, the dot product yields a numeric value, so the third parameter also has to be a numeric value.

A function like this that can take arguments with different types is called **polymorphic**.

As another example, consider the function `front_and_back`, which prints a list twice, forward and backward:

```
1 def front_and_back(front):
2     import copy
3     back = copy.copy(front)
4     back.reverse()
5     print(str(front) + str(back))
```

Because the `reverse` method is a modifier, we make a copy of the list before reversing it. That way, this function doesn't modify the list it gets as a parameter.

Here's an example that applies `front_and_back` to a list:

```
1 >>> my_list = [1, 2, 3, 4]
2 >>> front_and_back(my_list)
3 [1, 2, 3, 4][4, 3, 2, 1]
```

Of course, we intended to apply this function to lists, so it is not surprising that it works. What would be surprising is if we could apply it to a `Point`.

To determine whether a function can be applied to a new type, we apply Python's fundamental rule of polymorphism, called the **duck typing rule**: *If all of the operations inside the function can be applied to*

*the type, the function can be applied to the type.* The operations in the `front_and_back` function include `copy`, `reverse`, and `print`.

Not all programming languages define polymorphism in this way. Look up *duck typing*, and see if you can figure out why it has this name.

`copy` works on any object, and we have already written a `__str__` method for `Point` objects, so all we need is a `reverse` method in the `Point` class:

```
1 def reverse(self):
2     (self.x, self.y) = (self.y, self.x)
```

Then we can pass `Points` to `front_and_back`:

```
1 >>> p = Point(3, 4)
2 >>> front_and_back(p)
3 (3, 4)(4, 3)
```

The most interesting polymorphism is the unintentional kind, where we discover that a function we have already written can be applied to a type for which we never planned.

## 22.10. Glossary

```
1 dot product
2     An operation defined in linear algebra that multiplies two `Point`s and
3     yields a numeric value.
4
5 functional programming style
6     A style of program design in which the majority of functions are pure.
7
8 modifier
9     A function or method that changes one or more of the objects it receives
10    as parameters. Most modifier functions are void (do not return a value).
11
12 normalized
13    Data is said to be normalized if it fits into some reduced range or set
14    of rules. We usually normalize our angles to values in the range
15    \[0..360). We normalize minutes and seconds to be values in the range
16    \[0..60). And we'd be surprised if the local store advertised its cold
17    drinks at "One dollar, two hundred and fifty cents".
18
19 operator overloading
20    Extending built-in operators ( `+`, `-`, `*`, `>`, `<`, etc.) so that
21    they do different things for different types of arguments. We've seen
22    early in the book how `+` is overloaded for numbers and strings, and
23    here we've shown how to further overload it for user-defined types.
24
25 polymorphic
26    A function that can operate on more than one type. Notice the subtle
27    distinction: overloading has different functions (all with the same
28    name) for different types, whereas a polymorphic function is a single
29    function that can work for a range of types.
30
31 pure function
32    A function that does not modify any of the objects it receives as
33    parameters. Most pure functions are fruitful rather than void.
34
35 scalar multiplication
36    An operation defined in linear algebra that multiplies each of the
37    coordinates of a `Point` by a numeric value.
```

## 22.11. Exercises

1. Write a Boolean function `between` that takes two `MyTime` objects, `t1` and `t2`, as arguments, and

returns True if the invoking object falls between the two times. Assume  $t1 \leq t2$ , and make the test closed at the lower bound and open at the upper bound, i.e. return True if  $t1 \leq obj < t2$ .

2. Turn the above function into a method in the MyTime class.
3. Overload the necessary operator(s) so that instead of having to write :  
if  $t1.after(t2)$ : ...

we can use the more convenient :

if  $t1 > t2$ : ...

4. Rewrite increment as a method that uses our “Aha” insight.
5. Create some test cases for the increment method. Consider specifically the case where the number of seconds to add to the time is negative. Fix up increment so that it handles this case if it does not do so already. (You may assume that you will never subtract more seconds than are in the time object.)
6. Can physical time be negative, or must time always move in the forward direction? Some serious physicists  
think this is not such a dumb question. See what you can find on the Internet about this.

# Chapter 23: Collections of objects

## 23.1. Composition

By now, we have seen several examples of composition. One of the first examples was using a method invocation as part of an expression. Another example is the nested structure of statements; we can put an `if` statement within a `while` loop, within another `if` statement, and so on.

Having seen this pattern, and having learned about lists and objects, we should not be surprised to learn that we can create lists of objects. We can also create objects that contain lists (as attributes); we can create lists that contain lists; we can create objects that contain objects; and so on.

In this chapter and the next, we will look at some examples of these combinations, using `Card` objects as an example.

## 23.2. Card objects

If you are not familiar with common playing cards, now would be a good time to get a deck, or else this chapter might not make much sense. 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 we are playing, the rank of Ace may be higher than King or lower than 2. The rank is sometimes called the face-value of the card.

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. By `encode`, we do not mean what some people think, which is to encrypt or translate into a secret code. What a computer scientist means by `encode`

is to define a mapping between a sequence of numbers and the items I want to represent. For example:

1	Spades	-->	3
2	Hearts	-->	2
3	Diamonds	-->	1
4	Clubs	-->	0

An obvious feature of this mapping is that the suits map to integers in order, so we can compare suits by comparing integers. The mapping for ranks is fairly obvious; each of the numerical ranks maps to the corresponding integer, and for face cards:

```
1 Jack    --> 11
2 Queen  --> 12
3 King   --> 13
```

The reason we are using mathematical notation for these mappings is that they are not part of the Python program. They are part of the program design, but they never appear explicitly in the code. The class definition for the `Card` type looks like this:

```
1 class Card:
2     def __init__(self, suit=0, rank=0):
3         self.suit = suit
4         self.rank = rank
```

As usual, we provide an initialization method that takes an optional parameter for each attribute.

To create some objects, representing say the 3 of Clubs and the Jack of Diamonds, use these commands:

```
1 three_of_clubs = Card(0, 3)
2 card1 = Card(1, 11)
```

In the first case above, for example, the first argument, 0, represents the suit Clubs.

Save this code for later use ...

In the next chapter we assume that we have save the `Cards` class, and the upcoming `Deck` class in a file called `Cards.py`.

### 23.3. Class attributes and the `__str__` method

In order to print `Card` objects in a way that people can easily read, we want to map the integer codes onto words. A natural way to do that is with lists of strings. We assign these lists to **class attributes** at the top of the class definition:

```

1 class Card:
2     suits = ["Clubs", "Diamonds", "Hearts", "Spades"]
3     ranks = ["narf", "Ace", "2", "3", "4", "5", "6", "7",
4             "8", "9", "10", "Jack", "Queen", "King"]
5
6     def __init__(self, suit=0, rank=0):
7         self.suit = suit
8         self.rank = rank
9
10    def __str__(self):
11        return (self.ranks[self.rank] + " of " + self.suits[self.suit])

```

A class attribute is defined outside of any method, and it can be accessed from any of the methods in the class.

Inside `__str__`, we can use `suits` and `ranks` to map the numerical values of `suit` and `rank` to strings. For example, the expression `self.suits[self.suit]` means use the attribute `suit` from the object `self` as an index into the class attribute named `suits`, and select the appropriate string.

The reason for the "narf" in the first element in `ranks` is to act as a place keeper for the zero-eth element of the list, which will never be used. The only valid ranks are 1 to 13. This wasted item is not entirely necessary. We could have started at 0, as usual, but it is less confusing to encode the rank 2 as integer 2, 3 as 3, and so on.

With the methods we have so far, we can create and print cards:

```

1 >>> card1 = Card(1, 11)
2 >>> print(card1)
3 Jack of Diamonds

```

Class attributes like `suits` are shared by all `Card` objects. The advantage of this is that we can use any `Card` object to access the class attributes:

```

1 >>> card2 = Card(1, 3)
2 >>> print(card2)
3 3 of Diamonds
4 >>> print(card2.suits[1])
5 Diamonds

```

Because every `Card` instance references the same class attribute, we have an aliasing situation. The disadvantage is that if we modify a class attribute, it affects every instance of the class. For example, if we decide that Jack of Diamonds should really be called Jack of Swirly Whales, we could do this:

```
1 >>> card1.suits[1] = "Swirly Whales"
2 >>> print(card1)
3 Jack of Swirly Whales
```

The problem is that *all* of the Diamonds just became Swirly Whales:

```
1 >>> print(card2)
2 3 of Swirly Whales
```

It is usually not a good idea to modify class attributes.

## 23.4. Comparing cards

For primitive types, there are six relational operators (`<`, `>`, `==`, etc.) that compare values and determine when one is greater than, less than, or equal to another. If we want our own types to be comparable using the syntax of these relational operators, we need to define six corresponding special methods in our class.

We'd like to start with a single method named `cmp` that houses the logic of ordering. By convention, a comparison method takes two parameters, `self` and `other`, and returns 1 if the first object is greater, -1 if the second object is greater, and 0 if they are equal to each other.

Some types are completely ordered, which means that we can compare any two elements and tell which is bigger. For example, the integers and the floating-point numbers are completely ordered. Some types are unordered, which means that there is no meaningful way to say that one element is bigger than another. For example, the fruits are unordered, which is why we cannot compare apples and oranges, and we cannot meaningfully order a collection of images, or a collection of cellphones.

Playing cards are partially ordered, which means that sometimes we can compare cards and sometimes not. For example, we know that the 3 of Clubs is higher than the 2 of Clubs, and the 3 of Diamonds is higher than the 3 of Clubs. But 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 make cards comparable, we have to decide which is more important, rank or suit. To be honest, the choice is arbitrary. For the sake of choosing, we will say that suit is more important, because a new deck of cards comes sorted with all the Clubs together, followed by all the Diamonds, and so on.

With that decided, we can write `cmp`:

```

1 def cmp(self, other):
2     # Check the suits
3     if self.suit > other.suit: return 1
4     if self.suit < other.suit: return -1
5     # Suits are the same... check ranks
6     if self.rank > other.rank: return 1
7     if self.rank < other.rank: return -1
8     # Ranks are the same... it's a tie
9     return 0

```

In this ordering, Aces appear lower than Deuces (2s).

Now, we can define the six special methods that do the overloading of each of the relational operators for us:

```

1 def __eq__(self, other):
2     return self.cmp(other) == 0
3
4 def __le__(self, other):
5     return self.cmp(other) <= 0
6
7 def __ge__(self, other):
8     return self.cmp(other) >= 0
9
10 def __gt__(self, other):
11     return self.cmp(other) > 0
12
13 def __lt__(self, other):
14     return self.cmp(other) < 0
15
16 def __ne__(self, other):
17     return self.cmp(other) != 0

```

With this machinery in place, the relational operators now work as we'd like them to:

```

1 >>> card1 = Card(1, 11)
2 >>> card2 = Card(1, 3)
3 >>> card3 = Card(1, 11)
4 >>> card1 < card2
5 False
6 >>> card1 == card3
7 True

```

## 23.5. Decks

Now that we have objects to represent Cards, the next logical step is to define a class to represent a Deck. Of course, a deck is made up of cards, so each Deck object will contain a list of cards as an attribute. Many card games will need at least two different decks — a red deck and a blue deck.

The following is a class definition for the Deck class. The initialization method creates the attribute cards and generates the standard pack of fifty-two cards:

```

1 class Deck:
2     def __init__(self):
3         self.cards = []
4         for suit in range(4):
5             for rank in range(1, 14):
6                 self.cards.append(Card(suit, rank))

```

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. Since the outer loop iterates four times, and the inner loop iterates thirteen times, the total number of times the body is executed is fifty-two (thirteen times four). Each iteration creates a new instance of Card with the current suit and rank, and appends that card to the cards list.

With this in place, we can instantiate some decks:

```

1 red_deck = Deck()
2 blue_deck = Deck()

```

## 23.6. Printing the deck

As usual, when we define a new type we want a method that prints the contents of an instance. To print a Deck, we traverse the list and print each Card:

```

1 class Deck:
2     ...
3     def print_deck(self):
4         for card in self.cards:
5             print(card)

```

Here, and from now on, the ellipsis (...) indicates that we have omitted the other methods in the class.

As an alternative to print\_deck, we could write a `__str__` method for the Deck class. The advantage of `__str__` is that it is more flexible. Rather than just printing the contents of the object, it generates

a string representation that other parts of the program can manipulate before printing, or store for later use.

Here is a version of `__str__` that returns a string representation of a `Deck`. To add a bit of pizzazz, it arranges the cards in a cascade where each card is indented one space more than the previous card:

```
1 class Deck:  
2     ...  
3     def __str__(self):  
4         s = ""  
5         for i in range(len(self.cards)):  
6             s = s + " " * i + str(self.cards[i]) + "\n"  
7         return s
```

This example demonstrates several features. First, instead of traversing `self.cards` and assigning each card to a variable, we are using `i` as a loop variable and an index into the list of cards.

Second, we are using the string multiplication operator to indent each card by one more space than the last. The expression `" " * i` yields a number of spaces equal to the current value of `i`.

Third, instead of using the `print` command to print the cards, we use the `str` function. Passing an object as an argument to `str` is equivalent to invoking the `__str__` method on the object.

Finally, we are using the variable `s` as an **accumulator**. Initially, `s` is the empty string. Each time through the loop, a new string is generated and concatenated with the old value of `s` to get the new value. When the loop ends, `s` contains the complete string representation of the `Deck`, which looks like this:

```
1 >>> red_deck = Deck()  
2 >>> print(red_deck)  
3 Ace of Clubs  
4 2 of Clubs  
5 3 of Clubs  
6 4 of Clubs  
7 5 of Clubs  
8 6 of Clubs  
9 7 of Clubs  
10 8 of Clubs  
11 9 of Clubs  
12 10 of Clubs  
13 Jack of Clubs  
14 Queen of Clubs  
15 King of Clubs  
16 Ace of Diamonds  
17 2 of Diamonds  
18 ...
```

And so on. Even though the result appears on 52 lines, it is one long string that contains newlines.

## 23.7. Shuffling the deck

If a deck is perfectly shuffled, then any card is equally likely to appear anywhere in the deck, and any location in the deck is equally likely to contain any card.

To shuffle the deck, we will use the `randrange` function from the `random` module. With two integer arguments, `a` and `b`, `randrange` chooses a random integer in the range  $a \leq x < b$ . Since the upper bound is strictly less than `b`, we can use the length of a list as the second parameter, and we are guaranteed to get a legal index. For example, if `rng` has already been instantiated as a random number source, this expression chooses the index of a random card in a deck:

```
1 rng.randrange(0, len(self.cards))
```

An easy way to shuffle the deck is by traversing the cards and swapping each card with a randomly chosen one. It is possible that the card will be swapped with itself, but that is fine. In fact, if we precluded that possibility, the order of the cards would be less than entirely random:

```
1 class Deck:
2     ...
3     def shuffle(self):
4         import random
5         rng = random.Random()          # Create a random generator
6         num_cards = len(self.cards)
7         for i in range(num_cards):
8             j = rng.randrange(i, num_cards)
9             (self.cards[i], self.cards[j]) = (self.cards[j], self.cards[i])
```

Rather than assume that there are fifty-two cards in the deck, we get the actual length of the list and store it in `num_cards`.

For each card in the deck, we choose a random card from among the cards that haven't been shuffled yet. Then we swap the current card (`i`) with the selected card (`j`). To swap the cards we use a tuple assignment:

```
1 (self.cards[i], self.cards[j]) = (self.cards[j], self.cards[i])
```

While this is a good shuffling method, a random number generator object also has a `shuffle` method that can shuffle elements in a list, in place. So we could rewrite this function to use the one provided for us:

```

1 class Deck:
2     ...
3     def shuffle(self):
4         import random
5         rng = random.Random()          # Create a random generator
6         rng.shuffle(self.cards)        # Use its shuffle method

```

## 23.8. Removing and dealing cards

Another method that would be useful for the `Deck` class is `remove`, which takes a card as a parameter, removes it, and returns `True` if the card was in the deck and `False` otherwise:

```

1 class Deck:
2     ...
3     def remove(self, card):
4         if card in self.cards:
5             self.cards.remove(card)
6             return True
7         else:
8             return False

```

The `in` operator returns `True` if the first operand is in the second. If the first operand is an object, Python uses the object's `__eq__` method to determine equality with items in the list. Since the `__eq__` we provided in the `Card` class checks for deep equality, the `remove` method checks for deep equality.

To deal cards, we want to remove and return the top card. The list method `pop` provides a convenient way to do that:

```

1 class Deck:
2     ...
3     def pop(self):
4         return self.cards.pop()

```

Actually, `pop` removes the *last* card in the list, so we are in effect dealing from the bottom of the deck.

One more operation that we are likely to want is the Boolean function `is_empty`, which returns `True` if the deck contains no cards:

```
1 class Deck:  
2     ...  
3     def is_empty(self):  
4         return self.cards == []
```

## 23.9. Glossary

```
1 encode  
2     To represent one type of value using another type of value by  
3     constructing a mapping between them.  
4  
5 class attribute  
6     A variable that is defined inside a class definition but outside any  
7     method. Class attributes are accessible from any method in the class and  
8     are shared by all instances of the class.  
9  
10 accumulator  
11     A variable used in a loop to accumulate a series of values, such as by  
12     concatenating them onto a string or adding them to a running sum.
```

## 23.10. Exercises

1. Modify `cmp` so that Aces are ranked higher than Kings.

# Chapter 24: Inheritance

## 24.1. 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.

The primary advantage of this feature is that you can add new methods to a class without modifying the existing class. It is called inheritance because the new class inherits all of the methods of the existing class. Extending this metaphor, the existing class is sometimes called the **parent** class. The new class may be called the **child** class or sometimes subclass.

Inheritance is a powerful feature. Some programs that would be complicated without inheritance can be written concisely and simply with it. Also, 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 program 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 scattered among several modules. Also, many of the things that can be done using inheritance can be done as elegantly (or more so) without it. If the natural structure of the problem does not lend itself to inheritance, this style of programming can do more harm than good.

In this chapter we will demonstrate the use of inheritance as part of a program that plays the card game Old Maid. One of our goals is to write code that could be reused to implement other card games.

## 24.2. A hand of cards

For almost any card game, we need to represent a hand of cards. A hand is similar to a deck, of course. Both are made up of a set of cards, and both require operations like adding and removing cards. Also, we might like the ability to shuffle both decks and hands.

A hand is also different from a deck. Depending on the game being played, we might want to perform some operations on hands that don't make sense for a deck. For example, in poker we might classify a hand(straight, flush, etc.) or compare it with another hand. In bridge, we might want to compute a score for a hand in order to make a bid.

This situation suggests the use of inheritance. If `Hand` is a subclass of `Deck`, it will have all the methods of `Deck`, and new methods can be added.

We add the code in this chapter to our `Cards.py` file from the previous chapter. In the class definition, the name of the parent class appears in parentheses:

```
1 class Hand(Deck):  
2     pass
```

This statement indicates that the new `Hand` class inherits from the existing `Deck` class.

The `Hand` constructor initializes the attributes for the hand, which are `name` and `cards`. The string `name` identifies this hand, probably by the name of the player that holds it. The name is an optional parameter with the empty string as a default value. `cards` is the list of cards in the hand, initialized to the empty list:

```
1 class Hand(Deck):  
2     def __init__(self, name=""):  
3         self.cards = []  
4         self.name = name
```

For just about any card game, it is necessary to add and remove cards from the deck. Removing cards is already taken care of, since `Hand` inherits `remove` from `Deck`. But we have to write `add`:

```
1 class Hand(Deck):  
2     ...  
3     def add(self, card):  
4         self.cards.append(card)
```

Again, the ellipsis indicates that we have omitted other methods. The list `append` method adds the new card to the end of the list of cards.

## 24.3. Dealing cards

Now that we have a `Hand` class, we want to deal cards from the `Deck` into hands. It is not immediately obvious whether this method should go in the `Hand` class or in the `Deck` class, but since it operates on a

single deck and (possibly) several hands, it is more natural to put it in `Deck`.

`deal` should be fairly general, since different games will have different requirements. We may want to deal out the entire deck at once or add one card to each hand.

`deal` takes two parameters, a list (or tuple) of hands and the total number of cards to deal. If there are not enough cards in the deck, the method deals out all of the cards and stops:

```

1 class Deck:
2     ...
3     def deal(self, hands, num_cards=999):
4         num_hands = len(hands)
5         for i in range(num_cards):
6             if self.is_empty():
7                 break                      # Break if out of cards
8             card = self.pop()           # Take the top card
9             hand = hands[i % num_hands] # Whose turn is next?
10            hand.add(card)          # Add the card to the hand

```

The second parameter, `num_cards`, is optional; the default is a large number, which effectively means that all of the cards in the deck will get dealt.

The loop variable `i` goes from 0 to `num_cards-1`. Each time through the loop, a card is removed from the deck using the list method `pop`, which removes and returns the last item in the list.

The modulus operator (%) allows us to deal cards in a round robin (one card at a time to each hand). When `i` is equal to the number of hands in the list, the expression `i % num_hands` wraps around to the beginning of the list (index 0).

## 24.4. Printing a Hand

To print the contents of a hand, we can take advantage of the `__str__` method inherited from `Deck`. For example:

```

1 >>> deck = Deck()
2 >>> deck.shuffle()
3 >>> hand = Hand("frank")
4 >>> deck.deal([hand], 5)
5 >>> print(hand)
6 Hand frank contains
7 2 of Spades
8 3 of Spades
9 4 of Spades
10 Ace of Hearts
11 9 of Clubs

```

It's not a great hand, but it has the makings of a straight flush.

Although it is convenient to inherit the existing methods, there is additional information in a `Hand` object we might want to include when we print one. To do that, we can provide a `__str__` method in the `Hand` class that overrides the one in the `Deck` class:

```

1 class Hand(Deck):
2     def __str__(self):
3         s = "Hand " + self.name
4         if self.is_empty():
5             s += " is empty\n"
6         else:
7             s += " contains\n"
8         return s + Deck.__str__(self)

```

Initially, `s` is a string that identifies the hand. If the hand is empty, the program appends the words `is empty` and returns `s`.

Otherwise, the program appends the word `contains` and the string representation of the `Deck`, computed by invoking the `__str__` method in the `Deck` class on `self`.

It may seem odd to send `self`, which refers to the current `Hand`, to a `Deck` method, until you remember that a `Hand` is a kind of `Deck`. `Hand` objects can do everything `Deck` objects can, so it is legal to send a `Hand` to a `Deck` method.

In general, it is always legal to use an instance of a subclass in place of an instance of a parent class.

## 24.5. The CardGame class

The `CardGame` class takes care of some basic chores common to all games, such as creating the deck and shuffling it:

```

1 class CardGame:
2     def __init__(self):
3         self.deck = Deck()
4         self.deck.shuffle()

```

This is the first case we have seen where the initialization method performs a significant computation, beyond initializing attributes.

To implement specific games, we can inherit from `CardGame` and add features for the new game. As an example, we'll write a simulation of Old Maid.

The object of Old Maid is to get rid of cards in your hand. You do this by matching cards by rank and color. For example, the 4 of Clubs matches the 4 of Spades since both suits are black. The Jack of Hearts matches the Jack of Diamonds since both are red.

To begin the game, the Queen of Clubs is removed from the deck so that the Queen of Spades has no match. The fifty-one remaining cards are dealt to the players in a round robin. After the deal, all players match and discard as many cards as possible.

When no more matches can be made, play begins. In turn, each player picks a card (without looking) from the closest neighbor to the left who still has cards. If the chosen card matches a card in the player's hand, the pair is removed. Otherwise, the card is added to the player's hand. Eventually all possible matches are made, leaving only the Queen of Spades in the loser's hand.

In our computer simulation of the game, the computer plays all hands. Unfortunately, some nuances of the real game are lost. In a real game, the player with the Old Maid goes to some effort to get their neighbor to pick that card, by displaying it a little more prominently, or perhaps failing to display it more prominently, or even failing to fail to display that card more prominently. The computer simply picks a neighbor's card at random.

## 24.6. OldMaidHand class

A hand for playing Old Maid requires some abilities beyond the general abilities of a Hand. We will define a new class, OldMaidHand, that inherits from Hand and provides an additional method called remove\_matches:

```
1 class OldMaidHand(Hand):
2     def remove_matches(self):
3         count = 0
4         original_cards = self.cards[:]
5         for card in original_cards:
6             match = Card(3 - card.suit, card.rank)
7             if match in self.cards:
8                 self.cards.remove(card)
9                 self.cards.remove(match)
10                print("Hand {0}: {1} matches {2}"
11                     .format(self.name, card, match))
12                count += 1
13
14        return count
```

We start by making a copy of the list of cards, so that we can traverse the copy while removing cards from the original. Since self.cards is modified in the loop, we don't want to use it to control the traversal. Python can get quite confused if it is traversing a list that is changing!

For each card in the hand, we figure out what the matching card is and go looking for it. The match card has the same rank and the other suit of the same color. The expression 3 - card.suit turns a Club (suit 0) into a Spade (suit 3) and a Diamond (suit 1) into a Heart (suit 2). You should satisfy yourself that the opposite operations also work. If the match card is also in the hand, both cards are removed.

The following example demonstrates how to use remove\_matches:

```
1  >>> game = CardGame()
2  >>> hand = OldMaidHand("frank")
3  >>> game.deck.deal([hand], 13)
4  >>> print(hand)
5  Hand frank contains
6  Ace of Spades
7  2 of Diamonds
8  7 of Spades
9  8 of Clubs
10  6 of Hearts
11  8 of Spades
12  7 of Clubs
13  Queen of Clubs
14  7 of Diamonds
15  5 of Clubs
16  Jack of Diamonds
17  10 of Diamonds
18  10 of Hearts
19  >>> hand.remove_matches()
20  Hand frank: 7 of Spades matches 7 of Clubs
21  Hand frank: 8 of Spades matches 8 of Clubs
22  Hand frank: 10 of Diamonds matches 10 of Hearts
23  >>> print(hand)
24  Hand frank contains
25  Ace of Spades
26  2 of Diamonds
27  6 of Hearts
28  Queen of Clubs
29  7 of Diamonds
30  5 of Clubs
31  Jack of Diamonds
```

Notice that there is no `__init__` method for the `OldMaidHand` class. We inherit it from `Hand`.

## 24.7. `OldMaidGame` class

Now we can turn our attention to the game itself. `OldMaidGame` is a subclass of `CardGame` with a new method called `play` that takes a list of players as a parameter.

Since `__init__` is inherited from `CardGame`, a new `OldMaidGame` object contains a new shuffled deck:

```

1  class OldMaidGame(CardGame):
2      def play(self, names):
3          # Remove Queen of Clubs
4          self.deck.remove(Card(0,12))
5
6          # Make a hand for each player
7          self.hands = []
8          for name in names:
9              self.hands.append(OldMaidHand(name))
10
11         # Deal the cards
12         self.deck.deal(self.hands)
13         print("----- Cards have been dealt")
14         self.print_hands()
15
16         # Remove initial matches
17         matches = self.remove_all_matches()
18         print("----- Matches discarded, play begins")
19         self.print_hands()
20
21         # Play until all 50 cards are matched
22         turn = 0
23         num_hands = len(self.hands)
24         while matches < 25:
25             matches += self.play_one_turn(turn)
26             turn = (turn + 1) % num_hands
27
28         print("----- Game is Over")
29         self.print_hands()

```

The writing of `print_hands` has been left as an exercise.

Some of the steps of the game have been separated into methods. `remove_all_matches` traverses the list of hands and invokes `remove_matches` on each:

```

1  class OldMaidGame(CardGame):
2      ...
3      def remove_all_matches(self):
4          count = 0
5          for hand in self.hands:
6              count += hand.remove_matches()
7          return count

```

count is an accumulator that adds up the number of matches in each hand. When we've gone through every hand, the total is returned (count).

When the total number of matches reaches twenty-five, fifty cards have been removed from the hands, which means that only one card is left and the game is over.

The variable turn keeps track of which player's turn it is. It starts at 0 and increases by one each time; when it reaches num\_hands, the modulus operator wraps it back around to 0.

The method play\_one\_turn takes a parameter that indicates whose turn it is. The return value is the number of matches made during this turn:

```

1 class OldMaidGame(CardGame):
2     ...
3     def play_one_turn(self, i):
4         if self.hands[i].is_empty():
5             return 0
6         neighbor = self.find_neighbor(i)
7         picked_card = self.hands[neighbor].pop()
8         self.hands[i].add(picked_card)
9         print("Hand", self.hands[i].name, "picked", picked_card)
10        count = self.hands[i].remove_matches()
11        self.hands[i].shuffle()
12        return count

```

If a player's hand is empty, that player is out of the game, so he or she does nothing and returns 0.

Otherwise, a turn consists of finding the first player on the left that has cards, taking one card from the neighbor, and checking for matches. Before returning, the cards in the hand are shuffled so that the next player's choice is random.

The method find\_neighbor starts with the player to the immediate left and continues around the circle until it finds a player that still has cards:

```

1 class OldMaidGame(CardGame):
2     ...
3     def find_neighbor(self, i):
4         num_hands = len(self.hands)
5         for next in range(1, num_hands):
6             neighbor = (i + next) % num_hands
7             if not self.hands[neighbor].is_empty():
8                 return neighbor

```

If find\_neighbor ever went all the way around the circle without finding cards, it would return None and cause an error elsewhere in the program. Fortunately, we can prove that that will never happen (as long as the end of the game is detected correctly).

We have omitted the `print_hands` method. You can write that one yourself.

The following output is from a truncated form of the game where only the top fifteen cards (tens and higher) were dealt to three players. With this small deck, play stops after seven matches instead of twenty-five.

```
1  >>> import cards
2  >>> game = cards.OldMaidGame()
3  >>> game.play(["Allen", "Jeff", "Chris"])
4  ----- Cards have been dealt
5  Hand Allen contains
6    King of Hearts
7    Jack of Clubs
8    Queen of Spades
9    King of Spades
10   10 of Diamonds
11
12 Hand Jeff contains
13  Queen of Hearts
14  Jack of Spades
15  Jack of Hearts
16  King of Diamonds
17  Queen of Diamonds
18
19 Hand Chris contains
20  Jack of Diamonds
21  King of Clubs
22  10 of Spades
23  10 of Hearts
24  10 of Clubs
25
26 Hand Jeff: Queen of Hearts matches Queen of Diamonds
27 Hand Chris: 10 of Spades matches 10 of Clubs
28 ----- Matches discarded, play begins
29 Hand Allen contains
30  King of Hearts
31  Jack of Clubs
32  Queen of Spades
33  King of Spades
34  10 of Diamonds
35
36 Hand Jeff contains
37  Jack of Spades
```

```
38  Jack of Hearts
39  King of Diamonds
40
41 Hand Chris contains
42 Jack of Diamonds
43 King of Clubs
44 10 of Hearts
45
46 Hand Allen picked King of Diamonds
47 Hand Allen: King of Hearts matches King of Diamonds
48 Hand Jeff picked 10 of Hearts
49 Hand Chris picked Jack of Clubs
50 Hand Allen picked Jack of Hearts
51 Hand Jeff picked Jack of Diamonds
52 Hand Chris picked Queen of Spades
53 Hand Allen picked Jack of Diamonds
54 Hand Allen: Jack of Hearts matches Jack of Diamonds
55 Hand Jeff picked King of Clubs
56 Hand Chris picked King of Spades
57 Hand Allen picked 10 of Hearts
58 Hand Allen: 10 of Diamonds matches 10 of Hearts
59 Hand Jeff picked Queen of Spades
60 Hand Chris picked Jack of Spades
61 Hand Chris: Jack of Clubs matches Jack of Spades
62 Hand Jeff picked King of Spades
63 Hand Jeff: King of Clubs matches King of Spades
64 ----- Game is Over
65 Hand Allen is empty
66
67 Hand Jeff contains
68 Queen of Spades
69
70 Hand Chris is empty
```

So Jeff loses.

## 24.8. Glossary

```
1 inheritance
2     The ability to define a new class that is a modified version of a
3     previously defined class.
4
5 parent class
6     The class from which a child class inherits.
7
8 child class
9     A new class created by inheriting from an existing class; also called a
10    subclass.
```

## 24.9. Exercises

1. Add a method, `print_hands`, to the `OldMaidGame` class which traverses `self.hands` and prints each hand.
2. Define a new kind of Turtle, `TurtleGTX`, that comes with some extra features: it can jump forward a

given distance, and it has an odometer that keeps track of how far the turtle has travelled since it came off the production line. (The parent class has a number of synonyms like `fd`, `forward`, `back`, `backward`, and `bk`: for this exercise, just focus on putting this functionality into the `forward` method.) Think carefully about how to count the distance if the turtle is asked to move forward by a negative amount. (We would not want to buy a second-hand turtle whose odometer reading was faked because its previous owner drove it backwards around the block too often. Try this in a car near you, and see if the car's odometer counts up or down when you reverse.)

3. After travelling some random distance, your turtle should break down with a flat tyre. After this happens, raise an exception whenever `forward` is called. Also provide a `change_tyre` method that can fix the flat.

# Chapter 25: Linked lists

## 25.1. Embedded references

We have seen examples of attributes that refer to other objects, which we called **embedded references**. A common data structure, the **linked list**, takes advantage of this feature.

Linked lists are made up of **nodes**, where each node contains a reference to the next node in the list. In addition, each node contains a unit of data called the **cargo**.

A linked list is considered a **recursive data structure** because it has a recursive definition.

A linked list is either:

1. the empty list, represented by `None`, or
2. a node that contains a cargo object and a reference to a linked list.

Recursive data structures lend themselves to recursive methods.

## 25.2. The Node class

As usual when writing a new class, we'll start with the initialization and `__str__` methods so that we can test the basic mechanism of creating and displaying the new type:

```
1  class Node:  
2      def __init__(self, cargo=None, next=None):  
3          self.cargo = cargo  
4          self.next = next  
5  
6      def __str__(self):  
7          return str(self.cargo)
```

As usual, the parameters for the initialization method are optional. By default, both the cargo and the link, `next`, are set to `None`.

The string representation of a node is just the string representation of the cargo. Since any value can be passed to the `str` function, we can store any value in a list.

To test the implementation so far, we can create a `Node` and print it:

```

1 >>> node = Node("test")
2 >>> print(node)
3 test

```

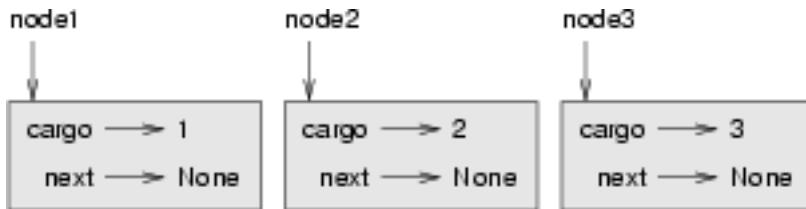
To make it interesting, we need a list with more than one node:

```

1 >>> node1 = Node(1)
2 >>> node2 = Node(2)
3 >>> node3 = Node(3)

```

This code creates three nodes, but we don't have a list yet because the nodes are not **linked**. The state diagram looks like this:



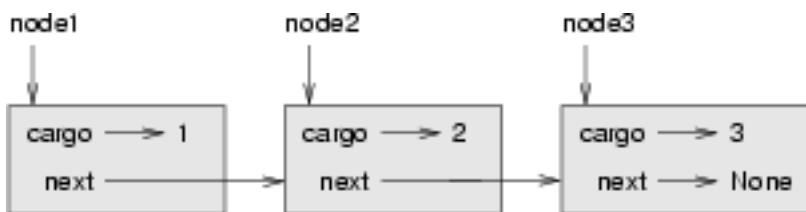
To link the nodes, we have to make the first node refer to the second and the second node refer to the third:

```

1 >>> node1.next = node2
2 >>> node2.next = node3

```

The reference of the third node is `None`, which indicates that it is the end of the list. Now the state diagram looks like this:



Now you know how to create nodes and link them into lists. What might be less clear at this point is why.

## 25.3. Lists as collections

Lists are useful because they provide a way to assemble multiple objects into a single entity, sometimes called a **collection**. In the example, the first node of the list serves as a reference to the entire list.

To pass the list as a parameter, we only have to pass a reference to the first node. For example, the function `print_list` takes a single node as an argument. Starting with the head of the list, it prints each node until it gets to the end:

```

1 def print_list(node):
2     while node is not None:
3         print(node, end=" ")
4         node = node.next
5     print()

```

To invoke this method, we pass a reference to the first node:

```

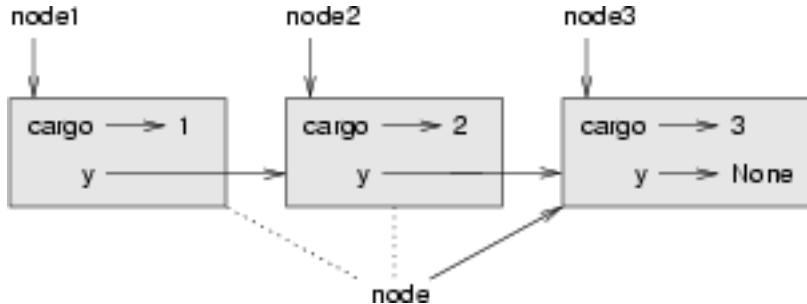
1 >>> print_list(node1)
2 1 2 3

```

Inside `print_list` we have a reference to the first node of the list, but there is no variable that refers to the other nodes. We have to use the `next` value from each node to get to the next node.

To traverse a linked list, it is common to use a loop variable like `node` to refer to each of the nodes in succession.

This diagram shows the value of `list` and the values that `node` takes on:



## 25.4. Lists and recursion

It is natural to express many list operations using recursive methods. For example, the following is a recursive algorithm for printing a list backwards:

1. Separate the list into two pieces: the first node (called the head); and the rest (called the tail).
2. Print the tail backward.
3. Print the head.

Of course, Step 2, the recursive call, assumes that we have a way of printing a list backward. But if we assume that the recursive call works — the leap of faith — then we can convince ourselves that this algorithm works.

All we need are a base case and a way of proving that for any list, we will eventually get to the base case. Given the recursive definition of a list, a natural base case is the empty list, represented by `None`:

```
1 def print_backward(list):
2     if list is None: return
3     head = list
4     tail = list.next
5     print_backward(tail)
6     print(head, end=" ")
```

The first line handles the base case by doing nothing. The next two lines split the list into `head` and `tail`. The last two lines print the list. The `end` argument of the `print` statement keeps Python from printing a newline after each node.

We invoke this method as we invoked `print_list`:

```
1 >>> print_backward(node1)
2 3 2 1
```

The result is a backward list.

You might wonder why `print_list` and `print_backward` are functions and not methods in the `Node` class. The reason is that we want to use `None` to represent the empty list and it is not legal to invoke a method on `None`. This limitation makes it awkward to write list – manipulating code in a clean object-oriented style.

Can we prove that `print_backward` will always terminate? In other words, will it always reach the base case? In fact, the answer is no. Some lists will make this method crash.

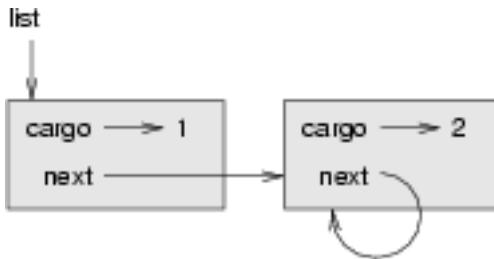
Revisit the Recursion chapter

In our earlier chapter on recursion we distinguished between the high-level view that requires a leap of faith, and the low-level operational view. In terms of mental chunking, we want to encourage the more abstract high-level view.

But if you'd like to see the detail you should use your single-stepping debugging tools to step into the recursive levels and to examine the execution stack frames at every call to `print_backward`.

## 25.5. Infinite lists

There is nothing to prevent a node from referring back to an earlier node in the list, including itself. For example, this figure shows a list with two nodes, one of which refers to itself:



If we invoke `print_list` on this list, it will loop forever. If we invoke `print_backward`, it will recurse infinitely. This sort of behavior makes infinite lists difficult to work with.

Nevertheless, they are occasionally useful. For example, we might represent a number as a list of digits and use an infinite list to represent a repeating fraction.

Regardless, it is problematic that we cannot prove that `print_list` and `print_backward` terminate. The best we can do is the hypothetical statement, “If the list contains no loops, then these methods will terminate.” This sort of claim is called a **precondition**. It imposes a constraint on one of the parameters and describes the behavior of the method if the constraint is satisfied. You will see more examples soon.

## 25.6. The fundamental ambiguity theorem

One part of `print_backward` might have raised an eyebrow:

```
1 head = list
2 tail = list.next
```

After the first assignment, `head` and `list` have the same type and the same value. So why did we create a new variable?

The reason is that the two variables play different roles. We think of `head` as a reference to a single node, and we think of `list` as a reference to the first node of a list. These roles are not part of the program; they are in the mind of the programmer.

In general we can't tell by looking at a program what role a variable plays. This ambiguity can be useful, but it can also make programs difficult to read. We often use variable names like `node` and `list` to document how we intend to use a variable and sometimes create additional variables to disambiguate.

We could have written `print_backward` without `head` and `tail`, which makes it more concise but possibly less clear:

```

1 def print_backward(list):
2     if list is None: return
3     print_backward(list.next)
4     print(list, end=" ")

```

Looking at the two function calls, we have to remember that `print_backward` treats its argument as a collection and `print` treats its argument as a single object.

The **fundamental ambiguity theorem** describes the ambiguity that is inherent in a reference to a node: *A variable that refers to a node might treat the node as a single object or as the first in a list of nodes.*

## 25.7. Modifying lists

There are two ways to modify a linked list. Obviously, we can change the cargo of one of the nodes, but the more interesting operations are the ones that add, remove, or reorder the nodes.

As an example, let's write a method that removes the second node in the list and returns a reference to the removed node:

```

1 def remove_second(list):
2     if list is None: return
3     first = list
4     second = list.next
5     # Make the first node refer to the third
6     first.next = second.next
7     # Separate the second node from the rest of the list
8     second.next = None
9     return second

```

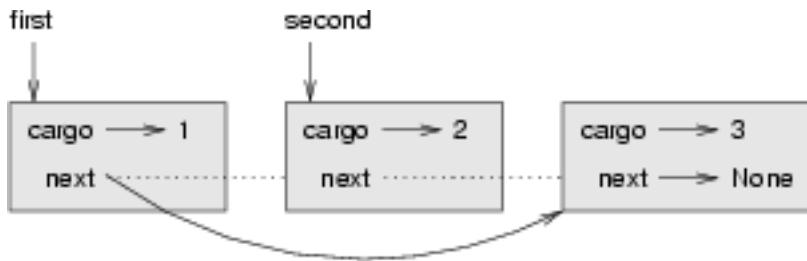
Again, we are using temporary variables to make the code more readable. Here is how to use this method:

```

1 >>> print_list(node1)
2 1 2 3
3 >>> removed = remove_second(node1)
4 >>> print_list(removed)
5 2
6 >>> print_list(node1)
7 1 3

```

This state diagram shows the effect of the operation:



What happens if you invoke this method and pass a list with only one element (a **singleton**)? What happens if you pass the empty list as an argument? Is there a precondition for this method? If so, fix the method to handle a violation of the precondition in a reasonable way.

## 25.8. Wrappers and helpers

It is often useful to divide a list operation into two methods. For example, to print a list backward in the conventional list format [3, 2, 1] we can use the `print_backward` method to print 3, 2, but we need a separate method to print the brackets and the first node. Let's call it `print_backward_nicely`:

```

1 def print_backward_nicely(list):
2     print("[", end=" ")
3     print_backward(list)
4     print("]")
  
```

Again, it is a good idea to check methods like this to see if they work with special cases like an empty list or a singleton.

When we use this method elsewhere in the program, we invoke `print_backward_nicely` directly, and it invokes `print_backward` on our behalf. In that sense, `print_backward_nicely` acts as a **wrapper**, and it uses `print_backward` as a **helper**.

## 25.9. The `LinkedList` class

There are some subtle problems with the way we have been implementing lists. In a reversal of cause and effect, we'll propose an alternative implementation first and then explain what problems it solves.

First, we'll create a new class called `LinkedList`. Its attributes are an integer that contains the length of the list and a reference to the first node. `LinkedList` objects serve as handles for manipulating lists of `Node` objects:

```
1 class LinkedList:  
2     def __init__(self):  
3         self.length = 0  
4         self.head = None
```

One nice thing about the `LinkedList` class is that it provides a natural place to put wrapper functions like `print_backward_nicely`, which we can make a method of the `LinkedList` class:

```
1 class LinkedList:  
2     ...  
3     def print_backward(self):  
4         print("[", end=" ")  
5         if self.head is not None:  
6             self.head.print_backward()  
7  
8 class Node:  
9     ...  
10    def print_backward(self):  
11        if self.next is not None:  
12            tail = self.next  
13            tail.print_backward()  
14        print(self.cargo, end=" ")
```

Just to make things confusing, we renamed `print_backward_nicely`. Now there are two methods named `print_backward`: one in the `Node` class (the helper); and one in the `LinkedList` class (the wrapper). When the wrapper invokes `self.head.print_backward`, it is invoking the helper, because `self.head` is a `Node` object.

Another benefit of the `LinkedList` class is that it makes it easier to add or remove the first element of a list. For example, `add_first` is a method for `LinkedLists`; it takes an item of cargo as an argument and puts it at the beginning of the list:

```
1 class LinkedList:  
2     ...  
3     def add_first(self, cargo):  
4         node = Node(cargo)  
5         node.next = self.head  
6         self.head = node  
7         self.length += 1
```

As usual, you should check code like this to see if it handles the special cases. For example, what happens if the list is initially empty?

## 25.10. Invariants

Some lists are well formed; others are not. For example, if a list contains a loop, it will cause many of our methods to crash, so we might want to require that lists contain no loops. Another requirement is that the `length` value in the `LinkedList` object should be equal to the actual number of nodes in the list.

Requirements like these are called **invariants** because, ideally, they should be true of every object all the time. Specifying invariants for objects is a useful programming practice because it makes it easier to prove the correctness of code, check the integrity of data structures, and detect errors.

One thing that is sometimes confusing about invariants is that there are times when they are violated. For example, in the middle of `add_first`, after we have added the node but before we have incremented `length`, the invariant is violated. This kind of violation is acceptable; in fact, it is often impossible to modify an object without violating an invariant for at least a little while. Normally, we require that every method that violates an invariant must restore the invariant.

If there is any significant stretch of code in which the invariant is violated, it is important for the comments to make that clear, so that no operations are performed that depend on the invariant.

## 25.11. Glossary

- 1 **embedded reference**  
2     A reference stored **in** an attribute of an object.
- 3
- 4 **linked list**  
5     A data structure that implements a collection using a sequence of linked  
6     nodes.
- 7
- 8 **node**  
9     An element of a list, usually implemented **as** an object that contains a  
10    reference to another object of the same type.
- 11
- 12 **cargo**  
13    An item of data contained **in** a node.
- 14
- 15 **link**  
16    An embedded reference used to link one object to another.
- 17
- 18 **precondition**  
19    An assertion that must be **true in** order **for** a method to work correctly.
- 20
- 21 **fundamental ambiguity theorem**

```
22     A reference to a list node can be treated as a single object or as the
23     first in a list of nodes.
24
25 singleton
26     A linked list with a single node.
27
28 wrapper
29     A method that acts as a middleman between a caller and a helper method,
30     often making the method easier or less error-prone to invoke.
31
32 helper
33     A method that is not invoked directly by a caller but is used by another
34     method to perform part of an operation.
35
36 invariant
37     An assertion that should be true of an object at all times (except
38     perhaps while the object is being modified).
```

## 25.12. Exercises

1. By convention, lists are often printed in brackets with commas between the elements, as in [1, 2, 3].

Modify `print_list` so that it generates output in this format.

# Chapter 26: Stacks

## 26.1. Abstract data types

The data types you have seen so far are all concrete, in the sense that we have completely specified how they are implemented. For example, the `Card` class represents a card using two integers. As we discussed at the time, that is not the only way to represent a card; there are many alternative implementations.

An **abstract data type**, or ADT, specifies a set of operations (or methods) and the semantics of the operations (what they do), but it does not specify the implementation of the operations. That's what makes it abstract.

Why is that useful?

1. It simplifies the task of specifying an algorithm if you can denote the operations you need without having to think at the same time about how the operations are performed.
2. Since there are usually many ways to implement an ADT, it might be useful to write an algorithm that can  
be used with any of the possible implementations.
3. Well-known ADTs, such as the Stack ADT in this chapter, are often implemented in standard libraries so

they can be written once and used by many programmers.

4. The operations on ADTs provide a common high-level language for specifying and talking about algorithms.

When we talk about ADTs, we often distinguish between the code that uses the ADT, called the **client** code, from the code that implements the ADT, called the **provider/implementor** code.

## 26.2. The Stack ADT

In this chapter, we will look at one common ADT, the **stack**. A stack is a collection, meaning that it is a data structure that contains multiple elements. Other collections we have seen include dictionaries and lists.

An ADT is defined by the operations that can be performed on it, which is called an **interface**. The interface for a stack consists of these operations:

`__init__`

Initialize a new empty stack.

push

Add a new item to the stack.

pop

Remove and return an item from the stack. The item that is returned is always the last one that was added.

is\_empty

Check whether the stack is empty.

A stack is sometimes called a “Last in, First out” or **LIFO** data structure, because the last item added is the first to be removed.

## 26.3. Implementing stacks with Python lists

The list operations that Python provides are similar to the operations that define a stack. The interface isn’t exactly what it is supposed to be, but we can write code to translate from the Stack ADT to the built-in operations.

This code is called an **implementation** of the Stack ADT. In general, an implementation is a set of methods that satisfy the syntactic and semantic requirements of an interface.

Here is an implementation of the Stack ADT that uses a Python list:

```
1 class Stack:
2     def __init__(self):
3         self.items = []
4
5     def push(self, item):
6         self.items.append(item)
7
8     def pop(self):
9         return self.items.pop()
10
11    def is_empty(self):
12        return (self.items == [])
```

A Stack object contains an attribute named `items` that is a list of items in the stack. The initialization method sets `items` to the empty list.

To push a new item onto the stack, `push` appends it onto `items`. To pop an item off the stack, `pop` uses the homonymous (*same-named*) list method to remove and return the last item on the list.

Finally, to check if the stack is empty, `is_empty` compares `items` to the empty list.

An implementation like this, in which the methods consist of simple invocations of existing methods, is called a **veneer**. In real life, veneer is a thin coating of good quality wood used in furniture-making

to hide lower quality wood underneath. Computer scientists use this metaphor to describe a small piece of code that hides the details of an implementation and provides a simpler, or more standard, interface.

## 26.4. Pushing and popping

A stack is a **generic data structure**, which means that we can add any type of item to it. The following example pushes two integers and a string onto the stack:

```
1 >>> s = Stack()
2 >>> s.push(54)
3 >>> s.push(45)
4 >>> s.push("+" )
```

We can use `is_empty` and `pop` to remove and print all of the items on the stack:

```
1 while not s.is_empty():
2     print(s.pop(), end=" ")
```

The output is `+ 45 54`. In other words, we just used a stack to print the items backward! Granted, it's not the standard format for printing a list, but by using a stack, it was remarkably easy to do.

You should compare this bit of code to the implementation of `print_backward` in the last chapter. There is a natural parallel between the recursive version of `print_backward` and the stack algorithm here. The difference is that `print_backward` uses the runtime stack to keep track of the nodes while it traverses the list, and then prints them on the way back from the recursion. The stack algorithm does the same thing, except that it uses a `Stack` object instead of the runtime stack.

## 26.5. Using a stack to evaluate postfix

In most programming languages, mathematical expressions are written with the operator between the two operands, as in `1 + 2`. This format is called **infix**. An alternative used by some calculators is called **postfix**. In postfix, the operator follows the operands, as in `1 2 +`.

The reason postfix is sometimes useful is that there is a natural way to evaluate a postfix expression using a stack:

1. Starting at the beginning of the expression, get one term (operator or operand) at a time.
  - If the term is an operand, push it on the stack.
  - If the term is an operator, pop two operands off the stack, perform the operation on them, and push the result back on the stack.

2. When you get to the end of the expression, there should be exactly one operand left on the stack. That operand is the result.

## 26.6. Parsing

To implement the previous algorithm, we need to be able to traverse a string and break it into operands and operators. This process is an example of **parsing**, and the results — the individual chunks of the string — are called **tokens**. You might remember these words from Chapter 1.

Python provides a `split` method in both string objects and the `re` (regular expression) module. A string's `split` method splits it into a list using a single character as a **delimiter**. For example:

```
1 >>> "Now is the time".split(" ")
2 ['Now', 'is', 'the', 'time']
```

In this case, the delimiter is the space character, so the string is split at each space.

The function `re.split` is more powerful, allowing us to provide a regular expression instead of a delimiter. A regular expression is a way of specifying a set of strings. For example, `[A-z]` is the set of all letters and `[0-9]` is the set of all numbers. The `^` operator negates a set, so `[^0-9]` is the set of everything that is not a number, which is exactly the set we want to use to split up postfix expressions:

```
1 >>> import re
2 >>> re.split("[^0-9]", "123+456*/")
3 ['123', '+', '456', '*', '', '/', '']
```

The resulting list includes the operands 123 and 456 and the operators \* and /. It also includes two empty strings that are inserted after the operands.

## 26.7. Evaluating postfix

To evaluate a postfix expression, we will use the parser from the previous section and the algorithm from the section before that. To keep things simple, we'll start with an evaluator that only implements the operators + and \*:

```

1 def eval_postfix(expr):
2     import re
3     token_list = re.split("([^\d])", expr)
4     stack = Stack()
5     for token in token_list:
6         if token == "" or token == " ":
7             continue
8         if token == "+":
9             sum = stack.pop() + stack.pop()
10            stack.push(sum)
11        elif token == "*":
12            product = stack.pop() * stack.pop()
13            stack.push(product)
14        else:
15            stack.push(int(token))
16    return stack.pop()

```

The first condition takes care of spaces and empty strings. The next two conditions handle operators. We assume, for now, that anything else must be an operand. Of course, it would be better to check for erroneous input and report an error message, but we'll get to that later.

Let's test it by evaluating the postfix form of  $(56 + 47) * 2$ :

```

1 >> eval_postfix("56 47 + 2 *")
2 206

```

That's close enough.

## 26.8. Clients and providers

One of the fundamental goals of an ADT is to separate the interests of the provider, who writes the code that implements the ADT, and the client, who uses the ADT. The provider only has to worry about whether the implementation is correct — in accord with the specification of the ADT — and not how it will be used.

Conversely, the client *assumes* that the implementation of the ADT is correct and doesn't worry about the details. When you are using one of Python's built-in types, you have the luxury of thinking exclusively as a client.

Of course, when you implement an ADT, you also have to write client code to test it. In that case, you play both roles, which can be confusing. You should make some effort to keep track of which role you are playing at any moment.

## 26.9. Glossary

```
1 abstract data type (ADT)
2     A data type (usually a collection of objects) that is defined by a set
3     of operations but that can be implemented in a variety of ways.
4
5 client
6     A program (or the person who wrote it) that uses an ADT.
7
8 delimiter
9     A character that is used to separate tokens, such as punctuation in a
10    natural language.
11
12 generic data structure
13    A kind of data structure that can contain data of any type.
14
15 implementation
16    Code that satisfies the syntactic and semantic requirements of an
17    interface.
18
19 interface
20    The set of operations that define an ADT.
21
22 infix
23    A way of writing mathematical expressions with the operators between the
24    operands.
25
26 parse
27    To read a string of characters or tokens and analyze its grammatical structure.
28
29 postfix
30    A way of writing mathematical expressions with the operators after the
31    operands.
32
33 provider
34    The code (or the person who wrote it) that implements an ADT.
35
36 token
37    A set of characters that are treated as a unit for purposes of parsing,
38    such as the words in a natural language.
39
40 veneer
```

41       A **class** definition that implements an ADT with method definitions that  
42       are invocations of other methods, sometimes with simple transformations.  
43       The veneer does no significant work, but it improves **or** standardizes the  
44       interface seen by the client.

## 26.10. Exercises

1. Apply the postfix algorithm to the expression  $1\ 2\ +\ 3\ *.$ This example demonstrates one of the advantages

of postfix—there is no need to use parentheses to control the order of operations. To get the same result in infix, we would have to write  $(1 + 2) * 3.$

2. Write a postfix expression that is equivalent to  $1 + 2 * 3.$

# Chapter 27: Queues

This chapter presents two ADTs: the Queue and the Priority Queue. In real life, a **queue** is a line of people waiting for something. In most cases, the first person in line is the next one to be served. There are exceptions, though. At airports, peoples whose flights are leaving soon are sometimes taken from the middle of the queue. At supermarkets, a polite person might let someone with only a few items go in front of them.

The rule that determines who goes next is called the **queueing policy**. The simplest queueing policy is called “First in, First out”, FIFO for short. The most general queueing policy is **priority queueing**, in which each person is assigned a priority and the person with the highest priority goes first, regardless of the order of arrival. We say this is the most general policy because the priority can be based on anything: what time a flight leaves; how many groceries the person has; or how important the person is. Of course, not all queueing policies are fair, but fairness is in the eye of the beholder.

The Queue ADT and the Priority Queue ADT have the same set of operations. The difference is in the semantics of the operations: a queue uses the FIFO policy; and a priority queue (as the name suggests) uses the priority queueing policy.

## 27.1. The Queue ADT

The Queue ADT is defined by the following operations:

`__init__`

Initialize a new empty queue.

`insert`

Add a new item to the queue.

`remove`

Remove and return an item from the queue. The item that is returned is the first one that was added.

`is_empty`

Check whether the queue is empty.

## 27.2. Linked Queue

The first implementation of the Queue ADT we will look at is called a **linked queue** because it is made up of linked `Node` objects. Here is the class definition:

```
1 class Queue:
2     def __init__(self):
3         self.length = 0
4         self.head = None
5
6     def is_empty(self):
7         return self.length == 0
8
9     def insert(self, cargo):
10        node = Node(cargo)
11        if self.head is None:
12            # If list is empty the new node goes first
13            self.head = node
14        else:
15            # Find the last node in the list
16            last = self.head
17            while last.next:
18                last = last.next
19            # Append the new node
20            last.next = node
21            self.length += 1
22
23     def remove(self):
24        cargo = self.head.cargo
25        self.head = self.head.next
26        self.length -= 1
27        return cargo
```

The methods `is_empty` and `remove` are identical to the `LinkedList` methods `is_empty` and `remove_first`. The `insert` method is new and a bit more complicated.

We want to insert new items at the end of the list. If the queue is empty, we just set `head` to refer to the new node.

Otherwise, we traverse the list to the last node and tack the new node on the end. We can identify the last node because its `next` attribute is `None`.

There are two invariants for a properly formed `Queue` object. The value of `length` should be the number of nodes in the queue, and the last node should have `next` equal to `None`. Convince yourself that this method preserves both invariants.

## 27.3. Performance characteristics

Normally when we invoke a method, we are not concerned with the details of its implementation. But there is one detail we might want to know: the performance characteristics of the method. How long does it take, and how does the run time change as the number of items in the collection increases?

First look at `remove`. There are no loops or function calls here, suggesting that the runtime of this method is the same every time. Such a method is called a **constant time** operation. In reality, the method might be slightly faster when the list is empty since it skips the body of the conditional, but that difference is not significant.

The performance of `insert` is very different. In the general case, we have to traverse the list to find the last element.

This traversal takes time proportional to the length of the list. Since the runtime is a linear function of the length, this method is called **linear time**. Compared to constant time, that's very bad.

## 27.4. Improved Linked Queue

We would like an implementation of the Queue ADT that can perform all operations in constant time. One way to do that is to modify the Queue class so that it maintains a reference to both the first and the last node, as shown in the figure:

The `ImprovedQueue` implementation looks like this:

```
1 class ImprovedQueue:
2     def __init__(self):
3         self.length = 0
4         self.head = None
5         self.last = None
6
7     def is_empty(self):
8         return self.length == 0
```

So far, the only change is the attribute `last`. It is used in `insert` and `remove` methods:

```

1 class ImprovedQueue:
2     ...
3     def insert(self, cargo):
4         node = Node(cargo)
5         if self.length == 0:
6             # If list is empty, the new node is head and last
7             self.head = self.last = node
8         else:
9             # Find the last node
10            last = self.last
11            # Append the new node
12            last.next = node
13            self.last = node
14        self.length += 1

```

Since `last` keeps track of the last node, we don't have to search for it. As a result, this method is constant time.

There is a price to pay for that speed. We have to add a special case to `remove` to set `last` to `None` when the last node is removed:

```

1 class ImprovedQueue:
2     ...
3     def remove(self):
4         cargo = self.head.cargo
5         self.head = self.head.next
6         self.length -= 1
7         if self.length == 0:
8             self.last = None
9         return cargo

```

This implementation is more complicated than the Linked Queue implementation, and it is more difficult to demonstrate that it is correct. The advantage is that we have achieved the goal — both `insert` and `remove` are constant time operations.

## 27.5. Priority queue

The Priority Queue ADT has the same interface as the Queue ADT, but different semantics. Again, the interface is:

`__init__`

Initialize a new empty queue.

`insert`

Add a new item to the queue.

`remove`

Remove and return an item from the queue. The item that is returned is the one with the highest priority.

`is_empty`

Check whether the queue is empty.

The semantic difference is that the item that is removed from the queue is not necessarily the first one that was added. Rather, it is the item in the queue that has the highest priority. What the priorities are and how they compare to each other are not specified by the Priority Queue implementation. It depends on which items are in the queue.

For example, if the items in the queue have names, we might choose them in alphabetical order. If they are bowling scores, we might go from highest to lowest, but if they are golf scores, we would go from lowest to highest. As long as we can compare the items in the queue, we can find and remove the one with the highest priority.

This implementation of Priority Queue has as an attribute a Python list that contains the items in the queue.

```

1 class PriorityQueue:
2     def __init__(self):
3         self.items = []
4
5     def is_empty(self):
6         return not self.items
7
8     def insert(self, item):
9         self.items.append(item)

```

The initialization method, `is_empty`, and `insert` are all veneers on list operations. The only interesting method is `remove`:

```

1 class PriorityQueue:
2     ...
3     def remove(self):
4         maxi = 0
5         for i in range(1, len(self.items)):
6             if self.items[i] > self.items[maxi]:
7                 maxi = i
8         item = self.items[maxi]
9         del self.items[maxi]
10        return item

```

At the beginning of each iteration, `maxi` holds the index of the biggest item (highest priority) we have seen *so far*. Each time through the loop, the program compares the *i*'th item to the champion. If the new item is bigger, the value of `maxi` is set to *i*.

When the `for` statement completes, `maxi` is the index of the biggest item. This item is removed from the list and returned.

Let's test the implementation:

```

1  >>> q = PriorityQueue()
2  >>> for num in [11, 12, 14, 13]:
3  ...     q.insert(num)
4  ...
5  >>> while not q.is_empty():
6  ...     print(q.remove())
7  ...
8  14
9  13
10 12
11 11

```

If the queue contains simple numbers or strings, they are removed in numerical or alphabetical order, from highest to lowest. Python can find the biggest integer or string because it can compare them using the built-in comparison operators.

If the queue contains an object type, it has to provide a `__gt__` method. When `remove` uses the `>` operator to compare items, it invokes the `__gt__` method for one of the items and passes the other as a parameter. As long as the `__gt__` method works correctly, the Priority Queue will work.

## 27.6. The Golfer class

As an example of an object with an unusual definition of priority, let's implement a class called `Golfer` that keeps track of the names and scores of golfers. As usual, we start by defining `__init__` and `__str__`:

```

1  class Golfer:
2      def __init__(self, name, score):
3          self.name = name
4          self.score= score
5
6      def __str__(self):
7          return "{0:16}: {1}".format(self.name, self.score)

```

`__str__` uses the `format` method to put the names and scores in neat columns.

Next we define a version of `__gt__` where the lowest score gets highest priority. As always, `__gt__` returns `True` if `self` is greater than `other`, and `False` otherwise.

```
1 class Golfer:
2     ...
3     def __gt__(self, other):
4         return self.score < other.score      # Less is more
```

Now we are ready to test the priority queue with the `Golfer` class:

```
1 >>> tiger = Golfer("Tiger Woods", 61)
2 >>> phil = Golfer("Phil Mickelson", 72)
3 >>> hal = Golfer("Hal Sutton", 69)
4 >>>
5 >>> pq = PriorityQueue()
6 >>> for g in [tiger, phil, hal]:
7     ...     pq.insert(g)
8 ...
9 >>> while not pq.is_empty():
10    ...     print(pq.remove())
11 ...
12 Tiger Woods      : 61
13 Hal Sutton       : 69
14 Phil Mickelson  : 72
```

## 27.7. Glossary

```
1 constant time
2     An operation whose runtime does not depend on the size of the data
3     structure.
4
5 FIFO (First In, First Out)
6     a queueing policy in which the first member to arrive is the first to be
7     removed.
8
9 linear time
10    An operation whose runtime is a linear function of the size of the data
11    structure.
12
13 linked queue
```

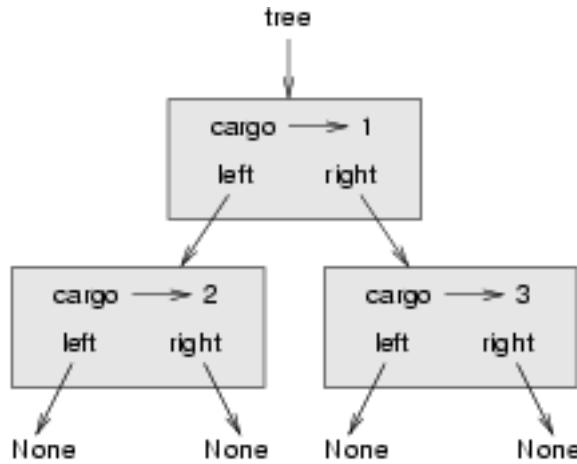
```
14     An implementation of a queue using a linked list.  
15  
16 priority queue  
17     A queueing policy in which each member has a priority determined by  
18     external factors. The member with the highest priority is the first to  
19     be removed.  
20  
21 Priority Queue  
22     An ADT that defines the operations one might perform on a priority  
23     queue.  
24  
25 queue  
26     An ordered set of objects waiting for a service of some kind.  
27  
28 Queue  
29     An ADT that performs the operations one might perform on a queue.  
30  
31 queueing policy  
32     The rules that determine which member of a queue is removed next.
```

## 27.8. Exercises

1. Write an implementation of the Queue ADT using a Python list. Compare the performance of this implementation to the ImprovedQueue for a range of queue lengths.
2. Write an implementation of the Priority Queue ADT using a linked list. You should keep the list sorted so that removal is a constant time operation. Compare the performance of this implementation with the Python list implementation.

# Chapter 28: Trees

Like linked lists, trees are made up of nodes. A common kind of tree is a **binary tree**, in which each node contains a reference to two other nodes (possibly `None`). These references are referred to as the left and right subtrees. Like list nodes, tree nodes also contain cargo. A state diagram for a tree looks like this:



To avoid cluttering up the picture, we often omit the `Nones`.

The top of the tree (the node `tree` refers to) is called the **root**. In keeping with the tree metaphor, the other nodes are called branches and the nodes at the tips with null references are called **leaves**. It may seem odd that we draw the picture with the root at the top and the leaves at the bottom, but that is not the strangest thing.

To make things worse, computer scientists mix in another metaphor: the family tree. The top node is sometimes called a **parent** and the nodes it refers to are its **children**. Nodes with the same parent are called **siblings**.

Finally, there is a geometric vocabulary for talking about trees. We already mentioned left and right, but there is also up (toward the parent/root) and down (toward the children/leaves). Also, all of the nodes that are the same distance from the root comprise a **level** of the tree.

We probably don't need three metaphors for talking about trees, but there they are.

Like linked lists, trees are recursive data structures because they are defined recursively. A tree is either:

1. the empty tree, represented by `None`, or
2. a node that contains an object reference (cargo) and two tree references.

## 28.1. Building trees

The process of assembling a tree is similar to the process of assembling a linked list. Each constructor invocation builds a single node.

```
1 class Tree:
2     def __init__(self, cargo, left=None, right=None):
3         self.cargo = cargo
4         self.left = left
5         self.right = right
6
7     def __str__(self):
8         return str(self.cargo)
```

The `cargo` can be any type, but the `left` and `right` parameters should be tree nodes. `left` and `right` are optional; the default value is `None`.

To print a node, we just print the cargo.

One way to build a tree is from the bottom up. Allocate the child nodes first:

```
1 left = Tree(2)
2 right = Tree(3)
```

Then create the parent node and link it to the children:

```
1 tree = Tree(1, left, right)
```

We can write this code more concisely by nesting constructor invocations:

```
1 >>> tree = Tree(1, Tree(2), Tree(3))
```

Either way, the result is the tree at the beginning of the chapter.

## 28.2. Traversing trees

Any time you see a new data structure, your first question should be, “How do I traverse it?” The most natural way to traverse a tree is recursively. For example, if the tree contains integers as cargo, this function returns their sum:

```

1 def total(tree):
2     if tree is None: return 0
3     return total(tree.left) + total(tree.right) + tree.cargo

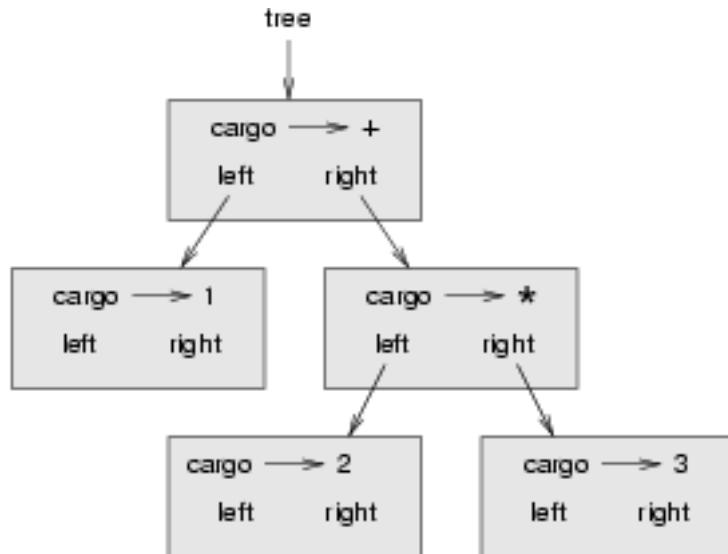
```

The base case is the empty tree, which contains no cargo, so the sum is 0. The recursive step makes two recursive calls to find the sum of the child trees. When the recursive calls complete, we add the cargo of the parent and return the total.

## 28.3. Expression trees

A tree is a natural way to represent the structure of an expression. Unlike other notations, it can represent the computation unambiguously. For example, the infix expression  $1 + 2 * 3$  is ambiguous unless we know that the multiplication happens before the addition.

This expression tree represents the same computation:



The nodes of an expression tree can be operands like 1 and 2 or operators like + and \*. Operands are leaf nodes; operator nodes contain references to their operands. (All of these operators are **binary**, meaning they have exactly two operands.)

We can build this tree like this:

```

1 >>> tree = Tree("+", Tree(1), Tree("*", Tree(2), Tree(3)))

```

Looking at the figure, there is no question what the order of operations is; the multiplication happens first in order to compute the second operand of the addition.

Expression trees have many uses. The example in this chapter uses trees to translate expressions to postfix, prefix, and infix. Similar trees are used inside compilers to parse, optimize, and translate programs.

## 28.4. Tree traversal

We can traverse an expression tree and print the contents like this:

```

1 def print_tree(tree):
2     if tree is None: return
3     print(tree.cargo, end=" ")
4     print_tree(tree.left)
5     print_tree(tree.right)

```

In other words, to print a tree, first print the contents of the root, then print the entire left subtree, and then print the entire right subtree. This way of traversing a tree is called a **preorder**, because the contents of the root appear *before* the contents of the children. For the previous example, the output is:

```

1 >>> tree = Tree("+", Tree(1), Tree("*", Tree(2), Tree(3)))
2 >>> print_tree(tree)
3 + 1 * 2 3

```

This format is different from both postfix and infix; it is another notation called **prefix**, in which the operators appear before their operands.

You might suspect that if you traverse the tree in a different order, you will get the expression in a different notation. For example, if you print the subtrees first and then the root node, you get:

```

1 def print_tree_postorder(tree):
2     if tree is None: return
3     print_tree_postorder(tree.left)
4     print_tree_postorder(tree.right)
5     print(tree.cargo, end=" ")

```

The result, 1 2 3 \* +, is in postfix! This order of traversal is called **postorder**.

Finally, to traverse a tree **inorder**, you print the left tree, then the root, and then the right tree:

```

1 def print_tree_inorder(tree):
2     if tree is None: return
3     print_tree_inorder(tree.left)
4     print(tree.cargo, end=" ")
5     print_tree_inorder(tree.right)

```

The result is  $1 + 2 * 3$ , which is the expression in infix.

To be fair, we should point out that we have omitted an important complication. Sometimes when we write an expression in infix, we have to use parentheses to preserve the order of operations. So an inorder traversal is not quite sufficient to generate an infix expression.

Nevertheless, with a few improvements, the expression tree and the three recursive traversals provide a general way to translate expressions from one format to another.

If we do an inorder traversal and keep track of what level in the tree we are on, we can generate a graphical representation of a tree:

```

1 def print_treeIndented(tree, level=0):
2     if tree is None: return
3     print_treeIndented(tree.right, level+1)
4     print(" " * level + str(tree.cargo))
5     print_treeIndented(tree.left, level+1)

```

The parameter `level` keeps track of where we are in the tree. By default, it is initially 0. Each time we make a recursive call, we pass `level+1` because the child's level is always one greater than the parent's. Each item is indented by two spaces per level. The result for the example tree is:

```

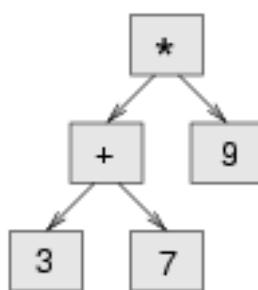
1 >>> print_treeIndented(tree)
2     3
3     *
4     2
5     +
6     1

```

If you look at the output sideways, you see a simplified version of the original figure.

## 28.5. Building an expression tree

In this section, we parse infix expressions and build the corresponding expression trees. For example, the expression  $(3 + 7) * 9$  yields the following tree:



Notice that we have simplified the diagram by leaving out the names of the attributes.

The parser we will write handles expressions that include numbers, parentheses, and the operators + and \*. We assume that the input string has already been tokenized into a Python list (producing this list is left as an exercise). The token list for  $(3 + 7) * 9$  is:

```
1 ["(", 3, "+", 7, ")", "*", 9, "end"]
```

The end token is useful for preventing the parser from reading past the end of the list.

The first function we'll write is `get_token`, which takes a token list and an expected token as parameters. It compares the expected token to the first token on the list: if they match, it removes the token from the list and returns True; otherwise, it returns False:

```
1 def get_token(token_list, expected):
2     if token_list[0] == expected:
3         del token_list[0]
4         return True
5     return False
```

Since `token_list` refers to a mutable object, the changes made here are visible to any other variable that refers to the same object.

The next function, `get_number`, handles operands. If the next token in `token_list` is a number, `get_number` removes it and returns a leaf node containing the number; otherwise, it returns None.

```
1 def get_number(token_list):
2     x = token_list[0]
3     if type(x) != type(0): return None
4     del token_list[0]
5     return Tree(x, None, None)
```

Before continuing, we should test `get_number` in isolation. We assign a list of numbers to `token_list`, extract the first, print the result, and print what remains of the token list:

```
1 >>> token_list = [9, 11, "end"]
2 >>> x = get_number(token_list)
3 >>> print_tree_postorder(x)
4 9
5 >>> print(token_list)
6 [11, "end"]
```

The next method we need is `get_product`, which builds an expression tree for products. A simple product has two numbers as operands, like  $3 * 7$ .

Here is a version of `get_product` that handles simple products.

```

1 def get_product(token_list):
2     a = get_number(token_list)
3     if get_token(token_list, "*"):
4         b = get_number(token_list)
5         return Tree("*", a, b)
6     return a

```

Assuming that `get_number` succeeds and returns a singleton tree, we assign the first operand to `a`. If the next character is `*`, we get the second number and build an expression tree with `a`, `b`, and the operator.

If the next character is anything else, then we just return the leaf node with `a`. Here are two examples:

```

1 >>> token_list = [9, "*", 11, "end"]
2 >>> tree = get_product(token_list)
3 >>> print_tree_postorder(tree)
4 9 11 *

```

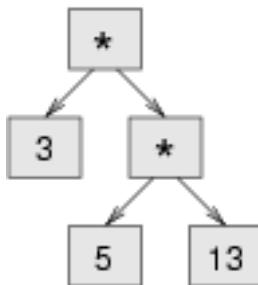
```

1 >>> token_list = [9, "+", 11, "end"]
2 >>> tree = get_product(token_list)
3 >>> print_tree_postorder(tree)
4 9

```

The second example implies that we consider a single operand to be a kind of product. This definition of product is counter-intuitive, but it turns out to be useful.

Now we have to deal with compound products, like like `3 * 5 * 13`. We treat this expression as a product of products, namely `3 * (5 * 13)`. The resulting tree is:



With a small change in `get_product`, we can handle an arbitrarily long product:

```

1 def get_product(token_list):
2     a = get_number(token_list)
3     if get_token(token_list, "*"):
4         b = get_product(token_list)           # This line changed
5         return Tree("*", a, b)
6     return a

```

In other words, a product can be either a singleton or a tree with \* at the root, a number on the left, and a product on the right. This kind of recursive definition should be starting to feel familiar.

Let's test the new version with a compound product:

```

1 >>> token_list = [2, "*", 3, "*", 5, "*", 7, "end"]
2 >>> tree = get_product(token_list)
3 >>> print_tree_postorder(tree)
4 2 3 5 7 * *

```

Next we will add the ability to parse sums. Again, we use a slightly counter-intuitive definition of sum. For us, a sum can be a tree with + at the root, a product on the left, and a sum on the right. Or, a sum can be just a product.

If you are willing to play along with this definition, it has a nice property: we can represent any expression (without parentheses) as a sum of products. This property is the basis of our parsing algorithm.

`get_sum` tries to build a tree with a product on the left and a sum on the right. But if it doesn't find a +, it just builds a product.

```

1 def get_sum(token_list):
2     a = get_product(token_list)
3     if get_token(token_list, "+"):
4         b = get_sum(token_list)
5         return Tree("+", a, b)
6     return a

```

Let's test it with 9 \* 11 + 5 \* 7:

```

1 >>> token_list = [9, "*", 11, "+", 5, "*", 7, "end"]
2 >>> tree = get_sum(token_list)
3 >>> print_tree_postorder(tree)
4 9 11 * 5 7 * +

```

We are almost done, but we still have to handle parentheses. Anywhere in an expression where there can be a number, there can also be an entire sum enclosed in parentheses. We just need to modify `get_number` to handle **subexpressions**:

```

1 def get_number(token_list):
2     if get_token(token_list, "("):
3         x = get_sum(token_list)           # Get the subexpression
4         get_token(token_list, ")")      # Remove the closing parenthesis
5         return x
6     else:
7         x = token_list[0]
8         if type(x) != type(0): return None
9         del token_list[0]
10        return Tree(x, None, None)

```

Let's test this code with  $9 * (11 + 5) * 7$ :

```

1 >>> token_list = [9, "*", "(", 11, "+", 5, ")", "*", 7, "end"]
2 >>> tree = get_sum(token_list)
3 >>> print_tree_postorder(tree)
4 9 11 5 + 7 * *

```

The parser handled the parentheses correctly; the addition happens before the multiplication.

In the final version of the program, it would be a good idea to give `get_number` a name more descriptive of its new role.

## 28.6. Handling errors

Throughout the parser, we've been assuming that expressions are well-formed. For example, when we reach the end of a subexpression, we assume that the next character is a close parenthesis. If there is an error and the next character is something else, we should deal with it.

```

1 def get_number(token_list):
2     if get_token(token_list, "("):
3         x = get_sum(token_list)
4         if not get_token(token_list, ")"):
5             raise ValueError('Missing close parenthesis')
6         return x
7     else:
8         # The rest of the function omitted

```

The `raise` statement throws the exception object which we create. In this case we simply used the most appropriate type of built-in exception that we could find, but you should be aware that you can create your own more specific user-defined exceptions if you need to. If the function that called `get_number`, or one of the other functions in the traceback, handles the exception, then the program can continue. Otherwise, Python will print an error message and quit.

## 28.7. The animal tree

In this section, we develop a small program that uses a tree to represent a knowledge base.

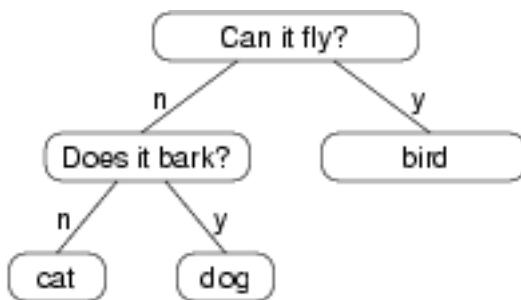
The program interacts with the user to create a tree of questions and animal names. Here is a sample run:

```

1 Are you thinking of an animal? y
2 Is it a bird? n
3 What is the animals name? dog
4 What question would distinguish a dog from a bird? Can it fly
5 If the animal were dog the answer would be? n
6
7 Are you thinking of an animal? y
8 Can it fly? n
9 Is it a dog? n
10 What is the animals name? cat
11 What question would distinguish a cat from a dog? Does it bark
12 If the animal were cat the answer would be? n
13
14 Are you thinking of an animal? y
15 Can it fly? n
16 Does it bark? y
17 Is it a dog? y
18 I rule!
19
20 Are you thinking of an animal? n

```

Here is the tree this dialog builds:



At the beginning of each round, the program starts at the top of the tree and asks the first question. Depending on the answer, it moves to the left or right child and continues until it gets to a leaf node. At that point, it makes a guess. If the guess is not correct, it asks the user for the name of the new animal and a question that distinguishes the (bad) guess from the new animal. Then it adds a node to the tree with the new question and the new animal.

Here is the code:

```
1 def yes(ques):
2     ans = input(ques).lower()
3     return ans[0] == "y"
4
5 def animal():
6     # Start with a singleton
7     root = Tree("bird")
8
9     # Loop until the user quits
10    while True:
11        print()
12        if not yes("Are you thinking of an animal? "): break
13
14        # Walk the tree
15        tree = root
16        while tree.left is not None:
17            prompt = tree.cargo + "? "
18            if yes(prompt):
19                tree = tree.right
20            else:
21                tree = tree.left
22
23        # Make a guess
24        guess = tree.cargo
25        prompt = "Is it a " + guess + "? "
26        if yes(prompt):
27            print("I rule!")
28            continue
29
30        # Get new information
31        prompt = "What is the animal's name? "
32        animal = input(prompt)
33        prompt = "What question would distinguish a {} from a {}? "
34        question = input(prompt.format(animal, guess))
35
36        # Add new information to the tree
37        tree.cargo = question
38        prompt = "If the animal were {} the answer would be? "
39        if yes(prompt.format(animal)):
40            tree.left = Tree(guess)
41            tree.right = Tree(animal)
```

```
42     else:
43         tree.left = Tree(animal)
44         tree.right = Tree(guess)
```

The function `yes` is a helper; it prints a prompt and then takes input from the user. If the response begins with `y` or `Y`, the function returns `True`.

The condition of the outer loop of `animal` is `True`, which means it will continue until the `break` statement executes, if the user is not thinking of an animal.

The inner `while` loop walks the tree from top to bottom, guided by the user's responses.

When a new node is added to the tree, the new question replaces the cargo, and the two children are the new animal and the original cargo.

One shortcoming of the program is that when it exits, it forgets everything you carefully taught it! Fixing this problem is left as an exercise.

## 28.8. Glossary

```
1 binary operator
2     An operator that takes two operands.
3
4 binary tree
5     A tree in which each node refers to zero, one, or two dependent nodes.
6
7 child
8     One of the nodes referred to by a node.
9
10 leaf
11     A bottom-most node in a tree, with no children.
12
13 level
14     The set of nodes equidistant from the root.
15
16 parent
17     The node that refers to a given node.
18
19 postorder
20     A way to traverse a tree, visiting the children of each node before the
21     node itself.
22
23 prefix notation
24     A way of writing a mathematical expression with each operator appearing
```

```
25      before its operands.  
26  
27 preorder  
28     A way to traverse a tree, visiting each node before its children.  
29  
30 root  
31     The topmost node in a tree, with no parent.  
32  
33 siblings  
34     Nodes that share a common parent.  
35  
36 subexpression  
37     An expression in parentheses that acts as a single operand in a larger  
38     expression.
```

## 28.9. Exercises

1. Modify `print_tree_inorder` so that it puts parentheses around every operator and pair of operands. Is the output correct and unambiguous? Are the parentheses always necessary?
2. Write a function that takes an expression string and returns a token list.
3. Find other places in the expression tree functions where errors can occur and add appropriate `raise` statements. Test your code with improperly formed expressions.
4. Think of various ways you might save the animal knowledge tree in a file. Implement the one you think is easiest.

# Appendix A: Debugging

Different kinds of errors can occur in a program, and it is useful to distinguish among them in order to track them down more quickly:

1. Syntax errors are produced by Python when it is translating the source code into byte code. They usually indicate that there is something wrong with the syntax of the program. Example: Omitting the colon at the end of a `def` statement yields the somewhat redundant message `SyntaxError: invalid syntax`.
2. Runtime errors are produced by the runtime system 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 a runtime error of maximum recursion depth exceeded.
3. Semantic errors are problems with a program that compiles and runs but doesn't do the right thing. Example: An expression may not be evaluated in the order you expect, yielding an unexpected 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. Check that indentation is consistent. You may indent with either spaces or tabs but it's best not to mix them. Each level should be nested the same amount.
4. Make sure that any strings in the code have matching quotation marks.
5. 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!
6. An unclosed bracket — `(`, `{`, or `[` — makes Python continue with the next line as part of the current statement. Generally, an error occurs almost immediately in the next line.
7. Check for the classic `=` instead of `==` inside a conditional.

If nothing works, move on to the next section...

## A.2. I can't get my program to run no matter what I do.

If the compiler says there is an error and you don't see it, that might be because you and the compiler 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 (or import) it again. If the compiler doesn't find the new error, there is probably something wrong with the way your environment is set up.

If this happens, 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 new program to the working one.

## A.3. Runtime errors

Once your program is syntactically correct, Python can import it and at least start running it. What could possibly go wrong?

## A.4. My program does absolutely nothing.

This problem is most common when your file consists of functions and classes but does not actually invoke anything 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 that you are invoking a function to start execution, or execute one from the interactive prompt. Also see the [Flow of Execution](#) section below.

## A.5. My program hangs.

If a program stops and seems to be doing nothing, we say it is hanging. Often that means that it is caught in an infinite loop or an infinite recursion.

1. 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.
2. 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.
3. 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.
4. 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.
5. If neither of those steps works, start testing other loops and other recursive functions and methods.
6. 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.

## A.6. 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:

```
1 while x > 0 and y < 0:
2     # Do something to x
3     # Do something to y
4
5     print("x: ", x)
6     print("y: ", y)
7     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.

In a development environment like PyScripter, one can also set a breakpoint at the start of the loop, and single-step through the loop. While you do this, inspect the values of `x` and `y` by hovering your cursor over them.

Of course, all programming and debugging require that you have a good mental model of what the algorithm ought to be doing: if you don't understand what ought to happen to `x` and `y`, printing or inspecting its value is of little use. Probably the best place to debug the code is away from your computer, working on your understanding of what should be happening.

## A.7. Infinite Recursion

Most of the time, an infinite recursion will cause the program to run for a while and then produce a `Maximum recursion depth exceeded` error.

If you suspect that a function or method is causing an infinite recursion, start by checking to make sure that there is a base case. In other words, there should be some condition that will cause the function or method to return without making a recursive invocation. If not, then 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 or method that prints the parameters. Now when you run the program, you will see a few lines of output every time the function or method is invoked, and you will see the parameters. If the parameters are not moving toward the base case, you will get some ideas about why not.

Once again, if you have an environment that supports easy single-stepping, breakpoints, and inspection, learn to use them well. It is our opinion that walking through code step-by-step builds the best and most accurate mental model of how computation happens. Use it if you have it!

## A.8. 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.

If you're not sure, step through the program with your debugger.

## A.9. 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.

Put a breakpoint on the line causing the exception, and look around!

The traceback identifies the function that is currently running, and then the function that invoked it, and then the function that invoked *that*, and so on. In other words, it traces the path of function invocations that got you to where you are. It also includes the line number in your file where each of these calls occurs.

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. 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:

1. You are trying to use a value improperly. Example: indexing a string, list, or tuple with something other than an integer.
2. 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.
3. You are passing the wrong number of arguments to a function or method. 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 value that the dictionary does not contain.

### AttributeError

You are trying to access an attribute or method that does not exist.

### 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 sequence. Is the sequence the right size? Is the index the right value?

## A.10. 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 sorting a sequence, sort a *small* sequence. 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 doesn't affect the program, and it does, that can tip you off.

You can also wrap your debugging `print` statements in some condition, so that you suppress much of the output. For example, if you are trying to find an element using a binary search, and it is not working, you might code up a debugging `print` statement inside a conditional: if the range of candidate elements is less than 6, then print debugging information, otherwise don't print.

Similarly, breakpoints can be made conditional: you can set a breakpoint on a statement, then edit the breakpoint to say "only break if this expression becomes true".

## A.11. Semantic errors

In some ways, semantic errors are the hardest to debug, because the compiler and the runtime system provide no information about what is wrong. Only you know what the program is supposed to do, and only you know that it isn't doing it.

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 walking the program to where the error is occurring.

## A.12. My program doesn't work.

You should ask yourself these questions:

1. 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.
2. 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.
3. 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 invocations to functions or methods in other Python modules. Read the documentation for the functions you invoke. Try them out by writing simple test cases and checking the results.

In order to program, you need to have a mental model of how programs work. If you write a program that doesn't do what you expect, very 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.13. 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:

```
1 self.hands[i].add_card(self.hands[self.find_neighbor(i)].pop_card())
```

This can be rewritten as:

```
1 neighbor = self.find_neighbor(i)
2 picked_card = self.hands[neighbor].pop_card()
3 self.hands[i].add_card(picked_card)
```

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 or inspect 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  $x/2\pi$  into Python, you might write:

```
1 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  $(x/2)\pi$ .

A good way to debug expressions is to add parentheses to make the order of evaluation explicit:

```
1 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 rules of precedence.

## A.14. I've got a function or method 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 `return` value before returning. Again, you can use a temporary variable. For example, instead of:

```
1 return self.hands[i].remove_matches()
```

you could write:

```
1 count = self.hands[i].remove_matches()
2 return count
```

Now you have the opportunity to display or inspect the value of count before returning.

## A.15. 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 effects:

1. Frustration and/or rage.
2. Superstitious beliefs (the computer hates me) and magical thinking  
(the program only works when I wear my hat backward).
3. 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. We often find bugs when we are away from the computer and let our minds wander. Some of the best places to find bugs are trains, showers, and in bed, just before you fall asleep.

## A.16. 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. A fresh pair of eyes is just the thing.

Before you bring someone else in, make sure you have exhausted the techniques described here. 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:

1. If there is an error message, what is it and what part of the program does it indicate?
2. 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?
3. What have you tried so far, and what have you learned?

Good instructors and helpers will also do something that should not offend you: they won't believe when you tell them "*I'm sure all the input routines are working just fine, and that I've set up the data correctly!*". They will want to validate and check things for themselves. After all, your program has a bug. Your understanding and inspection of the code have not found it yet. So you should expect to have your assumptions challenged. And as you gain skills and help others, you'll need to do the same for them.

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.

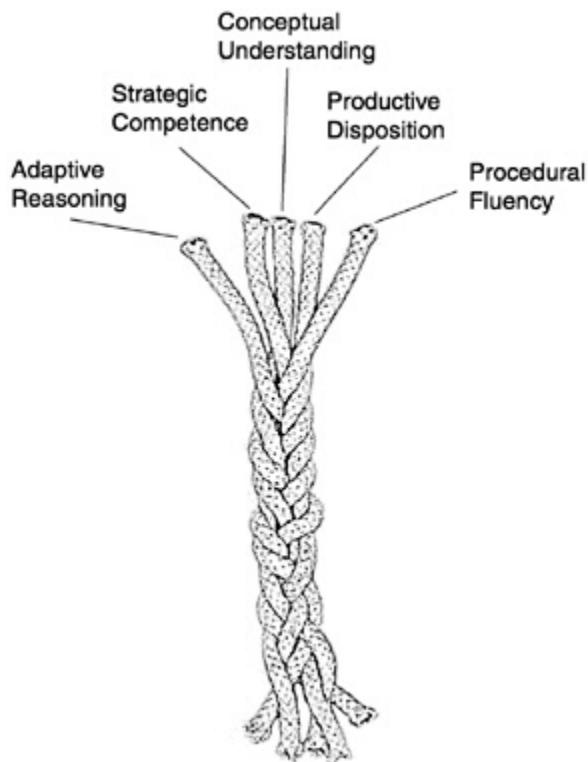
# Appendix B: An odds-and-ends Workbook

*This workbook / cookbook of recipes is still very much under construction.*

## B.1. The Five Strands of Proficiency

This was an important study commissioned by the President in the USA. It looked at what was needed for students to become proficient in maths.

But it is also an amazingly accurate fit for what we need for proficiency in Computer Science, or even for proficiency in playing Jazz!



1. **Procedural Fluency:** Learn the syntax. Learn to type. Learn your way around your tools. Learn and practice your scales. Learn to rearrange formulae.

2. **Conceptual Understanding:** Understand why the bits fit together like they do.

3. **Strategic Competence:** Can you see what to do next? Can you formulate this word problem into your notation? Can you take the music where you want it to go?

4. **Adaptive Reasoning:** Can you see how to change what you've learned for this new problem?

5. **A Productive Disposition:** We need that *Can Do!* attitude!

1. You habitually think it is worthwhile studying this stuff.
2. You are diligent and disciplined enough to grind through the tough stuff, and to put in your practice hours.
3. You develop a sense of *efficacy* — that you can make things happen!

Check out <http://mason.gmu.edu/~jsuh4/teaching/strands.htm>, or Kilpatrick's book at <http://www.nap.edu/openbook.php?isbn=0309069955>

## B.2. Sending Email

Sometimes it is fun to do powerful things with Python — remember that part of the “productive disposition” we saw under the five threads of proficiency included *efficacy* — the sense of being able to accomplish something useful. Here is a Python example of how you can send email to someone.

```
1 import smtplib, email.mime.text
2
3 me = "joe@my.org.com"                      # Put your own email here
4 fred = "fred@his.org.com"                    # And fred's email address here
5 your_mail_server = "mail.my.org.com"         # Ask your system administrator
6
7 # Create a text message containing the body of the email.
8 # You could read this from a file, of course.
9 msg = email.mime.text.MIMEText("""Hey Fred,
10
11 I'm having a party, please come at 8pm.
12 Bring a plate of snacks and your own drinks.
```

```

13
14 Joe""") 
15
16 msg["From"] = me           # Add headers to the message object
17 msg["To"] = fred
18 msg["Subject"] = "Party on Saturday 23rd"
19
20 # Create a connection to your mail server
21 svr = smtplib.SMTP(your_mail_server)
22 response = svr.sendmail(me, fred, msg.as_string()) # Send message
23 if response != {}:
24     print("Sending failed for ", response)
25 else:
26     print("Message sent.")
27
28 svr.quit()                 # Close the connection

```

In the context of the course, notice how we use the two objects in this program: we create a message object on line 9, and set some attributes at lines 16-18. We then create a connection object at line 21, and ask it to send our message.

## B.3. Write your own Web Server

Python is gaining in popularity as a tool for writing web applications. Although one will probably use Python to process requests behind a web server like Apache, there are powerful libraries which allow you to write your own stand-alone web server in a couple of lines. This simpler approach means that you can have a test web server running on your own desktop machine in a couple of minutes, without having to install any extra software.

In this cookbook example we use the `wsgi` (“wizz-gee”) protocol: a modern way of connecting web servers to code that runs to provide the services. See [http://en.wikipedia.org/wiki/Web\\_Server-Gateway\\_Interface](http://en.wikipedia.org/wiki/Web_Server-Gateway_Interface) for more on `wsgi`.

```

1 from codecs import latin_1_encode
2 from wsgiref.simple_server import make_server
3
4 def my_handler(environ, start_response):
5     path_info = environ.get("PATH_INFO", None)
6     query_string = environ.get("QUERY_STRING", None)
7     response_body = "You asked for {0} with query {1}" .format(
8                                     path_info, query_string)
9     response_headers = [ ("Content-Type", "text/plain"),

```

```

10         ("Content-Length", str(len(response_body)))]
11     start_response("200 OK", response_headers)
12     response = latin_1_encode(response_body)[0]
13     return [response]
14
15 httpd = make_server("127.0.0.1", 8000, my_handler)
16 httpd.serve_forever()  # Start the server listening for requests

```

When you run this, your machine will listen on port 8000 for requests. (You may have to tell your firewall software to be kind to your new application!)

In a web browser, navigate to `http://127.0.0.1:8000/catalogue?category=guitars`. Your browser should get the response

1 You asked for /catalogue with query category=guitars

Your web server will keep running until you interrupt it (Ctrl+F2 if you are using PyScripter).

The important lines 15 and 16 create a web server on the local machine, listening at port 8000. Each incoming html request causes the server to call `my_handler` which processes the request and returns the appropriate response.

We modify the above example below: `my_handler` now interrogates the `path_info`, and calls specialist functions to deal with each different kind of incoming request. (We say that `my_handler` *dispatches* the request to the appropriate function.) We can easily add other more request cases:

```

1 import time
2
3 def my_handler(environ, start_response):
4     path_info = environ.get("PATH_INFO", None)
5     if path_info == "/gettime":
6         response_body = gettime(environ, start_response)
7     elif path_info == "/classlist":
8         response_body = classlist(environ, start_response)
9     else:
10        response_body = ""
11        start_response("404 Not Found", [("Content-Type", "text/plain")])
12
13    response = latin_1_encode(response_body)[0]
14    return [response]
15
16 def gettime(env, resp):
17     html_template = """<html>
18     <body bgcolor='lightblue'>

```

```

19     <h2>The time on the server is {0}</h2>
20 <body>
21 </html>
22 """
23 response_body = html_template.format(time.ctime())
24 response_headers = [ ("Content-Type", "text/html"),
25                     ("Content-Length", str(len(response_body)))]
26 resp("200 OK", response_headers)
27 return response_body
28
29 def classlist(env, resp):
30     return # Will be written in the next section!

```

Notice how `gettime` returns an (admittedly simple) html document which is built on the fly by using `format` to substitute content into a predefined template.

## B.4. Using a Database

Python has a library for using the popular and lightweight `sqlite` database. Learn more about this self-contained, embeddable, zero-configuration SQL database engine at <http://www.sqlite.org>.

Firstly, we have a script that creates a new database, creates a table, and stores some rows of test data into the table: (Copy and paste this code into your Python system.)

We get this output:

- 1 Database table StudentSubjects has been created.
- 2 StudentSubjects table now has 18 rows of data.

Our next recipe adds to our web browser from the previous section. We'll allow a query like `classlist?subject=CompSci&year=2012` and show how our server can extract the arguments from the query string, query the database, and send the rows back to the browser as a formatted table within an html page. We'll start with two new imports to get access to `sqlite3` and `cgi`, a library which helps us parse forms and query strings that are sent to the server:

```

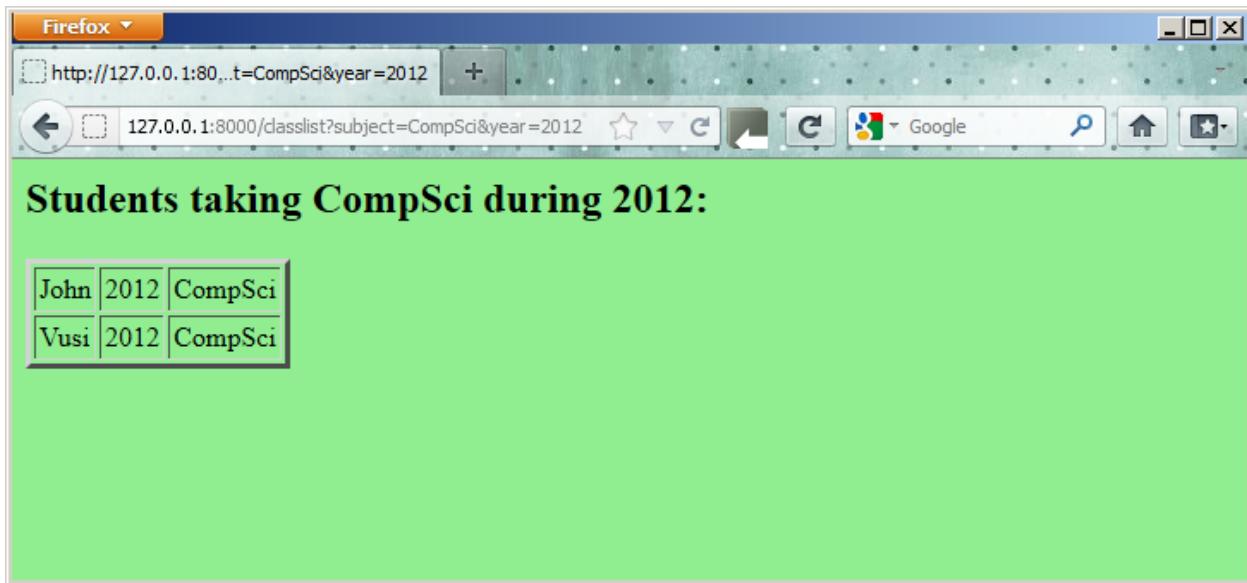
1 import sqlite3
2 import cgi

```

Now we replace the stub function `classlist` with a handler that can do what we need:

```
1 classlistTemplate = """<html>
2 <body bgcolor='lightgreen'>
3   <h2>Students taking {0} during {1}:</h2>
4   <table border=3 cellspacing=2 cellpadding=2>
5     {2}
6   </table>
7 </body>
8 </html>
9 """
10
11 def classlist(env, resp):
12
13     # Parse the field value from the query string (or from a submitted form)
14     # In a real server you'd want to check they were present!
15     the_fields = cgi.FieldStorage(environ = env)
16     subj = the_fields["subject"].value
17     year = the_fields["year"].value
18
19     # Attach to the database, build a query, fetch the rows.
20     connection = sqlite3.connect("c:\studentRecords.db")
21     cursor = connection.cursor()
22     cursor.execute("SELECT * FROM StudentSubjects WHERE subject=? AND year=?",
23                   (subj, year))
24     result = cursor.fetchall()
25     # Build the html rows for the table
26     table_rows = ""
27     for (sn, yr, subj) in result:
28         table_rows += "      <tr><td>{0}<td>{1}<td>{2}\n".format(sn, yr, subj)
29
30     # Now plug the headings and data into the template, and complete the response
31     response_body = classlistTemplate.format(subj, year, table_rows)
32     response_headers = [("Content-Type", "text/html"),
33                         ("Content-Length", str(len(response_body)))]
34     resp("200 OK", response_headers)
35     return response_body
```

When we run this and navigate to `http://127.0.0.1:8000/classlist?subject=CompSci&year=2012` with a browser, we'll get output like this:



It is unlikely that we would write our own web server from scratch. But the beauty of this approach is that it creates a great test environment for working with server-side applications that use the `wsgi` protocols. Once our code is ready, we can deploy it behind a web server like Apache which can interact with our handlers using `wsgi`.

# Appendix C: Configuring Ubuntu for Python Development

*Note:* the following instructions assume that you are connected to the Internet and that you have both the `main` and `universe` package repositories enabled. All unix shell commands are assumed to be running from your home directory (`$HOME`). Finally, any command that begins with `sudo` assumes that you have administrative rights on your machine. If you do not — please ask your system administrator about installing the software you need.

What follows are instructions for setting up an Ubuntu 9.10 (Karmic) home environment for use with this book. I use Ubuntu GNU/Linux for both development and testing of the book, so it is the only system about which I can personally answer setup and configuration questions.

In the spirit of software freedom and open collaboration, please contact me if you would like to maintain a similar appendix for your own favorite system. I'd be more than happy to link to it or put it on the Open Book Project site, provided you agree to answer user feedback concerning it.

Thanks!

[Jeffrey Elkner<sup>24</sup>](#)

Governor's Career and Technical Academy in Arlington  
Arlington, Virginia

## C.1. Vim

Vim<sup>25</sup> can be used very effectively for Python development, but Ubuntu only comes with the `vim-tiny` package installed by default, so it doesn't support color syntax highlighting or auto-indenting.

To use Vim, do the following:

1. From the unix command prompt, run:

```
$ sudo apt-get install vim-gnome
```

2. Create a file in your home directory named `.vimrc` that contains the following:

```
syntax enable  
filetype indent on  
set et
```

---

<sup>24</sup><mailto:jeff@elkner.net>  
<sup>25</sup><http://www.vim.org>

```
set sw=4  
set smarttab  
map <f2> :w!python %
```

When you edit a file with a .py extension, you should now have color syntax highlighting and auto indenting. Pressing the key should run your program, and bring you back to the editor when the program completes.

To learn to use vim, run the following command at a unix command prompt:

```
$ vimtutor
```

## C.2. \$HOME environment

The following creates a useful environment in your home directory for adding your own Python libraries and executable scripts:

1. From the command prompt in your home directory, create bin and lib/python subdirectories by running the

following commands:

```
$ mkdir bin lib  
$ mkdir lib/python
```

2. Add the following lines to the bottom of your .bashrc in your home directory:

```
PYTHONPATH=$HOME/lib/python  
EDITOR=vim
```

```
export PYTHONPATH EDITOR
```

This will set your preferred editor to Vim, add your own lib/python subdirectory for your Python libraries to your Python path, and add your own bin directory as a place to put executable scripts. You need to logout and log back in before your local bin directory will be in your search path<sup>26</sup>.

---

<sup>26</sup>[http://en.wikipedia.org/wiki/Path\\_\(variable\)](http://en.wikipedia.org/wiki/Path_(variable))

## C.3. Making a Python script executable and runnable from anywhere

On unix systems, Python scripts can be made *executable* using the following process:

1. Add this line as the first line in the script:

```
1      #!/usr/bin/env python3
```

2. At the unix command prompt, type the following to make myscript.py executable:

```
$ chmod +x myscript.py
```

3. Move myscript.py into your bin directory, and it will be runnable from anywhere.

# Appendix D: Customizing and Contributing to the Book

*Note:* the following instructions assume that you are connected to the Internet and that you have both the `main` and `universe` package repositories enabled. All unix shell commands are assumed to be running from your home directory (`$HOME`). Finally, any command that begins with `sudo` assumes that you have administrative rights on your machine. If you do not — please ask your system administrator about installing the software you need.

This book is free as in freedom, which means you have the right to modify it to suite your needs, and to redistribute your modifications so that our whole community can benefit.

That freedom lacks meaning, however, if you the tools needed to make a custom version or to contribute corrections and additions are not within your reach. This appendix attempts to put those tools in your hands.

Thanks!

[Jeffrey Elkner<sup>27</sup>](#)

Governor's Career and Technical Academy in Arlington  
Arlington, Virginia

## D.1. Getting the Source

This book is marked up<sup>28</sup> in [ReStructuredText](#)<sup>29</sup> using a document generation system called [Sphinx](#)<sup>30</sup>.

The source code is located at <https://code.launchpad.net/~thinkcspy-rle-team/thinkcspy/thinkcspy3-rle>.

The easiest way to get the source code on an Ubuntu computer is:

1. run `sudo apt-get install bzr` on your system to install `bzr`<sup>31</sup>.
2. run `bzr branch lp:thinkcspy`.

The last command above will download the book source from Launchpad into a directory named `thinkcspy` which contains the Sphinx source and configuration information needed to build the book.

---

<sup>27</sup><mailto:jeff@elkner.net>

<sup>28</sup>[http://en.wikipedia.org/wiki/Markup\\_language](http://en.wikipedia.org/wiki/Markup_language)

<sup>29</sup><http://en.wikipedia.org/wiki/ReStructuredText>

<sup>30</sup>[http://en.wikipedia.org/wiki/Sphinx\\_%28documentation\\_generator%29](http://en.wikipedia.org/wiki/Sphinx_%28documentation_generator%29)

<sup>31</sup>[http://en.wikipedia.org/wiki/Bazaar\\_%28software%29](http://en.wikipedia.org/wiki/Bazaar_%28software%29)

## D.2. Making the HTML Version

To generate the html version of the book:

1. run `sudo apt-get install python-sphinx` to install the Sphinx documentation system.
2. `cd thinkcspy` - change into the `thinkcspy` directory containing the book source.
3. `make html`.

The last command will run sphinx and create a directory named `build` containing the html version of the text.

*Note:* Sphinx supports building other output types as well, such as PDF<sup>32</sup>. This requires that LaTeX<sup>33</sup> be present on your system. Since I only personally use the html version, I will not attempt to document that process here.

---

<sup>32</sup><http://en.wikipedia.org/wiki/PDF>

<sup>33</sup><http://en.wikipedia.org/wiki/LaTeX>

# Appendix E: Some Tips, Tricks, and Common Errors

These are small summaries of ideas, tips, and commonly seen errors that might be helpful to those beginning Python.

## E.1. Functions

Functions help us with our mental chunking: they allow us to group together statements for a high-level purpose, e.g. a function to sort a list of items, a function to make the turtle draw a spiral, or a function to compute the mean and standard deviation of some measurements.

There are two kinds of functions: fruitful, or value-returning functions, which *calculate and return a value*, and we use them because we're primarily interested in the value they'll return. Void (non-fruitful) functions are used because they *perform actions* that we want done — e.g. make a turtle draw a rectangle, or print the first thousand prime numbers. They always return `None` — a special dummy value.

Tip: `None` is not a string

Values like `None`, `True` and `False` are not strings: they are special values in Python, and are in the list of keywords we gave in chapter 2 (Variables, expressions, and statements). Keywords are special in the language: they are part of the syntax. So we cannot create our own variable or function with a name `True` — we'll get a syntax error. (Built-in functions are not privileged like keywords: we can define our own variable or function called `len`, but we'd be silly to do so!)

Along with the fruitful/void families of functions, there are two flavors of the `return` statement in Python: one that returns a useful value, and the other that returns nothing, or `None`. And if we get to the end of any function and we have not explicitly executed any `return` statement, Python automatically returns the value `None`.

Tip: Understand what the function needs to return

Perhaps nothing — some functions exist purely to perform actions rather than to calculate and return a result. But if the function should return a value, make sure all execution paths do return the value.

To make functions more useful, they are given *parameters*. So a function to make a turtle draw a square might have two parameters — one for the turtle that needs to do the drawing, and another for the size of the square. See the first example in Chapter 4 (Functions) — that function can be used

with any turtle, and for any size square. So it is much more general than a function that always uses a specific turtle, say `tess` to draw a square of a specific size, say 30.

Tip: Use parameters to generalize functions

Understand which parts of the function will be hard-coded and unchangeable, and which parts should become parameters so that they can be customized by the caller of the function.

Tip: Try to relate Python functions to ideas we already know

In math, we're familiar with functions like  $f(x) = 3x + 5$ . We already understand that when we call the function  $f(3)$  we make some association between the parameter  $x$  and the argument 3. Try to draw parallels to argument passing in Python.

Quiz: Is the function  $f(z) = 3z + 5$  the same as function  $f$  above?

## E.2. Problems with logic and flow of control

We often want to know if some condition holds for any item in a list, e.g. “does the list have any odd numbers?” This is a common mistake:

```

1 def any_odd(xs): # Buggy version
2     """ Return True if there is an odd number in xs, a list of integers. """
3     for v in xs:
4         if v % 2 == 1:
5             return True
6         else:
7             return False

```

Can we spot two problems here? As soon as we execute a `return`, we'll leave the function. So the logic of saying “If I find an odd number I can return `True`” is fine. However, we cannot return `False` after only looking at one item — we can only return `False` if we've been through all the items, and none of them are odd. So line 6 should not be there, and line 7 has to be outside the loop. To find the second problem above, consider what happens if you call this function with an argument that is an empty list. Here is a corrected version:

```

1 def any_odd(xs):
2     """ Return True if there is an odd number in xs, a list of integers. """
3     for v in xs:
4         if v % 2 == 1:
5             return True
6     return False

```

This “eureka”, or “short-circuit” style of returning from a function as soon as we are certain what the outcome will be was first seen in Section 8.10, in the chapter on strings.

It is preferred over this one, which also works correctly:

```

1 def any_odd(xs):
2     """ Return True if there is an odd number in xs, a list of integers. """
3     count = 0
4     for v in xs:
5         if v % 2 == 1:
6             count += 1      # Count the odd numbers
7     if count > 0:
8         return True
9     else:
10        return False

```

The performance disadvantage of this one is that it traverses the whole list, even if it knows the outcome very early on.

Tip: Think about the return conditions of the function

Do I need to look at all elements in all cases? Can I shortcut and take an early exit? Under what conditions? When will I have to examine all the items in the list?

The code in lines 7-10 can also be tightened up. The expression `count > 0` evaluates to a Boolean value, either `True` or `False`. The value can be used directly in the `return` statement. So we could cut out that code and simply have the following:

```

1 def any_odd(xs):
2     """ Return True if there is an odd number in xs, a list of integers. """
3     count = 0
4     for v in xs:
5         if v % 2 == 1:
6             count += 1      # Count the odd numbers
7     return count > 0    # Aha! a programmer who understands that Boolean
8                         #   expressions are not just used in if statements!

```

Although this code is tighter, it is not as nice as the one that did the short-circuit return as soon as the first odd number was found.

Tip: Generalize your use of Booleans

Mature programmers won't write `if is_prime(n) == True:` when they could say instead `if is_prime(n):` Think more generally about Boolean values, not just in the context of `if` or `while` statements. Like arithmetic expressions, they have their own set of operators (`and`, `or`, `not`) and values (`True`, `False`) and can be assigned to variables, put into lists, etc. A good resource for improving your use of Booleans is [http://en.wikibooks.org/wiki/Non-Programmer%27s\\_Tutorial\\_for\\_Python\\_-3/Boolean\\_Expressions](http://en.wikibooks.org/wiki/Non-Programmer%27s_Tutorial_for_Python_-3/Boolean_Expressions)

Exercise time:

- How would we adapt this to make another function which returns

True if *all* the numbers are odd? Can you still use a short-circuit style?

- How would we adapt it to return True if at least three of the numbers are odd? Short-circuit the traversal when the third odd number is found — don't traverse the whole list unless we have to.

## E.3. Local variables

Functions are called, or activated, and while they're busy they create their own stack frame which holds local variables. A local variable is one that belongs to the current activation. As soon as the function returns (whether from an explicit return statement or because Python reached the last statement), the stack frame and its local variables are all destroyed. The important consequence of this is that a function cannot use its own variables to remember any kind of state between different activations. It cannot count how many times it has been called, or remember to switch colors between red and blue UNLESS it makes use of variables that are global. Global variables will survive even after our function has exited, so they are the correct way to maintain information between calls.

```
1 sz = 2
2 def h2():
3     """ Draw the next step of a spiral on each call. """
4     global sz
5     tess.turn(42)
6     tess.forward(sz)
7     sz += 1
```

This fragment assumes our turtle is tess. Each time we call h2() it turns, draws, and increases the global variable sz. Python always assumes that an assignment to a variable (as in line 7) means that we want a new local variable, unless we've provided a global declaration (on line 4). So leaving out the global declaration means this does not work.

Tip: Local variables do not survive when you exit the function

Use a Python visualizer like the one at <http://pythontutor.com/> to build a strong understanding of function calls, stack frames, local variables, and function returns.

Tip: Assignment in a function creates a local variable

Any assignment to a variable within a function means Python will make a local variable, unless we override with global.

## E.4. Event handler functions

Our chapter on event handling showed three different kinds of events that we could handle. They each have their own subtle points that can trip us up.

- Event handlers are void functions — they don't return any values.
- They're automatically called by the Python interpreter in response to an event, so we don't get to see the code that calls them.
- A mouse-click event passes two coordinate arguments to its handler, so when we write this handler we have to provide for two parameters (usually named `x` and `y`). This is how the handler knows where the mouse click occurred.
- A keypress event handler has to be bound to the key it responds to. There is a messy extra step when using keypresses: we have to remember to issue a `wn.listen()` before our program will receive any keypresses. But if the user presses the key 10 times, the handler will be called ten times.
- Using a timer to create a future-dated event only causes one call to the handler. If we want repeated periodic handler activations, then from within the handler we call `wn.ontimer(....)` to set up the next event.

## E.5. String handling

There are only four *really* important operations on strings, and we'll be able to do just about anything. There are many more nice-to-have methods (we'll call them sugar coating) that can make life easier, but if we can work with the basic four operations smoothly, we'll have a great grounding.

- `len(str)` finds the length of a string.
- `str[i]` the subscript operation extracts the `i`'th character of the string, as a new string.
- `str[i:j]` the slice operation extracts a substring out of a string.
- `str.find(target)` returns the index where target occurs within the string, or `-1` if it is not found.

So if we need to know if "snake" occurs as a substring within `s`, we could write

```

1 if s.find("snake") >= 0: ...
2 if "snake" in s: ...           # Also works, nice-to-know sugar coating!

```

It would be wrong to split the string into words unless we were asked whether the *word* “snake” occurred in the string.

Suppose we’re asked to read some lines of data and find function definitions, e.g.: `def some_function_name(x, y):`, and we are further asked to isolate and work with the name of the function. (Let’s say, print it.)

```

1 s = "..."
2 def_pos = s.find("def ")
3 if def_pos == 0:
4     op_index = s.find("(")
5     fname = s[4:op_index]
6     print(fname)

```

*# Get the next line from somewhere  
# Look for "def " in the line  
# If it occurs at the left margin  
# Find the index of the open parenthesis  
# Slice out the function name  
# ... and work with it.*

One can extend these ideas:

- What if the function def was indented, and didn’t start at column 0?
- The code would need a bit of adjustment, and we’d probably want to be sure that all the characters in front of the `def_pos` position were spaces. We would not want to do the wrong thing on data like this: `# I def initely like Python!`
- We’ve assumed on line 3 that we will find an open parenthesis. It may need to be checked that we did!
- We have also assumed that there was exactly one space between the keyword `def` and the start of the function name. It will not work nicely for `def f(x)`

As we’ve already mentioned, there are many more “sugar-coated” methods that let us work more easily with strings. There is an `rfind` method, like `find`, that searches from the end of the string backwards. It is useful if we want to find the last occurrence of something. The `lower` and `upper` methods can do case conversion. And the `split` method is great for breaking a string into a list of words, or into a list of lines. We’ve also made extensive use in this book of the `format` method. In fact, if we want to practice reading the Python documentation and learning some new methods on our own, the string methods are an excellent resource.

Exercises:

- Suppose any line of text can contain at most one url that starts with “[http://](#)” and ends at the next space in the line. Write a fragment of code to extract and print the full url if it is present.

(Hint: read the documentation for `find`. It takes some extra arguments, so you can set a starting point from which it will search.)

- Suppose a string contains at most one substring “`< ... >`”.

Write a fragment of code to extract and print the portion of the string between the angle brackets.

## E.6. Looping and lists

Computers are useful because they can repeat computation, accurately and fast. So loops are going to be a central feature of almost all programs you encounter.

Tip: Don’t create unnecessary lists

Lists are useful if you need to keep data for later computation. But if you don’t need lists, it is probably better not to generate them.

Here are two functions that both generate ten million random numbers, and return the sum of the numbers. They both work.

```
1 import random
2 joe = random.Random()
3
4 def sum1():
5     """ Build a list of random numbers, then sum them """
6     xs = []
7     for i in range(10000000):
8         num = joe.randrange(1000)    # Generate one random number
9         xs.append(num)             # Save it in our list
10
11    tot = sum(xs)
12
13    return tot
14
15 def sum2():
16     """ Sum the random numbers as we generate them """
17     tot = 0
18     for i in range(10000000):
19         num = joe.randrange(1000)
20         tot += num
21
22     return tot
23
24 print(sum1())
25 print(sum2())
```

What reasons are there for preferring the second version here? (Hint: open a tool like the Performance Monitor on your computer, and watch the memory usage. How big can you make the list before you get a fatal memory error in `sum1`?)

In a similar way, when working with files, we often have an option to read the whole file contents into a single string, or we can read one line at a time and process each line as we read it. Line-at-a-time is the more traditional and perhaps safer way to do things — you'll be able to work comfortably no matter how large the file is. (And, of course, this mode of processing the files was essential in the old days when computer memories were much smaller.) But you may find whole-file-at-once is sometimes more convenient!

MADE  
WITH  
CREATIVE  
COMMONS

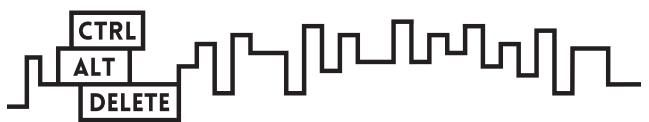




# MADE WITH CREATIVE COMMONS

---

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](http://www.cadb.dk)

[hey@cadb.dk](mailto: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/](http://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

---

<b>Foreword</b> . . . . .	<b>xi</b>
<b>Introduction</b> . . . . .	<b>xv</b>

## PART 1: THE BIG PICTURE

<b>1 The New World of Digital Commons by Paul Stacey</b> . . . . .	<b>3</b>
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
<b>2 How to Be Made with Creative Commons by Sarah Hinchliff Pearson</b> . . . . .	<b>19</b>
Problem Zero: Getting Discovered . . . . .	22
Making Money . . . . .	26
Making Human Connections . . . . .	30
<b>3 The Creative Commons Licenses</b> . . . . .	<b>39</b>

## 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.”<sup>1</sup>

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.<sup>2</sup> 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.<sup>3</sup> 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.<sup>4</sup>

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.<sup>5</sup> 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.

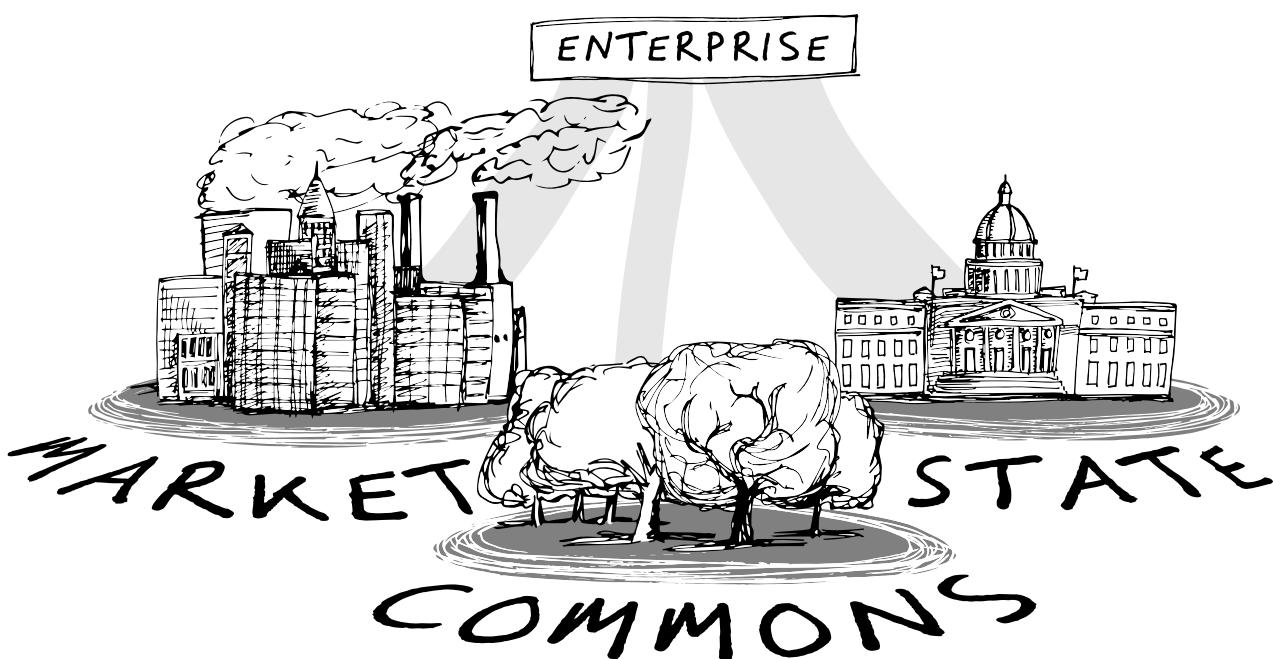


Fig. 1. Enterprise engagement with commons, state, and market.

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.<sup>6</sup> 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.<sup>7</sup> 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.

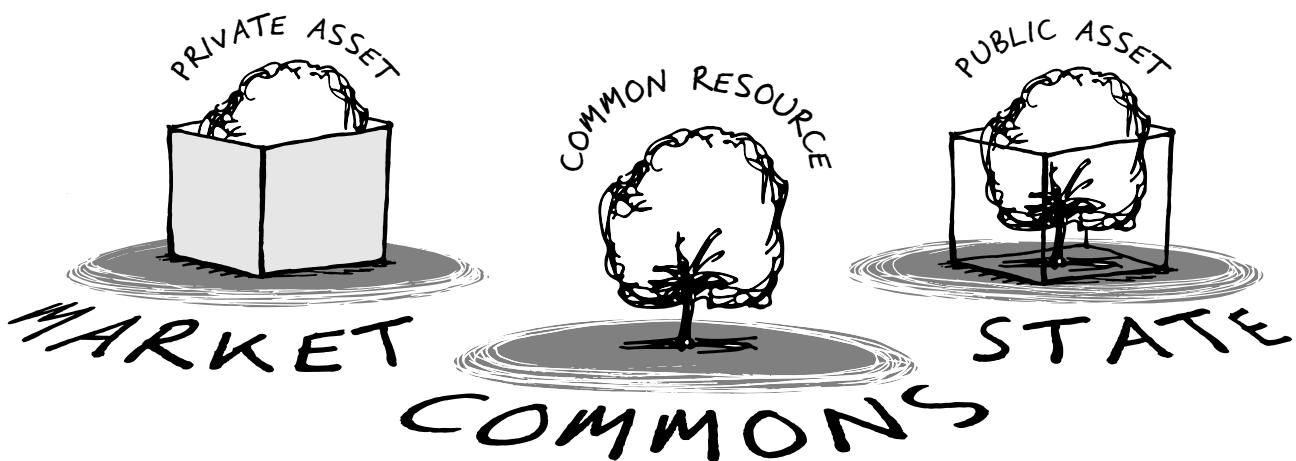


Fig. 3. How the market, commons, and state conceive of resources.

## **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.<sup>9</sup>

## **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.<sup>10</sup> 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.<sup>11</sup> 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:

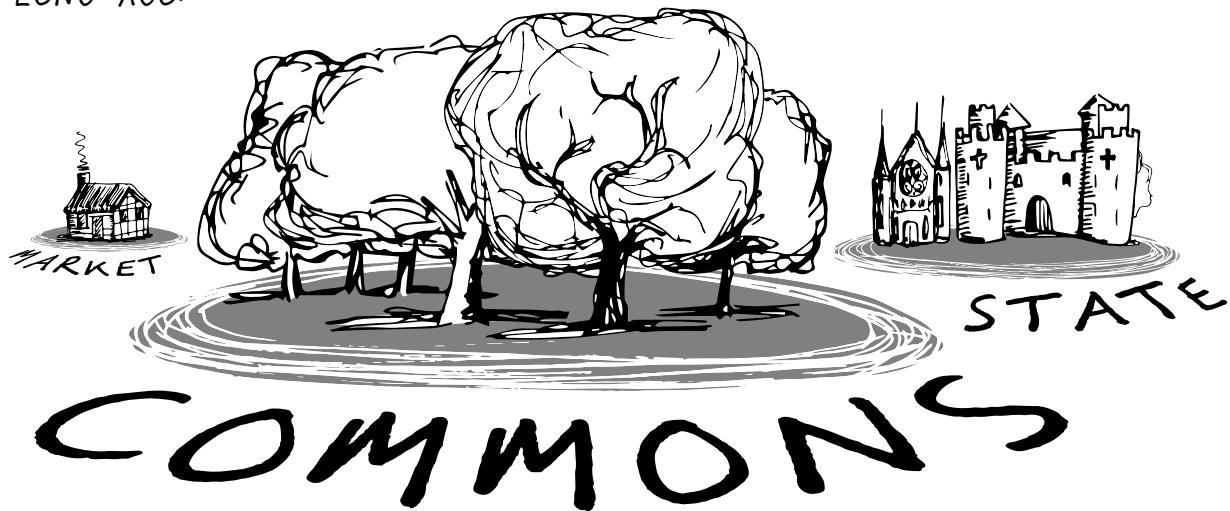


Fig. 4. In preindustrialized society.

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.<sup>12</sup> In olden days, "commoners" were evicted from the land, fences and hedges erected, laws passed, and security set up to forbid access.<sup>13</sup> 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.<sup>14</sup>

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.<sup>15</sup> The emergence of information technology and the Internet has led to an explosion in digital resources and new means of sharing

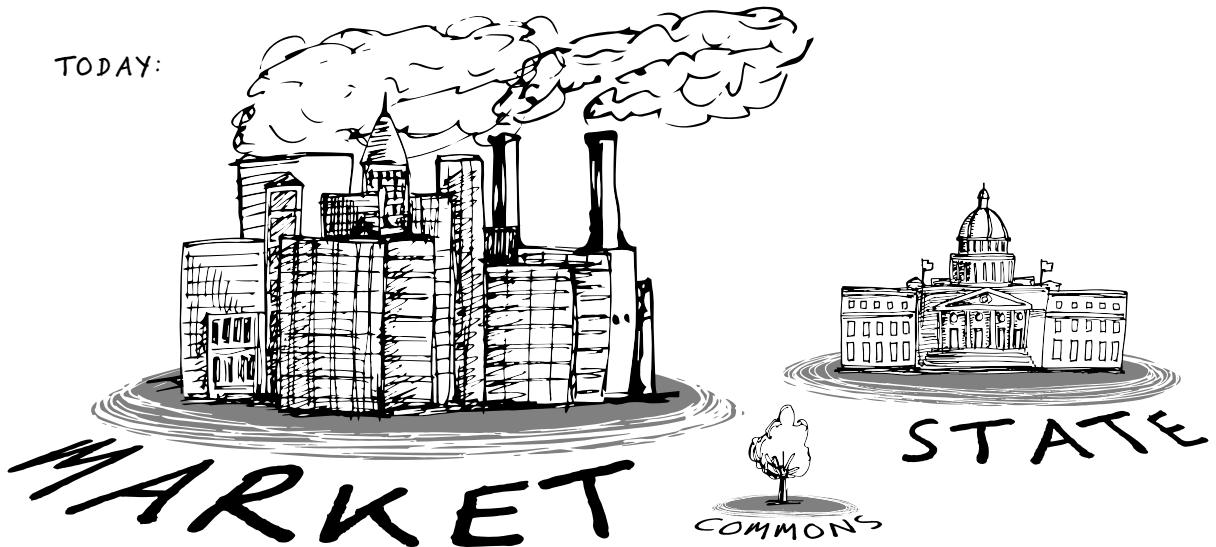


Fig. 6. How the market, the state, and the commons look today.

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.<sup>16</sup>

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.<sup>17</sup> 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.<sup>18</sup> 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.<sup>19</sup>

## 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.<sup>20</sup> 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.<sup>21</sup> 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.<sup>22</sup> 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.<sup>23</sup> 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.<sup>24</sup>

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.<sup>25</sup>

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.<sup>26</sup> 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.<sup>27</sup>

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.<sup>28</sup> 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.<sup>29</sup>

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.<sup>30</sup> 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.<sup>31</sup> 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.<sup>32</sup>

"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.<sup>33</sup>

It contains a "business model canvas," which conceives of a business model as having nine building blocks.<sup>34</sup> 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.*"<sup>35</sup> 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.)<sup>36</sup> 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.<sup>37</sup> 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](http://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/](http://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](http://www.iab.net/media/file/POSWhitePaper.pdf).
- 20 "Licensing Considerations," Creative Commons, accessed December 30, 2016, [creativecommons.org/share-your-work/licensing-considerations/](http://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/](http://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."<sup>1</sup> 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."<sup>2</sup> 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.<sup>3</sup> 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.<sup>4</sup> 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."<sup>5</sup> 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."<sup>6</sup> 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."<sup>7</sup> 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.<sup>8</sup> Anderson wrote, "The greatest change of the past decade has been the shift in time people spend consuming amateur content instead of professional content."<sup>9</sup> 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."<sup>10</sup> 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.<sup>11</sup> 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."<sup>12</sup>

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."<sup>13</sup>

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.<sup>14</sup> 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.<sup>15</sup> 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."<sup>16</sup> 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.<sup>17</sup> 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.<sup>18</sup>

## **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.<sup>19</sup> 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.<sup>20</sup> 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.<sup>21</sup> 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.”<sup>22</sup> 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.<sup>23</sup> We know that people will pay more for products they had a part in creating.<sup>24</sup> 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.<sup>25</sup>

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.”<sup>26</sup> 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.<sup>27</sup> 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.<sup>28</sup> 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.”<sup>29</sup>

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.<sup>30</sup> 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.<sup>31</sup>

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.<sup>32</sup> 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."<sup>33</sup>

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."<sup>34</sup> 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).<sup>35</sup> 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*.<sup>36</sup> 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.<sup>37</sup> The Internet has made this model more difficult because the number of potential channels available to reach those eyeballs has become essentially infinite.<sup>38</sup> 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.<sup>39</sup> 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.<sup>40</sup> We almost can't help but think of relationships in the market as being centered on an even-steven exchange of value.<sup>41</sup>

### 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.<sup>42</sup> 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."<sup>43</sup>

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.<sup>44</sup> 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."<sup>45</sup> 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.<sup>46</sup>

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.<sup>47</sup> 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.<sup>48</sup> 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.<sup>49</sup> 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.<sup>50</sup> 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.”<sup>51</sup> 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.<sup>52</sup> 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.”<sup>53</sup> 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.<sup>54</sup>

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.<sup>55</sup> 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.<sup>56</sup>

### 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.<sup>57</sup> 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.<sup>58</sup> 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."<sup>59</sup> 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."<sup>60</sup>

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."<sup>61</sup> Building true community requires giving people within the community the power to create or influence the rules that govern the community.<sup>62</sup> 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.<sup>63</sup> 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.<sup>64</sup> 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.<sup>65</sup> 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.<sup>66</sup> 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.<sup>67</sup> 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.<sup>68</sup>

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.<sup>69</sup>

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.<sup>70</sup> 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.”<sup>71</sup>

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.<sup>72</sup> 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.<sup>73</sup>

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.<sup>74</sup> 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.<sup>75</sup>

## 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](http://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](http://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](http://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](http://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/](http://creativecommons.org/share-your-work/).



# **Part 2**

## **THE CASE STUDIES**

---

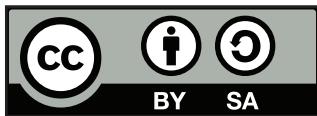


The twenty-four case studies in this section were chosen from hundreds of nominations received from Kickstarter backers, Creative Commons staff, and the global Creative Commons community. We selected eighty potential candidates that represented a mix of industries, content types, revenue streams, and 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](http://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.<sup>1</sup>

---

## **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.<sup>2</sup>

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."

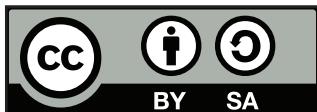
Arduino has been hugely successful in making technology and electronics reach a larger audience. For Tom, Arduino has been about "the democratization of technology." Tom sees Arduino's open-source strategy as helping the world get over the idea that technology has to be protected. Tom says, "Technology 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](http://www.arduino.cc/en/Main/Products)
- 2 [blog.arduino.cc/2013/07/10/send-in-the-clones/](http://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](http://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.

---

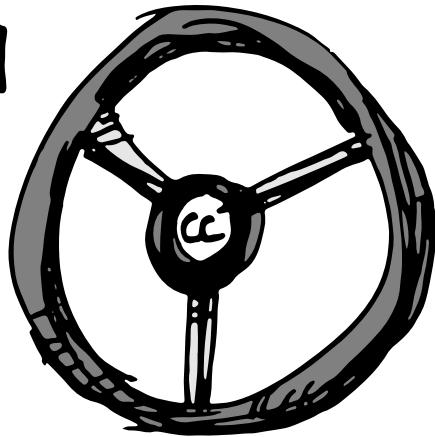
At the core of everything Ártica does is a set of values. "Good content is not enough," Jorge said. "We also think that it is very important to take a stand for some things in the cultural sector." Mariana and Jorge are activists. They defend free culture (the movement promoting the freedom to modify 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](http://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](http://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.**

---

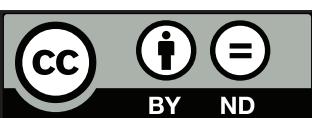
For all of their success, the creators of Cards Against Humanity are only partially motivated by money. Max says they have always been interested in the Walt Disney philosophy of financial success. "We don't make jokes and games to make money—we make money so we can make more jokes and games," he said.

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](http://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.<sup>1</sup> 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](http://theconversation.com/us/charter)



# CORY DOCTOROW



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

[raphound.com](http://raphound.com) and [boingboing.net](http://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.<sup>1</sup> 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.<sup>2</sup>

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.<sup>3</sup> 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](http://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](http://retr0.shinyapps.io/journal_costs/?year=2014&inst=19,22,38,42,59,64,80,95,136)
- 3 [figshare.com/features](http://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,<sup>1</sup> 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.<sup>2</sup> 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.<sup>3</sup>

Figure.NZ also has patrons.<sup>4</sup> 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](http://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/](http://www.ict.govt.nz/guidance-and-resources/open-government/new-zealand-government-open-access-and-licensing-nzgoal-framework/)
- 3 [figure.nz/business/](http://figure.nz/business/)
- 4 [figure.nz/patrons/](http://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](http://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.<sup>1</sup>

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.<sup>4</sup> 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.)<sup>5</sup>

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](http://www.pinter.org.uk/pdfs/Toward_an_Open.pdf)
- 2 [www.oopen.org](http://www.oopen.org)
- 3 [www.hathitrust.org](http://www.hathitrust.org)
- 4 [collections.knowledgeunlatched.org/collection-availability-1/](http://collections.knowledgeunlatched.org/collection-availability-1/)
- 5 [www.knowledgeunlatched.org/featured-authors-section/](http://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](http://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.<sup>1</sup> 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/](http://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](http://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.<sup>1</sup> 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.<sup>2</sup> 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.<sup>2</sup> 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.

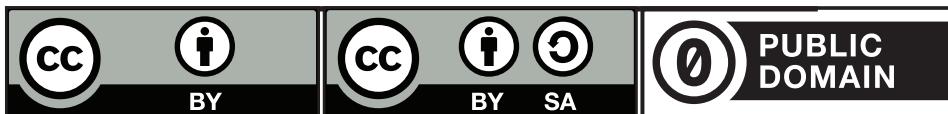
Providing a free version of their product that satisfies a lot of their customers' needs has actually enabled the Noun Project to build the paid version, using a service-oriented

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](http://www.kickstarter.com/projects/tnp/building-a-free-collection-of-our-worlds-visual-sy)/description
- 2 [thenounproject.com/handbook/royalties/#getting\\_paid](http://thenounproject.com/handbook/royalties/#getting_paid)
- 3 [thenounproject.com/iconathon/](http://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](http://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.<sup>1</sup>

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.)<sup>2</sup>

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.<sup>3</sup>

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.<sup>4</sup>

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-

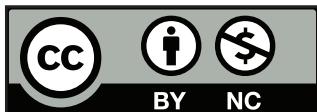
dicators. Here are a few metrics as of April 27, 2016:

- 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
- 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](http://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.<sup>1</sup>

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.<sup>2</sup>

---

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)<sup>3</sup>

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.<sup>4</sup> 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.

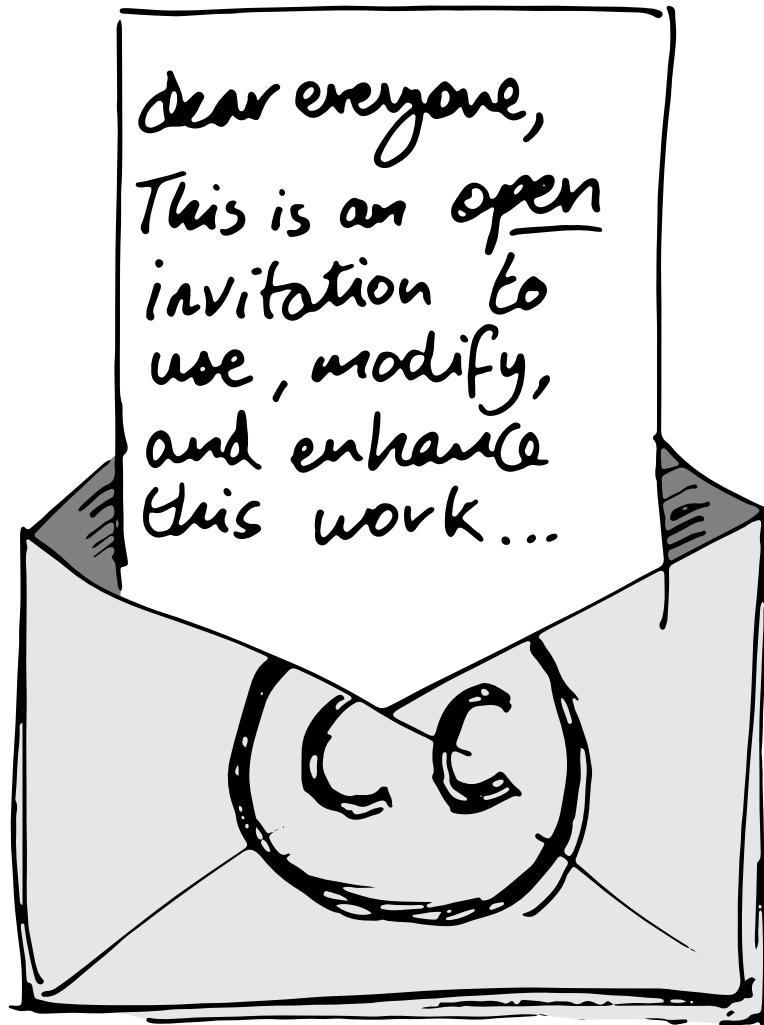
---

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](http://www.opendesk.cc/designers)
- 2 [www.opendesk.cc/open-making/makers/](http://www.opendesk.cc/open-making/makers/)
- 3 [www.opendesk.cc/open-making/join](http://www.opendesk.cc/open-making/join)
- 4 [openmaking.is](http://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](http://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.<sup>1</sup> 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.<sup>2</sup>

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](http://news.rice.edu/files/2016/01/0119-OPENSTAX-2016Infographic-lg-1tahxiu.jpg)
- 2 [openstax.org/adopters](http://openstax.org/adopters)

# AMANDA PALMER



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

[amandapalmer.net](http://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.<sup>1</sup> 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](http://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.<sup>1</sup> 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.<sup>2</sup> 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](http://collections.plos.org)
- 2 [plos.org/article-level-metrics](http://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](http://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.<sup>1</sup> 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.<sup>2</sup>

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*.<sup>3</sup>

In 2013 the Rijksmuseum organized their first high-profile design competition, known as the Rijksstudio Award.<sup>4</sup> 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.<sup>5</sup> 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](http://www.europeana.eu/portal/en)
- 2 [www.rijksmuseum.nl/en/rijksstudio](http://www.rijksmuseum.nl/en/rijksstudio)
- 3 [www.etsy.com/ca/listing/175696771/fringe-kimono-silk-kimono-kimono-robe](http://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;](http://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](http://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](http://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](http://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.<sup>1</sup> 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.<sup>2</sup> 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.<sup>3</sup> 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.<sup>4</sup> 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](http://www.gnu.org/licenses/fdl)
- 2 [www.capetowndeclaration.org](http://www.capetowndeclaration.org)
- 3 [cnx.org](http://cnx.org)
- 4 [www.siyavula.com/products-primary-school.html](http://www.siyavula.com/products-primary-school.html)

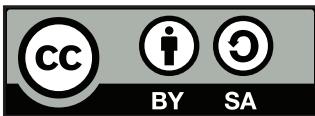
# BUSINESS

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](http://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](http://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.

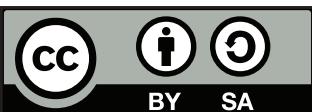
Importantly, the value to corporate sponsors goes beyond commercial considerations. As a nonprofit with a clearly articulated social mission, corporate sponsorships are donations to a cause. "This is something companies can be proud of internally," Shuman said. Some companies have even built public-

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](http://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.<sup>1</sup>

---

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  $US\$12 * 1000 \text{ stores} * 0.425 * 0.0143 = US\$73 \text{ per month}$ .<sup>2</sup>

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.

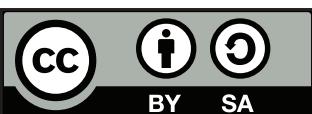
---

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](http://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](http://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.<sup>1</sup>

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/](http://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](http://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](http://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/](http://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](http://bollier.org/democratic-money-and-capital-commons-report-pdf). For more information, see [bollier.org/blog/democratic-money-and-capital-commons](http://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/](http://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](http://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/](http://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](http://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](http://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](http://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](http://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](http://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](http://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](http://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](http://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](http://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/](http://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](http://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/](http://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 Umanathan, 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, Mattias 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





