

Preface	4
Chapter 1 Introduction	9
Exercises - IntrotoPicobot	23
Exercises - IntrotoPicobot2	28
Exercises - PicobotDiamond	30
Exercises - PicobotStalactite	32
Exercises - PicobotPebbles	34
Chapter 2 Functional Programming	36
Exercises - PythonWelcome	63
Exercises - FunctionFun	84
Exercises - InteractiveFiction	90
Exercises - RockPaperScissors	94
Exercises - SequencesData	98
Exercises - TurtleGraphics	105
Exercises - FunctionFrenzy	120
Exercises - SleepwalkingStudent	132
Exercises - MoreTurtle	140
Exercises - ReadItandWeep	142
Exercises - RecursionMuscles	144
Exercises - GooglesSecret	149
Exercises - MakingChange	154
Chapter 3 Functional Programming- Part Deux	158
Exercises - CaesarSorting	178
Exercises - Integrals	187
Exercises - ScrabbleWords	196
Exercises - SoundsGood	200
Exercises - HigherOrderFunctions	213
Exercises - 42andMe	215
Exercises - RNAFolding	219
Exercises - WordBreak	223

Chapter 4 Computer Organization	227
Exercises - Binary	258
Exercises - AdditionCircuits	268
Exercises - ChangingBases	277
Exercises - Division	285
Exercises - MultiplicationMadness	289
Exercises - RecursivePower	293
Exercises - MemoryMadness	297
Exercises - HmmCountdown	301
Exercises - Randhmmm	307
Exercises - HmmmPower	314
Exercises - FibonacciFun	316
Exercises - ImageCompression	318
Exercises - RecursiveFibonacci	322
Exercises - Cryptography	326
Exercises - Ackerman	328
Exercises - TowersofHanoi	332
RecursionExamples	335
HMM advanced topics	338
HMM Documentation	344
Chapter 5 Imperative Programming	352
Exercises - PythonBat	384
Exercises - LoopingBack	386
Exercises - PifromPie	392
Exercises - TTSecurities	395
Exercises - 2DGameboard	403
Exercises - ASCIIArt	410
Exercises - GameofLife	415
Exercises - Mandelbrot	426
Exercises - SpellingatMillisoft	439

Exercises - MarkovText1	445
Exercises - MarkovText2	454
Exercises - Quadtrees	459
Chapter 6 Fun and Games with OOPs Object-Oriented Programs	465
Chapter 7 How Hard is the Problem	485
Index	498

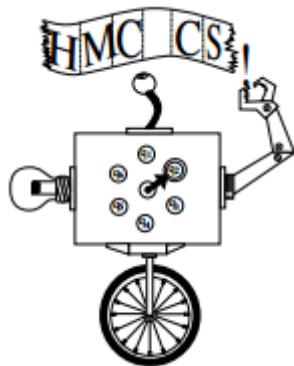
CS for All — cs5book 1 documentation

Navigation

- [index](#)
- [next](#) |
- [cs5book 1 documentation »](#)

CS for All

Christine Alvarado (UC San Diego), Zachary Dodds (Harvey Mudd), Geoff Kuenning (Harvey Mudd), Ran Libeskind-Hadas (Harvey Mudd)



To The Reader

Welcome! This book (and course) takes a unique approach to “Intro CS.” In a nutshell, our objective is to provide an introduction to *computer science* as an intellectually rich and vibrant field rather than focusing exclusively on *computer programming*. While programming is certainly an important and pervasive element of our approach, we emphasize concepts and problem-solving over syntax and programming language features.



Our name is Mudd!

This book is a companion to the course “CS for All” developed at Harvey Mudd College. At Mudd, this course is taken by almost every first-year student—irrespective of the student’s ultimate major—as part of our core curriculum. Thus, it serves as a first computing course for future CS majors and a first and last computing course for many other students. The course also enrolls a significant number of students from the other Claremont Colleges, many of whom are not planning to major in the sciences or engineering. At other schools, versions of this course have also been taught to students with varying backgrounds and interests.

The emphasis on problem-solving and big ideas is evident beginning in the introductory chapter, where we describe a very simple programming language for controlling the virtual “Picobot” robot. The syntax takes ten minutes to master but the computational problems posed here are deep and intriguing.

The remainder of the book follows in the same spirit. We use the Python language due to the simplicity of its syntax and the rich set of tools and packages that allow a novice programmer to write useful programs. Our introduction to programming with Python in Chapter 2 uses only a limited subset of the language’s syntax in the spirit of a functional programming language. In this approach, students master recursion early and find that

they can write interesting programs with surprisingly little code. Chapter 3 takes another step in functional programming, introducing the concept of higher-order functions.

Chapter 4 addresses the question “How does my computer do all this?” We examine the inner-workings of a computer, from digital logic through the organization of a machine and programming the machine in its native machine and assembly language.

Now that the computer has been demystified and students have a physical representation of what happens “under the hood,” we move on in Chapter 5 to explore more complex ideas in computation and, concomitantly, concepts such as references and mutability, and constructs including loops, arrays, and dictionaries. We explain these concepts and constructs using the physical model of the computer introduced in the previous chapter. In our experience, students find these concepts are much easier to comprehend when there is an underlying physical model.

Chapter 6 explores some of the key ideas in object-oriented programming and design. The objective here is *not* to train industrial-strength programmers but rather to explain the rationale for the object-oriented paradigm and allow students to exercise some key concepts. Finally, Chapter 7 examines the “hardness” of problems—providing a gentle but mathematically sound treatment of some of the ideas in complexity and computability and ultimately proving that there are many computational problems that are impossible to solve on a computer. Rather than using formal models of computation (e.g., Turing Machines), we use Python as our model.

This book is intended to be used with the substantial resources that we have developed for the course, which are available on the Web at <https://www.cs.hmc.edu/twiki/bin/view/ModularCS1>. These resources include complete lecture slides, a rich collection of weekly assignments, some accompanying software, documentation, and papers that have been published about the course.



New! Improved! With many “marginally” useful comments!

We have kept this book relatively short and have endeavored to make it fun and readable. The content of this book is an accurate reflection of the content of the course rather than an intimidating encyclopedic tome that can’t possibly be covered in a single semester. We have written this book in the belief that a student can read all of it comfortably as the course proceeds. In an effort to keep the book short (and hopefully sweet), we have *not* included the exercises and programming assignments in the text but rather have posted these on the course Web site.

We wish you happy reading and happy computing!

Acknowledgements

The authors gratefully acknowledge support from the National Science Foundation under CPATH grant 0939149 for supporting many aspects of the development of the “CS For All” course. This book benefitted significantly from feedback from many Harvey Mudd students over the past several years. Professor Dan Hyde at Bucknell University provided detailed comments that have substantially improved this book. In addition, Professor Richard Zacccone at Bucknell University, Professors David Naumann and Dan Duchamp at the Stevens Institute of Technology, Dr. Dave Sullivan at Boston University, Mr. Eran Segev, and several anonymous reviewers have provided many valuable comments and suggestions. While we’ve tried hard to be accurate and correct, all errors in this text are solely the responsibility of the authors. Finally, the authors thank Professors Brad Miller and David Ranum at Luther College who developed the Runestone Interactive ebook tools, Harvey Mudd students Akhil Bagaria, Alison Kingman, and Sarah Trisorus for “runestoning” our manuscript, and Mr. Tim Buchheim for system administration support.

Table of Contents

Introduction

- Chapter 1: Introduction
 - 1.1 What is Computer Science?
 - * 1.1.1 Data
 - * 1.1.2 Algorithms
 - * 1.1.3 Programming

- * 1.1.4 Abstraction
- * 1.1.5 Problem Solving and Creativity
- 1.2 PicoBot
 - * 1.2.1 The Roomba Problem
 - * 1.2.2 The Environment
 - * 1.2.3 State
 - * 1.2.4 Think locally, act globally
 - * 1.2.5 Whatever
 - * 1.2.6 Algorithms and Rules
 - * 1.2.7 The Picobot challenge
 - * 1.2.8 A-Maze Your Friends!
 - * 1.2.9 Uncomputable environments

Functional Programming

- Chapter 2 : Functional Programming
 - 2.1 Humans, Chimpanzees, and Spell Checkers
 - 2.2 Getting Started in Python
 - * 2.2.1 Naming Things
 - * 2.2.2 What's in a Name?
 - 2.3 More Data: From Numbers to Strings
 - * 2.3.1 A Short Note on Length
 - * 2.3.2 Indexing
 - * 2.3.3 Slicing
 - * 2.3.4 String Arithmetic
 - 2.4 Lists
 - * 2.4.1 Some Good News!
 - 2.5 Functioning in Python
 - * 2.5.1 A Short Comment on Docstrings
 - * 2.5.2 An Equally Short Comment on Comments
 - * 2.5.3 Functions Can Have More Than One Line
 - * 2.5.4 Functions Can Have Multiple Arguments
 - * 2.5.5 Why Write Functions?
 - 2.6 Making Decisions
 - * 2.6.1 A second example
 - * 2.6.2 Indentation
 - * 2.6.3 Multiple Conditions
 - 2.7 Recursion!
 - 2.8 Recursion, Revealed
 - * 2.8.1 Functions that Call Functions
 - * 2.8.2 Recursion, Revealed, Really!
 - 2.9 Building Recursion Muscles
 - 2.10 Use It Or Lose It
 - 2.11 Edit distance!
 - 2.12 Conclusion
- Chapter 3: Functional Programming, Part Deux
 - 3.1 Cryptography and Prime Numbers
 - 3.2 First-Class Functions
 - 3.3 Generating Primes
 - 3.4 Filtering
 - 3.5 Lambda
 - 3.6 Putting Google on the Map!
 - * 3.6.1 Map
 - * 3.6.2 Reduce
 - * 3.6.3 Composition and MapReduce
 - 3.7: Functions as Results
 - * 3.7.1: Python Does Calculus!
 - * 3.7.2: Higher Derivatives
 - 3.8: RSA Cryptography Revisited
 - 3.9: Conclusion

Computer Organization

- Chapter 4: Computer Organization
 - 4.1 Introduction to Computer Organization
 - 4.2 Representing Information
 - * 4.2.1 Integers
 - * 4.2.2 Arithmetic
 - * 4.2.3 Letters and Strings
 - * 4.2.4 Structured Information
 - 4.3 Logic Circuitry
 - * 4.3.1 Boolean Algebra
 - * 4.3.2 Making Other Boolean Functions
 - * 4.3.3 Logic Using Electrical Circuits
 - * 4.3.4 Computing With Logic
 - * 4.3.5 Memory
 - 4.4 Building a Complete Computer
 - * 4.4.1 The von Neumann Architecture
 - 4.5 Hmmm
 - * 4.5.1 A Simple Hmmm Program
 - * How Does It Work?
 - * Trying It Out
 - * 4.5.2 Looping
 - * 4.5.3 Functions
 - * 4.5.4 Recursion
 - * Stacks
 - * Saving Precious Possessions
 - * 4.5.5 The Complete Hmmm Instruction Set
 - * 4.5.6 A Few Last Words
 - 4.6 Conclusion

Imperative Programming

- Chapter 5: Imperative Programming
 - 5.1 A Computer that Knows You (Better than You Know Yourself?)
 - * 5.1.1 Our Goal: A Music Recommender System
 - 5.2 Getting Input from the User
 - 5.3 Repeated Tasks—Loops
 - * 5.3.1 Recursion vs. Iteration at the Low Level
 - * 5.3.2 Definite Iteration: `for` loops
 - * 5.3.3 How Is the Control Variable Used?
 - * 5.3.4 *Accumulating* Answers
 - * 5.3.5 Indefinite Iteration: `while` Loops
 - * 5.3.6 `for` Loops vs. `while` Loops
 - * 5.3.7 Creating Infinite Loops On Purpose
 - * 5.3.8 Iteration Is Efficient
 - 5.4 References and Mutable vs. Immutable Data
 - * 5.4.1 Assignment by *Reference*
 - * 5.4.2 Mutable Data Types Can Be Changed Using Other Names!
 - 5.5 Mutable Data + Iteration: Sorting out Artists
 - * 5.5.1 Why Sort? Running Time Matters
 - * 5.5.2 A Simple Sorting Algorithm: Selection Sort
 - * 5.5.3 Why `selectionSort` Works
 - * 5.5.4 A Swap of a Different Sort
 - * 5.5.5 2D Arrays and Nested Loops
 - * 5.5.6 Dictionaries
 - 5.6 Reading and Writing Files
 - 5.7 Putting It All Together: Program Design
 - 5.8 Conclusion

Object-Oriented Programming

- Chapter 6: Fun and Games with OOPs: Object-Oriented Programs

- 6.1 Introduction
- 6.2 Thinking Objectively
- 6.3 The Rational Solution
- 6.4 Overloading
- 6.5 Printing an Object
- 6.6 A Few More Words on the Subject of Objects
- 6.7 Getting Graphical with OOPs
- 6.8 Robot and Zombies, Finally!
- 6.9 Conclusion

Problem “Hardness”

- Chapter 7: How Hard is the Problem?
 - 7.1 The Never-ending Program
 - 7.2 Three Kinds of Problems: Easy, Hard, and Impossible.
 - * 7.2.1 Easy Problems
 - * 7.2.2 Hard Problems
 - 7.3 Impossible Problems!
 - * 7.3.1 “Small” Infinities
 - * “Larger” Infinities
 - * 7.3.2 Uncomputable Functions
 - 7.4 An Uncomputable Problem
 - * 7.4.1 The Halting Problem
 - 7.5 Conclusion

Indices and tables

- *Index*
- *Search Page*

Table Of Contents

- CS for All
 - To The Reader
 - Acknowledgements
 - Table of Contents
 - * Introduction
 - * Functional Programming
 - * Computer Organization
 - * Imperative Programming
 - * Object-Oriented Programming
 - * Problem “Hardness”
 - Indices and tables

Next topic Chapter 1: Introduction

Quick search

Enter search terms or a module, class or function name.

Navigation

- index
- next |
- cs5book 1 documentation »

© Copyright 2013, hmc. Created using Sphinx 1.2b1.

Chapter 1: Introduction — cs5book 1 documentation

1 capture

10 Sep 2019

Aug	SEP	Oct
10		
2018	2019	2020

success

fail

About this capture

COLLECTED BY

Organization: Internet Archive

The Internet Archive discovers and captures web pages through many different web crawls. At any given time several distinct crawls are running, some for months, and some every day or longer. View the web archive through the Wayback Machine.

Collection: Live Web Proxy Crawls

Content crawled via the Wayback Machine Live Proxy mostly by the Save Page Now feature on web.archive.org.

Liveweb proxy is a component of Internet Archive's wayback machine project. The liveweb proxy captures the content of a web page in real time, archives it into a ARC or WARC file and returns the ARC/WARC record back to the wayback machine to process. The recorded ARC/WARC file becomes part of the wayback machine in due course of time.

TIMESTAMPS



The Wayback Machine - <https://web.archive.org/web/20190910144836/https://www.cs.hmc.edu/csforallbook/Introduction/Introduction.html>

Navigation

- [index](#)
- [next |](#)
- [previous |](#)
- [cs5book 1 documentation »](#)

Chapter 1: Introduction

Computer science is to the information revolution what mechanical engineering was to the industrial revolution.

—Robert Keller

1.1 What is Computer Science?

You might be uncertain about what computer science (CS) is, but you use it every day. When you use Google or your smartphone, or watch a movie with special effects, there's lots of CS in there. When you order a product over the Internet, there is CS in the web site, in the cryptography used to keep your credit card number secure, and in the way that FedEx routes their delivery vehicle to get your order to you as quickly as possible. Nonetheless, even computer scientists can struggle to answer the question "What *exactly* is CS?"

Many other sciences try to understand how things work: physics tries to understand the physical world, chemistry tries to understand the composition of matter, and biology tries to understand life. So what is computer science trying to understand? Computers? Probably not: computers are designed and built by humans, so their inner workings are known (at least to some people!).

Perhaps it's all about programming. Programming is indeed important to a computer scientist, just as grammar is important to a writer or a telescope is important to an astronomer. But nobody would argue that writing is about grammar or that astronomy is about telescopes. Similarly, programming is an important piece of computer science but it's not what CS is all about.

If we turn to origins, computer science has roots in disparate fields that include engineering, mathematics, and cognitive science, among others. Some computer scientists design things, much like engineers. Others seek new ways to solve computational problems, analyze their solutions, and prove that they are correct, much like mathematicians. Still others think about how humans interact with computers and software, which is closely related to cognitive science and psychology. All of these pieces are a part of computer science.

Zoogenesis refers to the origin of a particular animal species. Computational biology is a field that uses CS to help solve zoogenetic questions, among many others.

One theme that unifies (nearly) all computer scientists is that they are interested in the *automation of tasks* ranging from **artificial intelligence** to **zoogenesis**. Put another way, computer scientists are interested in finding solutions for a wide variety of computational problems. They analyze those solutions to determine their "goodness," and they implement the good solutions to create useful software for people to work with. This diversity of endeavors is, in part, what makes CS so much fun.

There are several important concepts at the heart of computer science; we have chosen to emphasize six of them: data, problem solving, algorithms, programming, abstraction, and creativity.

1.1.1 Data

That's Astronomical!

When you Google the words "pie recipe," Google reports that it finds approximately 38 million pages, ranked in order of estimated relevance and usefulness. Facebook has approximately 1 billion active users who generate over 3 billion comments and "Likes" each day. GenBank, a national database of DNA sequences used by biologists and medical researchers studying genetic diseases, has over 100 million genetic sequences with over 100 billion DNA base pairs. According to the International Data Corporation, in 2010 the size of our "Digital Universe" reached 1.2 zettabytes. How much is that? Jeffrey Heer, a computer scientist who specializes in managing and visualizing large amounts of data, puts it this way: A stack of DVDs that reached to the moon and back would store approximately 1.2 zettabytes of data.

Without computer science, all of this data would be junk. Searching for a recipe on Google, a friend on Facebook, or genes in GenBank would all be impossible without ideas and tools from computer science.

Doing meaningful things with data is challenging, even if we're not dealing with millions or billions of things. In this book, we'll do interesting things with smaller sets of data. But much of what we'll do will be applicable to very large amounts of data too.

1.1.2 Algorithms

Making Pie and Making \(\pi\)

When presented with a computational problem, our first objective is to find a computational solution, or "algorithm," to solve it. An *algorithm* is a precise sequence of steps for carrying out a task, such as ranking web pages in Google, searching for a friend on Facebook, or finding closely related genes in Genbank. In some cases, a single good algorithm is enough to launch a successful company (e.g., Google's initial success was due to its Page Rank algorithm).

Algorithms are commonly compared to recipes that act on their ingredients (the data). For example, imagine that an alien has come to Earth from a distant planet and has a hankering for some pumpkin pie. The alien does a Google search for pumpkin pie and finds the following:

I've come to Earth for pumpkin pie!

1. Mix 3/4 cup sugar, 1 tsp cinnamon, 1/2 tsp salt, 1/2 tsp ginger and 1/4 tsp cloves in a small bowl.
2. Beat two eggs in a large bowl.
3. Stir 1 15-oz. can pumpkin and the mixture from step 1 into the eggs.
4. Gradually stir in 1 12 fl. oz. can evaporated milk into the mixture.
5. Pour mixture into unbaked, pre-prepared 9-inch pie shell.
6. Bake at 425°F for 15 minutes.
7. Reduce oven temperature to 350°F.
8. Bake for 30-40 minutes more, or until set.
9. Cool for 2 hours on wire rack.

No! Don't lick the spoon - there are raw eggs in there!

Assuming we know how to perform basic cooking steps (measuring ingredients, cracking eggs, stirring, licking the spoon, etc.), we could make a tasty pie by following these steps precisely.

Out of respect for our gastronomical well-being, computer scientists rarely write recipes (algorithms) that have anything to do with food. As a computer scientist, we would be more likely to write an algorithm to calculate π very precisely than we would be to write an algorithm to make a pie. Let's consider just such an algorithm:

1. Draw a square that is 2 by 2 feet.
2. Inscribe a circle of radius 1 foot (diameter 2 feet) inside this square.
3. Grab a bucket of n darts, move away from the dartboard, and put on a blindfold.

Please don't try this at home!

4. Take each dart one at a time and for each dart:
 - (a) With your eyes still covered, throw the dart randomly (but assume that your throwing skills ensure that it will land somewhere on the square dartboard).
 - (b) Record whether or not the dart landed inside the circle.
5. When you have thrown all the darts, divide the number that landed inside the circle by the total number, n , of darts you threw and multiply by 4. This will give you your estimate for π .

Figure 1.1 shows the scenario.

Figure 1.1: Using a dartboard to approximate π

Hey, watch it! That dart almost hit me!

That's the description of the algorithm, but why does it work? Here's why: The area of the circle is πr^2 which is π in this case because we made the radius of the board to be 1. The area of the square is 4. Since we're assuming that darts are equally likely to end up anywhere in the square, we expect the proportion of them that land in the circle to be the ratio of the area of the circle to the area of the square: $\frac{\pi}{4}$. Therefore, if we throw n darts and determine that some number k land inside the circle, then $\frac{k}{n}$ should be approximately $\frac{\pi}{4}$. So multiplying the ratio by 4 gives us an approximation of π .

Happily, the computer does not have to robotically throw physical darts; instead we can simulate this dart throwing process on a computer by generating random coordinates that describe where the darts land. The computer can throw millions of virtual darts in a fraction of a second and will never miss the square—making things considerably safer for your roommate!

1.1.3 Programming

Although we noted earlier that computer science is not exclusively about programming, ultimately we usually want to have a program—that is, software—that implements the algorithm that will operate on our data.

Learning to program is a bit like learning to speak or write in a new language. The good news is that the *syntax* of a programming language—the vocabulary and grammar—is not nearly as complicated as for a spoken language. In this book, we'll program in a language called Python, whose syntax is particularly easy to learn. But don't be fooled into thinking it's not a real programming language—Python is a very real language used

by real programmers to write real software. Moreover, the ideas that you'll learn here will be transferable to learning other languages later.

1.1.4 Abstraction

While data, algorithms, and programming might seem like the whole story, the truth is that there are other important ideas behind the scenes. Software is often immensely complex and it can be difficult or even impossible for any single person to keep all of the interacting pieces in mind. To deal with such complex systems, computer scientists use the the notion of *abstraction*—the idea that when designing one part of a program, we can ignore the inessential details of other parts of the program as long as we have a high level understanding of what they do.

For example, a car has an engine, a drivetrain, an electrical system, and other components. These components can be designed individually and then assembled to work together. The designer of the drivetrain doesn't need to understand every aspect of how the engine works, but just enough to know how the drivetrain and the engine will be connected. To the drivetrain designer, the engine is an “abstraction.” In fact, the engine itself is divided into components such as the engine block, distributor, and others. These parts too can be viewed as abstract entities that interact with one another. When designing the engine block, we don't need to think about every detail of how the distributor works.

Software systems can be even more complicated than a car. Designing software requires that we think about abstractions in order to ensure that many people can contribute to the project without everyone needing to understand everything, in order to test the software methodically, and in order to be able to update it in the future by simply replacing one “component” by a new and improved component. Abstraction, therefore, is a key idea in the design of any large system, and software in particular.

1.1.5 Problem Solving and Creativity

This book strives to prepare you to write well-designed programs that do interesting things with data. In the process, we hope to convey to you that computer science is an enormously creative endeavor that requires innovative problem-solving, exploration, and even experimentation. Often times, there's more than one way to solve a problem. In some cases there's not even a clear “best” way to solve a problem. Different solutions will have different merits. While Google, Facebook, GenBank are wonderfully easy to use, many challenges arose—and continue to arise—in the design and continual updating of such systems. These challenges often lead to groups of computer scientists working together to find different solutions and evaluate their relative merits. While the challenges that we'll confront in this book are of a more modest scope, we hope to share with you the sense of problem solving and creativity that are at the heart of computer science.

Takeaway message: *In a nutshell, the objective of this book is to demonstrate the breadth of activities that comprise computer science, show you some fundamental and beautiful ideas, and provide you with the skills to design, implement, and analyze your own programs.*

1.2 PicoBot

Leap before you look.

—W.H. Auden

The best way for you to get a feel for computer science is to jump right in and start solving a computer science problem. So let's do just that. In this section, we'll examine solutions to an important problem: How to make sure you'll never have to clean—or at least vacuum—your room again. To solve this problem we'll use a simple programming language named Picobot that controls a robot loosely based on the Roomba vacuum cleaner robot.

This web site offers a simulation environment for exploring Picobot's capabilities

You're probably wondering what happened to Python, the programming language we said we would be using throughout this book. Why are we sweeping Python under the carpet and brushing aside the language that we plan to use for the remainder of the book? The answer is that although Python is a simple (but powerful!) programming language that's easy to learn, Picobot is an *even simpler* language that's *even easier* to learn. The entire language takes only a few minutes to learn and yet it allows you to do some very powerful and interesting computation. So, we'll be able to start some serious computer science before we get sucked into a discussion of a full-blown programming language. This will be new and fun—and whether you have programmed before, it should offer a “Eureka!” experience. So, dust off your browser and join us at <http://www.cs.hmc.edu/picobot>.

Or, at least, the “breakout” app that enable the industry's first large-scale profits.



An iRobot Roomba. You'll notice that we use the word "Picobot" to refer to both the Roomba robot and the language that we will use to program it. Actually, Picobot might not be able to actually "see" at all. Instead, it might sense its environment through one of many possible sensors including bump sensors, infrared, camera, lasers, etc.

1.2.1 The Roomba Problem

It is the humblest of tasks—cleaning up—that has turned out to be the “killer app” for household robots. Imagine yourself as a Roomba vacuum named Picobot: your goal is to suck up the debris from the free space around you—ideally without missing any nooks or crannies. The robotics community calls this *the coverage problem*: it is the task of ensuring that all the grass is mown, all the surface receives paint, or all the Martian soil is surveyed.

At first this problem might seem pretty easy. After all, if your parents gave you a vacuum cleaner and told you to vacuum your room without missing a spot, you'd probably do a pretty great job without even thinking too much about it. Shouldn't it be straightforward to convey your strategy to a robot?

Unfortunately, there are a couple of obstacles that make the Picobot's job considerably more difficult than yours. First, Picobot has very limited “sight”; it can only sense what's directly around it. Second, Picobot is totally unfamiliar with the environment it is supposed to clean. While you could probably walk around your room blindfolded without crashing into things, Picobot is not so lucky. Third, Picobot has a very limited memory. In fact, it can't even remember which part of the room it has seen and which part it has not.

While these challenges make Picobot's job (and our job of programming Picobot) more difficult, they also make the coverage problem an interesting and non-trivial computer science problem worth serious study.

1.2.2 The Environment

“Discretize” is CS-speak for “break up into individual pieces”.

Our first task in solving this problem is to represent it in a way that the computer can handle. In other words, we need to define the data we will be working with to solve this problem. For example, how will we represent where the obstacles in the room are? Where Picobot is? We could represent the room as a plane, and then list the coordinates of the object's corners and the coordinates of Picobot's location. While this representation is reasonable, we will actually use a slightly simpler approach.

Whether lawn or sand, an environment is simpler to cover if it is discretized into cells as shown in Figure 1.2. This is our first example of an abstraction: we are ignoring the details of the environment and simplifying it into something we can easily work with. You, as Picobot, are similarly simplified: you occupy one grid square (the green one), and you can travel one step at a time in one of the four compass directions: north, east, west, or south.

Picobot cannot travel onto obstacles (the blue cells—which we will also call—“walls”); as we mentioned above, it does not know the positions of those obstacles ahead of time. What Picobot can sense is its immediate surroundings: the four cells directly to its north, east, west, or south. The surroundings are always reported as a string of four letters in “NEWS” order, meaning that we first see what is in our neighboring cell to the North, next what's to the East, then West, and finally South. If the cell to the north is empty, the letter in the first position is an x. If the cell to the north is occupied, the letter in that first position is an N. The second letter, an x or an E, indicates whether the eastern neighbor is empty or occupied; the third, x or W, is the west; the fourth, x or S, is the south. At its position in the lower-left-hand corner of Figure 1.2, for example, Picobot's sensors would report its four-letter surroundings as xxWS. There are sixteen possible surroundings for Picobot, shown in Figure 1.3 with their textual representations.

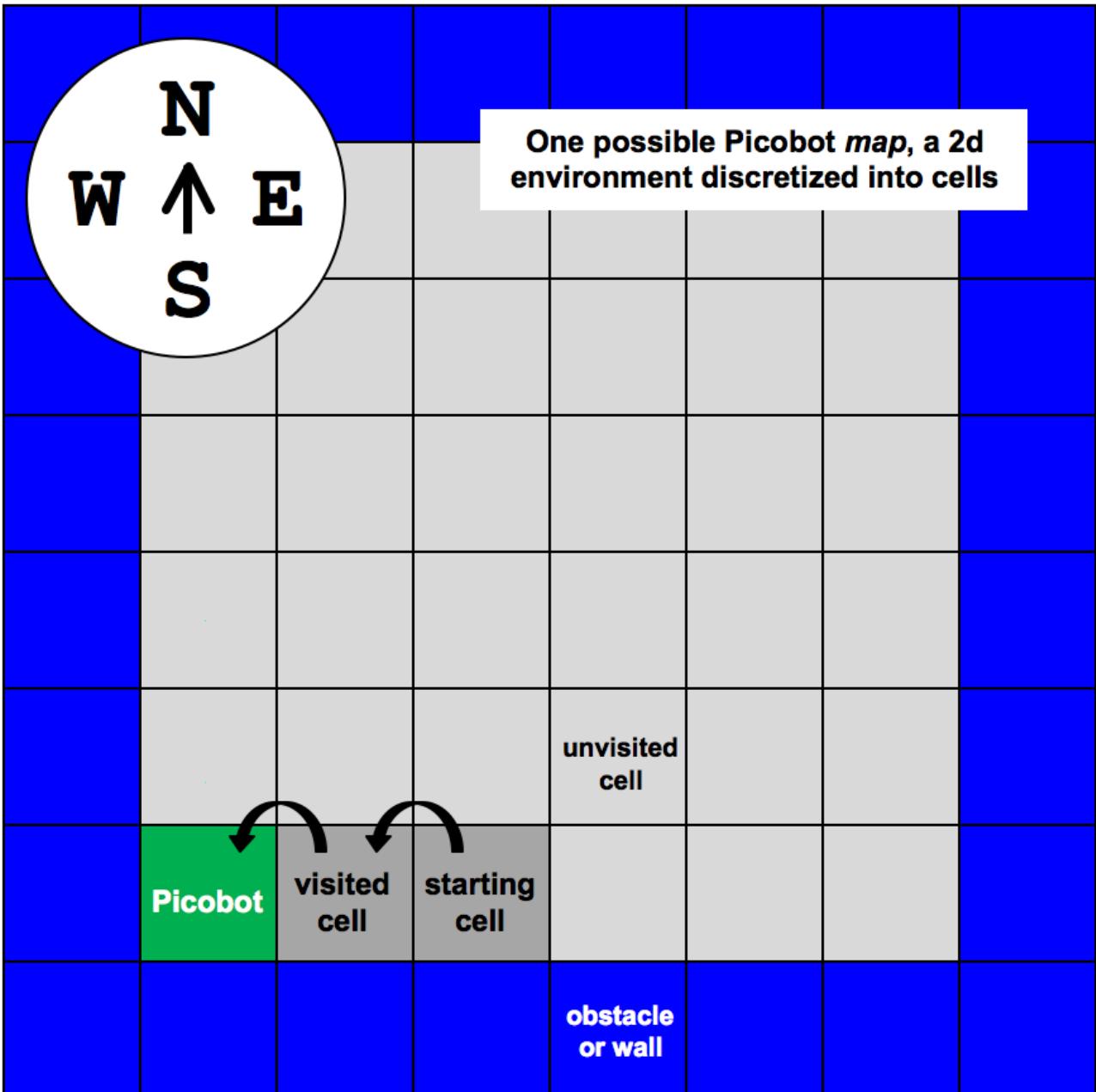


Figure 1.2: There are four types of cells in a Picobot environment, or map: green is Picobot itself, blue cells are walls, and gray cells are free space. Picobot can't sense whether an empty cell has been visited or not (dark or light gray), but it can sense whether each of its four immediate neighbors is free space or an obstacle.

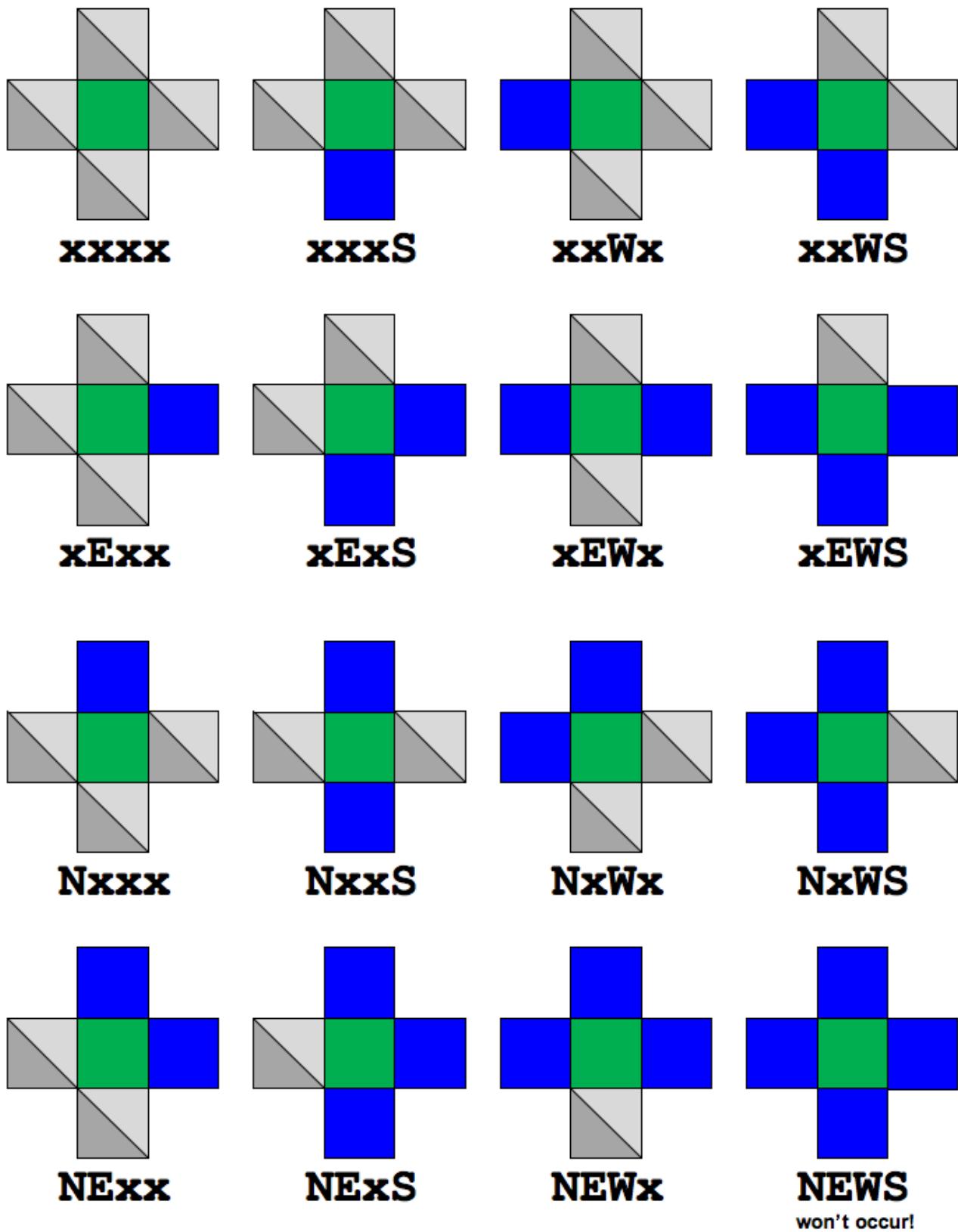


Figure 1.3: There are sixteen possible surroundings strings for Picobot. The one in which Picobot is completely enclosed will not occur in our simulator!

1.2.3 State

As we've seen, Picobot can sense its immediate surroundings. This will be important in its decision-making process. For example, if Picobot is in the process of moving north and it senses that the cell to its north is a wall, it should not try to continue moving north! In fact, the simulator will not allow it.

I'm currently in an inquisitive state.

But how does Picobot “know” whether it is moving north or some other direction? Picobot doesn’t have an innate sense of direction. Instead, we make use of a powerful concept called *state*. The state of a computer (or a person or almost any other thing) is simply its current condition: on or off, happy or sad, underwater or in outer space, etc. In computer science, we often use “state” to refer to the internal information that describes what a computer is doing.

Picobot’s state is extremely simple: it is a single number in the range 0-99. Somewhat surprisingly, that’s enough to give Picobot some pretty complex behaviors. **Picobot always starts in state 0.**

The state of anything can be described with a set of numbers.. but describing human states would take at least trillions of values

Although Picobot’s state is numeric, it’s helpful to think of it in English terms. For example, we might think of state 0 as meaning “I’m heading north until I can’t go any further.” However, it’s important to note that none of the state numbers has any special built-in meaning; it is up to us to make those decisions. Moreover, Picobot doesn’t actually have a sense of which directions it is pointing. But we can define our own conception of which direction Picobot is “pointing” by defining an appropriate set of states.

For example, imagine that Picobot wants to perform the task of continually moving north until it gets to a wall. We might decide that state 3 means “I’m heading north until I can’t go any further (and when I get to a wall to my north, then I’ll consider what to do next!).” When Picobot gets to a wall, it might want to enter a new state such as “I’m heading west until I can’t go any further (and when I get to a wall to my west, I’ll have to think about what to do then!).” We might choose to call that state 42 (or state 4; it’s entirely up to us).

Figure 1.4: The five parts of two Picobot rules. One useful way to interpret the idea of state is to attribute a distinct intention to each state. With these two rules, Picobot’s initial state (state 0) represents “go west as far as possible.”

As we’ll see next, your job as the Picobot programmer is to define the states and their meanings; this is what controls Picobot and makes it do interesting things!

Takeaway message: *The state is simply a number representing a task that you would like Picobot to undertake.*

1.2.4 Think locally, act globally

Now we know how to represent Picobot’s surroundings, and how to represent its state. But how do we make Picobot *do* anything?

Picobot moves by following a set of rules that specify actions and possibly state changes. Which rule Picobot chooses to follow depends on its current state and its current surroundings. Thus, Picobot’s complete “thought process” is as follows:

1. I take stock of my current state and immediate surroundings.
2. Based on that information, I find a rule that tells me (1) a direction to move and (2) the state I want to be in next.

Picobot uses a five-part rule to express this thought process. Figure 1.4 shows two examples of such rules.

The first rule,

0 xxWx -> E 1

re-expressed in English, says “If I’m in state 0 and only my western neighbor contains an obstacle, take one step east and change into state 1.” The second rule,

0 xxxx -> W 0

Go west, young Picobot!

says “If I’m in state 0 with no obstacles around me, move one step west and stay in state 0.” Taken together, these two rules use local information to direct Picobot across an open area westward to a boundary.

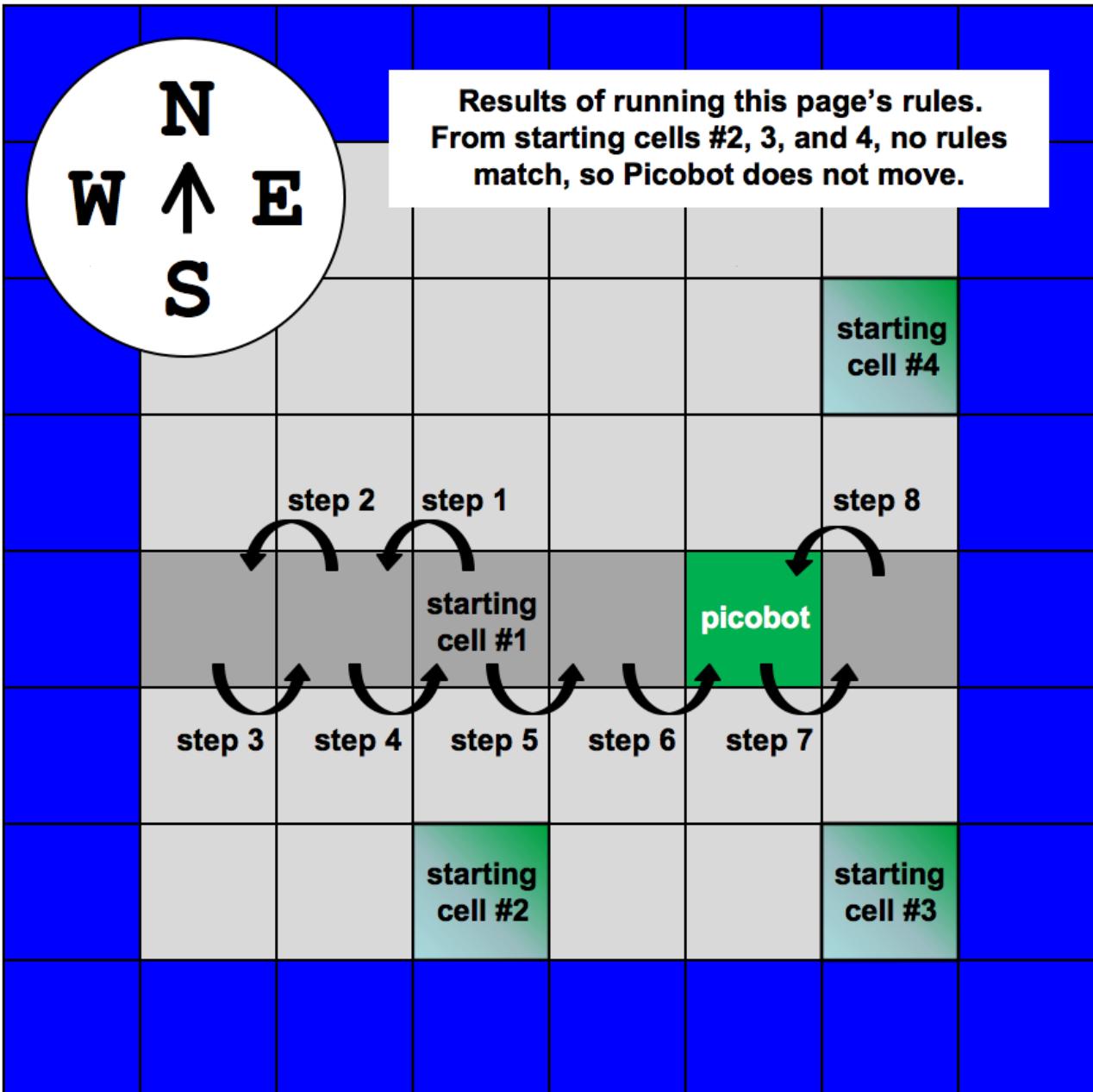


Figure 1.5: The result of running Picobot with this section’s four rules.

Remember that Picobot always begins its mission in state 0

At each step, Picobot examines the list of rules that you’ve written looking for the *one* rule that applies. A rule applies if the state part of the rule matches Picobot’s current state and the surroundings part of the rule matches the current surroundings. What happens if there are NO rules that match Picobot’s current state and surroundings? The Picobot simulator will let you know about this in its *Messages* box and the robot will stop running. Similarly, if more than one rule applies, Picobot will also complain. Figure 1.5 shows how Picobot follows the first rule that matches its current state and surroundings at each time step. But what about state 1? No rules specify Picobot’s actions in state 1-yet! Just as state 0 represents the “go west” task, we can specify two rules that will make state 1 be the “go east” task:

```
1 xxxx -> E 1
```

```
1 xExx -> W 0
```

Picobot cannot sense whether or not a cell has been visited. This limitation is quite realistic: the Roomba, for example, does not know whether a region has already been cleaned.

These rules transition back to state 0, creating an infinite loop back and forth across an open row. Try it out! Note that the Picobot website starts Picobot at a randomly selected empty cell. Note also that if Picobot starts along a top or bottom wall, no rules match and it does not move! We will remedy this defect in the next section.

Table 1.1: Two equivalent formulations of a more general “go-west-go-east” behavior for Picobot. Both sets of rules use only two states, but the wildcard character * allows for a much more succinct representation on the left than on the right!

By the way, sometimes you might not want Picobot to move as the result of applying a rule. Rather than specifying a move direction (“E”, “W”, “N”, or “S”), you may use the upper-case letter “X” to indicate “stay where you are”. For example, the rule

```
0 Nxxx -> X 1
```

is saying “if I’m in state 0 and there is a wall to the north, don’t move but enter state 1.”

1.2.5 Whatever

The problem with the previous “go-west-go-east” example is that the rules are too specific. When going west, we really don’t care whether or not walls are present to the north, south, or east. Similarly, when going east, we don’t care about neighboring cells to the north, south, or west. The wildcard character * indicates that we don’t care about the surroundings in the given position (N, E, W, or S). Table 1.1’s rules use the wildcard to direct Picobot to forever visit (vacuum) the east-west row in which it starts.

Picobot needs to get over its “don’t care” attitude!

1.2.6 Algorithms and Rules

So far we’ve looked at how to write rules that make Picobot move. But in trying to solve problems with Picobot, it’s usually helpful to take a more global view of how Picobot is accomplishing its task, and then to translate that approach into rules. In other words, we want to develop an algorithm that allows Picobot to accomplish the desired task, where that task is usually to cover the entire room. In the previous section, Picobot had the more modest goal of simply moving back and forth in an empty room. The algorithm for accomplishing this task was the following:

1. Move west until Picobot hits a wall to the west
2. Then move east until Picobot hits a wall to the east
3. Then go back to step 1

Now the question becomes: how do we translate this algorithm into the rules from the previous section:

```
0 **x* -> W 0
0 **W* -> E 1
1 *x** -> E 1
1 *E** -> W 0
```

As written, it is difficult to see the connection between the steps of the algorithm and the Picobot rules. We can see that Picobot will need two states to keep track of which direction it is moving (i.e., is it in step 1 or step 2), but it’s still not exactly clear how the algorithm translates into precise rules. Essentially, each of Picobot’s rules applies in an “if-then” fashion. In other words, if Picobot is in a particular state and sees a particular environment, then it takes a certain action and potentially enters a new state. With some minor modifications, we can rewrite the algorithm above to follow Picobot’s “if-then” rule structure more directly:

1. Repeat the following steps forever:
 - (a) If Picobot is moving west and there is no wall to the west, then keep moving west.
 - (b) If Picobot is moving west and there is a wall to the west, then start moving east.
 - (c) If Picobot is moving east and there is no wall to the east, then keep moving east.
 - (d) If Picobot is moving east and there is a wall to the east, then start moving west.

Now we can see more clearly the direct translation between the steps of this algorithm and the Picobot rules: each step in the algorithm translates directly into a rule in Picobot, where state 0 represents “Picobot is moving West” and state 1 represents “Picobot is moving East”. Formulating algorithms in this way is the key to writing successful programs in Picobot.

1.2.7 The Picobot challenge

This back-and-forth nwork saw euqinhcet to ancient Greek ox- ti dellac ohw srevird “boustrophedon.” Text in some classical nwonk si stpircsunam to show the same .nrettap

Table 1.1’s rules direct Picobot to visit the entirety of its starting row. This section’s challenge is to develop a set of rules that direct Picobot to cover the entirety of an empty rectangular room, such as the rooms in Figure 1.2 and 1.5. The set of rules—that is, your program—should work regardless of how big the room is and regardless of where Picobot initially begins.

Because Picobot does not distinguish already-visited from unvisited cells, it may not know when it has visited every cell. The online simulator, however, will detect and report a successful, complete traversal of an environment.

Try it out. You might find it helpful to simply play around with modifying the rules we’ve given you here. For example, you might start by altering the rules in Figure 1.1 so that they side-step into a neighboring row after clearing the current one. However, once you have an idea for how you might solve the problem, we encourage you to plan your algorithm, and then express that algorithm in a way that is easily translatable into Picobot rules.

Thank you for sparing us from any corny maize jokes.

1.2.8 A-Maze Your Friends!

Once you’ve developed a Picobot program that completely traverses the empty room, try to write other programs for more complex environments. You’ll see a “MAP” option on the Picobot Web page where you can scroll forward or backward through a collection of maps that we’ve created. You can also edit these maps by clicking on a cell with your mouse; clicking on an empty cell turns it into a wall and clicking on a wall turns it into an empty cell. *Remember that your program should work no matter where Picobot begins.*

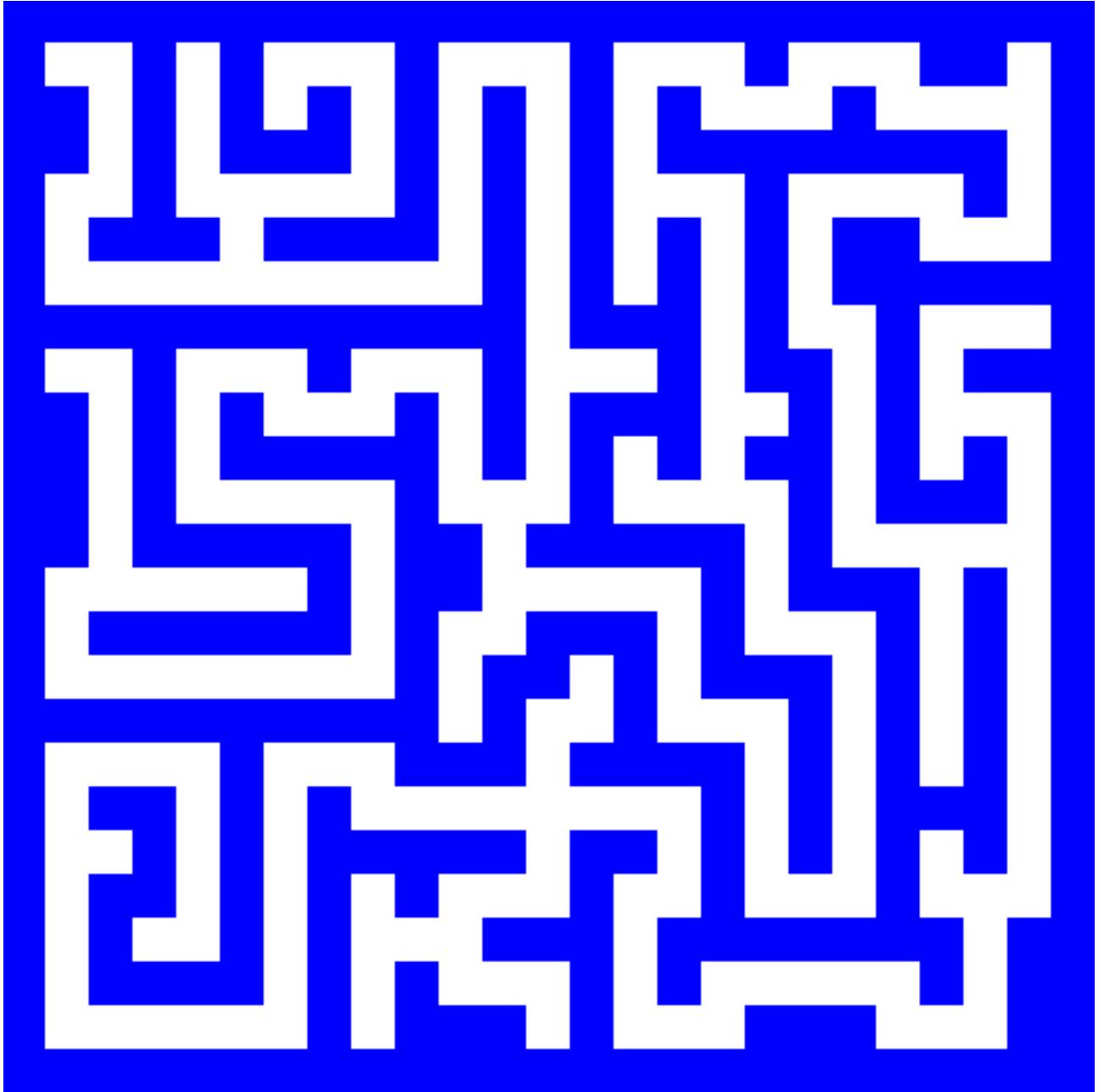


Figure 1.6: Picobot's maze.

One environment that is particularly interesting is the maze shown in Figure 1.6. Notice that in this maze, all the walls are connected to the outer boundary and all empty cells are adjacent to a wall. A smaller maze with this property is shown in Figure 1.7(a). Any maze with this property can be completely explored with a simple algorithm called the *right-hand rule* (or the *left-hand rule* if you prefer).

Imagine for a moment that you are in the maze rather than Picobot. In contrast to Picobot, you have a clear sense of the direction you're pointing and you have two hands. You start facing north with your right hand touching the wall. Now, you can visit every empty cell by simply walking through the maze, making sure that your right hand is always touching the wall. Pause here for a moment to convince yourself that this is true. Notice also that this algorithm will not visit every cell if some walls are not connected to the outer boundary, as shown in the maze in Figure 1.7(b) or if some empty cells are not adjacent to a wall, as shown in Figure 1.7(c).

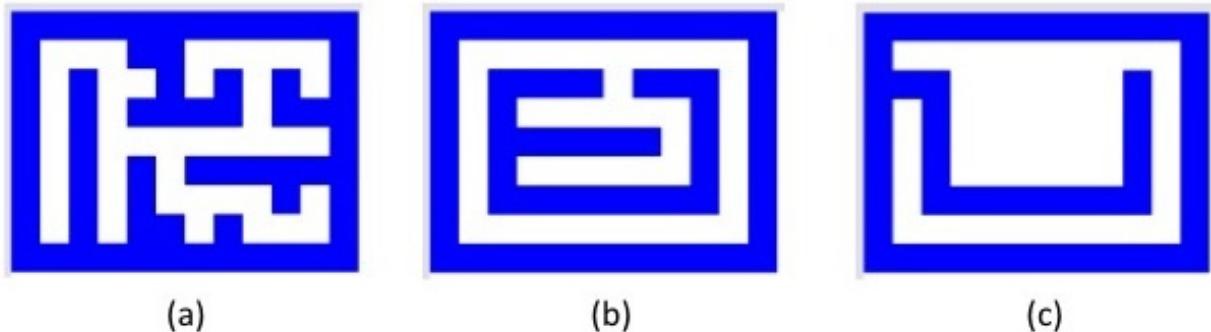


Figure 1.7: (a) A maze in which all walls are connected to the outer boundary and all empty cells are adjacent to a wall. (b) A maze in which some walls are not connected to the outer boundary. (c) A maze in which some empty cells are not adjacent to walls.

Converting the right-hand rule into a set of Picobot rules is an interesting computational challenge. After all, you have a sense of direction and you have a right hand that was guiding you around the walls, whereas Picobot has neither hands nor a sense of orientation. To “teach” Picobot the right-hand rule, we’ll again need to use states to represent the direction that Picobot is pointing. It may seem that an impossibly large number of situations must be considered, but in fact, the number of situations is finite and actually quite small, which makes it possible to program Picobot for this task.

To get started, it seems pretty natural to use the four states 0, 1, 2, and 3 to represent Picobot pointing north, south, east, or west. Now, we’ll need to introduce rules that allow Picobot to behave as if it had a right hand to touch against the wall.

Of course, all empty cells must be reachable. If some cells are isolated from others, the problem is just physically impossible.

Assume we are in state 0, which we (arbitrarily) choose to correspond to representing Picobot pointing north. Picobot’s imaginary right hand is then pointing east. If there is a wall to the east and none to the north, the right-hand rule would tell us to take a step to the north and keep pointing north. Taking a step to the north is no problem. “Keep pointing north” means “stay in state 0.” On the other hand, if we are in state 0 and there is no wall to the east, Picobot should take a step to the east and think of itself as pointing to the east. “Pointing east” will mean changing to another state that is intended to encode that information. This is a fun challenge and we encourage you to stop here and try it. (Remember, your program should work regardless of where Picobot starts and for any maze with the property that all walls are connected to the outer boundary and all empty cells are adjacent to a wall.)

1.2.9 Uncomputable environments

Is it possible to write a Picobot program that will fully explore any room that we give it? Surprisingly, the answer is “no,” and it’s possible to prove that fact mathematically. Picobot’s computational capabilities aren’t enough to guarantee coverage of all environments. However, by adding one simple feature to Picobot, it can be programmed to fully explore any room. That feature is the ability to drop, sense, and pick up “markers” along the way.

The fact that computational challenges as elementary as Picobot lead us to *provably unsolvable problems* suggests that computation and computers are far from omnipotent. And by the time you’re done reading this book, you’ll have learned how to prove that certain problems are beyond the limits of what computers can solve.

Table Of Contents

- Chapter 1: Introduction
 - 1.1 What is Computer Science?
 - * 1.1.1 Data
 - * 1.1.2 Algorithms
 - * 1.1.3 Programming
 - * 1.1.4 Abstraction
 - * 1.1.5 Problem Solving and Creativity
 - 1.2 PicoBot
 - * 1.2.1 The Roomba Problem
 - * 1.2.2 The Environment

- * 1.2.3 State
- * 1.2.4 Think locally, act globally
- * 1.2.5 Whatever
- * 1.2.6 Algorithms and Rules
- * 1.2.7 The Picobot challenge
- * 1.2.8 A-Maze Your Friends!
- * 1.2.9 Uncomputable environments

Previous topic [CS for All](#)

Next topic [Chapter 2 : Functional Programming](#)

Quick search

Enter search terms or a module, class or function name.

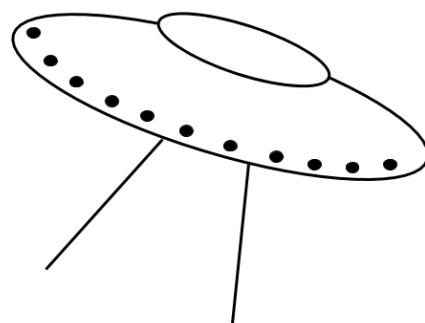
Navigation

- [index](#)
- [next |](#)
- [previous |](#)
- [cs5book 1 documentation »](#)

© Copyright 2013, hmc. Created using Sphinx 1.2b1.

CSforAll - IntrotoPicobot

CS for All



CSforAll Web > Chapter1 > IntrotoPicobot

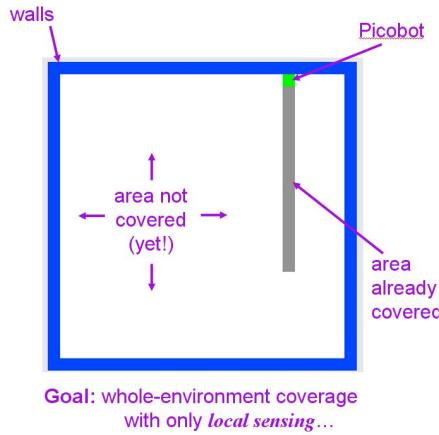
This website accompanies the textbook *CS For All* by Alvarado, Dodds, Kuennen, and Libeskind-Hadas

Introduction to Picobot!

This problem explores a simple "robot," named "picobot," whose goal is to completely traverse its environment.

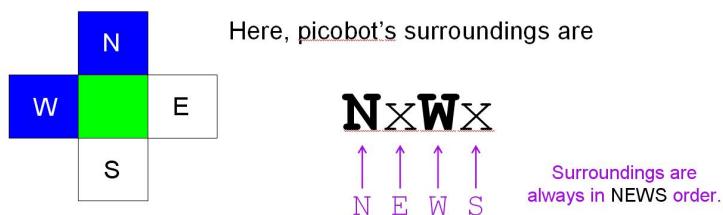
Here is a link to the Picobot page.

Picobot starts at a *random* location in a room—you don't have control over Picobot's initial location. The walls of the room are blue; picobot is green, and the empty area is white. Each time picobot takes a step, it leaves a grey trail behind it. When Picobot has completely explored its environment, it stops automatically.



Surroundings

Not surprisingly, picobot has limited sensing power. It can only sense its surroundings immediately to the north, east, west, and south of it.

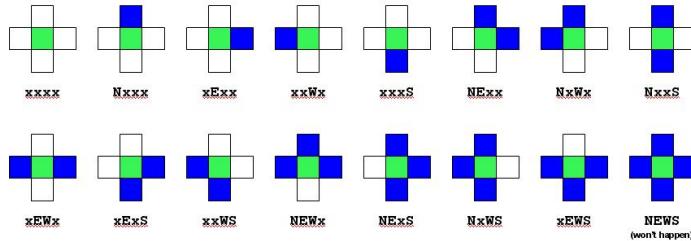


In the above image, Picobot sees a wall to the north and west and it sees nothing to the east or south. This set of surroundings would be represented as follows:

NxWx

The four squares surrounding picobot are always considered in NEWS order: an x represents empty space, the appropriate direction letter (N, E, W, and S)

represents a wall blocking that direction. Here are all of the possible picobot surroundings:



State

Picobot's memory is also limited. In fact, it has only a single value from 0 to 99 available to it. This number is called picobot's **state**. In general, "state" refers to the relevant context in which a computation takes place. Here, you might think of each "state" as one piece—or behavior—that the robot uses to achieve its overall goal.

Picobot always begins in state 0.

The state and the surroundings are all the information that picobot has available to make its decisions!

Rules

Picobot moves according to a set of rules of the form

`StateNow Surroundings -> MoveDirection NewState`

For example,

`0 xxxxS -> N 0`

is a rule that says "if picobot starts in state 0 and sees the surroundings `xxxxS`, it should move North and stay in state 0."

The `MoveDirection` can be `N`, `E`, `W`, `S`, or `X`, representing the direction to move or, in the case of `X`, the choice not to move at all.

If this were picobot's only rule and if picobot began (in state 0) at the bottom of an empty room, it would move up (north) one square and stay in state 0. However, **picobot would not move any further**, because its surroundings would have changed to `xxxx`, which does not match the rule above.

Wildcards

The asterisk * can be used inside surroundings to mean "I don't care whether there is a wall or not in that position." For example, `xE**` means "there is no wall North, there *is* a wall to the East, and there may or may not be a wall to the West or South."

As an example, the rule

```
0  x***  ->  N  0
```

is a rule that says "if picobot starts in state 0 and sees *any surroundings without a wall to the North*, it should move North and stay in state 0."

If this new version (with wildcard asterisks) were picobot's only rule and if picobot began (in state 0) at the bottom of an empty room, it would first see surroundings `xxxS`. These match the above rule, so picobot would move North and stay in state 0. Then its surroundings would be `xxxx`. These *also* match the above rule, so picobot would again move North and stay in state 0. In fact, this process would continue until it hit the "top" of the room, when the surroundings `Nxxx` no longer match the above rule.

Comments

Anything after a pound sign (#—you might think of it as a "hashtag" but computer scientists usually call it a pound sign) on a line is a comment (as in Python). Comments are human-readable explanations of what is going on, but are ignored by picobot. Blank lines are ignored as well.

An Example

Consider the following set of rules:

```
# state 0 goes N as far as possible
0 x*** -> N 0    # if there's nothing to the N, go N
0 N*** -> X 1    # if N is blocked, switch to state 1

# state 1 goes S as far as possible
1 ***x -> S 1    # if there's nothing to the S, go S
1 ***S -> X 0    # otherwise, switch to state 0
```

Recall that picobot always starts in state 0. Picobot now consults the rules from top to bottom until it finds the first rule that applies. It uses that rule to make its move and enter its next state. It then starts all over again, looking at the rules and finding the first one from the top that applies.

In this case, picobot will follow the first rule up to the "top" of its environment, moving north and staying in state 0 the whole time. Eventually, it encounters a wall to its north. At this point, the topmost rule no longer applies. However, the next rule " $0\ N^{***} \rightarrow X\ 1$ " does apply now! So, picobot uses this rule which causes it to stay put (due to the "X") and ***switch to state 1***. Now that it is in state 1, neither of the first two rules will apply. Picobot follows state 1's rules, which guide it back to the "bottom" of its environment. And so it continues....

The Assignment

For this assignment, your task is to design one set of rules that will allow picobot to completely cover an empty square room.

Remember to click on the "Enter rules for Picobot" before you try to run picobot.

Website design by Madeleine Masser-Frye and Allen Wu

CSforAll - IntrotoPicobot2

CS for All

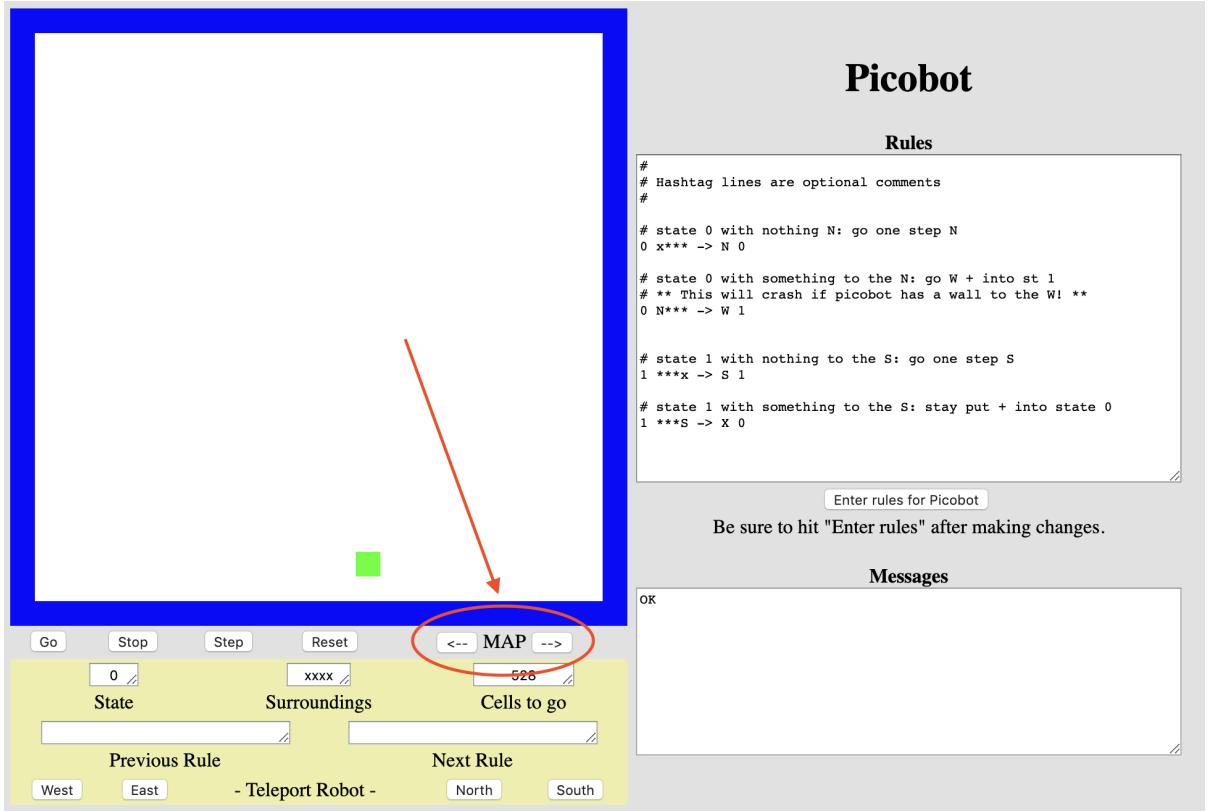
CSforAll Web > Chapter1 > IntrotoPicobot2

This website accompanies the textbook *CS For All* by Alvarado, Dodds, Kuenning, and Libeskind-Hadas

Introduction to Picobot (II)

Covered the empty room? Try some other picobot environments!

Click the arrows under the bottom right corner of the room to change the map picobot is navigating.



The Assignment

For this assignment, your task is to design one set of rules that will allow picobot to traverse a maze.

The second picobot room should be a maze where the width of all hallways is one square and all walls connect to the edge of the room. Your program should work on all mazes like this where no open spaces are completely enclosed.

Remember to click on the "Enter rules for Picobot" before you try to run picobot.

Website design by Madeleine Masser-Frye and Allen Wu

CSforAll - PicobotDiamond

CS for All

CSforAll Web > Chapter1 > PicobotDiamond

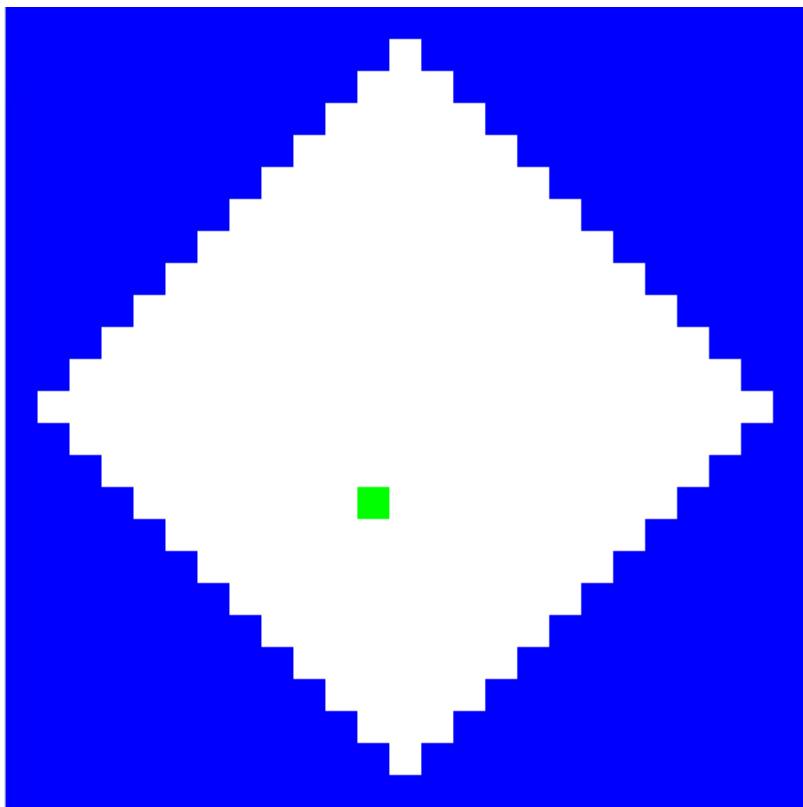
This website accompanies the textbook *CS For All* by Alvarado, Dodds, Kuenning, and Libeskind-Hadas

Picobot: Diamond Map

Other Picobot challenges exist, such as the diamond map. To access it use the MAP arrows under the bottom right of the picobot room.

The Assignment

Solve the "diamond" map that looks like this (with any number of rules):



Website design by Madeleine Masser-Frye and Allen Wu

CSforAll - PicobotStalactite

CS for All

CSforAll Web > Chapter1 > PicobotStalactite

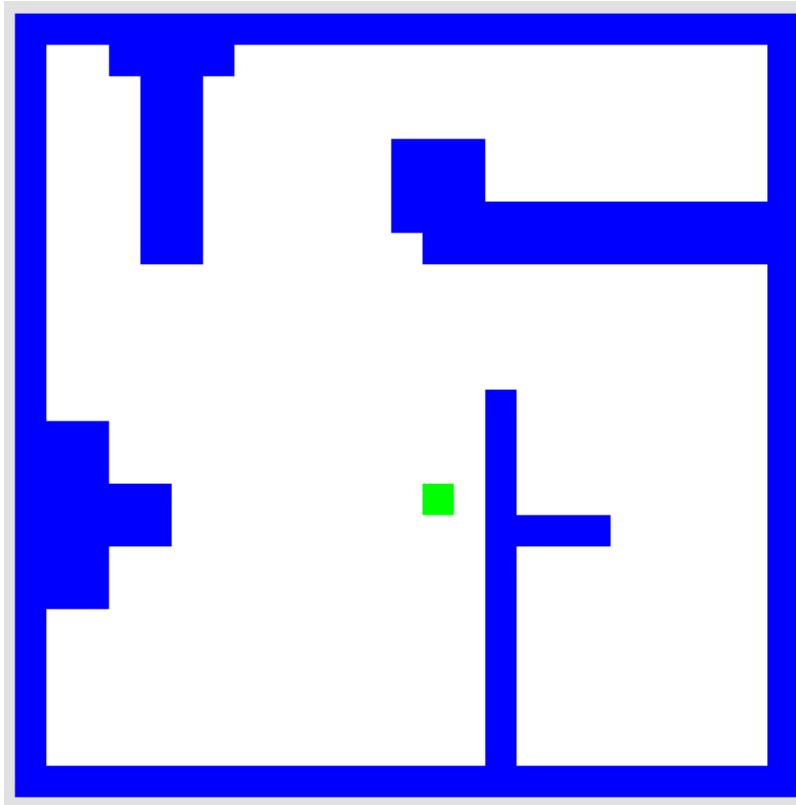
This website accompanies the textbook *CS For All* by Alvarado, Dodds, Kuenning, and Libeskind-Hadas

Picobot: Stalactite Map

Other Picobot challenges exist, such as the stalactite map. To access it use the MAP arrows under the bottom right of the picobot room.

The Assignment

Solving the "stalactite" map that looks like this one (with any number of rules):



Website design by Madeleine Masser-Frye and Allen Wu

CSforAll - PicobotPebbles

CS for All

CSforAll Web > Chapter1 > PicobotPebbles

This website accompanies the textbook *CS For All* by Alvarado, Dodds, Kuenning, and Libeskind-Hadas

Picobot with Pebbles

A version of Picobot with pebbles is available here.

This simulator recognizes regular Picobot commands of the form:

STATE SURROUNDINGS → MOVE NEW_STATE

Recall that Picobot always begins in state 0 but you may define new states numbered up to 99. The SURROUNDINGS are in NEWS format.

This augmented version of Picobot also allows commands that look like this:

STATE SURROUNDINGS PEBBLE → NEW_PEBBLE MOVE NEW_STATE

The PEBBLE can be one of the following:

- m means "do this rule only if there is a pebble on this cell."
- xm means "do this rule only if there NO pebble on this cell."
- Omitted, in which case it makes no difference whether or not there is a pebble here.

The NEW_PEBBLE can be one of the following:

- dm means "drop a pebble here."
- pm means "pick up the pebble here."
- Omitted, in which case no action is taken with a pebble.

Part 1

Consider a room that is some arbitrarily large rectangle. Inside that room are some arbitrary number of rectangular obstacles of arbitrary size. The obstacles are not connected to the outer boundary of the room and none of the obstacles

touch each other. (Remember, you can edit the room by clicking on cells to create and destroy wall cells.)

The Picobot starts in the northwest corner of the room. (The Picobot tool has buttons on the lower left that allow you to "Teleport" the Picobot in any direction. Be sure to teleport it to the northwest corner.)

The objective is to have the Picobot visit every cell in the room. A few pebbles will be needed, but try to use no more than five pebbles (it's possible to do it with three!).

This problem is conceptually challenging, but the program itself is not very long.

Part 2

Now, extend your program so that it can handle boundaries that are not necessarily rectangular and obstacles that are not necessarily rectangular either!

Website design by Madeleine Masser-Frye and Allen Wu

Chapter 2 : Functional Programming — cs5book 1 documentation

1 capture

10 Sep 2019

Aug	SEP	Oct
◀	10	▶
2018	2019	2020

success

fail

About this capture

COLLECTED BY

Organization: Internet Archive

The Internet Archive discovers and captures web pages through many different web crawls. At any given time several distinct crawls are running, some for months, and some every day or longer. View the web archive through the Wayback Machine.

Collection: Live Web Proxy Crawls

Content crawled via the Wayback Machine Live Proxy mostly by the Save Page Now feature on web.archive.org.

Liveweb proxy is a component of Internet Archive's wayback machine project. The liveweb proxy captures the content of a web page in real time, archives it into a ARC or WARC file and returns the ARC/WARC record back to the wayback machine to process. The recorded ARC/WARC file becomes part of the wayback machine in due course of time.

TIMESTAMPS



The Wayback Machine - <https://web.archive.org/web/20190910144901/https://www.cs.hmc.edu/csforall-book/FunctionalProgramming/functionalprogramming.html>

Navigation

- index
- next |
- previous |
- cs5book 1 documentation »

Chapter 2 : Functional Programming

2.1 Humans, Chimpanzees, and Spell Checkers

For many years, scientists were uncertain whether humans were more closely related to chimpanzees or gorillas. New technologies and clever computational methods have allowed us to resolve this issue in recent years: humans

are evidently more closely related to chimps than to gorillas. Humans and chimps diverged from their common ancestor approximately 4–6 million years ago, while gorillas diverged about 2 million years before that. How did we come to that conclusion? One of the primary methods involves computational analysis of DNA, the genetic code—or genome—that is essentially the “program” for all living creatures.

As you may recall from your biology class, DNA is a sequence of molecules fondly known as “A”, “T”, “C”, and “G”. Each living organism has a long sequence of these “letters” that make up its genome. Computer scientists refer to a sequence of symbols as a *string*.

Imagine that we look at a DNA string from the human genome and a string from the corresponding location of the chimp genome. For example, in the human we might see the string “ATTCG” and in the chimp we might see “ACTCG.” The human and chimp DNA differ in only one position (the second one). In contrast, imagine that the gorilla’s DNA string at the corresponding part of its genome is “AGGCG.” Notice that the gorilla differs from the human in two positions (the second and third) and also differs from the chimp in two positions (again, the second and third). So the gorilla exhibits more differences from the human and the chimp than those two species differ from one another. This kind of analysis (on a larger scale and with more complex ways of comparing differences) allows scientists to make inferences about when species diverged. In a nutshell, the key computational idea here is determining the level of similarity between two strings.

But how exactly do we measure the similarity between two strings? We’re glad you asked! Biologists know that there are three fundamental ways in which DNA changes over time. First, a single letter in the genome can change to another letter. This is called *substitution*. Second, a letter in the genome can be deleted, and third, a new letter can be inserted. A reasonable definition of “similarity” is to find the smallest number of substitutions, insertions, and deletions required to get from our first string to our second string. For example, to get from the DNA string “ATC” to the string “TG”, we can delete the “A” in “ATC” to get “TC”. Then we can change the “C” to a “G” to get “TG.” This took two operations—and that’s the best that we can do in this case. We say that the *edit distance* between these two strings is 2.



It seems that biology beat computer science to programming!

Interestingly, this problem also arises in spell checking. Many word processing programs have built-in spell checkers that will try to offer you a number of alternatives to a word that you’ve misspelled. For example, when we typed “spim” in one spell checker it offered us a list of alternatives that included “shim”, “skim”, “slim”, “spam”, “spin”, “spit”, “swim”, among others. You can probably see why those words were suggested: They are all legitimate English words that differ from “spim” by only a little bit. In general, a spell checker might check every word in its dictionary against the word that we typed in, measure the difference between those two words, and then show us a short list of the most similar words. Here again, we need a way to compute the similarity between two strings.

For example, consider the pair of strings “spam” and “poems”. One way to get from “spam” to “poems” is through the following sequence of operations: Delete “s” from “spam” resulting in “pam” (a deletion), replace the “a” with an “o” resulting in “pom” (a substitution), insert an “e” after the “o” resulting in “poem” (an insertion), and insert an “s” at the end of “poem” resulting in “poems” (another insertion). This took a total of four operations. Indeed, that’s the smallest number of operations required to get from “spam” to “poems.” In other words, the edit distance between “spam” and “poems” is 4. By the way, you’ll notice that we defined the edit distance to be the smallest number of operations needed to get from the first string to the second. A few moments of thought will convince you that this is exactly the same as the number of operations required to get from the second string to the first. In other words, edit distance is symmetric—it doesn’t depend on which of the two strings we start with.

Our ultimate goal in this chapter is to write a program to compute the edit distance. We’ll begin with the foundations of programming in the Python programming language and then explore a beautiful technique called “recursion.” Recursion will allow us to write short and very powerful computer programs to compute the edit distance between two strings—and many other useful things.

2.2 Getting Started in Python

Python is a programming language that, according to its designers, aims to combine “remarkable power with very clear syntax.” Indeed, in very short order you’ll be able to write programs that solve interesting and useful

problems. While we hope that this and later chapters will be pleasant reading, there's no better way to learn the material than by trying it yourself. Therefore, we recommend that you pause frequently and try some of the things that we're doing here on a computer. Moreover, this book is relatively short and to-the-point. To truly digest this material, it will be important for you to do the exercises on the book's Web site or assignments given to you by your instructor. Let's get started! When you start up Python, it presents you with a "prompt" that looks like this:

```
>>>
```

That's your invitation to type things in. For now, let's just type in arithmetic—essentially using Python as a calculator. (We'll do fancier things soon.) For example, below we've typed $3+5$.

```
>>> 3 + 5  
8
```

Next, we can do more complex things, like this:

```
>>> (3 + 5) * 2 - 1  
15
```

Notice that parentheses were used here to control the order of operations. Normally, multiplication and division have higher precedence than addition or subtraction, meaning that Python does multiplications and divisions first and addition and subtractions afterwards. So without parentheses we would have gotten:

```
>>> 3 + 5 * 2 - 1  
12
```

You can always use parentheses to specify your desired order of operations. Here are a few more examples of arithmetic in Python:

```
>>> 6 / 2  
3  
>>> 2 ** 5  
32  
>>> 10 ** 3  
1000  
>>> 52 % 10  
2
```

You may have inferred what `/`, `**`, and `%` do. In particular, $52 \% 10$ is the remainder when 52 is divided by 10—it's pronounced "52 mod 10". Arithmetic symbols like `+`, `-`, `/`, `*`, `**`, and `%` are called *operators*.

We pause our regularly scheduled program to make a quick observation about division. In Python 2, division can be a bit surprising. Here's an example:

```
>>> 11 / 2  
5
```

In Python 2, when the numerator and denominator are both integers, Python assumes that you want the result to be an integer as well. Although 11 divided by 2 is 5.5, Python gives just the integer part of the solution which is 5. If you want Python 2 to give you the digits after the decimal point, you need to use a decimal point somewhere. For example, you could do any of these:

```
>>> 11.0 / 2  
5.5  
>>> 11 / 2.0  
5.5  
>>> 11.0 / 2.0  
5.5
```

Python 3, on the other hand, always gives you the digits after the decimal point. If you want to do integer division in Python 3 (dividing two integers and getting an integer back), you'll need to use the special integer division `//` as in:

```
>>> 11 // 2  
5
```

2.2.1 Naming Things

Python lets you give names to *values* — the results of calculations. Here's an example:

```
>>> pi = 3.1415926  
>>> pi * (10 ** 2)  
314.15926
```

In the first line, we defined `pi` to be `3.1415926`. In the second line, we used that value to compute the area of a circle with radius 10. Computer scientists call a name like “`pi`” a *variable* because we can assign it any value that we like. In fact, we could, if we wanted to, give “`pi`” a new value later. We could even give it some crazy value like `42` (although that's probably not a great idea if we're going to use it to compute the area of a circle). The point here is that a “variable” in the computer science sense is different from a variable in the mathematical sense. No sane mathematician would say that the number `pi` is a variable!



Calling `pi` a “variable” would be irrational.

Notice that the “`=`” sign is used to assign a value to a variable. On the left of the equal sign is the name of the variable. On the right of the equal sign is an expression that Python evaluates and then assigns that value to the variable. For example, we could have done this:

```
>>> pi = 3.1415926  
>>> area = pi * (10 ** 2)  
>>> area  
314.15926
```

In this case, we define a variable called `pi` in the first line. In the second line, the expression `pi * (10 ** 2)` is evaluated (its value is `314.15926`) and that value is assigned to another variable called `area`. Finally, when we type `area` at the prompt, Python displays the value. Notice, also, that the parentheses weren't actually necessary in this example. However, we used them just to help remind us which operations will be done first. It's often a good idea to do things like this to make your code more readable to other humans. Note that the equals sign represents an action, namely changing the variable on the left (in this case, `pi` or `area`) so that it has a new value. That value can be changed later. It is important to distinguish this notation from the use of the equals sign in mathematics, where it means that the left and right side are forever the same. That's not the case in CS!

2.2.2 What's in a Name?

Python is not too picky about the names of variables. For example naming a variable `joe` or `joe42` is fine. However, naming a variable `joe+Sally` is not permitted. You can probably imagine why: if Python sees `joe+Sally` it will think that you are trying to add the values of two variables `joe` and `Sally`. Similarly, there are built-in Python special words that can't be used as variable names. If you try to use them, Python will give you an error message. Rather than listing here the words that cannot be used as Python variable names, just keep in mind that if you get a Python error when trying to assign a variable, it's probably because you've stumbled upon the relatively small number of names that are not permitted. Finally, try to use descriptive variable names to help the reader understand your program. For example, if a variable is going to store the area of a circle, calling that variable `area` (or even `areaOfCircle`) is a much better choice than calling it something like `z` or `x42b` or `harriet`.

2.3 More Data: From Numbers to Strings

One of our central themes for this book is data. In the previous section we worked with one kind of data: numbers. That was all good, but numbers are not the only useful kind of data. In the rest of this chapter we'll introduce a few other types of data that are central to solving problems in Python. Indeed, at the start of the chapter we noted that we're going to want to compare strings. In Python, a string is any sequence of symbols within quotation marks. Python allows you to use either double quotes or single quotes around a string. However, Python displays strings in single quotes, regardless of whether you used single or double quotes. Here are some examples:

```
>>> name1 = "Ben"  
>>> name2 = 'Jerry'  
>>> name1
```

```
'Ben'  
>>> name2  
'Jerry'
```

Again, `name1` and `name2` are just variables that we've defined. There's nothing particularly special about those variable names. While there are many things that we can do with strings, we want to show you just a few of the most important things here. In the examples that follow, we assume that we've defined the strings `name1` and `name2` as shown above.

2.3.1 A Short Note on Length

First, we can find the length of a string by using Python's `len` function:

```
>>> len(name1)  
3  
>>> len('I love spam!')  
12
```

In the first example, `name1` is the string '`Ben`' and the length of that string is 3. In the second example, the string contains two spaces (a space is a regular symbol so it gets counted in the length) so the total length is 12.

2.3.2 Indexing

Another thing that we can do with strings is find the symbol at any given position or index. In most computer languages, including Python, the first symbol in a string has index 0. So,

```
>>> name1[0]  
'B'  
>>> name1[1]  
'e'  
>>> name1[2]  
'n'  
>>> name1[3]  
IndexError: string index out of range
```

Notice that although `name1`, which is '`Ben`', has length 3, the symbols are at indices 0, 1, and 2 according to the funny way that Python counts. There is no symbol at index 3, which is why we got an error message. (Computer scientists start counting from 0 rather than from 1.)

2.3.3 Slicing

Python lets you find parts of a string using its special slicing notation. Let's look at some examples first:

```
>>> bestFood = 'spam burrito'  
>>> bestFood[0:3]  
'spa'  
>>> bestFood[0:4]  
'spam'
```



I don't really want a slice of your spam burrito.

What's going on here? First, we've defined a variable `bestFood` and given it the string '`spam burrito`' as its value. The notation `bestFood[0:3]` is telling Python to give us the part—or “slice”—of that string beginning at index 0 and going up to, but not including, index 3. So, we get the part of the string with symbols at indices 0, 1, and 2, which are the three letters `s`, `p`, `a` — resulting in the string `spa`. It may seem strange that the last index is not used, so we don't actually get the symbol at index 3 when we ask for the slice `bestFood[0:3]`. It turns out that there are some good reasons why the designers of Python chose to do this, and we'll see examples of that later. By the way, we don't have to start at index 0. For example:

```
>>> bestFood[2:6]  
'am b'
```

This is giving us the slice of the string 'spam burrito' from index 2 up to, but not including, index 6. We can also do things like this:

```
>>> bestFood[1:]  
'pam burrito'
```

When we leave out the number after the colon, Python assumes we mean “go until the end.” So, this is just saying “give me the slice that begins at index 1 and goes to the end.” Similarly,

```
>>> bestFood[:4]  
'spam'
```

Because there was nothing before the colon, it assumes that we meant 0. So this is the same as `bestFood[0:4]`.

2.3.4 String Arithmetic

Strings can be “added” together. Adding two strings results in a new string that is simply the first one followed by the second. This is called *string concatenation*. For example,

```
>>> 'yum' + 'my'  
'yummy'
```

Once we can add, we can also multiply!

```
>>> 'yum' * 3  
'yumyumyum'
```

In other words, `'yum' * 3` really means `'yum' + 'yum' + 'yum'`, which is `'yumyumyum'`: the concatenation of `'yum'` three times.

2.4 Lists

So far we’ve looked at two different types of data: numbers and strings. Sometimes it’s convenient to “package” a bunch of numbers or strings together. Python has another type of data called a *list* that allows us to do this. Here’s an example:

```
>>> oddNumbers = [1, 3, 5, 7, 9, 11]  
>>> friends = ['rachel', 'monica', 'phoebe', 'joey', 'ross']
```

In the first case, `oddNumbers` is a variable and we’ve assigned that variable to be a list of six odd numbers. In the second case, `friends` is a variable and we’ve given it a list of five strings. If you type `oddNumbers` or `friends`, Python will show you what those values are currently storing. Notice that a list starts with an open bracket `[`, ends with a close bracket `]`, and each item inside the list is separated from the next by a comma. Check this out:

```
>>> stuff = [2, 'hello', 2.718]
```

Notice that `stuff` is a variable that is assigned to be a list, and that list contains two numbers and a string. Python has no objection to different kinds of things being inside the same list! In fact, `stuff` could even have other lists in it, as in this example:

```
>>> stuff = [2, 'hello', 2.718, [1, 2, 3]]
```

This ability to have multiple types of data living together in the same list is called *polymorphism* (meaning “many types”), which is a common feature in functional programming languages.

2.4.1 Some Good News!

Here’s some good news: Almost everything that works on strings also works on lists. For example, we can ask for the length of a list just as we ask for the length of a string:

```
>>> len(stuff)  
4
```

Notice that the list `stuff` really only has four things in it: The number 2, the string 'hello', the number 2.718, and the list `[1, 2, 3]`. Indexing and slicing also work on lists just as on strings:

```
>>> stuff[0]  
2  
>>> stuff[1]  
'hello'
```

```
>>> stuff[2:4]
[2.718, [1, 2, 3]]
```

Just like strings, lists can be added and multiplied as well. Adding two lists together creates a new list that contains all of the elements in the first list followed by all of the elements in the second. This is called *list concatenation* and is similar to string concatenation.

```
>>> mylist = [1, 2, 3]
>>> mylist + [4, 5, 6]
[1, 2, 3, 4, 5, 6]
>>> mylist * 3
[1, 2, 3, 1, 2, 3, 1, 2, 3]
```

Finally, it's worth noting that doing something like:

```
>>> mylist + [4 , 5 , 6]
```

doesn't actually change `mylist`. Instead, it gives you a NEW list which is the result of concatenating `mylist` and the list `[4, 5, 6]`. You can verify this by typing `mylist` at the Python prompt and seeing that it wasn't changed:

```
>>> mylist + [4 , 5 , 6]
[1, 2, 3, 4, 5, 6]
>>> mylist
[1, 2, 3]
```

2.5 Functioning in Python

We've covered the basics of Python including how to represent and work with several types of data including numbers, strings and lists! Next, we're going to write actual programs. Throughout this and the remaining sections, we will keep in mind the problem that is motivating us in the first place—calculating edit distance—by looking at examples that help us build toward a solution to that problem. However, we wouldn't want you to get bored by looking at just one single problem, so we'll also throw in some other interesting ones along the way.

Let's begin with an analogy to something that you know and love: mathematical functions. In math, a function can be thought of as a “box” that takes some data as input, or argument, and returns some data as output, which we call its result, return value, or simply value. For example, the function $f(x) = 2x$ has an argument called x , and for any value that we “stick into” x we get back a value that is twice as large. Python lets us define functions too, and as in math, these functions take some data as arguments, process that data in some way, and then return some data as a result. Here's a Python function that we've named `f`, which takes an argument we are calling x and returns a result that is two times x .

In other words:

```
def f( x ):
    return 2 * x
```



Actually, we'll learn later that unlike in math, Python functions do not have to take arguments or return results. But for now we'll focus on functions that do.

In Python, the syntax for functions uses the special word `def`, which means: “I'm defining a function.” Then comes the name that we've chosen for the function; in our case we chose to call it `f`. Then come the arguments to the function in parentheses (just like the definition of a function in math!), followed by a colon (“:”). Then, we start a new line, indented a few spaces, and begin the function. In this case, our function computes $2*x$ and then returns it. The word `return` tells Python to give us back that value as the function's result. *Don't forget to indent the line (or lines) after the first line.* Python absolutely demands this. We'll talk a bit more about indentation shortly. We can now run the function in the Python interpreter like this:

```
>>> f(10)
20
>>> f (-1)
-2
```

When we type `f(10)` we say that this is a *function call* because we are metaphorically placing a telephone call to `f` with argument 10 and asking it to give us an answer. When we call `f`, Python takes the value inside the parentheses and assigns it to the name `x`. Then it executes the statement or statements inside the function in order until there are no more statements to execute in the function. In this case, there is only a single statement inside the function, which doubles the value of `x` and returns the result to where the function was called.



The Python IDLE environment is named after Eric Idle, one of the members of the Monty Python comedy group.

The best way to define a function is to open up an editor and define the function there. If you are using IDLE go to the File menu and select “New Window.” A new window will open up. Now you have two windows: the Python interpreter that you had originally and a new editor window where you can type your function definitions. In the editor window, you can edit comfortably, moving the cursor with the arrow keys and mouse. When you’ve completed your function, use the “File” menu to “Save” the file. Then, you can click on “Run” and the function will be available for you to call in the original Python window. (If you are using some other version of Python, your professor will need to provide you with instructions on how to open a file for editing a function.)

As we noted, you can name a function more or less anything you want (as with variable names, there are a few exceptions, but it’s not worth enumerating them). In this example, we called the function `\(f\)` simply to make the analogy with the mathematical function `\(f(x) = 2x\)` that we defined at the beginning. To be honest, it’s better to give functions descriptive names, just as we’ve advocated doing for variables. So, calling the function something like `double` would probably have been a better choice.

2.5.1 A Short Comment on Docstrings

A function is actually a computer program. We’ve just written our first program! Admittedly, a program that doubles a number is probably not one that you would show off to your friends and family. However, we’re getting closer to writing some amazing programs. As our programs become more interesting, it will be important for their users to be able to quickly understand what the program is for. To that end, every function that we write from now on will begin with something called a *docstring*, which stands for “documentation string”. Here’s our snazzy doubling function `f` with its docstring:

```
def f(x):  
    """ Takes a number x as an argument and returns 2*x.  
    return 2 * x
```

The docstring is a string that begins and ends with three single quote marks. If you now type `help(f)`, the docstring will be displayed. *You should always provide a docstring for every function that you write.*

2.5.2 An Equally Short Comment on Comments

Docstrings provide a way for users of your program to find out what the function is about. In addition to docstrings, you should always leave yourself (the programmer) or other programmers little notes, called *comments*, which explain the internal details of your program. Any text following a `#` is understood by Python to be a comment. The comment lasts until the end of that line. Python doesn’t try to read or understand the comment (though it would be neat if it could!). Just to be clear, the difference between a docstring and a comment is that a docstring is intended for the user of the function. The user might not even understand Python. A comment lets the programmer share details about how the program works with other people who might want to understand or modify it.

2.5.3 Functions Can Have More Than One Line

The function above, `f`, had only one statement, which doubled its argument and returned that value. However, in general functions are not limited to a single Python statement, but rather may have as many statements as you choose. For example, we could have written the exact same function from above as follows:

```
def f(x):  
    """Takes a number x as an argument and returns 2*x.  
    twoTimesX = 2 * x  
    return twoTimesX
```

Notice that when there is more than one statement in the function they must all be indented to the same position. This is how Python knows which statements are inside the function and which are not. We will say even more on indentation below. Here, you might be wondering which definition for `f` is better: the one with one line or the one with two. The answer is that both are equally valid and correct. Which you prefer is really up to you, the programmer. Some people prefer the one-line implementation because it's more compact, while others prefer to the two-line implementation because they prefer to store the values of intermediate calculations rather than immediately returning them. There's always more than one way to write a program!

2.5.4 Functions Can Have Multiple Arguments

Just as in math, functions can have more than one argument. For example here's a function that takes two arguments, $\langle x \rangle$ and $\langle y \rangle$, and returns $\langle x^2 + y^2 \rangle$:

```
def sumOfSquares(x , y):
    """ Computes the sum of squares of its arguments """
    return x**2 + y**2 # Here's our first comment!
```

In fact, the arguments to a function need not be numbers. They can be strings, lists, and a number of other things (more on this in the next chapter). Here's a mystery function. It takes two strings as arguments and returns another string. What is it doing? Once you think you have an idea, run the Codelens example and see if it matches your expectations. Codelens will allow you to step through the program and visualize what the computer is doing at each step as it executes the program.

(ch02_mystery)

What's this `print` thing? In Python, `print` is a function that takes an argument (e.g., a number, a string, or any other value) and prints it on the screen. In this example, the argument that is being printed is whatever is being returned by the `mystery` function. There are two important things to note about `print`: First, it's a function, so its argument must be in parentheses. Second, it's different from `return` in an important way. The `return` statement is not a function - it simply leaves the function and returns a value to whoever called that function. In contrast, `print` is a function that displays a value on the screen. You can have many uses of `print` inside a function. Each one will print what it's told to print and then the function will continue on from there. On the other `return` is powerful stuff. The moment that a `return` statement is encountered, the function returns the value and it's done.

2.5.5 Why Write Functions?

At this point you might be wondering why we write functions. If we wanted to calculate twice the values of 10 and -1, wouldn't it be easier to just type what we have below rather than going through the hassle of defining a function?

```
>>> 2 * 10
20
>>> 2 * -1
-2
```

In this case, perhaps direct (or even mental!) calculation is simpler, but in general (and as we will shortly see), functions do much more complicated calculations that would be a pain to have to type over and over. Functions allow us to "package up" a bunch of calculations that we know we will want to perform over and over.

Remember our spiel about abstraction in Chapter 1? Packaging a computation into a function like this is a form of abstraction: we're hiding the details (the precise calculations in the body of the function) so that whoever calls the function can focus on its end result, instead of exactly how that result is calculated.

2.6 Making Decisions

So far our programs have just made straightforward calculations. But sometimes we need to write programs that make decisions, so that they perform different actions depending on some condition. For example, to solve our edit distance problem, we'll need to compare characters in our two strings and take different actions depending on whether those characters are the same or different. Before we get back to edit distance, let's consider a famous problem in mathematics called the " $3n + 1$ " problem. It goes like this. Consider a function that takes a positive integer n as an argument. If the integer is even, the function returns the value $n/2$. If the integer is odd, the function returns $3n + 1$. The problem, which is really a conjecture, states that if we start with any positive integer n and repeatedly apply this function, eventually we will produce the value 1. For example starting with $n = 2$, the function returns 1 because n is even. Starting with $n = 3$, the first application of the function returns

10. Now applying the function to 10 gives 5; applying it to 5 gives 16, which in turns gives 8, then 4, then 2, and finally 1. Ta-dah!



It seems like too many big names for just one little problem!

This seemingly benign little problem also has many other names, including the “Collatz problem”, the “Syracuse problem”, “Kakutani’s problem”, and “Ulam’s problem”. So far, nobody has been able to prove that this conjecture with many names is true in general. In fact, the famous mathematician Paul Erdos stated that “Mathematics is not yet ready for such problems.”

Let’s write the function in Python and then you can experiment with it!



My guess is that two equals signs means “very equal”

Before we look at this in detail, notice the expression `n % 2 == 0`. Recall that `n % 2` is just the remainder when `\(n\)` is divided by 2. If it is even, its remainder when divided by 2 will be 0; if it is odd, the remainder will be 1.

OK, but how about the `==`? You’ll recall that a single `=` sign is used in assignment statements to assign the value of an expression on the right-hand side of `=` to the variable on the left-hand side. The syntax `==` is doing something quite different. It evaluates the expressions on both sides of the `==` sign and determines whether or not they have the same value. This kind of expression always evaluates to either `True` or `False`. You might wish to pause here and try this in the Python interpreter. See what Python says to `42 % 2 == 0` or to `2 * 21 == 84/2` or to `42 % 2 == 41 % 2`. The special values `True` and `False` are called Boolean values. An expression that has a value of `True` or `False` is called a *Boolean expression*.



George Boole (1815-1864) was an English mathematician and philosopher. His work in logic forms part of the foundation of computer science and electrical engineering.

By the way, the Booleans `True` or `False` should not be thought of as either numbers or strings. They are special kinds of values that are intended for letting your function “reason” logically.

Back to our `collatz` function. It begins with an `if` statement. Right after the Python keyword `if` is the Boolean expression `n % 2 == 0`, followed by a colon. Python interprets this *conditional statement* as follows: “If the expression that you gave me (in this case, `n % 2 == 0`) is `True` then I will do all of the stuff that is indented on the following lines.” In this case there is only one indented line, and that line says to return the value `n/2`. On the other hand, if the Boolean expression that was tested had the value `False`, Python would execute the indented lines that come after the `else:` line. You can think of that as the “otherwise” option. In this case, the function returns `3*n+1`.

It turns out that `else` statements are not required by Python, and sometimes we can do without them. For example, take a look at a slight modification of our `collatz` program shown below.

```
def collatz( n ):  
    """Takes a single number as argument and applies the Collatz to it"""  
    if n % 2 == 0:  
        return n/2  
    return 3*n + 1
```



How do you humans know about 42? Has someone told you that it is the answer to the ultimate question of the universe, and everything?! I'll have to Google 42 to see what it says.

If n is even, this function computes $n/2$ and returns that value; returning that value causes the function to end—which means that it stops evaluating any more statements. This last sentence is important and often confusing so let's highlight it again: **Executing a return statement always causes a function to end immediately**. Thus, if n is even, Python will never get to the line `return 3*n + 1`. However, if n is odd then the expression $n \% 2$ evaluates to `False`. In this case, Python drops down to the first line after the `if` statement that is at the same level of indentation as the `if`; in this case this is the line `return 3*n+1`. So we see that this version of the function behaves just like the first version with the `else` clause. But forty-one out of forty-two surveyed computer scientists advocate using the `else` version simply because it is easier to read and understand.

2.6.1 A second example

Now let's consider a second example, more closely related to the edit distance problem we are building up to. Our next function will determine whether the first character in each of two strings is the same or different. For example:

```
>>> matchFirst ( 'spam' , 'super')
True
>>> matchFirst ( 'AAGC' , 'GAG')
False
```

Here's one way to write the `matchFirst` function:

```
def matchFirst ( s1 , s2 ):
    """Compare the first characters in s1 and s2
    and return True if they are the same.
    False if not"""

    if s1 [0] == s2 [0]:
        return True
    else :
        return False
```

As always, there's more than one way to write a program. Take a look at this shorter (and seemingly stranger) version:

```
def matchFirst ( s1 , s2 ):
    """Compare the first characters in s1 and s2
    and return True if they are the same.
    False if not"""

    return s1[0] == s2[0]
```

What's going on here!? Remember that our goal is to return a Boolean—a value that is either `True` or `False`. The expression `s1[0] == s2[0]` is either `True` or `False`, and whichever it is, that's what we want to return. So the statement `return s1[0] == s2[0]` will produce our desired answer.

However, both versions of our `matchFirst` function have a subtle problem: there are some arguments on which the functions will fail to work. Take a moment to see if you can identify a case where things will go awry before you read on.



Many computer scientists confuse measuring with counting and also count from zero!

Did you find the problem? When either (or both) of our arguments are the empty string (a string with no characters inside it), Python will complain. Why? The empty string has no symbol at index 0—because it has no symbols in it at all! (Remember, the symbol at index 0 actually refers to the first symbol in the string. When we index into strings, we are actually measuring the distance from the beginning to the desired character, and the first symbol is at a distance of zero characters from the start.)

We can fix this by using another if statement to check for the special case that one or both of the arguments is the empty string. The built-in `len` function will tell us whether the string has length 0. It's a bit weird, but a string of length 0 has no symbols in it at all, not even one at index 0.

```
def matchFirst ( s1 , s2 ):  
    """Compare the first characters in s1 and s2  
    and return True if they are the same.  
    False if not"""  
  
    if len(s1) == 0 or len(s2) == 0:  
        return False  
    else :  
        return s1[0] == s2[0]
```



if both strings are empty, the function returns false. Do you think this is the correct result? How would you change it to return True in that case?

Now, if either string is empty the function will return `False`. Notice the use of the word `or` in the `if` statement here. It's saying “if the length of `s1` is 0 or the length of `s2` is 0 then return `False`.”

Python expects that the condition being tested in an `if` will have a Boolean value—that is, its value is either `True` or `False`. In this case, `len(s1) == 0` has a Boolean value; it's either `True` or `False`. Similarly, `len(s2) == 0` is Boolean. The connector `or` is Boolean “glue” much like addition is arithmetic glue. Just as the plus sign adds the two numbers on its left and right and gives us back another number (the sum), the `or` sign looks at the Booleans on its left and right and gives us back another Boolean—`True` if at least one of the Booleans is `True` and otherwise it gives us back `False`.

By the way, Python also has another piece of Boolean glue called `and`. Not surprisingly, it gives us back `True` if both of the Booleans on its left and right are `True` and gives us back `False` otherwise. So, the statement:

```
len(s1) == 0 and len(s2) == 0
```

will be `True` if *both* strings are empty.

Finally, Python has something called `not` that “negates” a Boolean, flipping `True` to `False` and `False` to `True`. For example:

```
not 1 == 2
```

will be `True` because `1 == 2` is `False` and its negation is therefore `True`. Incidentally, we can mix and match our new friends `or`, `and`, and `not`. For example, the expression:

```
1 == 2 or not 41 == 42
```



But later we'll see a better way to write `not 41 == 42`

will evaluate to `True` because although `1 == 2` is `False`, `not 41 == 42` is `True`, and the result of `or`-ing `False` and `True` is `True`.

2.6.2 Indentation

We have noted that Python is persnickety about indentation. After the first line (the one containing the `def`), Python expects all of the remainder of the function to be indented. As we've seen above, indentation is also used in `if` and `else` statements. Immediately after an `if` statement, we indent the line or lines that we want Python to execute if the expression is `True`. Similarly, the line or lines that should be executed after the `else` are indented as well.

For example, here is another way that we could have written our `collatz` function; this one uses one line to compute the desired value and then returns it in a second. Most programmers wouldn't write the function this way because it's more verbose than necessary, but when programs get more involved and complicated it can be handy (and sometimes necessary) to have multiple lines like this. And again, if you prefer to write it this way even in this simple case, there's nothing wrong with that!

```
def collatz(n):
    """Takes a single number as an argument and
       applies the Collatz function to it."""
    if n % 2 == 0:
        result = n/2    # Create a variable called result
        return result   # Now we return the value of result
    else:
        result = 3*n + 1  # Create a variable called result
        return result    # Now we return the value of result
```

2.6.3 Multiple Conditions

We noted that sometimes we use `if` without a matching `else`. On the flip side, sometimes it's useful to have more than one alternative to the condition tested in the `if` statement.

Consider again our edit distance problem. While we're not quite ready to solve the whole thing yet, we can solve it for the case that one or both of the argument strings is empty. In this case, the edit distance between the two strings is simply the length of the non-empty one, because it necessarily will take that many insertions to the empty string (or, conversely, deletions from the non-empty one) to make the two the same.

Here is a function that solves the edit distance problem only in this simple case:

```
def simpleDistance(s1, s2):
    """Takes two strings as arguments and returns the edit
       distance between them if one of them is empty.
       Otherwise it returns an error string."""
    if len(s1) == 0:
        return len(s2)
    elif len(s2) == 0:
        return len(s1)
    else:
        return 'Help! We don't know what to do here!'
```



Dissection!? Is this CS or biology?

Let's dissect this function.



Computer scientists are infamous for claiming that anything that their software does is a “feature”—making even their mistakes sound like they were intentional and useful!

It starts by checking whether the string `s1` is empty; if so the function returns the length of `s2`. If `s1` is not empty, then the function will use an `elif` statement to see whether `s2` is empty. If so, it will return the length of `s1`.

Finally, if neither `s1` nor `s2` is empty, it will return a string reporting that it does not know what to do.

(By the way, notice that if *both* strings are empty then we'll return 0, which is the correct answer. Do you see why 0 gets returned?)

The `elif` statements are pronounced “else if”. The `elif` in this function is only reached if `s1` was not empty. The `elif` is saying “else (otherwise), if the string `s2` is empty, then return the length of `s1`.“ Although we only have one `elif` in this function, in general, after an `if` we can have as many `elif` conditions as we wish. Then, at the end we can have zero or one `else` statements, but not more than one! The final `else` statement specifies what to do if all of the preceding conditions failed.

There's nothing wrong with a function returning a string! Returning means we're done—the function is over and we get back the value in the return statement. Generally you would not want your function to return a string in some cases and a number in others (like ours currently does), but in this case we're doing that just to make it clear what's happening in each case. Our final edit distance function won't have this “feature.”

Note

A Colorful Application of if, elif, and else

In 1852, Augustus DeMorgan revealed in a letter to fellow British mathematician William Hamilton how he'd been stumped by one of his student's questions:

A student of mine asked me today to give him a reason for a fact which I did not know was a fact—and do not yet. He says that if a figure be anyhow divided and the compartments differently coloured so that figures with any portion of common boundary line are differently coloured—four colours may be wanted, but not more...

DeMorgan had articulated the *four-color problem*: whether or not a flat map of regions would ever need more than four colors to ensure that neighboring regions had different colors. The problem haunted DeMorgan for the rest of his life, and he died before Alfred Kempe published a proof that four colors suffice in 1879.

Kempe received great acclaim for his proof. He was elected a Fellow of the Royal Society and served as its treasurer for many years; he was knighted for his accomplishments. He also continued to investigate the four color theorem, publishing improved versions of his proof and inspiring other mathematicians to do so.

However, in 1890 a colleague showed that Kempe's proof (and its variants) were all incorrect—and the mathematical community resumed its efforts. The four-color problem stubbornly resisted proof until 1976, when it became the first major mathematical theorem proved using a computer. Kenneth Appel and Wolfgang Haken of the University of Illinois first established that *every flat map* must contain one of 1,936 particular sub-maps that they defined. They next showed, using over 1200 hours of computer time, that each of those 1,936 cases could not be part of a counterexample to the theorem. Four colors did, in fact, suffice!

Their program essentially used a giant conditional statement with 1,936 occurrences of `if`, `elif`, or `else` statements. Such *case analyses* are quite common in computational problems (although 1,936 cases is admittedly more than we might usually encounter).

We suspect that DeMorgan would feel better knowing that the question he couldn't answer wouldn't be answered at all for over a century.

2.7 Recursion!

So far we've built up some powerful programming tools that have allowed us to do some interesting things, but we still can't solve the edit distance problem except for the very special case that one of the strings is empty.

What about the cases we really care about, where neither string is empty? Imagine for a moment that we know that both of our strings are exactly four characters long. In that case, we don't need any deletions or insertions to get from the first string to the second string—we only need substitutions. For example, to get from “spam” to “spim” we just substitute the “a” with an “i”. For the case of two strings of length four, we could compute the distance this way:

```
def distance(s1, s2):
    """Return the distance between two strings,
    each of which is of length four."""
    editDist = 0
    if s1[0] != s2[0]:
        editDist = editDist + 1
    if s1[1] != s2[1]:
```

```

editDist = editDist + 1
if s1[2] != s2[2]:
    editDist = editDist + 1
if s1[3] == s2[3]:
    editDist = editDist + 1
return editDist

```

The notation `!=`, which is read as “not equals”, offers a much nicer notation than the clumsy but equivalent `not s1[0] == s2[0]`. Notice that this function starts by setting a variable named `editDist` to 0 and then adds 1 to that variable each time it finds a pair of corresponding symbols that don’t match and thus require a substitution. (By the way, also notice that we used `if` s and not `elif` s. Do you see why `elif` s would give us the wrong answer here?)

While this program will work for four-letter words, it doesn’t help us if the words are longer or shorter. Moreover, it can’t deal with insertions or deletions. We might be tempted to add more `if` s to handle longer strings, but how many would we add? No matter how many we added, we would still run into trouble for sufficiently long strings.

To add both the ability to handle strings of arbitrary length and the ability to handle insertions and deletions, we will need a beautiful and elegant new ingredient called *recursion*.



A recursion excursion!

Before using recursion for the edit distance problem, let’s take an excursion to visit a group of aliens who have come to Earth from a distant planet for the debut of the seventeenth Harry Potter movie.

At the moment there are, let’s see, 42 aliens in line. One alien muses to itself, “I wonder how many different ways I could arrange us 42 aliens?”

You may know the answer: it’s $(42 \times 41 \times 40 \dots 3 \times 2 \times 1)$, also known as “42 factorial” and written $(42!)$. (There are 42 choices for the alien that could be first in line, 41 who could get the next spot, and so forth.)

The alien decides that it would like to write a Python program to compute the factorial of any positive integer. Fortunately, the alien has done some shopping at the local mall, where it purchased a laptop that runs Python. Unfortunately, the alien is vexed by how to write such a program. Fortunately, we observe that $(42! = 42 \times 41!)$ and in general, $(n! = n(n-1)!)$. That is, the factorial of (n) can be expressed as (n) times the result of solving another smaller factorial problem. This is called a *recursive definition*: it expresses the problem in terms of a smaller version of the same problem; the definition *recurs* in the solution.



Unfortunately, it’s not clear to me how this helps.

Before writing a program to compute the factorial, let’s just observe that this recursive definition is indeed useful. Imagine that we want to compute $(3!)$. According to the recursive definition, that’s just $(3 \times 2!)$. So now we’re off to find $(2!)$. Using the same definition again, we see that $(2! = 2 \times 1!)$. If we can figure out what $(1!)$ is, we’ll be in good shape. According to the definition, $(1!)$ is $(1 \times 0!)$. According to the definition, $(0!)$ is $(0 \times (-1) \dots)$ Uh oh, this is bad—the process will never stop. Moreover, we are not really interested in $(0!)$ or the factorial of a negative number.

To work around the difficulty, we should add a rule that tells us when to stop this recursive process. A reasonable stopping place is to say, “If you get to the point that you’re trying to compute $(1!)$, stop using the recursive rule and just report that the answer is 1.” This is called the *base case* of the recursive definition. It tells us when to stop applying the rule. Of course, we should always check the base case *before* deciding whether to continue applying the recursive rule.

Coming back to the example of $(3!)$, we get down to the case of $(1!)$ and the base case says “Aha! Stop! That’s 1.” So, we have $(1! = 1)$. Remember that it was $(2! = 2 \times 1!)$ that wanted to know the value of $(1!)$. So now we plug the 1 in for $(1!)$ and determine that $(2! = 2 \times 1 = 2)$. But $(3! = 3 \times 2!)$

was the one that asked about $\backslash(2!\\)$ and is waiting patiently for the answer. So, now $\backslash(3!\\)$ determines that its result is $\backslash(3 \times 2 = 6\\)$. That's it, we've computed $\backslash(3!\\)$ using the recursive definition.

Let's try to capture the recursive definition as closely as we can in Python. This may seem weird or even downright wrong at first, but let's try. Here's the program. (Run it, and try it out!)

(ch02_secondeexample)

Looking at this function, we see that it looks like a translation from math into Python. Try running this function. The factorial of 5 is 120 and the factorial of 70 is larger than the number of particles in the universe—but Python will gladly compute it.

The fact that this function correctly computes the factorial function might seem mysterious and magical. We will see shortly that there is no magic here and not even any mystery! However, for just a moment, let's take a leap of faith that Python will do what we intend when we run this function (hopefully corroborated by your computational experiment that the function seems to work on the arguments that you tried). In a moment we'll come back to convince ourselves that this recursion really *must* work.

A typical recursive function has two main parts:

- **A base case** : This is the value that the function returns for the “simplest” argument.
- **A recursive step** : This is the solution to a smaller version of the problem, computed by calling the function with a smaller or simpler argument. This solution to the smaller problem is then used in some way to solve the original problem.

In the factorial function, the base case is when n is 1. In this case, our function simply returns 1, since $\backslash(1! = 1\\)$, and we're done. The recursive step, the part inside the `else` statement, computes the factorial of $n-1$, multiplies the result by n , and returns this value.

Now let's return to the edit distance function. We're still not ready to solve it in its entirety, but we're getting closer! Let's now consider the situation where the two strings are guaranteed to be of the same length (so that no insertions or deletions need to be used), but their length could be anything—not just four! In this case, the edit distance is the number of positions where the two strings differ.

The base case now is when both strings are empty—that's the simplest case in which the two strings could have the same length. In that case, the edit distance is 0 and we're done.

If the base case does not apply—that is the strings are of some non-zero length—then what happens next depends on whether or not the two strings match at position 0. If they don't match, that position contributes 1 to the edit distance. Now, we have to compare the remainder of the two strings to one another to find the number of differences between them. But that's exactly the same problem, just for the two strings with their leading symbols chopped off! So, if the characters at position 0 don't match, the edit distance between s_1 and s_2 is $\backslash(1\\)$ plus the edit distance between $s_1[1:]$ and $s_2[1:]$. Remember, the notation $s_1[1:]$ is a string just like s_1 except with the symbols at position 0 chopped off.

On the other hand, if the two strings match on the first symbol then the edit distance between s_1 and s_2 is just the edit distance between $s_1[1:]$ and $s_2[1:]$. This results in the function below.

```
def simpleDistance(s1, s2):
    """Takes two strings of the same length and returns the
    number of positions in which they differ."""
    if len(s1) == 0:      # len(s2) is also 0 since strings
        # have the same length
        return 0          # base case
    elif s1[0] != s2[0]: # recursive step, case 1
        return 1 + simpleDistance(s1[1:], s2[1:])
    else:                # recursive step, case 2:
        # s1[0] == s2[0]
        return simpleDistance(s1[1:], s2[1:])
```



And don't forget the base case!

Takeaway message : *The secret to thinking about recursion is to ask yourself “would it help if I had the answer to a slightly smaller version of the same problem?” If the answer is “yes,” then you can write a recursive function that calls itself to get the answer to the slightly smaller version of the problem and then use that result to solve your original problem.*

2.8 Recursion, Revealed

2.8.1 Functions that Call Functions

We promise that in this section we will reveal the “magic” behind recursion. But for right now, we’ve consulted with our lawyers and they told us that first we could sneak in a short section on a slight tangent. Actually, it’s not quite as much of a tangent as it may seem at first. Did you know that the word tangent was evidently first used in 1583 by the Danish mathematician Thomas Fincke? Speaking of Denmark, did you know that Legos were invented there? We digress.

Take a look at the code in the example below. How did Python arrive at 42 when we called `demo` with argument 13? Does Python get upset or confused by the fact that each of the functions here has a variable named `x` and a variable named `r`? How does changing the value of `x` in one function affect the value of `x` in another function?

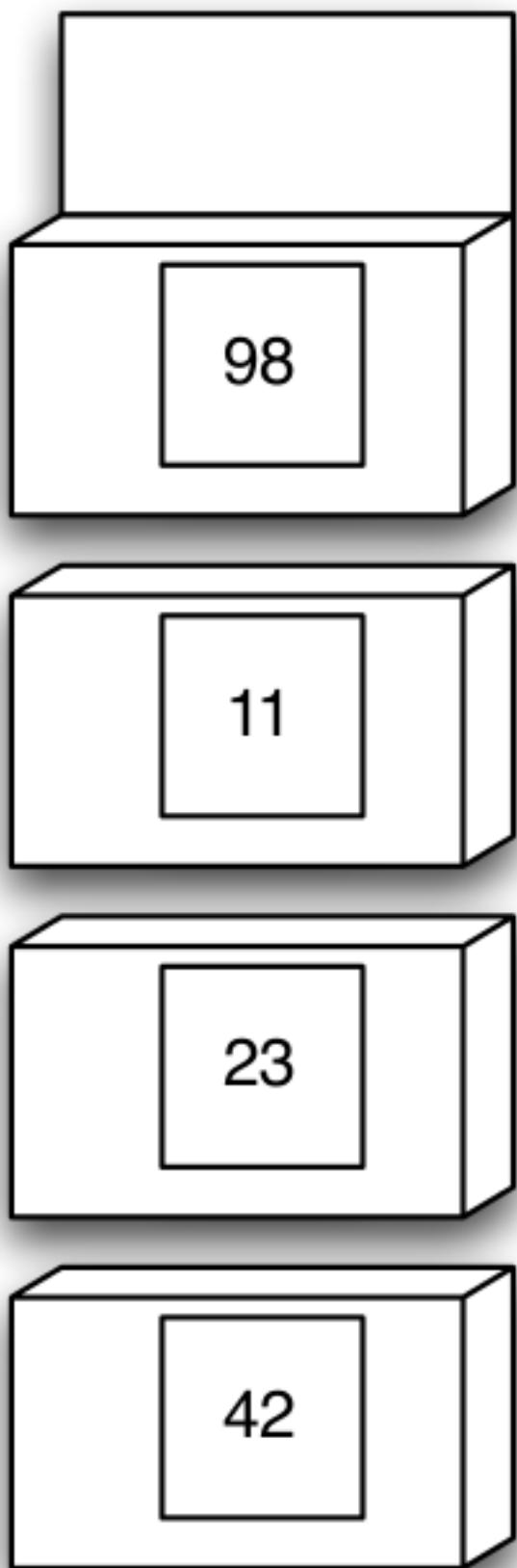


Figure 2.1: A stack of storage boxes. Only the box on the top has its door accessible.

```
def demo(x):  
    r = f(x+6) + x  
    return r
```

```
def f(x):
    r = g(x-1)
    x = 1
    return r + x
```

```
def g(x):
    r = x + 10
    return r
```

```
>>> demo(13)
42
```

The secret here has to do with the way that Python (and, indeed, any self-respecting programming language) deals with variables. Python has a large supply of metaphorical storage boxes that it can use to store “precious” commodities.

These storage boxes have the special feature that their doors are on top and they are stackable. Figure 2.1 shows an artist’s rendition of a stack of boxes. The box on the top of the stack is the only one that you can tinker with. You’ll have to remove that box before you can tinker with the contents of the one below it. However, we’ve put little windows in all of the boxes just for the sake of explanation—allowing you to peek at what’s in there.



And letting the numbers get some natural light!

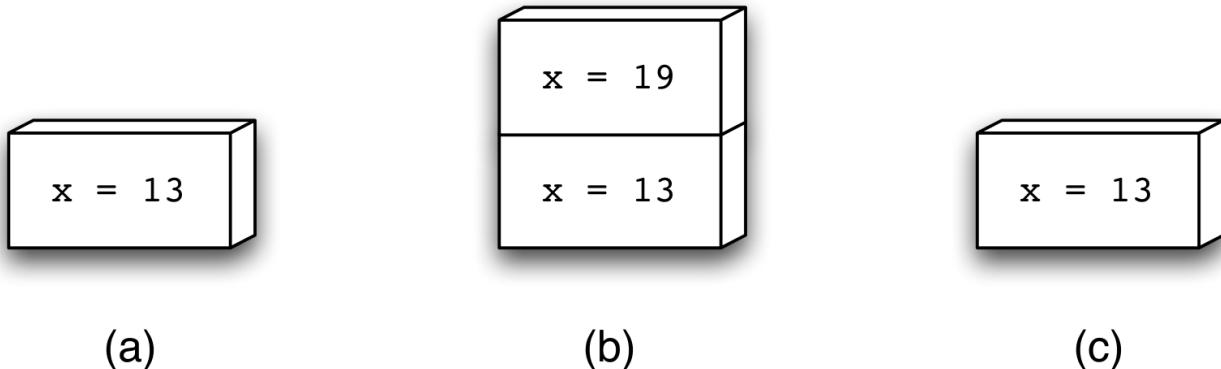


Figure 2.2: The use of stack when `demo(13)` is evaluated. (a) `demo` places its value of `x` on the stack. (b) `f` places its value of `x` in a box on top of the stack. (c) The stack after `g` is done and `f` has retrieved its value of `x`.

Storage boxes? Doors? Windows? Are you CS professors crazy? Perhaps, but that’s not really relevant here. Let’s take a closer look at our program above. When the call `demo(13)` is made, the value 13 is passed into the `demo` function’s argument named `x`. This variable is owned by `demo` and cannot be seen by “anyone” outside of this function. Python, like most programming languages, likes to enforce privacy.

The function `demo` needs to call `f(x+6)` since this is the first expression in the right-hand side of the statement `r = f(x+6) + x`. However, `demo` wants to ensure that the “precious” value of its variable `x`, currently, 13, is preserved. It rightfully worries that it might get changed by the function `f` or perhaps one of `f`’s pernicious friends (like `g`). So, right before the function call to `f`, Python automatically stores all of `demo`’s variables in a storage box for safe-keeping. In this case, there is a variable called `x` and Python stores its value, `\(13\)`. This is shown in Figure 2.2(a).

When the function `f` is called, it gets the argument `\(13+6 = 19\)`. The value 19 is going into `f`’s argument `x`. Now `x` has the value 19. The function `demo` is not worried about this change in the value of `x` because it has locked up its own value of `x` in its secure box. It will retrieve that value when `f` is done and returns control to `demo`.

Next, `f` calls `g(19-1)`. Again, before doing so, it saves its own value of `x`, 19, in its own box for safe-keeping. This box is stacked on top of the previous box as shown in Figure 2.2(b). Now function `g` gets 18 in its argument of `x`. It computes `\(10+18 = 28\)` and returns that value. When we return to function `f`, that function immediately

finds the storage box on the top of the stack, retrieves its value of x from the box, and then removes that box from the stack. Now, f has restored its original value 19 for x . The current situation is shown in Figure 2.2(c). Now x is changed to 1—though that’s kind of silly since it’s about to be thrown away. Finally f returns $\lambda(28+1 = 29\lambda)$ to the `demo` function. At this point `demo` finds its box at the top of the stack, opens it, restores its original value of x to 13, and tosses the box. This value is now used in the rest of computation, and the `demo` returns the value $\lambda(29+13 = 42\lambda)$. Whew!

In computer science, this pile of storage boxes is called the *stack*. In practice it is implemented using the computer’s memory rather than storage boxes with cute doors and windows, but we like the metaphor.

Where a variable’s value can be seen is called its *scope*. Our example demonstrates that the scope of a variable is limited to the function in which it resides. That’s all cool, but our lawyers have warned us that if we don’t talk about recursion now, you’ll have grounds to sue us, so here we go!

2.8.2 Recursion, Revealed, Really!

OK, now back to our first recursive function, `factorial`:

(ch02_factorial)

Amazing! But even though it *seems* to run correctly and implements a recursive definition that *seems* correct, the function still *seems* rather like magic.



To me it still seems unseemly!

So let’s take a look at what Python is doing when we run `factorial(3)`. We’ll explain it step-by-step and summarize the process in Figure 2.3.

The function begins with n equal to 3. Since n is not equal to 1, the condition in the `if` statement is `False` and we continue down to the `else` part. At this point, we see that we need to evaluate $n * \text{factorial}(n-1)$ which requires calling the function `factorial` with argument 2. Python doesn’t realize—or even care—that the function that is about to be called is the very same function that we’re currently running. It simply uses the same policy that we’ve seen before: “Aha! A function call is coming up. I’d better put my precious belongings in a storage box on the stack for safekeeping so that I can retrieve them when this function call returns.”

In this case, n , with value 3, is the only variable that `factorial` owns at the moment. So, `factorial(3)` puts the value of n equal to 3 away in the box for later retrieval. Then, it calls `factorial(2)`. Now `factorial(2)` runs. That means that `factorial` starts at the beginning with an input of 2 so that n now has the value 2. Fortunately, the original value of n has been locked away for safekeeping because we’ll need it later. For now, though, `factorial(2)` again goes to the `else` part where it sees that it needs to call `factorial(1)`. Before doing so, `factorial(2)` puts its value of n , namely 2, in its own storage box at the top of the stack. Then it calls `factorial(1)`.

Finally, `factorial(1)` is executed. Notice that n is now 1, so the expression $n == 1$ evaluates to `True` and `factorial(1)` simply returns 1. But this 1 must be returned to the function that called it. That’s true anytime there is a function call! Recall that `factorial(2)` made that call. At this point, control is returned to `factorial(2)`, which immediately goes to its storage box at the top of the stack, opens it, retrieves its value of n (which is 2), and discards the box from the top of the stack. Now, `factorial(2)` resumes its work. It computes $2 * 1$, assigns that value ($\lambda(2\lambda)$) to the variable `result`, and returns that.

That value is returned to the place where `factorial(2)` was called. That was in `factorial(3)`, which now goes to the top of the stack, opens the storage box, retrieves its value of n , and tosses the box. Now, n is 3 and `factorial(3)` computes $3 * 2$, which is 6, and returns that value. The value is returned to us because we called `factorial(3)` at the prompt and, voila, we have our answer!

Remember that the test `if n == 1`, the base case, is critically important.



The base case in a recursive function is analogous to the base case in a proof by induction. In fact, you may have noticed that recursion and induction are very similar in spirit.

Without it, the program has no way of knowing when to stop. In general, we advocate **trying to write the base case first** because it gets an easy case out of the way and lets you concentrate on the recursive part. Moreover, the base case needs to be the first thing that your recursive function tests, to make sure that it can eventually stop.

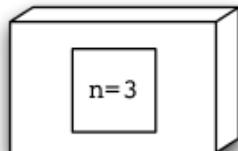
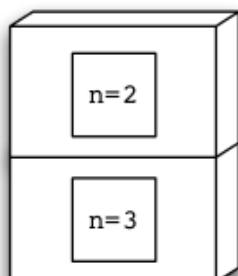
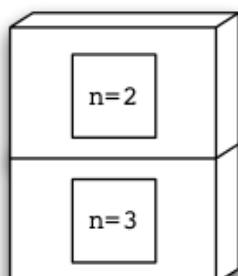
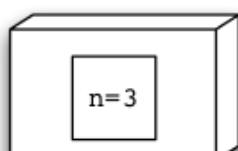
<u>Function call</u>	<u>What happens here...</u>	<u>What the stack looks like at this moment...</u>
factorial(3):	Places its value of n=3 on the top of the stack and then calls factorial(2)	
factorial(2):	Places its value of n=2 on the top of the stack and then calls factorial(1)	
factorial(1):	Returns the value 1 to the place where it was called, namely factorial(2)	
factorial(2):	Takes the value 1 returned from factorial(1), finds its own value of n=2 at the top of the stack, computes the product, 2*1, and returns this value to the place where it was called, namely factorial(3)	
factorial(3):	Takes the value 2 returned from factorial(2), finds its own value of n=3 at the top of the stack, computes the product (3*2), and returns this value, 6, to the place where it was called...	

Figure 2.3: What happens when we invoke factorial(3)

One way to help you think of what the base case should be is to ask yourself the question, “What is the ‘easiest’ input that I might get for this problem?” In the case of factorial, it seems that $\lfloor(1!) \rfloor$ is the easiest reasonable factorial problem you can imagine. (Although we could have also defined $\lfloor(0! = 1) \rfloor$ as the base case and this would have worked too. Indeed, a mathematician could give us convincing arguments why we should allow for $\lfloor(0!) \rfloor$.)

Takeaway message: *Recursion is not magic! It is simply one function calling another, but it just happens that*

the function that we're calling has the same name as the function that we're in!

2.9 Building Recursion Muscles

Let's do another example with recursion to flex our recursion muscles. Imagine that we want to write a function that takes a string as an argument and returns the reversal of that string—that is, the string in reverse order. So, if we give this function “spam”, it should return the string “maps.” If we give it “alien”, it should return “neila”.



Did you know that “Neila” is a town in Spain? Population 235.

The secret to writing a recursive function is to try to identify how solving a “smaller” version of the problem would allow us to get closer to solving the original problem. In the case of computing the factorial, we observed that computing the factorial of $\backslash(n\backslash)$ is easy if we know the factorial of $\backslash(n-1\backslash)$. Aha! Now we can use recursion to compute the factorial of $\backslash(n-1\backslash)$ and when we get that answer, we can multiply it by $\backslash(n\backslash)$ and we've solved our problem. This is what is sometimes called the *recursive substructure property*. That's a fancy term that really means that computing the factorial of a number can be viewed as first computing a slightly simpler factorial problem and then doing just a bit of extra work (in this case, multiplying by $\backslash(n\backslash)$ at the end).

Similarly, we'd like to find the recursive substructure in reversing a string. If we want to reverse a string like “spam”, we observe that if we chopped off the first letter, resulting in “pam” and then reversed “pam”, we'd be nearly done. The reversal of “pam” is “map”. Once we have “map”, we can add that chopped-off “s” to the *end* and we get “maps”. Generalizing this, we can say that to reverse a string, we can chop off the first letter, reverse the remainder, and then add that first letter to the end.

Using this rule, we see that to reverse “spam” we will first reverse “pam” (and then add the “s” to the end). To reverse “pam”, we'll first reverse “am” (and then add the “p” to the end). To reverse “am”, we'll first reverse “m” (and then add “a” to the end). To reverse “m” what will we do? Well, continuing in this spirit, we'll chop the “m” off and we'll be left with “”—a string with no symbols. This is called the *empty string*. It seems like a strange and perhaps even invalid string, but one could say the same thing about the number zero and yet we all agree that zero is a perfectly OK number.



You Earthlings are geniuses for inventing zero. It's like nothing I've ever seen!

But now we have a problem. How do we reverse the empty string? Simple: this is the base case of our recursion. If we're asked to give the reversal of the empty string, it's clearly just the empty string itself!

So, continuing with our example, when we reverse “”, we return “”. Now, we concatenate the “m” to that and we return “m”. That's the reversal of “m”. Recall that it was the reversal of “am” that requested the reversal of “m”. Now that we have that, the reversal of “am” is “m” with the “a” concatenated at the end, which is “ma”. It was the reversal of “pam” that was waiting patiently for the reversal of “am”. “Thank you,” it says. “I've been waiting for the answer, which I see now is ‘ma’ and I will concatenate my ‘p’ to the end of it and get ‘map’.” Finally, it was the reversal of “spam” that requested the reversal of “pam”. It gets “map”, concatenates the “s”, and returns “maps”. Wow!

While it's instructive to trace through this logic step by step to understand what's happening, it can be aggravating to do this every time you write a recursive function. So now we'll take a small leap of faith and try to write this recursive function in Python, based on our observation of the recursive substructure.

We've agreed that if the string is empty, the problem is easy—we just return the empty string and we're done. That's the base case and we take care of that first. If the string is not empty, we'll find the first symbol in the string and “store” it away for later. In other words, we'll have a variable that keeps that first symbol for later use. Then, we'll slice off the first symbol, reverse the remaining string, and add the first symbol to the end of the resulting string.

Remember that a string can be defined using either single quotes or double quotes. We'll use single quotes throughout, just for consistency.

```

def reverse(string):
    """Takes a string as an argument and returns
    its reversal."""
    if string == "":
        # Is the string empty?
        return "" # If so, reversing it is easy!
    else:
        firstSymbol = string[0] # Hold on to the first symbol
        return reverse(string[1:]) + firstSymbol

```

At this point you might be starting to notice a pattern for writing recursive functions with strings. The base case is (often) when the string(s) are empty, and the recursive call is (often) made on the string without its first character. We'll see this pattern come up again and again. What differs from function to function is what happens with the result of the recursive call, and what is returned in each case.

2.10 Use It Or Lose It

We're almost ready to completely solve our edit distance problem. In fact, we have all the programming tools we need to do so. However, the general edit distance problem is a substantially more complicated problem than reversing a string, and we're missing some problem-solving tools that will help considerably. So in this section we're going to introduce a general approach to solving a large class of problems (including the edit distance problem), which we call "Use It Or Lose It."

Our alien has a predicament: it's planning to return to its home planet soon (after the Harry Potter 17 debut and a massive shopping spree at the local mall) and has acquired way more stuff than will fit in its suitcase. The alien would therefore like to select a subset of items whose total weight is as close as possible to the suitcase capacity, but without exceeding it. (Greedy creature!)

For example, imagine that the suitcase capacity is 42 units and there are items with weights 5, 10, 18, 23, 30, and 45. In this case, the best we can do is to choose the weights 18 and 23 for a total of 41. On the other hand, if an item with weight 2 is also available, we can get exactly 42 by choosing the weights 2, 10, and 30.

So, what we'd like is a function that takes two arguments: a number representing the suitcase capacity and a *list* of *positive* numbers (in no particular order) representing the weights of the items. The function should then return the largest total weight of items that could be chosen without exceeding the suitcase capacity. We'll call our function `subset` and we can imagine using it (once it's written!) this way:

```

>>> subset(42, [5, 10, 18, 23, 30, 45])
41
>>> subset(42, [2, 5, 10, 18, 23, 30, 45])
42

```

A slightly more ambitious task would be to actually report the set of items that gives us this best solution, but let's not worry about that for now.

We start by thinking about the base case. What are the "easy" cases where the `subset` function needs to do almost no work? Clearly, if the capacity that we're given is 0, we can't take any items, so we should return 0 to indicate "sorry, the maximum sum that you can attain is 0." So, we can start this way:

```

def subset(capacity, items):
    """Given a suitcase capacity and a list of items
    consisting of positive numbers, returns a number
    indicating the largest sum that can be made from a
    subset of the items without exceeding the capacity."""

```

```

if capacity == 0:
    return 0

```

Actually, we could also return zero if the capacity is *less than* zero; we'll see that below. But there's another "easy" case that we haven't handled. What if the list of items is empty? In that case, we also can't take any items. We could handle this with an `elif` after the `if` statement:

```

elif items == []:
    return 0

```

Alternatively, since we plan to return 0 if the capacity is less than or equal to 0 *or* the list is empty, we could simply modify the `if` statement above to say:

```

if capacity <= 0 or items == []:
    return 0

```

Recall that we advocate taking care of the base cases first, since this takes care of the “easy” situations. Notice that since there were two arguments to subset, we should expect that there will be two base cases to handle: either of the arguments could be “easy” (capacity is ≤ 0) or list of items is empty). Often, the number of base cases is equal to the number of arguments to the function.

OK, now for the actual recursion! Somehow we need to find the recursive substructure in this problem. Let’s take a look at the first item in the given list. That list isn’t necessarily in any particular order, but it’s easy to pick on the first item so let’s start with it. (We could have just as well looked at the last item or any other, but Python makes it particularly easy to look at the first one, since it’s just `items[0]`.)

If that first item happens to be larger than the capacity of the suitcase, we have no choice but to toss it out. In that case, we’re confronted with a simpler problem: Find the best solution with the given capacity but with a list that has had the first item removed. We’d like to call some function to help us find that solution. What function can do that? Our own `subset` function! So, here’s what we have so far:

```

def subset(capacity, items):
    """Given a suitcase capacity and a list of items
       consisting of positive numbers, returns a number
       indicating the largest sum that can be made from a
       subset of the items without exceeding the capacity."""
    if capacity <= 0 or items == []:
        return 0
    elif items[0] > capacity:
        return subset(capacity, items[1:])

```

Remember, there is no magic here! We’re simply going to call a function that happens to be the same function as the one we are in.

Finally, if that first item, `items[0]`, is not greater than the capacity, we might want to use it, but not for sure. For example, if the capacity was 10 and the list of items was [8, 4, 6] we *could* use the item with value 8, but if we do use it we can’t take any other items (because the remaining items in the list will exceed the remaining capacity). A better solution would be *not* to use it and take the items with values 4 and 6 to get a solution with total value 10. So, the question here is should we “use it or lose it”? (The “it” here being the item with value 8.)

We don’t yet know the answer to that question. However, we can make *two* recursive calls, one that finds the best solution that “uses” the item with value 8 and one that finds the best solution that “loses” the value 8. The better of these two solutions must be the best solution overall. After all, with respect to that item with value 8, any solution must either use it or lose it!



In this case, the recursion is trying to “e-value-8” the two options!

The case that we lose the first item in the list is easy. We just want to find the best solution with the same capacity and the list `items[1:]`. If we choose to use that item, we now get the value of that item in our solution, but our capacity is now reduced by the weight of that item. So our complete function will look like this:

(ch02_using)

By the way, the function `max` is built in to Python; it can take any number of arguments and returns the maximum of all of them. In this case, we’re using `max` to return the better of our two options: “use it” and “lose it.”

The “use it or lose it” paradigm is a powerful problem-solving strategy. It allows us to exploit recursion to explore all possible ways to construct a solution. We’re now (finally) ready to return to the problem that started off this chapter.

2.11 Edit distance!

Now that we've seen recursion and the "use it or lose it" strategy, we are ready to solve our original problem: Finding the edit distance between two strings. Recall that the objective is to find the least number of substitutions, insertions, and deletions required to get from one string to another. As we noted at the beginning of the chapter, it doesn't matter whether we choose to change the first string into the second or vice versa, so we'll choose to start with the first string and try to transform it into the second string.



Too bad! I love sales.

Just as a quick reminder, here's an example of the full version of the problem: Consider transforming the string "alien" into the string "sales." We can begin by inserting an "s" at the front of "alien" to make "salien". Then we delete the "i" to make "salen." Then we replace the "n" with an "s" to make "sales." That took three operations, and indeed it is not possible to transform "alien" to "sales" with fewer than three operations.

Our objective is to write a function called `distance` that will take two strings as arguments; we'll call them `first` and `second`. Our solution will use recursion, so we'll start with the base case. Since there are two arguments, we should expect that there will be two base cases and that they will be the "easy" or "extreme" cases.

One extreme is that one (or both) of the strings are empty. For example, imagine that `first` is the empty string. For the moment, let's assume that `second` is not empty—for example it might be "spam". Then the distance between the two strings must be the length of `second` since we must insert that many letters into the empty string to get to `second`. Similarly, if `second` is empty then the distance must be the length of `first`, since we must delete that many symbols from `first`. So, let's start our function accordingly:

```
def distance(first, second):
    """Returns the edit distance between first and second."""

    if first == "":
        return len(second)
    elif second == "":
        return len(first)
```

We're not done yet, but let's pause here and ask ourselves what would happen in the case that both strings were empty. In this case, the distance should be 0; notice that this is what we'll get since `first` is empty and we'll get the length of `second`—which is 0. It's always good to check these kinds of special cases to make sure we haven't missed anything.

Now for the recursion. If `first` and `second` begin with the same symbol it's pretty clear that we should consider ourselves fortunate and not mess with those matching characters. In this case, the distance between the two strings is just the distance between the two strings with the first symbol of each sliced off. That is, we want the distance between `first[1:]` and `second[1:]`. For example, when computing the distance between `spam` and `spim` (notice that the distance is 1), the fact that they begin with the same letter lets us conclude that the distance is going to be the same as the distance between `pam` and `pim`. So, we can make a recursive call `distance(first[1:], second[1:])`.

On the other hand, if the two strings begin with different letters, the problem is more interesting. Since the first letters don't match, some sort of change will be required. We can either change the first symbol of the first string to match the first symbol of the second (a substitution), remove the first symbol of the first string (a deletion), or add a new symbol at the very front of the first string (an insertion). We don't know which of these is best, so we'll let recursion explore all three options for us. This is like the "use it or lose it" strategy but now we have *three* choices to consider rather than two.

Let's consider each of the three options. If we are going to perform a substitution, it's not hard to see that we should change the first letter in `first` to be the same as the first letter in `second`. Now these letters will match and we can remove those two letters from further consideration. Therefore, the best solution that begins with a substitution will have cost $1 + \text{distance}(\text{first}[1:], \text{second}[1:])$ because the number of operations will be 1 (the substitution) plus however many operations are required to get from the remaining string `first[1:]` to the remaining string `second[1:]`.

The second option is deleting the first symbol in `first`. That's one operation, and now we would need to find the distance from `first[1:]` to `second`.

Lastly, we need to consider inserting a new symbol to the front of `first`. It's not hard to see that the new symbol should match the first symbol in `second`. That will require one operation, and then the remaining problem is to find the distance from `first` to `second[1:]`. That's because we haven't yet used the first symbol in `first`, but we've just matched the first symbol in `second`.

The best of these three options will be our best solution! Putting it all together, here's our function:

```
def distance(first, second):
    """Returns the edit distance between first and second."""

    if first == "":
        return len(second)
    elif second == "":
        return len(first)
    elif first[0] == second[0]:
        return distance(first[1:], second[1:])
    else:
        substitution = 1 + distance(first[1:], second[1:])
        deletion = 1 + distance(first[1:], second)
        insertion = 1 + distance(first, second[1:])
        return min(substitution, deletion, insertion)
```

Like `max`, `min` is built in to Python and returns the minimum of all of its arguments.

Wow! That's a remarkably short program that solves a challenging and important computational problem. Here are a few examples of us using this function.

```
>>> distance('spam', 'poems')
4
>>> distance('alien', 'sales')
3
```

We encourage you to try it for yourself!

2.12 Conclusion

This chapter has led us from the basics of Python to writing powerful recursive functions. This style of programming—using functions and recursion—is called *functional programming*.



I tried Googling “recursion”. It came back and said “Did you mean: recursion”

Amazingly, what we've seen so far is enough to write *any* possible program. To be a bit more precise, the Python features that we've seen here are sufficient to write *any* program that can be written in *any other language*. You may wonder how we dare make such a bold assertion. The answer is that there is a lovely part of computer science called *computability theory*, which allows us to actually prove such statements. We'll see some aspects of computability theory in Chapter 7.

In spite of the fact that we are now functional (in both senses of the word) programmers, there are some beautiful ideas that will allow us to write more efficient, succinct, and elegant programs, and we'll see some of these ideas in the next chapter. But before going there, you'll probably want to write some programs of your own to become comfortable with recursion.

In the meantime, we leave you with this quip from a former student of ours: “To understand recursion, you must first understand recursion.”

Table Of Contents

- Chapter 2 : Functional Programming

- 2.1 Humans, Chimpanzees, and Spell Checkers
- 2.2 Getting Started in Python
 - * 2.2.1 Naming Things
 - * 2.2.2 What's in a Name?
- 2.3 More Data: From Numbers to Strings
 - * 2.3.1 A Short Note on Length
 - * 2.3.2 Indexing
 - * 2.3.3 Slicing
 - * 2.3.4 String Arithmetic
- 2.4 Lists
 - * 2.4.1 Some Good News!
- 2.5 Functioning in Python
 - * 2.5.1 A Short Comment on Docstrings
 - * 2.5.2 An Equally Short Comment on Comments
 - * 2.5.3 Functions Can Have More Than One Line
 - * 2.5.4 Functions Can Have Multiple Arguments
 - * 2.5.5 Why Write Functions?
- 2.6 Making Decisions
 - * 2.6.1 A second example
 - * 2.6.2 Indentation
 - * 2.6.3 Multiple Conditions
- 2.7 Recursion!
- 2.8 Recursion, Revealed
 - * 2.8.1 Functions that Call Functions
 - * 2.8.2 Recursion, Revealed, Really!
- 2.9 Building Recursion Muscles
- 2.10 Use It Or Lose It
- 2.11 Edit distance!
- 2.12 Conclusion

[Previous topic](#) Chapter 1: Introduction

[Next topic](#) Chapter 3: Functional Programming, Part Deux

Quick search

Enter search terms or a module, class or function name.

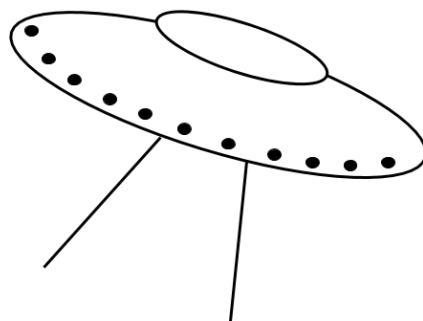
Navigation

- [index](#)
- [next](#) |
- [previous](#) |
- [cs5book 1 documentation](#) »

© Copyright 2013, hmc. Created using Sphinx 1.2b1.

CSforAll - PythonWelcome

CS for All



CSforAll Web > Chapter2 > PythonWelcome

This website accompanies the textbook *CS For All* by Alvarado, Dodds, Kuenning, and Libeskind-Hadas

Welcome to Python!

The *goals* of this assignment are:

- To make sure you've successfully installed IPython and a text editor
- To introduce you to a command-line interface, text editing, and Python
- To read, edit, and run a "Hello" program in Python

- To challenge you to solve some "four fours" problems in Python

Installing Software

The assignments use some software you might not already have. As a default, the programs we recommend using are:

- **Python**—our version is the Anaconda scientific distribution (Anaconda version 5.2, which includes Python 3.6). *You'll need Python3.x*
- **A text editor**—not a Word processor—for editing your Python files. By default, we'll use VSCode, a free, widely adopted text editor available for all operating systems (Mac, Win, Linux). *Already have a favorite? Your text editor is OK.*

The Mac and Windows links above are HMC-local; the Linux links are directly to the applications' sites.

Go to this page for instructions on how to download and install:

- Anaconda's distribution of Python3
- A text editor, *VSCode*, a free and modern text editor

Try Out VSCode, A *Text Editor*

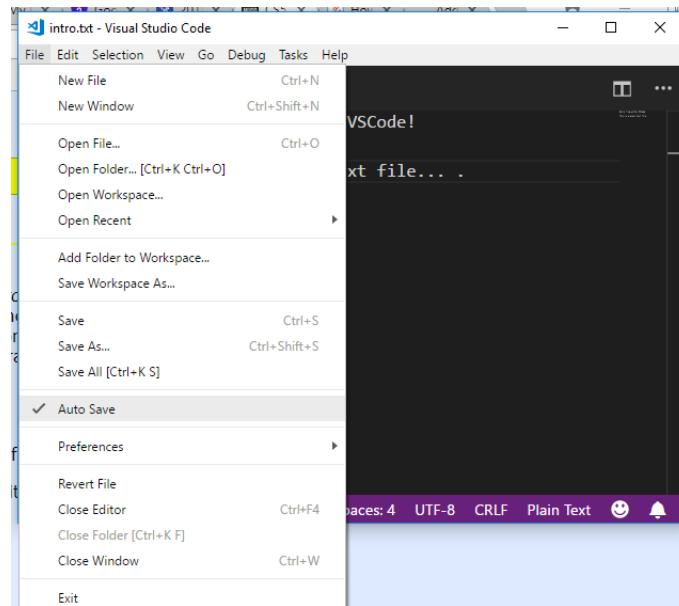
- Text editors are different from word processors.
 - Microsoft Word or Google Docs or Apple's Pages or any *word processor* can format things beautifully
 - But they *don't* provide direct access to the actual contents of the file!
 - Special, non-visible characters carry around the formatting information
 - Since all programming languages use ***raw text*** (strings of characters), (plain-)text editors are the right toolset for CS!
- So, start your text editor—likely VSCode, if you just installed it...
- Get rid of the intro tabs (you might tell VSCode not to bother you with those) and get to a "blank file" (an empty window).
 - Then, type some text, for example:

```
Untitled-1 - Visual Studio Code
File Edit Selection View Go Debug Tasks Help
1 Welcome to VSCode!
2
3 This is a plain-text file... .
Ln 3, Col 33 Spaces: 4 UTF-8 CRLF Plain Text
```

- Go ahead and save that file as `intro.txt`
 - Save it to your Desktop or to a dedicated folder for the class—it's up to you. It'll look something like this:

```
intro.txt - Visual Studio Code
File Edit Selection View Go Debug Tasks Help
intro.txt x
1 Hello from within VSCode!
2
3 This is a plain-text file... .
Ln 3, Col 32 Spaces: 4 UTF-8 CRLF Plain Text
```

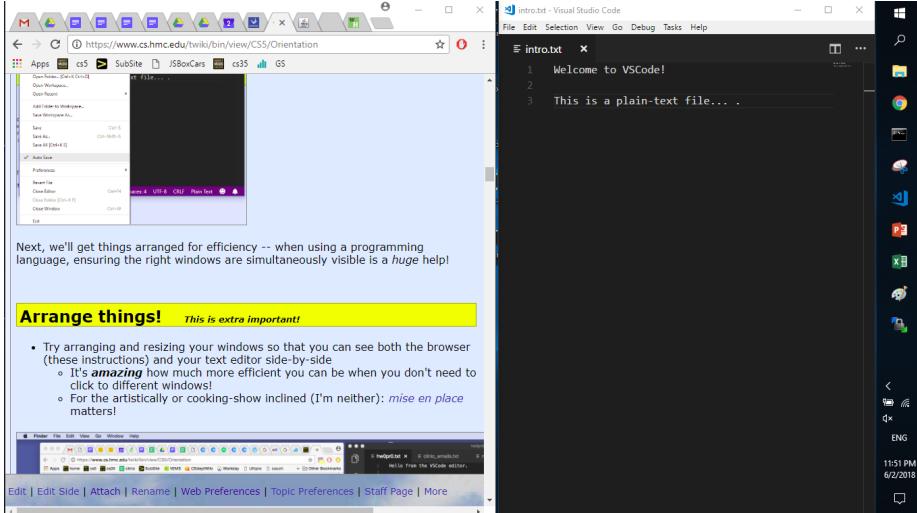
- *Autosave!* - it is a great idea to set VSCode to autosave all of your files. Here is where the option is located:



Next, we'll get things arranged for efficiency -- when using a programming language, ensuring the right windows are simultaneously visible is a *huge* help!

Arrange Things! *This is Extra Important!*

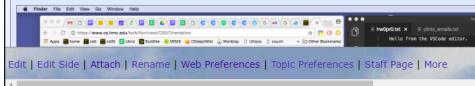
- Try arranging and resizing your windows so that you can see both the browser (these instructions) and your text editor side-by-side
 - It's **amazing** how much more efficient you can be when you don't need to click to different windows!
 - For the artistically or cooking-show inclined (I'm neither): *mise en place* matters!
- Here's an example of a Windows setup:



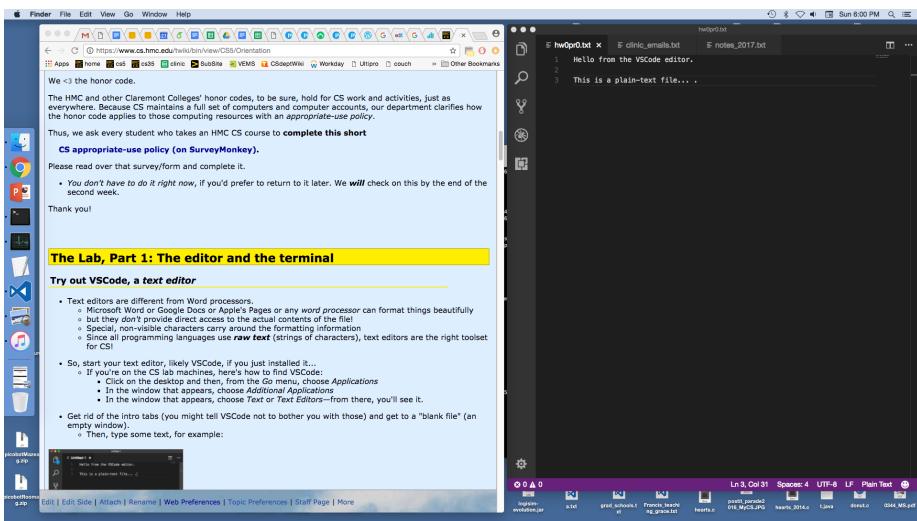
Next, we'll get things arranged for efficiency -- when using a programming language, ensuring the right windows are simultaneously visible is a *huge* help!

Arrange things! This is extra important!

- Try arranging and resizing your windows so that you can see both the browser (these instructions) and your text editor side-by-side
 - It's **amazing** how much more efficient you can be when you don't need to click to different windows!
 - For the artistically or cooking-show inclined (I'm neither): *mise en place* matters!



- and a Mac setup:



- Explore the VSCode menus a little bit. I usually choose *View* and then *Hide Status Bar* and *Hide Activity Bar...*. But changing the color theme is even more fun!
- *Color theme!* Don't spend too much time on it, but feel free to choose your favorite color theme for VSCode (menus: File/Code ... Preferences ... Color Theme) This is great fun -- perhaps it can be too much fun...!)

Next, you'll open a *Terminal Window...*

The Terminal Window!

Terminal Window?

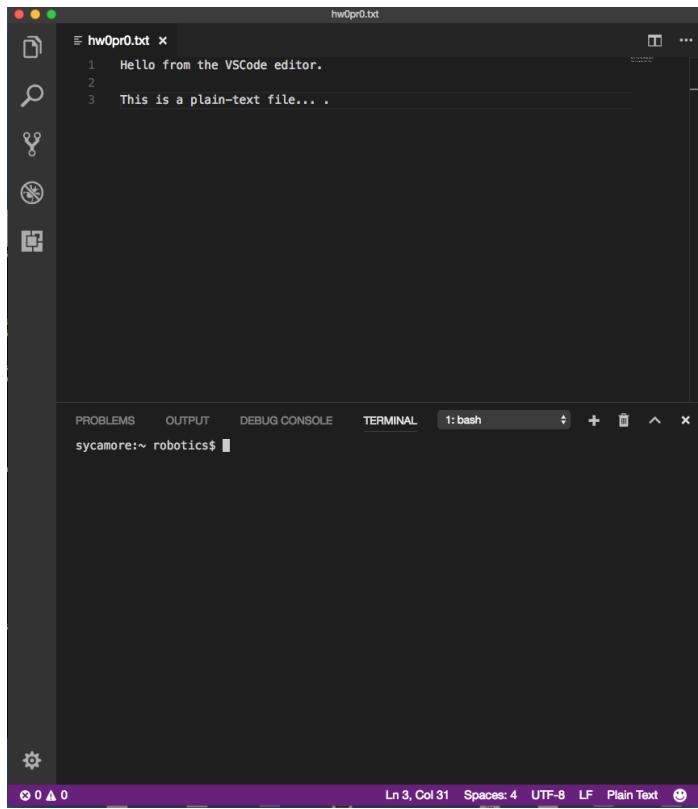
Most of our interactions with computers are through the windows provided by the OS. The "OS," short for *Operating System*, is usually either Windows or Mac OS, although there are many others, Linux being the most common. The OS you use provides a *windowing system* that lets you use a mouse, interact with the computer in an intuitive fashion, and even watch movies. For sure, the click-and-drag interface of today's windowing systems is a great convenience!

However, the graphical interface is also a curtain—one that disconnects users from what's happening with the various files on their device. It's a flexible and really useful skill to have a clear picture of how files are being used *behind* the windowing system's curtain. The Terminal is a program that gets "behind" that curtain. It uses textual commands, the so-called "command line", to handle files and actions on your machine.

Feeling comfortable around the command line will come in handy for a lot of pathways that *create* computation. The OS already handles *consuming* computation brilliantly!

Start your terminal!

So, pull back that OS "curtain," go to your VSCode text-editor and, among the menus, choose *View* and then *Integrated Terminal*. You should see a split screen:



The upper half is still the text editor.

The lower half is now your terminal window. The terminal goes by many names:

- The terminal
- The shell
- The command line

If this is your first time using the terminal, Yay!

Start IPython...

The next section introduces command-line navigation—it's great stuff!

First, start `ipython`

To do this, make sure the terminal window has the focus, and type

```
ipython
```

at the terminal's prompt.

If ipython isn't found, then

- perhaps it's not installed -- that's earlier in this lab...
- perhaps you're on windows and windows can't find where ipython was installed (this is common)
- if you suspect it's this, try pasting or typing C:\Users\zdodds\Anaconda3\scripts\ipython3.exe
 - ... be sure to replace zdodds with your windows username
- and/or, if you're having any troubles, please ask! This is where the grutors/instructors can help!!

For future, convenience, it's possible to add the location of ipython and ipython3 to your Windows "PATH" - this link explains how. The location to add is C:\Users\zdodds\Anaconda3\scripts\, but be sure to replace zdodds with your username.

The Command-Line Terminal

Everything you can do with the windows of your OS, you can do at the terminal and command-line. (Actually, you can do much *more* with the command-line... .)

For the assignments ahead, you'll need to know three terminal commands. We'll dedicate a short section to each. Here's a preview:

- **pwd** short for *print working directory*. It prints your current location (folder).
- **ls** short for *list (files)*. It prints all the files at your current location.
- **cd** short for *change directory*. It lets you move around your computer from folder to folder.

pwd

First, the *prompt*. The prompt is the bit of text on the left that's waiting for you to type something to the command line:



The **pwd** command is short for *print working directory*. It prints your current location (folder). Try it:

```
In [1]: pwd  
Out[1]: 'C:\\Users\\zdodds'  
  
In [2]: █
```

you'll see the location at which your terminal and command-line are currently running. Unless your name is really similar to mine, your results will be different!

- Also, the formatting will be different in you're using a Mac - no problem at all
- Notice, too, that a next prompt has appeared, waiting for another command...

The output is the *name of the folder* at which I'm currently located in the terminal. "Folder" and "directory" are the same thing.

The slash characters / show subfolders. On Windows, they're usually backward-slashes \ or double backward-slashes: not an important difference.

Thus, I'm currently in the subfolder named `zdodds`, within the folder named `Users` on the drive named `C`. This is my "home directory."

You'll likely be in your own home directory (and you'll see its full pathname).

Next, we'll see what's around with the `ls` command... .

ls: the *List* Command

The `ls` command stands for *list*.

Running `ls` lists everything in your current directory. For me, typing `ls` does this:

```
In [5]: ls
Volume in drive C has no label.
Volume Serial Number is 2E4A-FCEA

Directory of C:\Users\zdodds

06/03/2018  12:14 AM    <DIR>        .
06/03/2018  12:14 AM    <DIR>        ..
06/03/2018  12:14 AM    <DIR>        .atom
04/02/2018  11:20 AM    <DIR>        .conda

06/03/2018  12:06 AM    <DIR>        Desktop
05/11/2018  02:56 AM    <DIR>        Documents
05/11/2018  02:56 AM    <DIR>        Downloads
05/11/2018  02:56 AM    <DIR>        Favorites
05/11/2018  02:56 AM    <DIR>        Links
05/11/2018  02:56 AM    <DIR>        Music
06/02/2018  05:44 PM    <DIR>        OneDrive
05/11/2018  02:56 AM    <DIR>        Pictures
12/19/2016  11:19 PM    <DIR>        pip
05/11/2018  02:56 AM    <DIR>        Saved Games
05/11/2018  02:56 AM    <DIR>        Searches
05/11/2018  02:56 AM    <DIR>        Videos
          0 File(s)           0 bytes
         27 Dir(s)  326,749,069,312 bytes free

In [6]: []
```

The output is a list of all of the files and (sub)folders in the current directory. (Remember, "directory" and "folder" are the same thing—the terms are used interchangeably.)

Since this is Windows, you're seeing lots of the default subfolders in every Windows home directory, plus a few extra things. On a Mac, there will be some differences:

```
sycamore:~ robotics$ pwd  
/Users/robotics  
sycamore:~ robotics$ ls  
Applications Terminal Saved Output.txt  
Desktop anaconda  
Documents anaconda2  
Downloads doddsLocal  
Library doddsLocal_oldMacG4Tower  
Movies doddsTabletPC_May2011  
Music dwhelper  
Pictures nltk_data  
Public software_files_ISOs  
Sites t.txt  
sycamore:~ robotics$ █
```

Try it to see the list of names of the files and subfolders in your current directory at the terminal.

Next, you'll "move around" from directory to directory with `cd... .`

cd: the *Change Directory* Command

The **cd** command is the most important. It stands for *change directory*.

The **cd** command lets you move from your current folder to other folders (directories) around your computer. To use it, you need to have a place to go!

I'd recommend the Desktop... so, type `cd Desktop` Here's an example:

```
In [6]: cd Desktop  
C:\Users\zdodds\Desktop
```

Not much happened...until you type `pwd` and see you're in a new place:

```
In [7]: pwd  
Out[7]: 'C:\\Users\\zdodds\\Desktop'
```

Also, type `ls` and there could be a *lot* more files—well, depending on how cluttered your Desktop is! *Try it!* Here's a bit of mine:

```
In [9]: ls
Volume in drive C has no label.
Volume Serial Number is 2E4A-FCEA

Directory of C:\Users\zdodds\Desktop

06/03/2018  12:25 AM    <DIR>      .
06/03/2018  12:25 AM    <DIR>      ..
05/30/2018  08:23 AM   4,017,144 2017-Taulbee-Survey-Report.pdf
06/02/2018  07:04 PM   14,806 2018 and 2017 summer startup participants for HMCEN.docx
09/22/2016  07:25 AM   23,168 alien.png
06/02/2018  10:26 PM   661,987,080 Anaconda3-5.2.0-Windows-x86_64.exe
06/02/2018  11:40 PM   30,414 autosave.png
09/08/2017  09:03 PM   <DIR>      biocs_data
05/25/2018  09:09 PM   5,298,525 class03-gold-18-data.pptx
```

Your *filage* may vary!

Next, you'll move "up" within the directory structures...

cd.. : Moving "Up" One Folder

Ok! You cd'ed to the Desktop—*how do you get back?!*

The special symbol of two periods in a row .. means "the next directory up."

So, if you type cd .. and hit return, you will be back in the folder that *contains* your Desktop. Try it:

```
In [10]: cd ..
C:\Users\zdodds

In [11]: pwd
Out[11]: 'C:\\Users\\zdodds'

In [12]: ls
Volume in drive C has no label.
Volume Serial Number is 2E4A-FCEA

Directory of C:\\Users\\zdodds

06/03/2018  12:14 AM    <DIR>      .
06/03/2018  12:14 AM    <DIR>      ..
06/03/2018  12:14 AM    <DIR>      .atom
04/02/2018  11:20 AM    <DIR>      .conda
```

Then, try `pwd` and `ls`. You might notice that two of the directories listed are `..` (the next directory "up") and `.` which refers to the current directory.

Practice!

Try these -- at least for your set-up. You'll find that Mac doesn't show the directory by default. Use `pwd` in that case!

```
In [14]: cd .
C:\Users\zdodds

In [15]: cd ..
C:\Users

In [16]: cd zdodds
C:\Users\zdodds

In [17]: cd Desktop
C:\Users\zdodds\Desktop

In [18]: cd ..
C:\Users\zdodds

In [19]: cd Desktop
C:\Users\zdodds\Desktop
```

That's it! You're set for the command-line. There are *very* worthwhile shortcuts that make the command line *much* more efficient—more efficient than the drag-and-drop windows interface! For instance,

- *tab-completion* From your home directory, type `cd Des` and hit the *tab* key. The command-line will try to complete your thoughts. Experienced (and lazy!) users nearly always tab-complete long names, instead of typing them. It's a great time-saver!
- *up-arrow* and *down-arrow* The up-arrow and down-arrow keys remember everything you've done before. Enter a command once, and you can just up-arrow to get it back. You can also re-edit it (the left- and right-arrow keys will work) if you made a mistake...err, can I left-arrow that?

Ok! We're ready for Python. We'll use ***ipython*** next.

Python!

Python is a programming language, sometimes called a *scripting* language. Scripting languages have an interactive command line.

In fact, you've been using the Python command line!

Your ipython command-line is a place to experiment with the Python language.

- The "prompt" tells you that Python is ready to go.
- You might try `6*7` as a first computation at the prompt.
- ***Caution:*** deep wisdom may result!

Here it is

```
In [1]: 6*7  
Out[1]: 42
```

If everything is working so far, try some larger computations...

Try computing a googol (ten to the hundredth power). Google is (loosely) named for this number.

- The power operator in Python is two asterisks `**`.
- So, at the prompt you would type

`10**100`

```
In [5]: 10**100
```

All right... but can we go further! (***Note: this is actually NOT a good idea***) So, only if you're feeling reckless or angry at your computer, you can distress-or-crash ipython, and maybe your whole machine, by asking it to compute a *googolplex*, which is ten to the googol power:

- ```
In [7]: 10**10**100
```
- It won't work. In fact, this really is *not* such a good idea: you will likely have to kill and restart at least Python, and your computer might slow down frustratingly in the meantime.
- But, this *does* show how easy it is to reach computers' numeric limits!
- **IF** for some reason you do decide to test this, you can cancel the calculation by pressing `ctrl-c`, or by killing the terminal with the `x` at its upper right,

or by killing the whole VSCode application, or by holding down the power button of your machine, or unplugging both it and its battery...

Continuing for the less reckless. Now, copy-and-paste (or type) this line of Python code:

- `print("Zero is", 4+4-4-4)`
  - You should see the output line: Zero is 0
  - Notice that you used exactly four fours to create the numeric value 0 here. Here's ours:

Extending this idea is the next -- and primary -- challenge of this week's lab!

First, we'll see how to use a *file* to hold our Python code, so that it can be submitted...

## Running Python From a *File*

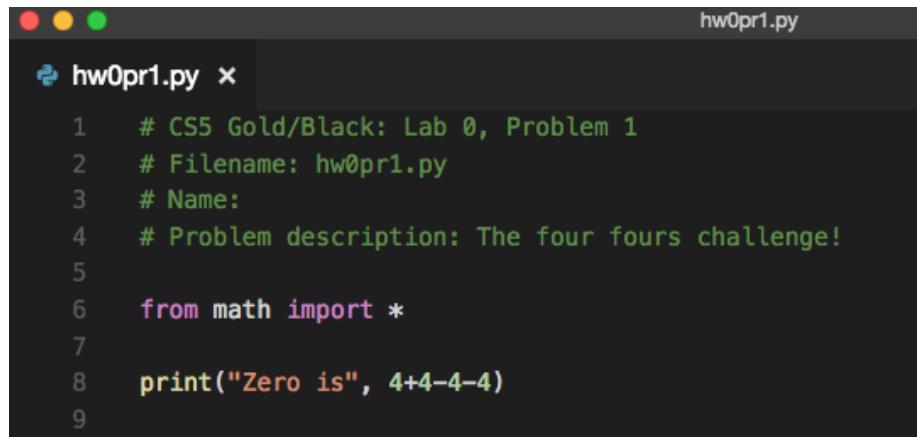
We already have a text editor open!

So, in a new tab, create a new empty file (menus: *File - New File*)

Then, into this new empty file paste (or type)

```
from math import *
print("Zero is", 4+4-4-4)
Save this file on your Desktop
```

- Later on, you can save your Python programs anywhere—it's best to keep them organized in folders.
- But for the purposes of this lab, we'll presume you've saved it to your Desktop.
- Also, you **do** need to type the .py extension
- When you save it with a .py extension (as it's called), you will see the Python source code *colorize*
  - This is also called *syntax highlighting*
  - If yours doesn't colorize, ask! It's important to have those colorful cues as to the structure of your programs. Ours:



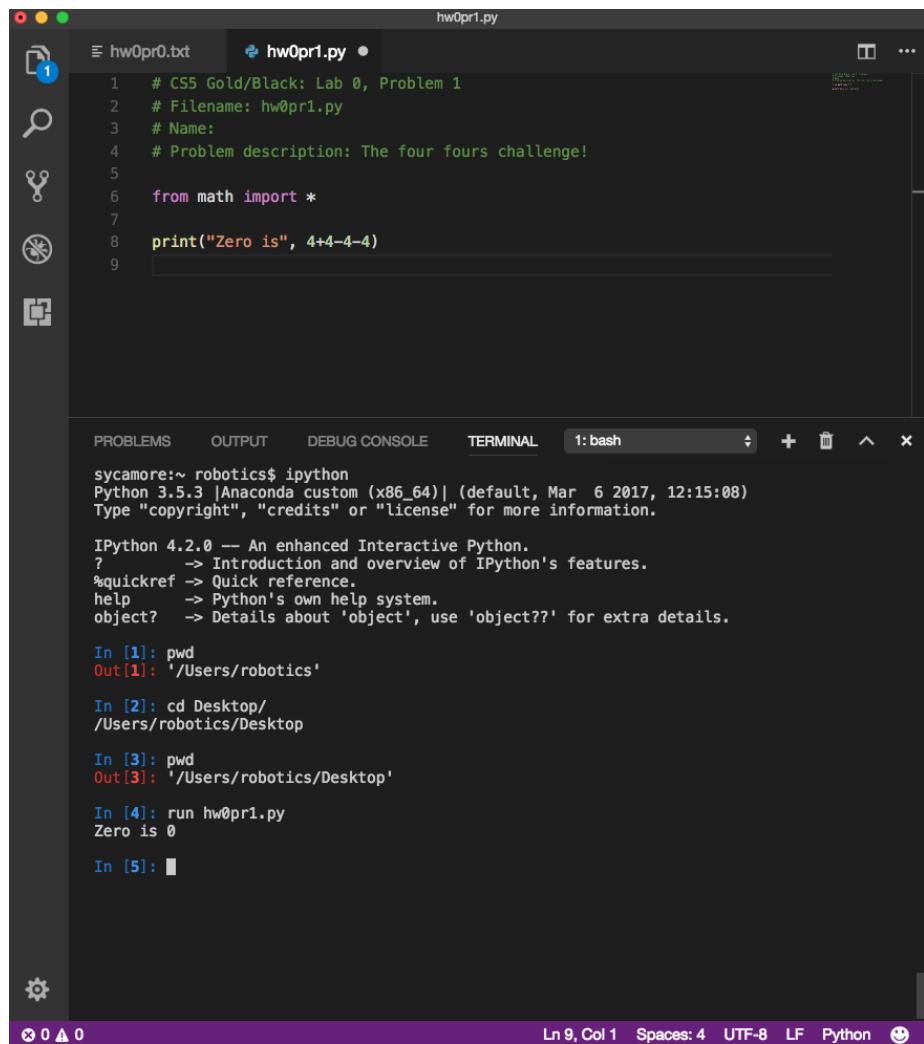
```
hwOpr1.py
hwOpr1.py ×
1 # CS5 Gold/Black: Lab 0, Problem 1
2 # Filename: hwOpr1.py
3 # Name:
4 # Problem description: The four fours challenge!
5
6 from math import *
7
8 print("Zero is", 4+4-4-4)
9
```

- **Windows users:** please show all filename extensions!
  - If you're on Windows, we ask that you turn *on* the view of file extensions (such as .py, .txt, .doc, and so on).
  - *It can be confusing if Windows is hiding the extensions*, because the .py or .txt extension needed might—or might not!—be there.
  - If your Windows' folders are not showing the .py file extensions, please enable all file extensions by following the directions on this page.
- **Running a file!**

To run your file, go back over to the terminal.

- Be sure you're running ipython
- Type ls (windows or mac) to see the files in the current directory
- Make sure your file is there!
  - \* If not, use cd .. or cd Desktop or other combinations to get to the correct directory. Ask for help!
- At the ipython prompt, type your file name (tab completion will work)
- This should run the file
- If all goes well, the program should run and you should see the output

- If not, please ask!
  - Now, you can edit your file, save it, and hit *up-arrow* to re-run it.
- Awesome! Here's ours:



The screenshot shows a Jupyter Notebook interface. At the top, there are two tabs: "hw0pr0.txt" and "hw0pr1.py". The "hw0pr1.py" tab is active, displaying the following Python code:

```

1 # CS5 Gold/Black: Lab 0, Problem 1
2 # Filename: hw0pr1.py
3 # Name:
4 # Problem description: The four fours challenge!
5
6 from math import *
7
8 print("Zero is", 4+4-4-4)
9

```

Below the code editor, there are several tabs: PROBLEMS, OUTPUT, DEBUG CONSOLE, TERMINAL, and 1:bash. The TERMINAL tab is selected, showing a Python session:

```

sycamore:~ robotics$ ipython
Python 3.5.3 |Anaconda custom (x86_64)| (default, Mar 6 2017, 12:15:08)
Type "copyright", "credits" or "license" for more information.

IPython 4.2.0 -- An enhanced Interactive Python.
? --> Introduction and overview of IPython's features.
%quickref --> Quick reference.
help --> Python's own help system.
object? --> Details about 'object', use 'object??' for extra details.

In [1]: pwd
Out[1]: '/Users/robotics'

In [2]: cd Desktop/
Out[2]: '/Users/robotics/Desktop'

In [3]: pwd
Out[3]: '/Users/robotics/Desktop'

In [4]: run hw0pr1.py
Zero is 0

In [5]:

```

At the bottom of the interface, there is a status bar with the following information: 0 0 ▲ 0, Ln 9, Col 1, Spaces: 4, UTF-8, LF, Python, and a help icon.

### The Assignment: *Four Fours!*

- The *four fours challenge!* Now, add several more lines similar to this one so that you compute at least ***16 of the 20 values*** from 0 through 20 using ***exactly four fours***. (We ask for only 16 so that you can skip a couple if they give you trouble!) You may use any of Python's arithmetic

operations:

- + addition
- - subtraction or negation
- \* multiplication
- / division
- ( ) parentheses for grouping
- \*\* power

- Results with decimals: 1.0, 2.0, etc. are totally ok!
- You may also use 44 or 4.4, each of which count as two fours,
- or .4, which counts as one four.
- or, `sqrt`, for example `sqrt(4)` (or others)
- or, `factorial`, for example `factorial(4)` (or others)
- both `sqrt` and `factorial` are from Python's `math` library
- aside: the line `from math import *` is what imports that math library...
- Here are what the results, *but not the source code*, will look like.  
Remember you need only 16 of them:

```
Zero is 0
One is 1
Two is 2
Three is 3
Four is 4
Five is 5
Six is 6
Seven is 7
Eight is 8
Nine is 9
Ten is 10
Eleven is 11
Twelve is 12
Thirteen is 13
Fourteen is 14
Fifteen is 15
Sixteen is 16
Seventeen is 17
Eighteen is 18
Nineteen is 19
Twenty is 20
```

- You may find the four fours game addicting, or frustrating, or both! **Hint:** the power operator is helpful!
- *Extra karma:* feel free to go further than 20, if you'd like.

## Running an *Interactive* Program...

It's not over! To be ready for the rest of the assignments, try running and editing a larger, *interactive* Python program.

Create a new file and paste in the following starting code:

```
import time # includes a library named time
import random # includes a library named random

def rps():
 """ this plays a game of rock-paper-scissors
 (or a variant of that game ...)
 inputs: no inputs (prompted text doesn't count as input)
 outputs: no outputs (printing doesn't count as output)
 """
 name = input('Hi...what is your name? ')
 print()
 print("Hmmm...")
 print()

 if name == 'Eliot' or name == 'Ran':
 print('I\'m "offline" Try later.')

 elif name == 'Zach':
 print('Do you mean Jeff?')
 time.sleep(1)
 print('No?')
 time.sleep(1)
 print('Oh.')

 else:
 print('Welcome, ', name)
 my_choice = random.choice(['rock','paper','scissors'])
 print('By the way, I choose ', my_choice)
```

- Load and run the file with `run [your file name]`
- If it's successful, there will be *no message at all*
- This file's code is in a function named `rps()`
- To run the function (after loading the file), type `rps()` and hit enter at the prompt.
- If it runs, great! (If not, seek out help.) Here's our picture, just before running it:

```
In [14]: run hw0pr2.py
In [15]: rps() █
```

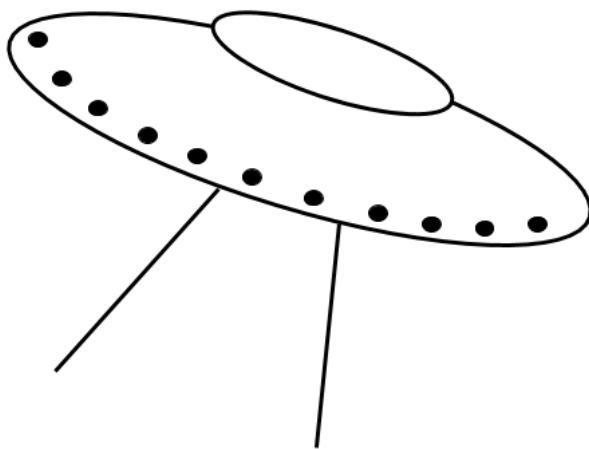
## Handling Errors!

- *To see how Python handles errors*, remove one of the two equals signs in the `elif` line
  - so that the line reads `elif name = 'Zach':`
  - Then, try to re-load the file with `run [your file name]`
  - Python will tell you there is a "syntax error" and give you a chance to fix it.
- **Customize!** Make other edits to the program so that
  - It prints a different message if the user enters your name (or another name)
  - Be sure to run it after you make changes to try it out!

Website design by Madeleine Masser-Frye and Allen Wu

# CSforAll - FunctionFun

## CS for All



CSforAll Web > Chapter2 > FunctionFun

This website accompanies the textbook *CS For All* by Alvarado, Dodds, Kuenning, and Libeskind-Hadas

### Function Fun!

#### Using Built-In Functions

First, try out some of Python's many *built-in* functions. These will *not* go into your file:

```
In [1]: list(range(0, 100))
```

```
Out[1]: [0, 1, 2, ..., 99]
```

`range` returns an iterator of integers. (Don't worry if you're not quite sure what an "iterator" is; the important thing is that `list` turns the iterator into a *list* of integers.)

Note that when you use `range`, as with almost everything in python, the **right endpoint is omitted!**

```
In [2]: sum(range(3, 11))
```

```
Out[2]: 52
sum sums a list of numbers, and range creates a list of integers.
```

```
In [3]: sum([40, 2])
Out[3]: 42
a roundabout way of adding 40 + 2
```

```
In [4]: help(sum)
(an explanation of sum)
help is a good thing—ask for it by name!
Depending on your computer, you may need to hit q in order to leave the help interface...
```

```
In [5]: import math
You do need to import math first...or else you'll get an error!
```

```
In [6]: dir(math)
(All of the functions in the math library...)
```

```
In [7]: dir(__builtins__)
(a huge list of the built-in functions)
```

This `dir` command is often useful—it lists everything from a particular library (or *module*, as it's also called). The special `__builtins__` module holds all of the built-in functions!

To `type__builtins__`, there are two *underscores* both before and after the lowercase letters: the underscore, `_`, is shift-hyphen on most keyboards (between the right parenthesis and the plus sign).

If you look carefully in the big list of stuff returned from `dir(builtins)`, you will find the `sum` function you used above. You will **not**, however, find some other important functions, for example, `sin` or `cos` or `sqrt`. All of these functions (and many more) are in the `math` module. There are many modules (libraries) of functions available for you to use as part of Anaconda Python, along with even more available for download beyond that foundation.

## Importing Other Code (or "Modules")

To access functions that are not built in by default, you need to load them from their modules or libraries. (We use those terms interchangeably.) Try out these examples to get familiar with how to access Python's many libraries:

(1) You can import a module, i.e., a library, and then access its functions with that module name:

```
In [1]: import math
(no response from Python)
```

```
In [2]: math.sqrt(9)
Out[2]: 3.0
Note that sqrt returns a float even if its argument is an int.
```

```
In [3]: math.cos(3.14159)
Out[3]: -0.999...
Note that cos et al. take their arguments in radians. Also, 3.14159 is less than math.pi.
```

(2) Tired of typing `math.` in front of things? You can avoid this with

```
In [1]: from math import *
(no response from Python)
```

The asterisk `*` here means "everything." This will bring all of the functions and constants from the `math` module into your current python environment—and now you can use them without prefacing them with `math!`

```
In [2]: cos(pi)
```

```
Out[2]: -1.0
```

This would have had to be `math.cos(math.pi)` before the new “`from math import *`” import statement.

*Note:* It's not *always* the best idea to do this...especially when you're using lots of libraries—some libraries may share the same function name.

## Create a New File

Next, you'll create a few functions *of your own* in a new file, so use your text-editor to create one.

Functions are the fundamental building blocks of computation. What distinguishes Python from other computing environments is the ease and power of creating your own functions!

Start by pasting the following comments and a definition of a function named `dbl`:

```
def dbl(x):
 """Result: dbl returns twice its argument
 Argument x: a number (int or float)
 Spam is great, and dbl("spam") is better!
 """
 return 2*x
```

## Run Your File

When you run this file in the usual way:

```
run YOUR_FILE_NAME.py
```

you won't see anything! However, your newly defined function, `dbl` is now available for you to use.

From here, *try using* the newly defined `dbl` function:

```
In [1]: dbl(21)
Out[1]: 42
```

```
In [2]: dbl('wow! ')
Out[2]: 'wow! wow! '
```

## Signature and Docstring

The first line of a Python function is called its *signature*. The function signature includes the keyword `def`, the name of the function, and a parenthesized list of arguments to the function.

**Docstring** Directly underneath the signature is a string inside triple quotes ““—this is called the *docstring*, short for “documentation string.” CS5 asks you to include a docstring in all of your functions (even simple ones such as these, so that you will develop the habit early on).

A docstring should describe the function's arguments and result (return value). As you see above, it may include other important information, too. Docstrings are how *your* functions become part of Python's built-in help system.

To see this, type

```
In [2]: help(dbl)
```

and you will see that Python provides the docstring as the help! The language's help system is docstrings! This self-documenting feature in Python is especially important for making your functions understandable, both to others and to yourself.

**Important Warning:** the first set of triple quotes of a docstring needs to be indented underneath the function definition `def` line, at the same level of indentation as the rest of block of code that defines the function.

## Writing Your Own Functions...

Let's go!

For each of these functions, be sure to include a docstring that describes **what your function does** and **what its arguments are** for each function. See the `tpl` example, below, for a reasonable starting point and guide:

**Example problem:** Write the function `tpl(x)`, which accepts a numeric argument and returns three times that argument.

**Answer to example problem:** Copy the following solution (after a few blank lines to leave space and help readability) into your file:

```
def tpl(x):
 """Return value: tpl returns thrice its argument
 Argument x: a number (int or float)
 """
 return 3*x
```

## The Five Functions to Write...

1. Write `sq(x)`, which accepts a numeric argument named `x`. Then, `sq` should return the square of its argument. Note that this is the *square*, not the square root. (The square is `x` times itself...)
2. `interp(low, hi, fraction)` accepts three numbers as its arguments: `low`, `hi`, and `fraction`, and should return the floating-point value that is `fraction` of the way between `low` and `hi`.

### What!?

That is to say, if `fraction` is zero, `low` will be returned. If `fraction` is one, `hi` will be returned, and values of `fraction` between 0 and 1 will lead to results between `low` and `hi`. (In fact, values of `fraction` can go below 0, yielding return values less than `low`, and they can go above 1, producing return values greater than `high`. Purists would call this extrapolation, rather than interpolation, however.)

From the above description, it might be tempting to divide this function into several cases and use `if`, `elif`, and the like. Yet, this function can be written using *no* conditional (`if/elif/else`) constructions at all! Try it *without* using `if`!

As noted, your function should also work if `fraction` is less than zero or greater than one. In this case, it will be linearly extrapolating, rather than interpolating. We'll stick with the name `interp` anyway.

Here are examples that will help clarify how `interp` works:

```
In [1]: interp(1.0, 9.0, 0.25)
A quarter (.25) of the way from 1.0 to 9.0
Out[1]: 3.0
```

```
In [2]: interp(1.0, 3.0, 0.25)
A quarter of the way from 1.0 to 3.0
Out[2]: 1.5
```

```
In [3]: interp(2, 12, 0.22)
```

```
22% of the way from 2 to 12
Out[3]: 4.2
```

Hint

If you're unsure of where to begin on this problem, look at the first example above. In it `low` is 1.0 `hi` is 9.0 `fraction` is 0.25

See if you can determine how to combine those three values to yield the correct return value of 3.0. (Consider starting with `(hi - low)`.)

Here are two more examples to try:

```
In [1]: interp(24, 42, 0)
0% of the way from 24 to 42
Out[1]: 24.0
```

```
In [2]: interp(102, 117, -4.0)
-400% of the way from 102 to 117 (whoa!)
Out[2]: 42.0
```

3. Write a function `checkends(s)`, which takes in a string `s` and returns `True` if the first character in `s` is the same as the last character in `s`. It returns `False` otherwise. The `checkends` function does not have to work on the empty string (the string `"`).

There is a hint below, but read through the examples first.

These examples will help explain `checkends`—read them over now, and be sure to try them once you have a first draft of your function. Notice that the final, fourth example below is the *string of one space character*, which is different from the empty string, which contains no characters:

```
In [1]: checkends('no match')
Out[1]: False
```

```
In [2]: checkends('hah! a match')
Out[2]: True
```

```
In [3]: checkends('q')
Out[3]: True
```

```
In [4]: checkends(' ')
Out[4]: True
```

Make sure to check that this last example (the string of a single space) works for your `checkends` function. The empty string does not need to work.

4. Write a function `flipside(s)`, which accepts a string `s` and returns a string whose first half is `s`'s second half and whose second half is `s`'s first half. If `len(s)` (the length of `s`) is odd, the "first half" of `s` is considered to have one fewer character than the second half. (Accordingly, the second half of the returned string will be one shorter than the first half in these cases.) There's also a hint after the examples below.

Here you may want to use the built-in function `len(s)`, which returns the length of its argument string, `s`.

Examples:

```
In [1]: flipside('homework')
Out[1]: workhome
```

```
In [2]: flipside('carpets')
Out[2]: petscar
```

5. Write `convertFromSeconds(s)`, which takes in a nonnegative `integer` number of seconds `s` and returns a list (we'll call it `L`) of four nonnegative integers that represents that number of seconds in more conventional

units of time, such that:

- The initial element represents a number of days
- The next element represents a number of hours
- The next element represents a number of minutes
- The final one represents a number of seconds

You should be sure that

- $0 \leq \text{seconds} < 60$
- $0 \leq \text{minutes} < 60$
- $0 \leq \text{hours} < 24$

There are no limits on the number of days.

For instance,

```
In [1]: convertFromSeconds(610)
```

```
Out[1]: [0, 0, 10, 10]
```

```
In [2]: convertFromSeconds(100000)
```

```
Out[2]: [1, 3, 46, 40]
```

# CSforAll - InteractiveFiction

## CS for All

CSforAll Web > Chapter2 > InteractiveFiction

This website accompanies the textbook *CS For All* by Alvarado, Dodds, Kuenning, and Libeskind-Hadas

### Interactive Fiction

#### ***Choose Your Own Adventure***

For this problem, you-the-author should create a Python program (built on the starting template below) that is a work of interactive fiction, a.k.a., a *choose-your-own-adventure* program. It needs to include at least 5 if-based control structures, as noted below:

#### **An example from which to build**

Start by creating a file and pasting this skeleton of a story into it:

```
"""
Title for your adventure: The Quest.

Notes on how to "win" or "lose" this adventure:
 To win, choose the table.
 To lose, choose the door.

"""

import time

def adventure():
 """ this function runs one session of interactive fiction
 Well, it's "fiction," depending on the pill color chosen...
```

```

 arguments: no arguments (prompted text doesn't count as an argument)
 results: no results (printing doesn't count as a result)
"""

delay = 0.0 # change to 0.0 for testing or speed runs,
..larger for dramatic effect!

username = input("What do they call you, worthy adventurer? ")

print()
print("Welcome, " + username + " to the Libra Complex, a labyrinth")
print("of weighty wonders and unreal quantities...of poptarts!")
print()

print("Your quest: To find--and partake of--a poptart!")
print()
flavor = input("What flavor do you seek? ")
if flavor == "strawberry":
 print("Wise! You show deep poptart experience.")
elif flavor == "s'mores":
 print("The taste of the campfire: well chosen, adventurer!")
else:
 print("Each to their own, then.")
print()

print("On to the quest!\n\n")
print("A corridor stretches before you; its dim lighting betrays, to one side,")
print("a table supporting nameless forms of inorganic bulk and, to the other,")
print("a door ajar, leaking laughter--is that laughter?--of lab-goers.")
time.sleep(delay)
print()

choice1 = input("Do you choose the table or the door? [table/door] ")
print()

if choice1 == "table":
 print("As you approach the table, its hazy burdens loom ever larger, until...")
 time.sleep(delay)
 print("...they resolve into unending stacks of poptarts, foil shimmering.")
 print("You succeed, sumptuously, in sating the challenge--and your hunger.")
 print("Go well, " + username + " !")

else:
 print("You push the door into a gathering of sagefowl, athenas, and stags alike,")
 print("all relishing their tasks. Teamwork and merriment abound here, except...")
 time.sleep(delay)
 print("...they have consumed ALL of the poptarts! Drifts of wrappers coat the floor")

```

```
print("Dizzy, you grasp for a pastry. None is at hand. You exhale and slip")
print("under the teeming tide of foil as it finishes winding around you.")
print("Farewell, " + username + ".")
```

### Try It Out!

Try out the above story by

- Opening a terminal, probably in VSCode
- cd to the directory where YOUR\_FILE\_NAME.py is located
- Starting ipython (just typing ipython should do it)
- Typing run YOUR\_FILE\_NAME.py
- If it doesn't find that file, you can cd to the correct directory *from within ipython*
- If it loads successfully, paste or type adventure() and hit enter.

### Run At Least Twice!

Once you've loaded it, try it at least twice to show the different paths through the story.

- To run it again, you only need to type adventure() and hit enter
- adventure() is the function defined in the file
- Only if you make a *change* to the file would you have to retype run YOUR\_FILE\_NAME.py

## Modify or Rewrite Your Own Interactive Fiction

You're ready to go! Partner up with someone, if you'd like (entirely optional!) and create your own adventure.

Certainly feel free to modify or continue the poptart-related story—or, if you wish, do something completely different to compose your own work of interactive fiction. The story above includes two if-based control structures: (a) an if-elif-else and (b) an if-else.

The requirement of this problem is that you use *each* of five conditional control structures in a way that affects the story or dialog.

- They don't have to appear in this order
- You're welcome to have many more than this, if you wish!

Here are the control structures to be sure to include:

1. An if, elif, and else control structure (with exactly one elif)
2. An if, elif, elif, ... and else control structure (with at least two elifs)
3. An if, else control structure (with zero elifs)
4. An if, elif, ... control structure (*with one or more elifs but no trailing else at all*)

## 5. An `if` control structure (*with no trailing `elif` nor trailing `else` at all*)

Please keep things light! There are forums for all forms of expression—this one's for contextualizing conditionals in CS.

So, keep in mind that, potentially, lots of people (our grutors!) will read—and run—your story (we look forward to it!) Also, ***please be mindful of the time you spend!*** The goal here is ***not*** to take a lot of time. (It's easy to let time slip away: the poptart adventure above took much longer than I originally thought it would—and it has only one control structure!)

Happy adventuring!

Website design by Madeleine Masser-Frye and Allen Wu

# CSforAll - RockPaperScissors

## CS for All

CSforAll Web > Chapter2 > RockPaperScissors

This website accompanies the textbook *CS For All* by Alvarado, Dodds, Kuenning, and Libeskind-Hadas

### Rochambeau

#### A Rock, Paper, Scissors Program

This second problem asks you to practice:

- Creating new Python files (in this case, by copying old ones or starting with the code below)
- Writing a bit of your own code (by altering our starter program, if you wish)
- Text-based input and output in Python

#### Start With a New File

Start by pasting in the following code, which gets you on the way:

```
import random # imports the library named random

def rps():
 """ this plays a game of rock-paper-scissors
 (or a variant of that game)
 arguments: no arguments (prompted text doesn't count as an argument)
 results: no results (printing doesn't count as a result)
 """
 user = input("Choose your weapon: ")
 comp = random.choice(['rock','paper','scissors'])
```

```

print()

print('The user (you) chose', user)
print('The computer (I) chose', comp)
print()

if user == 'rock':
 print('Ha! I really chose paper--I WIN!')

print("Better luck next time...")

```

### What's Next? What's Required?

- From within ipython—and from within the correct directory!—run the file with `run FILE_NAME`
  - Doing so loads the file. You should up-arrow to do this each time you change the file!
  - When it successfully runs, then type `rps()` to run the `rps` function.
- The basic requirements are these: use the starter code (below) to create a program that:
  - Invites the user to play a game of "rock-paper-scissors."
  - Lets the user choose from at least three options
    - \* They don't have to be rock, paper, and scissors—feel free to vary the actors!
    - \* *But your program does have to work differently for each of three distinct inputs* (well, at least three inputs)
  - Your program is welcome to play an honest game of RPS
    - \* But you're also welcome to create a player that always wins (or, if you prefer, that always loses)
  - Your program should echo the choice the user made
    - \* You may assume the user will type her/his choice correctly, for your game's choices...
  - Your program should reveal the choice that it makes (whether fair or not)
  - Your program should print out who won that round (or whether it was a tie, or some other outcome)
  - Adding side comments to the graders who will be running your program is optional, but strongly encouraged!

## More Details

- In brief, the program should ask the user to choose rock, paper, or scissors (or your own variants!). Then it should repeat that choice back to the user, reveal its own choice, and finally report the results. The program can play fairly, can always win, or can always lose—it's up to you. If RPS is unfamiliar, in the game of rock-paper-scissors, rock defeats scissors, scissors defeat paper, and paper defeats rock.
- You may assume that the user will input one of `rock`, `paper`, or `scissors`. Case matters! We'll stick with lower case...
- You may write the dialog however you like—below is an example dialog if you'd like one to follow. We are *positive* that you can improve on this interaction, however! Here are two distinct runs of the program:

```
In [1]: run FILE_NAME.py
```

```
In [2]: rps()
Choose your weapon: rock
The user (you) chose rock
The computer (I) chose scissors
```

```
Ha! I really chose paper--I WIN!
Better luck next time...
```

```
In [3]: rps()
Choose your weapon: paper
the user (you) chose paper
the computer (I) chose dynamite
```

```
Dynamite!? I REALLY WIN!
You can't play again.
```

## Other Possibilities

- Too much time on your hands? Add "lizard" and "spock" as noted at this RPSSL link.  
Even more time? Consider RPS-25, a strict superset of RPS.  
But if you have enough time for RPS-101, there's a problem!
- Want your program to continue playing many times? Use a `while True:` loop.

We'll provide two examples instead of detailed explanations:

```
while True:
 print("Still running...")
 response = input("Play again? ")
 if response == 'n':
 break
```

Here is another possibility:

```
running = True
while running:
 response = input("Play again? ")
 if response == 'n':
 running = False
```

Website design by Madeleine Masser-Frye and Allen Wu

# CSforAll - SequencesData

## CS for All

CSforAll Web > Chapter2 > SequencesData

This website accompanies the textbook *CS For All* by Alvarado, Dodds, Kuenning, and Libeskind-Hadas

### Sequences and Data

There are two parts to this:

- In the first you'll gain experience splicing and interacting with Python data
- In the second you'll write a number of *functions*, the fundamental building blocks of software

### Trying Out the Python Interpreter (or *shell*)—IPython!

There's no file nor anything to hand in for this part—just open ipython's shell by typing `ipython` at the command prompt and try out the Python commands below...

You can tell if you're in the ipython shell by the prompt: `In [n]` (where n is some number) or, on its own line,

`In [1]:`

Note that you don't have to type that prompt—it's already there!

### Using IPython's Numbered Prompts

The numbered prompts are great, because you can access all inputs (as strings) and outputs (as whatever type they produce) via their numbers. To see this, try

these examples:

```
In [1]: 6*7
Out[1]: 42

In [2]: Out[1]*2
Out[2]: 84

In [3]: In[1]*2
Out[3]: '6*76*7'
```

Cool!

Python's `eval` will evaluate strings. Try `eval(Out[3])` in the example above!

Our example won't necessarily match which number you're currently on, so **don't worry if your prompt number doesn't match ours—experiment!**

### Arithmetic with numbers, lists, strings, booleans, ...

To get started, try a few arithmetic, string, and list expressions in the Python interpreter, e.g.,

```
In [1]: 40 + 2
Out[1]: 42

In [2]: 40**2
Out[2]: 1600

In [3]: 40 % 7 # 40 "mod" 7: the remainder of 40 divided by 7
Out[3]: 5

In [3]: 40 // 11 # integer division: throwing the fraction away
Out[3]: 3

In [4]: 'hi there!'
Out[4]: 'hi there!' # (notice Python's politeness!)

In [5]: 'who are you?'
Out[5]: 'who are you?' # (though sometimes it's a bit touchy.)

In [6]: L = [0, 1, 2, 3] # You can label data (here, a list) with a name (here, the name L)
 # (no response from Python)

In [7]: L
Out[7]: [0, 1, 2, 3] # You can see the data (here, a list) referred to by a name (here,
```

```

In [8]: L[1:]
Out[8]: [1, 2, 3] # You can slice lists (here, using the name L)

In [9]: L[::-1]
Out[9]: [3, 2, 1, 0] # You can reverse lists (<i>or strings!</i>) using "skip" slicing with

In [10]: [6, 7, 8, 9][1:]
Out[10]: [7, 8, 9] # You can slice lists using the raw list instead of the name (Not that th

In [11]: 100*L + [42]*100
Out[11]: (a list with 500 elements)

In [12]: L = 42 # You can reassign another value to the name L, even one of a different type
 (no response from Python)

In [13]: L == 42 # Two equals are different than !=! This <i>tests for equality</i>.
Out[13]: True

In [14]: L != 42 # This tests for "not equal."
Out[14]: False

```

### Errors and *Exceptions*

Mistakes are unavoidable! So, you'll encounter Python errors. They're sometimes called *exceptions*, as well.

One of the most important habits we hope you'll practice in CS 5 is this:  
*If an error happens, consider it an opportunity, not a problem!*

It's true, in that an error is a chance to:

1. Improve your intuition about how computation works, i.e., the "machine's mindset,"
2. Improve the software you're developing (or your understanding of it), and
3. Build your debugging skills.

So, let's create some errors, which Python calls *exceptions*:

***Give yourself two minutes.*** In that time, see how many of these Python exceptions you can cause!

If you create others, all the better—let an instructor or tutor know, and we'll add them to this list:

- `NameError` (an unrecognized variable!)
- `TypeError` (try slicing an integer for example!)
- `ZeroDivisionError` (perhaps clear from its name)
- `SyntaxError` (the error that kittens most often produce when walking over the keyboard)
- `IndexError` (try an out-of-bounds index into a sequence)
- `OverflowError`  
Remember that integers won't overflow—if they get too big to fit in memory, they'll simply crash Python. To obtain this error, therefore, you'll need to use a floating-point value, such as `42.0`, in some mathematical expression that produces very large values. For example, use the power `**` operator!

## Lists! Challenges with Slicing and Indexing

This problem will exercise your slicing-and-indexing skills.

**To do:** Then, copy the following starting lines into your new plain-text file:

```
pi = [3, 1, 4, 1, 5, 9]
e = [2, 7, 1]

Example problem (problem 0): [2, 7, 5, 9]
answer0 = e[0:2] + pi[-2:]
print(answer0)
```

**A couple of notes on this code:**

- Be sure to save this as a plain-text file with the `.py` extension.
- After the initial comment, this code defines the list named `pi` and the list named `e`.
- When you run the file, the line `answer0 = e[0:2] + pi[-2:]` will define the value held by the variable `answer0`.
- Then, the code will print the value of the variable `answer0`.

### To run the code:

- First, at your command line, make sure you're in the folder (location) where `YOUR_FILE_NAME.py` is located. Perhaps it's the desktop, or maybe you put it in another folder.
- Run `ipython`.
- Then, within Python, enter `YOUR_FILE_NAME.py`
- (Tab-completion and up-arrow can make things easier.)

### Composing New Lists From `pi`, `e`, and List Operations

The problems below ask you to create several lists using **only** the list named `pi`, the list named `e`, and these list operations:

- List indexing, such as `pi[0]`
- List slicing, such as `e[1:]`
- Skip slicing, such as `pi[0:6:2]`
- List concatenation with `+`, such as `e[0:2] + pi[-2:]`  
(for this problem, we ask you *not* to use `+` to add values numerically)
- Please **leave a blank line** or two between your answers (to keep things readable—this makes the graders happy)!
- Once you've run the file once, you can experiment at Python's command-line—try it by typing `e[0:1] + pi[0:1]`
- **For fun only**, you might try using as few operations as possible, to keep your answers elegant and efficient.
- Here are the problems:
- **0.** Use `pi` and `e` (or just one!) to create the list `[2, 7, 5, 9]`. This is the example above, stored in the variable `answer0`.
- **1.** Use `pi` and `e` (or just one—for all the problems here you can always use just one) to create the list `[7, 1]`.
  - As above, store this list in the variable `answer1`. Here is a start, to copy and paste:

```
Problem 1: creating [7, 1]
answer1 = e[1:2] # not the right answer, but a start...
print(answer1)
```

- **2.** Use `pi` and `e` to create the list `[9, 1, 1]`. Store this list in the variable `answer2`.
- **3.** Use `pi` and `e` to create the list `[1, 4, 1, 5, 9]`. Store this list in the variable `answer3`.

- 4. Use `pi` and `e` to create the list [1, 2, 3, 4, 5]. Store this list in the variable `answer4`.

## Strings! Slicing and Indexing

This problem continues in the style of the last one, but uses strings rather than lists.

First, copy these lines into your file underneath the previous problems (with some blank lines to keep things apart!):

```
string practice

h = 'harvey'
m = 'mudd'
c = 'college'
```

You may use any combination of these four string operations:

- String indexing, e.g., `h[0]`
- String slicing, e.g., `m[1:]`
- String concatenation, `+`, e.g., `h + m`
- Repetition, `*`, e.g., `42*c`

The number of operations in the shortest answers that we know about are in parentheses. If you'd like, you might see if your answers are equally or more concise.

However, **any correct answer is OK**—there's no requirement to use a small number of operations.

**Example problem (#5):** Use `h`, `m`, and `c` to create 'hey'. Store this string in the variable `answer5`. We used 3 operations.

**Answer to example 5—please copy and paste this into your file:**

```
Problem 5: 'hey'
answer5 = h[0] + h[4:6]
print(answer5)
```

The 3 operations are 1 use of list indexing, 1 slice, and 1 concatenation with `+`.

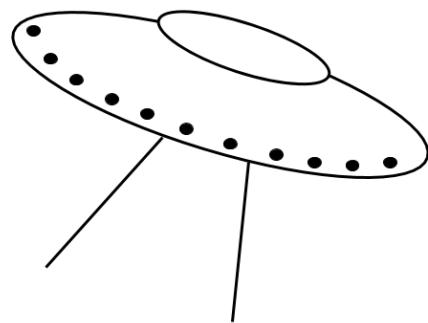
Here are the string-creation challenges (and, in parentheses, our most efficient answers, at least so far):

- Remember that the "most efficient" answers are not at all needed (they may be fun, but **any** working answer is 100% OK!)
- 5. (The example from above) Create `hey` and store this string in the variable `answer5`. (3 ops.)
- 6. Create `collude` and store this string in the variable `answer6`. (Our best: 5 ops.)
- 7. Create `arveyudd` and store this string in the variable `answer7`. (Our best: 3 ops.)
- 8. Create `hardeharharhar` and store this string in the variable `answer8`. (Our best: 7 ops.) with `(h+m)[-2:3:-4]`
- 9. Create `legomyego` and store this string in the variable `answer9`. (Our best: 8 ops.)
- 10. Create `clearcall` and store this string in the variable `answer10`. (Our best: 8 ops.)

Website design by Madeleine Masser-Frye and Allen Wu

## CSforAll - TurtleGraphics

### CS for All



CSforAll Web > Chapter2 > TurtleGraphics

This website accompanies the textbook *CS For All* by Alvarado, Dodds, Kuennen, and Libeskind-Hadas

**Python Turtles!**

## Trying Out the `tri` Example...

First, you'll see how your version of Python interacts with turtle graphics.

Start by creating a new file in your editor . Paste into your new file this top-of-file comment and example function, `tri(n)` :

```

Name:

Turtle graphics and recursion

import turtle
import time
from turtle import *
from random import *
t = Turtle()
t.color("green")
t.shape("turtle")
t.speed(2)

def resetTurtle():
 userInput = input("Press enter to reset the turtle.")
 if userInput == "":
 t.reset()
 t.color("green")
 return

def tri(n):
 """Draws n 100-pixel sides of an equilateral triangle.
 Note that n doesn't have to be 3 (!)
 """
 if n == 0:
 resetTurtle()
 return # No sides to draw, so stop drawing
 else:
 t.forward(100)
 t.left(120)
 tri(n-1)
```

Then, to try this out:

- From your ipython prompt, type `run YOURFILENAME`.
- Then, paste or type `tri(3)`.

- You should see the turtle appear and draw a triangle...
  - *Sometimes the window opens **behind** other applications...* You may need to bring it to the front.
- When it's finished, you'll see a prompt that says "Press enter to reset the turtle." When you do that, the shape should disappear and the turtle should go home (to the center of the screen, if it isn't already).
- If everything works, great!
- If something doesn't go as planned (it is likely this will happen at some point int the assignment as the program is inherently finicky), there are a couple things you can try...
  - `run YOURFILENAME` (yes, again, and if you get a blank turtle window, do it once more)
  - close out of the turtle window with the x in the corner
  - `t.reset()()`
  - some combination of these things

Now, the trick -- *try it again!*

- Type or up-arrow to again execute the function `tri(3)`
- **You may get a "Terminator" error** -- this is ok!
- If you do, simply type (or up-arrow) and try running `tri(3)` again.
- Almost always on this second attempt, it works -
- This is because closing the window does not reset the system fully (but the error does)

If running twice works for your system, it's ***totally ok*** to run turtle functions or commands twice!

If your system or the turtle window does become non-responsive...

- Ask for help if you're running into trouble or other oddities.
  - Python turtling is fun, but it's idiosyncratic -- it's a *very* old library.
- If things go super wrong, you can always kill the ipython terminal...
  - we hope that that's not needed! (Ask for help if it seems to be.)
- If we **really** can't figure it out, you can do the turtle part of the homework with a partner, or try using this web interface. If you use the web interface, you should *edit* your code in VSCode, then copy and paste it into the web editor to run it. That way you'll be able to keep a copy of your work saved on your machine, and it will be easier to submit.

### What's happening in this code???

- Turtle/Turtleness/Turtlality/Turtlation/Turtlocity is a "class" in python and you'll learn what that means later this semester
- Right now this matters to you because `t` is an "object" of class `Turtle`
- We care about that because to control your turtle you will use `t.some_command`

- Look for examples of this in the starter code!

## Trying Other Options and Colors!

Now, to turtling!

First, change your `tri` function and re-run it in order to try some of the many `turtle` options:

- (*Line thickness*) Put the line `t.width(10)` as the first overall line of the function.
  - **Or** Put the line `t.width(2*n+1)` as the first line of the `else` block.
- The two functions `t.clear()` and `t.reset()` are useful!
  - Try them out: `t.reset()` is the more "comprehensive"!
- (*Colorfulness*) Put the line `t.color('darkgreen')` as the first line of the function.
  - **Or** put the two lines:
 

```
clr = choice(['darkgreen', 'red', 'blue'])
t.color(clr)
```

 as the first lines of the function.
  - This colors page<sup>7</sup> lists the various colors available.
  - In addition, you can create your own colors, e.g., `clr = (0.8, 0.6, 0.0)`, followed by `t.color(clr)`, creates an HMC-gold-ish color, with red at 0.8 strength (out of 1), green at 0.6 out of 1, and blue at 0.0 out of 1.
- (*Turtle shape*) Put the line `t.shape('turtle')` as the first line of the function.
  - This library reference lists the default shapes.
- (*Other Python turtle-drawing commands*) Put the line `t.dot(10, 'red')` as the first line of the `else` block.
  - This should draw a filled circle (a dot) just before it runs the other lines of the `else` block.
  - Note that you can make the dot a different color than the line color by including the color as the second argument to `dot`.
  - Here is the overall turtle reference page

**Big picture:** The `tri` example demonstrates **single-path** recursion: the recursive calls are made a single time, so that there is a single, step-by-step path taken.

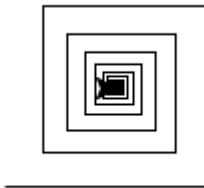
## The `spiral` Function

Next, you'll build another *single-path* recursive function, `spiral`. It will be very similar to `tri`.

To begin, underneath the `tri` function, write another named `spiral`. Here is its signature, and outline, with docstring, for easy copy-and-paste:

```
def spiral(initialLength, angle, multiplier):
 """Spiral-drawing function. Arguments:
 initialLength = the length of the first leg of the spiral
 angle = the angle, in degrees, turned after each spiral's leg
 multiplier = the fraction by which each leg of the spiral changes
 """
 if initialLength <= 1 or initialLength >= 1000:
 resetTurtle()
 return # No more to draw, so stop this call to spiral
 else:
 # You will want a call to forward here...
 # You will want a turn here...
 # You will want to recur here! That is, make a new call to spiral...
```

Here's a picture from the call `spiral(100, 90, 0.9)`



The `spiral` function should use the `turtle` drawing functions to create a spiral that

- Draws its first segment of length `initialLength`, and then
- Turns `angle` degrees to the left after that segment, and then
- Calls `spiral` recursively, using `multiplier` to change its first argument (the second and third arguments won't change!)

For example,

- With a `multiplier` of 0.5 each side in the spiral should be *half* the length of the previous side (getting smaller)
- with a `multiplier` of 1.5 each side in the spiral should be *one and a half times* the length of the previous side (getting larger)

### ***Base cases!***

For your base cases, have `spiral` stop drawing—just with `return`—when it has reached a side length of

- ***Less than 1 pixel*** (this one is already implemented, above!) or
- ***Greater than 1000 pixels***
  - You can use `or initialLength ...` in the same base-case test, above, to implement this.

For example, try some of these calls:

- `spiral(100, 90, 0.9)`
- `spiral(100, 170, 0.95) # More of a "star" (hit control-c to stop...)`
- `spiral(400, 120, 0.8) # A triangular spiral...`
- `spiral(2, 1, 0.999) # An ACTUAL spiral (again, control-c to stop...)`

### **The `chai` Function: *Branching Recursion***

Next, you'll build a *branching-recursion* example. When branching, recursion is at its most "magical." However, it's also true that in *writing*, branching recursion can be the most mind-bending! What's remarkable is that the "magic," in the end, reveals all of its tricks: it's fully understandable.

Start by pasting the `chai` function:

```
def chai(size):
 """Our chai function!
 if size < 5:
 resetTurtle()
 return
```

```

else:
 t.forward(size)
 t.left(90)
 t.forward(size/2)
 t.right(90)

 t.right(90)
 t.forward(size)
 t.left(90)

 t.left(90)
 t.forward(size/2.0)
 t.right(90)
 t.backward(size)
return

```

Then, try running it with `chai(100)`

Next, add one branch of recursion to `chai`:

- Paste this recursive call: `chai(size/2)` between the two calls to `t.right(90)`

Try it out!

Next, add a second branch.

It's nothing more than a second branch, but because it's called recursively on *all* of the subbranches, the resulting work (and visual intricacy) is more than just doubled!

To add this second branch, you should

- Paste a second recursive call to `chai(size/2)` between the two calls to `t.left(90)`

Again, try it out!

In the end, branching recursion works by creating a smaller version of the overall structure at ***more than one location*** within that structure.

### Key Idea: *Ending Where You Began*

The **key** to making "branching recursion" work is *making sure that your turtle ends at the same location that it begins, and ends up facing the same direction*.

That is how you ensure that the statements *after* the recursive calls are moving the turtle as expected.

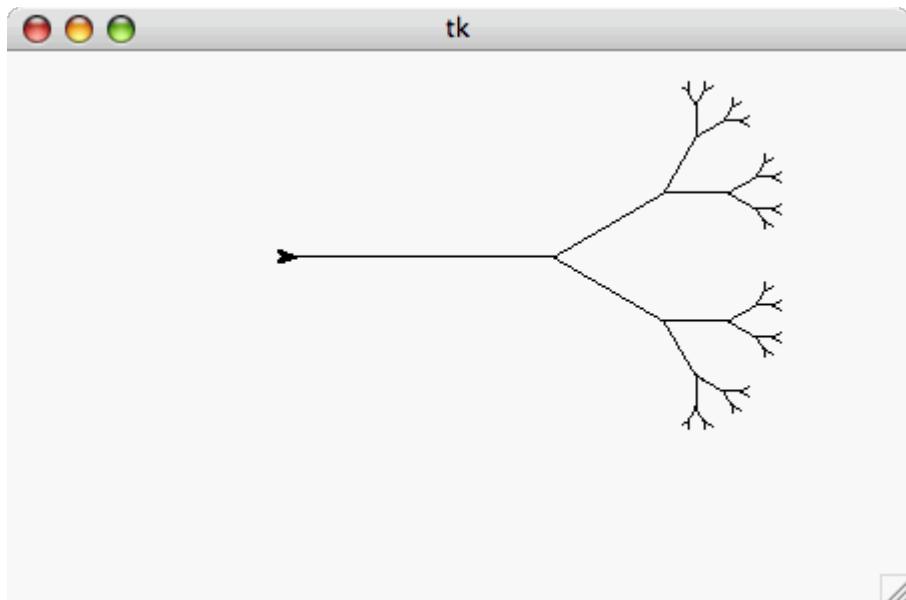
### The `svtree` Function

Next, you'll write another branching example: the *side-view* tree. Here, "branching" seems like a particularly appropriate descriptor!

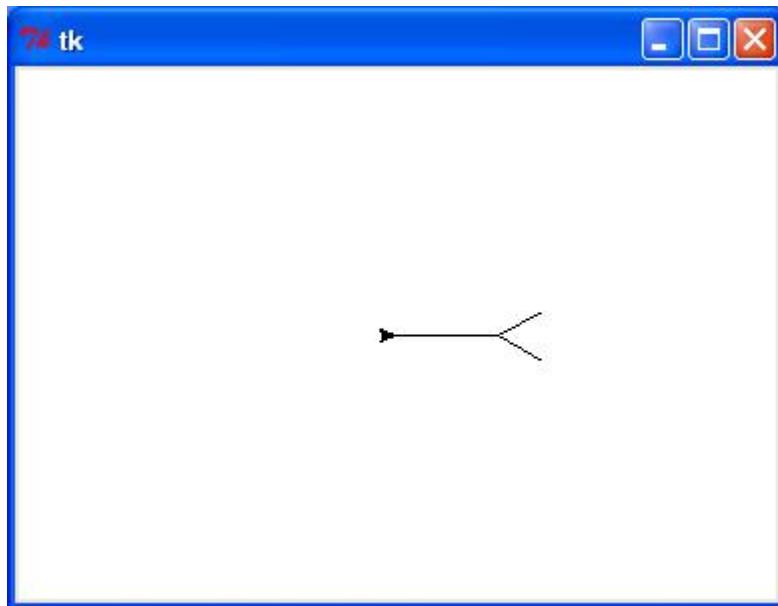
The idea here is to create a function that draws the *side-view* of a tree, hence `svtree`. Read over this docstring in order to get a sense of the two arguments. Then, look over the outline and images below:

```
def svtree(trunklength, levels):
 """svtree: draws a side-view tree
 trunklength = the length of the first line drawn ("the trunk")
 levels = the depth of recursion to which it continues branching
 """
 if levels == 0:
 resetTurtle()
 return
 else:
 # Draw the original trunk (1 line)
 # Turn a little bit to position the first subtree (1 line)
 # Recur! with both a smaller trunk and fewer levels (1 line)
 # Turn the other way to position the second subtree (1 line)
 # Recur again! (1 line)
 # Turn and go BACKWARDS (2 steps: 2 lines)
```

Before diving in, take a look at two examples and some analysis. First, here is an example of the output from my function when `svtree(128, 6)` is run:



Note that this is **really** side view! Here is an example of the possible output when `svtree(50, 2)` is run:



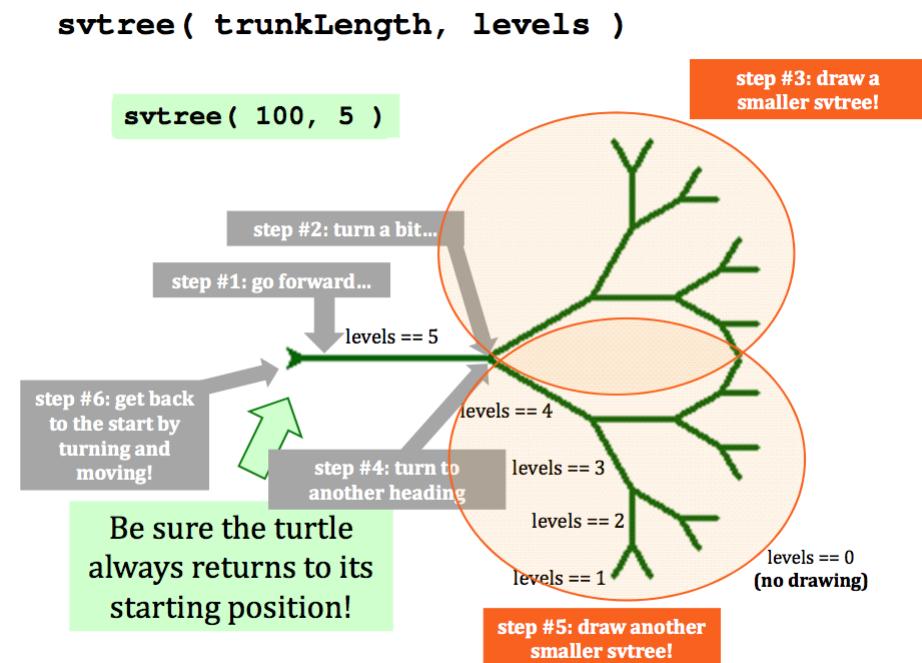
Also, if `svtree(50, 0)` is run, the result should be no drawing at all. *The base case occurs when levels equals 0.*

## Analysis

**Base case:** for `svtree`, you want to draw nothing (and return from `svtree`) when `levels == 0`.

**Recursive case:** We tackle it conceptually.

Here is a picture showing the self-similar breakdown of `svtree` (from the Gold slides). *This is, in fact, an almost complete map of the `svtree` code!*



The key to happiness with recursive drawing is this: *the pen must be back at the start (root) of the tree at the end of the function call*, and the turtle must be facing in the original direction! That way, each portion of the recursion "takes care of itself" relative to the other parts of the image. Here are the steps:

- Go forward the `trunklength`.
- Turn left some amount.
- Recur! (call `svtree` with a fraction of the `trunklength` and 1 fewer levels).
- Turn right some amount (if you want to be symmetric, turn right double the amount you turned left).

- Recur again! (make the same call to `svtree` with a fraction of the `trunklength` and 1 fewer levels).
- *Finish up, part 1:* Turn left so that the turtle is facing its original direction (usually the same as the original left turn).
- *finish up, part 2:* Go **backward** the original `trunklength`.

Again, here is a code-based sketch of how `svtree` will work:

```
def svtree(trunklength, levels):
 """svtree: draws a side-view tree
 trunklength = the length of the first line drawn ("the trunk")
 levels = the depth of recursion to which it continues branching
 """
 if levels == 0:
 return
 else:
 # Draw the original trunk (1 line)
 # Turn a little bit to position the first subtree (1 line)
 # Recur! with both a smaller trunk and levels (1 line)
 # Turn the other way to position the second subtree (1 line)
 # Recur again! (1 line)
 # Turn and go backwards (2 steps: 2 lines)
```

**Notes:** Don't worry about the exact angle of branching or the amount of reduction of the `trunklength` in sub-branches, etc.

- Design your own tree by making aesthetic choices for each of these.

Calling `t.left(90); svtree(100, 5)` will yield a more traditional, skyward, tree pose!

### Try More Branches!

Once you have the `svtree` function working, alter it so that it has *three or more branches*, instead of only two...

- You can get some very dense "foliage" very quickly.
- Even more "life-like" results are possible if you use non-identical branching angles and size multipliers.
- Also, you could have the `width` or `color` depend on the value of `levels`.
- If you make the final "level" red, you can create an apple tree.

- See the `dot` example from `tri` for other ways to add fruit.
- Or, if you make that final "level" a random color, you can produce fall-foliage-type effects...

### The `snowflake` and `flakeside` Functions

A challenge! The Koch Snowflake is an example of very deeply branching recursion.

The Koch snowflake is a fractal with three identical sides—it's the sides themselves that are defined recursively, not the triangle.

Because of this, we provide the overall `snowflake` function for you to use—it's here. Feel free to copy-and-paste:

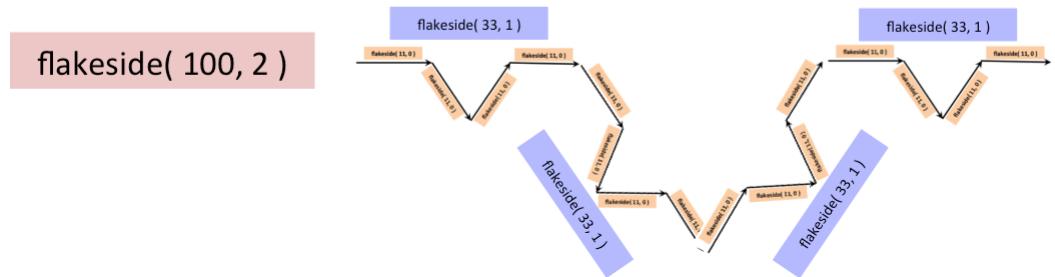
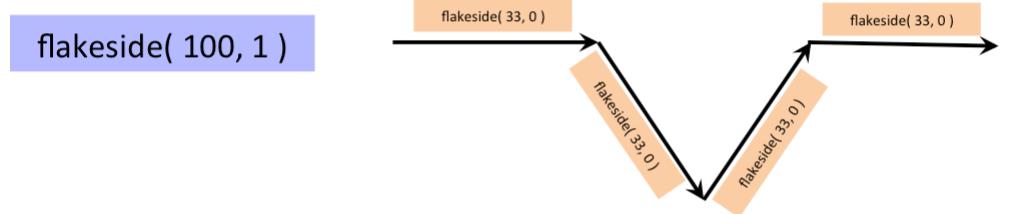
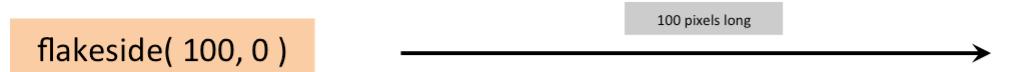
```
def snowflake(sidelen, levels):
 """Fractal snowflake function, complete.
 sidelen: pixels in the largest-scale triangle side
 levels: the number of recursive levels in each side
 """
 flakeside(sidelen, levels)
 t.left(120)
 flakeside(sidelen, levels)
 t.left(120)
 flakeside(sidelen, levels)
 t.left(120)
 resetTurtle()
```

Note that you can't use this until you define `flakeside`—that's next.

### *Your Task*

Your task is to implement `flakeside(sidelen, levels)`, which will draw one snowflake side.

First, here is a graphical summary of a snowflake side's structure:



### Hint

- A base-case Koch snowflake side is simply a straight line of length `sidelength`!
- Each recursive level replaces the *middle third* of the snowflake's side with a "bump," i.e., two sides that would be part of a one-third-scale equilateral triangle.
- Notice there are four *subsides* to each flakeside. This means that `flakeside` will call itself recursively ***four times!***
- At the three spots *between* those four calls, there will be an appropriate turn...
- Thus, the recursive case will include seven total lines (4 recursions and

three turns).

### Calls to try...

- Try `flakeside(300, 0)`—make sure you get a straight line
- Try `flakeside(300, 1)`—make sure you get a four-segment contour
- Try `flakeside(300, 2)`—make sure you get a four "level-1" flakesides
- Try `flakeside(300, 3)`—pretty! and pretty cool! Olaf approves.

Remember that `flakeside` is creating ***only one*** of the three sides of the snowflake!

- Because of this, it does ***not*** have to end in precisely the same location as it begins.
- After all, if it did, all three sides of the overall snowflake would be on top of one another.
- Don't include the `resetTurtle()` function within this one because you don't want to reset before the whole snowflake is done

### From Flakeside to Snowflake

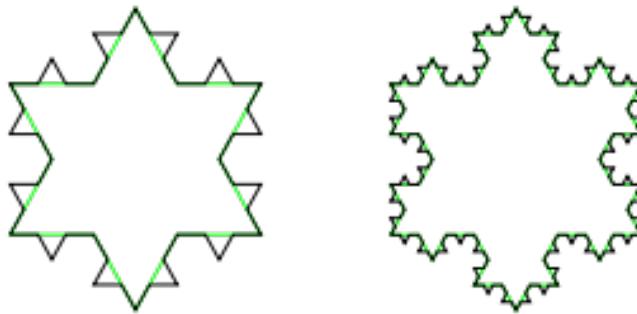
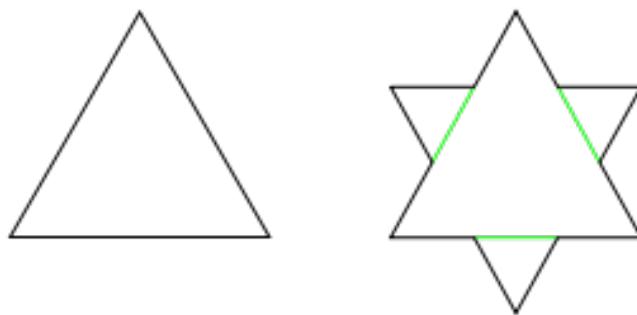
Once your `flakeside` function works, try out `snowflake`!

The `snowflake` function simply calls `flakeside` three times.

Depending on the directions that `flakeside` uses, you may need to change the `lefts` in `snowflake` to `rights`.

Examples to try might include `snowflake(300, 2)` and `snowflake(300, 3)`

Here are images of four different values of `levels` for a snowflake, 0, 1, 2, and 3:



You can learn much more about the Koch snowflake fractal curve by Googling!

### Extra Turtle Art!

If you enjoy creating turtle graphics, whether recursive or not, you're invited to try more!

Happy turtling!

Website design by Madeleine Masser-Frye and Allen Wu

# CSforAll - FunctionFrenzy

## CS for All

CSforAll Web > Chapter2 > FunctionFrenzy

This website accompanies the textbook *CS For All* by Alvarado, Dodds, Kuenning, and Libeskind-Hadas

### Function Frenzy!

This problem asks you to use recursion to write several Python functions. (There are also extra-credit opportunities...)

#### Starting Your File

Create a new plain-text file using VSCode or another plain-text editor.

Here is a little bit of code to paste, if you'd like to use it to get started:

```
def leng(s):
 """leng returns the length of s
 Argument: s, which can be a string or list
 """
 if s == '' or s == []:
 return 0
 else:
 return 1 + leng(s[1:])
```

In addition, the above code includes the `leng` example function we wrote in class. Here are all of the RecursionExamples.

*Be sure to name your functions exactly as specified—including capitalization!*

## Use Recursion!

For this homework, the `mult`, `dot`, `ind`, `scrabbleScore`, and `transcribe` functions should all be done using *recursion*. Compare them to the `power`, `max`, `leng`, and `vwl` functions we saw in class this week. Those examples are linked at this page and beside each problem for you to test and use as the basis for your design.

## Visualize Recursion!

Some people have used this [online Python visualizer](#) to build intuition about how recursion works. A couple of details: to visualize a recursive call, you'll need to (1) define your recursive function and then (2) make a test call to it, perhaps immediately underneath it.

Here is an example that shows how to use the online Python visualizer to test `mylen('cs5')`, one of the examples from class. Paste this code into the visualizer linked above:

```
def mylen(s):
 if s == '':
 return 0
 else:
 return 1 + mylen(s[1:])

test = mylen('cs5')
print('test is', test)
```

You can adapt this for other examples from class or from your own code, as well. Try it!

## Use docstrings!

Also, for each function be sure to include a docstring that indicates what the function's arguments mean and what its return value means, i.e., what the function "does." Here's an example of a docstring, that you are welcome to use for `mult` and as a template for the others:

```
def mult(n, m):
 """mult returns the product of its two arguments
 Arguments: n and m are both integers
 Return value: the result of multiplying n and m
 """
 code here ...
```

**Warning!** Notice that the docstring needs to be indented to the same level as body of the function it's in. (Python is picky about this...)

## Test!

Be sure to test your functions! It's tempting to write a function and feel like it works, but if it hasn't been tested, it may have a small (or big!) error that causes it to fail... .

For this week's assignments, we provide a set of tests that you can (and should!) paste into your code. Then, when you run your file, the tests will run and you can check (by sight, in this case) whether any of the tests has not passed... .

For this week, if your functions pass the provided tests, they will pass all of the graders' tests, too. (In the extra credit and in future assignments, we may add more tests of our own...)

Here's an example using the `flipside(s)` function from Lab 1. Paste this into your file and run it:

```
def flipside(s):
 """flipside swaps s's sides
 Argument s: a string
 """
 x = len(s)//2
 return s[x:] + s[:x]

#
Tests
#
assert flipside('carpets') == 'petscar'
assert flipside('homework') == 'workhome'
assert flipside('flipside') == 'sideflip'
assert flipside('az') == 'za'
assert flipside('a') == 'a'
assert flipside('') == ''
```

When you run this, you should get `silence` from the tests. However, if one of your test fails you will get a message reading `AssertionError:`" along with a pointer to the line that failed. You can try this out by temporarily changing the last line above to read:

```
assert flipside('') == 'verywrong'
```

(which is obviously very wrong!) and running your file again. It will fail on that line. Don't forget to put it back!

For all the functions below, be sure to paste in the tests, too—and run them!

## The Functions to Write...

- **Function 1:** First, write `mult(n, m)`. Here is a full description of how it should work:

`mult(n, m)` should return the product of the two integers `n` and `m`. Since this would be a bit *too* easy if the multiplication operator `*` were used, for this function, you are limited to using the addition, subtraction, and negation operators, along with recursion. (Use the `power` function we did in class as a guide.) Some examples:

```
In [1]: mult(6, 7)
Out[1]: 42
```

```
In [2]: mult(6, -3)
Out[2]: -18
```

[RecursionExamples](#) This link contains the recursive `power` function you wrote in class.

Here are the tests to try:

```

Tests
#
assert mult(6, 7) == 42
assert mult(6, -7) == -42
assert mult(-6, 7) == -42
assert mult(-6, -7) == 42
assert mult(6, 0) == 0
assert mult(0, 7) == 0
assert mult(0, 0) == 0
```

- **Function 2:** Next, write `dot(L, K)`. Here is this function's description:

`dot(L, K)` should return the *dot product* of the lists `L` and `K`. If the two argument lists are not of equal length, `dot` should return `0.0`. If the two lists are both empty, `dot` also should return `0.0`. You should assume that the argument lists contain only numeric values. (Compare this with the `mylen` example we did in class, but be sure to account for *both* lists—and remember they're lists, not strings! Here is the `leng` example, modified slightly to handle both lists and strings!

*What's the dot product?* The dot product of two vectors or lists is the sum of the products of the elements in the same position in the two vectors. for example, the first result below is  $5*6$  plus  $3*4$ , which is 42. The result here is 42.0 rather than 42, because we used a float of 0.0 in the base case.

You're welcome to use the multiplication operator \* for this problem, for sure!

```
In [1]: dot([5, 3], [6, 4])
Out[1]: 42.0

In [2]: dot([1, 2, 3, 4], [10, 100, 1000, 10000])
Out[2]: 43210.0

In [3]: dot([5, 3], [6])
Out[3]: 0.0
```

Here are the tests to try:

```

Tests
#
assert dot([5, 3], [6, 4]) == 42.0
assert dot([1, 2, 3, 4], [10, 100, 1000, 10000]) == 43210.0
assert dot([5, 3], [6]) == 0.0
assert dot([], [6]) == 0.0
assert dot([], []) == 0.0
```

- **Function 3:** Next, write `ind(e, L)`. Here is its description:

Write `ind(e, L)`, which takes in a sequence `L` and an element `e`. `L` might be a string, or it might be a list.

Your function `ind` should return the *index* at which `e` is first found in `L`. The index begins at 0, as is usual with lists. If `e` is NOT an element of `L`, then `ind(e, L)` should return the integer equal to `len(L)`. You may **not** use the built-in `index` function of Python. Here are a few examples:

```
In [1]: ind(42, [55, 77, 42, 12, 42, 100])
Out[1]: 2

In [2]: ind(42, list(range(0, 100)))
Out[2]: 42

In [3]: ind('hi', ['hello', 42, True])
Out[3]: 3

In [4]: ind('hi', ['well', 'hi', 'there'])
Out[4]: 1
```

```
In [5]: ind('i', 'team')
Out[5]: 4

In [6]: ind(' ', 'outer exploration')
Out[6]: 5
```

In this last example, the first argument to `ind` is a string of a single space character, *not* the empty string.

Hint

Just as you can check whether an element is in a sequence with

```
if e in L:
```

you can also check whether an element is *not* in a sequence with

```
if e not in L:
```

This latter syntax is useful for the `ind` function! As with `dot`, `ind` is probably most similar—but not identical—to `leng` from the RecursionExamples.

Here are the tests to try:

```

Tests

assert ind(42, [55, 77, 42, 12, 42, 100]) == 2
assert ind(42, list(range(0, 100))) == 42
assert ind('hi', ['hello', 42, True]) == 3
assert ind('hi', ['well', 'hi', 'there']) == 1
assert ind('i', 'team') == 4
assert ind(' ', 'outer exploration') == 5
```

- **Function 4:** Next, write `letterScore(let)`. (Watch for capitalization in the name!) Here is its description:

`letterScore(let)` should take a single argument, which is a single-character string, and return the value of that character as a Scrabble tile. If the argument is not one of the letters from 'a' to 'z', the function should return 0.

To write this function you will need to use this mapping of letters to scores

|                |                |                |                 |                |                |                |                |                |                |                |                |                 |
|----------------|----------------|----------------|-----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|-----------------|
| A <sub>1</sub> | B <sub>3</sub> | C <sub>3</sub> | D <sub>2</sub>  | E <sub>1</sub> | F <sub>4</sub> | G <sub>2</sub> | H <sub>4</sub> | I <sub>1</sub> | J <sub>8</sub> | K <sub>5</sub> | L <sub>1</sub> | M <sub>3</sub>  |
| N <sub>1</sub> | O <sub>1</sub> | P <sub>3</sub> | Q <sub>10</sub> | R <sub>1</sub> | S <sub>1</sub> | T <sub>1</sub> | U <sub>1</sub> | V <sub>4</sub> | W <sub>4</sub> | X <sub>8</sub> | Y <sub>4</sub> | Z <sub>10</sub> |

**What!?** Do I have to write 25 or 26 *if elif* or *else* statements? **No!** Instead, use the `in` keyword:

```
In [1]: 'a' in 'this is a string including a'
Out[1]: True
```

```
In [2]: 'q' in 'this string does not have the letter before r'
Out[2]: False
```

*OK!* ... but how does this help...?

Consider a conditional such as this:

```
if let in 'qz':
 return 10
```

One note: `letterScore` does **not** require recursion. But recursion *is* used in the next one...

Here are some examples of `letterScore` in action:

```
In [1]: letterScore('w')
Out[1]: 4
```

```
In [2]: letterScore('%')
Out[2]: 0
```

**Tests?** Write a few tests for this one yourself...it will also be tested in conjunction with the next function! **But be sure to test `letterScore` separately, or you will have great difficulty with `scrabbleScore`!**

- **Function 5:** Next, write `scrabbleScore(S)`. (Again, watch for capitalization!) Here is `scrabbleScore`'s description: `scrabbleScore(S)` should take a string argument `S`, which will have only lowercase letters, and should return the Scrabble score of that string. Ignore the fact that, in reality, the availability of each letter tile is limited.

Hint

Use the above `letterScore` function and recursion. (Compare this with the `vwl` example we did in class, but consider adding *different* values for each letter. Here are the `RecursionExamples`).

Here are some examples:

```
In [1]: scrabbleScore('quetzal')
Out[1]: 25
```

```
In [2]: scrabbleScore('jonquil')
Out[2]: 23
```

```
In [3]: scrabbleScore('syzygy')
Out[3]: 25
```

Here are the tests to try:

```

Tests

assert scrabbleScore('quetzal') == 25
assert scrabbleScore('jonquil') == 23
assert scrabbleScore('syzygy') == 25
assert scrabbleScore('abcdefghijklmnopqrstuvwxyz') == 87
assert scrabbleScore('?!@#$%^&*()') == 0
assert scrabbleScore('') == 0
```

- **Function 6:** Finally, write `transcribe(S)`. Here is its description:

**DNA -> RNA transcription** In an incredible molecular feat called *transcription*, your cells create molecules of messenger RNA that mirror the sequence of nucleotides in your DNA. The RNA is then used to create proteins that do the work of the cell.

Write a recursive function `transcribe(S)`, which should take an argument that is a string `S`, which will have DNA nucleotides (capital letter As, Cs, Gs, and Ts).

There may be other characters, too, though they should be ignored by your `transcribe` function by simply disappearing from the return value. These might be spaces or other characters that are not really DNA nucleotides.

`transcribe` should return the messenger RNA that would be produced from that string `S`. The correct return value simply uses replacement:

- As in the argument become Us in the result.
- Cs in the argument become Gs in the result.
- Gs in the argument become Cs in the result.
- Ts in the argument become As in the result.
- any other input characters should disappear from the result altogether

As with the previous problem, you will want a helper function that converts one nucleotide. Feel free to use this as a start for this helper function:

```

def one_dna_to_rna(c):
 """Converts a single-character c from DNA
 nucleotide to complementary RNA nucleotide """
 if c == 'A':
 return 'U'
 # you'll need more here...

```

You'll want to adapt the `vwl` example, but adding together *strings*, instead of numbers! Here are the RecursionExamples.

Here are some examples of `transcribe`:

```
In [1]: transcribe('ACGT TGCA') # space should be removed
Out[1]: 'UGCAACGU'
```

```
In [2]: transcribe('GATTACA')
Out[2]: 'CUAAUGU'
```

```
In [3]: transcribe('cs5') # lowercase doesn't count
Out[3]: ''
```

**Not quite working?** One common problem that can arise is that `one_dna_to_rna` lacks an `else` case to capture all of the non-legal characters. Since all non-nucleotide characters should be dropped, this can be fixed by including code similar to this:

```

else:
 return '' # return the empty string if it's not a legal nucleotide

```

There are different ways around this, too, but this is one problem that has appeared a few times. Note that the `else` above is only for `one_dna_to_rna`, *not* for `transcribe` itself.

Here are the tests to paste and try:

```

#
Tests
#
assert transcribe('ACGTTGCA') == 'UGCAACGU'
assert transcribe('ACG TGCA') == 'UGCACGU' # Note that the space disappears
assert transcribe('GATTACA') == 'CUAAUGU'
assert transcribe('cs5') == '' # Note that the other characters disappear
assert transcribe('') == ''

```

## **Extra!**

This week's extra-credit shows off a wonderful practice website for Python functions, called CodingBat. In addition, it offers a challenge "pig-Latin" function to write...

There are three opportunities:

- Practice with strings on CodingBat with Python strings
- Practice with lists on CodingBat with Python lists
- Write a pig-Latin-izing function (more with strings...)

### **Extra #1: CodingBat for Python Strings**

For extra credit, complete *all of the Python string problems* on CodingBat's "String-1" Python string page. Use as many attempts as you'd like.

### **Extra #2: CodingBat for Python Lists**

If you like the CodingBat practice-problem site, try some more! Complete *all of the Python list problems* on CodingBat's "List-1" Python list page. Use as many attempts as you'd like.

### **Extra #3: Pig Latin!**

This problem asks you to write two functions that implement an English-to-Pig-Latin translator.

Be sure to name and test your functions carefully. In each, include a doc-string, which should indicate what the function computes (returns) and what its arguments are or what they mean.

This problem is inspired by



**Warm up:**

Write `pigletLatin(s)`, which accepts an argument that is a string `s`. `s` will be a single word consisting of lowercase letters. Then, `pigletLatin` should return the translation of `s` to "piglet latin," which has these rules:

- If the argument has no letters at all (the empty string), your function should return the empty string
- If the argument begins with a vowel, the piglet latin result simply appends the string 'way' at the end. 'y' will be considered a consonant, and not a vowel, for this problem.

**Example:** `pigletLatin('one')` returns 'oneway'

- If the argument begins with a consonant, the piglet latin result is identical to the argument, except that the argument's initial consonant is at the end of the word instead of the beginning and it's followed by the string 'ay'.

**Example:** `pigletLatin('be')` returns 'ebay'

- Be sure to write some tests using `assert!`
- Of course, this is not full pig Latin, because it does not handle words beginning with multiple consonants correctly. For example, `pigletLatin('string')` returns 'tringsay'.

You'll fix this next!

**The real pig Latin challenge:**

Create a function called `pigLatin(s)` that handles the rules above *and* handles more than one initial consonant correctly in the translation to pig Latin. That is, `pigLatin` moves *all* of the initial consonants to the end of the word before adding 'ay'. (You may want to write and use a helper function to do this—see the hint below.)

Also, `pigLatin` should handle an initial 'y' either as a consonant **OR** as a vowel,

depending on whether the y is followed by a vowel or consonant, respectively. For example, 'yes' has an initial y acting as a consonant. The word 'yttrium', however, (element #39) has an initial y acting as a vowel. Here are some additional examples:

```
In [1]: pigLatin('string')
Out[1]: ingstray
```

```
In [2]: pigLatin('yttrium')
Out[2]: yttriumway
```

```
In [3]: pigLatin('yoohoo')
Out[3]: oohooyay
```

*Tests?* These we're leaving up to you!

Hint

One way to use recursion to assist in this is to write a function

```
def initial_consonants(s):
```

that returns a string of all of the initial consonants in its string argument `s`. Thus, if `s` starts with a vowel, the empty string '' will be returned.

If you think about this problem thoroughly, you'll find that not every possible case has been accounted for—you are free to decide the appropriate course of action for those "corner cases." We won't test these...

Oodgay ucklay!

Website design by Madeleine Masser-Frye and Allen Wu

# CSforAll - SleepwalkingStudent

## CS for All

CSforAll Web > Chapter2 > SleepwalkingStudent

This website accompanies the textbook *CS For All* by Alvarado, Dodds, Kuenning, and Libeskind-Hadas

### The Sleepwalking Student

You will write Python functions to investigate the behavior of a sleepwalking student, a.k.a., a "random walk." The student repeatedly takes steps inside a long hallway, but being asleep, each step is in a random direction (either left or right). Will the student be stuck in the hallway forever? Or will they eventually reach the CS labs (or bed)?

#### Part 1: Copy This Function: `rs()`

Start your file with this header and function, named `rs()`

```
import random

def rs():
 """rs chooses a random step and returns it.
 note that a call to rs() requires parentheses
 arguments: none at all!
 """
 return random.choice([-1, 1])
```

You can call `rs()` function whenever you want to obtain a new, random step: either 1 or -1.

An advantage of this is that it is easy to change what is meant by "random step" in the future, without changing any other code!

```
import vs from ... import * If you use the library-import statement
import random
```

you can use the `random` library, but you will need to preface each call with the library's name:

```
random.choice([-1, 1])
```

Another way to import libraries is to use

```
from random import *
```

In this case, you can simply type `choice([-1, 1])`, which is a bit shorter.

A problem could arise if you had *another* function named `choice` already. For the moment, that isn't a concern.

**A Reminder About String Multiplication** For this problem, string *multiplication* is very useful. Here is a reminder:

```
In [1]: print('spam' * 3)
```

```
spamspamspam
```

```
In [2]: print('start|' + '_'*10 + '|end')
```

```
start|_____|end
```

In this latter example, `'_*10` specified how much space to place between `'start|'` and `'|end'`.

It's nice to use underscores (or some other "ocean") for your sleepwalker to wander in!

### Part 2: Write: `rwpos(start, nsteps)`

Next, write a function named `rwpos(start, nsteps)` which takes two arguments:

- An integer, `start`, representing the starting position of our sleepwalker, and
- A nonnegative integer, `nsteps`, representing the number of random steps to take from this starting position.

The name, `rwpos` is a reminder that this function should return the `random` walker's `position`.

Write `rwpos` so that it returns the `position` of the sleepwalker after `nsteps` random steps, where each step moves according to `rs()`, which means either plus 1 or minus 1 from the previous position.

**Example random Code...** Here is some of random-number-guessing code, if you'd like to use it as a starting point...

**Debugging Code to Include** As part of your `rwpos` function, include a line of debugging code that prints what `start` is each time the function is called. Include the string `start is`, too, as in the examples below.

Remember that, because each step is random, the exact values your function produces will almost certainly be different than these, though the overall behavior should be the same:

```
In [1]: rwpos(40, 4)
start is 40
start is 41
start is 42
start is 41
start is 42
Out[1]: 42

In [2]: rwpos(40, 4) # won't be the same each time...
start is 40
start is 39
start is 38
start is 37
start is 36
Out[2]: 36
```

**Is It 4 or 5 Printed Lines?** You may have four lines of output instead of five—this most likely depends on whether or not you print when the base case is hit. Either way is completely fine for this problem.

**No Loops!** Even if you've used `while` or `for` loops in the past, for this problem we ask you to **use recursion**.

These assignments are primarily to develop *design* skills—specifically, recursive design. Don't worry—there will be plenty of loops later in the term.

### Part 3: Write `rwsteps(start, low, hi)`

Next, write `rwsteps(start, low, hi)` which takes three arguments:

- An integer, `start`, representing the starting position of our sleepwalker,

- An integer, `low`, which will always be nonnegative, representing the smallest value our sleepwalker will be allowed to wander to, and
- An integer, `hi`, representing the highest value our sleepwalker will be allowed to wander to.

You may assume that `hi >= start >= low`.

**What should `rwsteps` do?** It should simulate a random walk, printing each step (see below). Also, as soon as the sleepwalker reaches *at or beyond* the `low` or `hi` value, the random walk should stop. When it does stop, `rwsteps` must return the **number of steps** that the sleepwalker took in order to finally reach the lower or upper bound.

**Printing/debugging code:** In `rwsteps` include a line of debugging code that prints a visual representation of your sleepwalker's position while wandering!

Feel free to be more creative than a simple '`S`' character. For example, consider `0->-<` (a true sleepwalker!)

As an extra-credit challenge (a fun one), you might create a more elaborate sleepwalker simulation that changes its looks depending on which direction it's heading (eyes looking left or right?). Or, it could interact with some other items, people, or things on its path—see the extra credit, below.

**Examples** Here are two plain-wandering examples, one using spaces and one using the underscore character (making it easier to see what's going on than with spaces!). One has walls on either side and one does not. The specifics of spacing, walls, etc, are entirely up to you—be creative! Also, as a reminder, you can create a string of 10 underscore characters with `10*'_'`; string multiplication is helpful here!

```
In [1]: rwsteps(10, 5, 15)
 |____S____|
 |__S_____|
 |_S_____|
 |_S_____|
 |_S_____|
 |_S_____|
 |_S_____|
 |_S_____|
 |_S_____|
 |S_____|
Out[1]: 9 # here is the return value!

In [2]: rwsteps(10, 7, 20)
 S
 S
 S
 S
```

```
S
S
S
S
S
S
S
S
S
Out[2]: 11
```

**Use recursion** to implement `rwsteps` for this problem.

Hint

This problem can be tricky because you are *both* adding a random step **and** adding to the ongoing count of the total number of steps!

One way to do this is to use the line `rest_of_steps = rwsteps(newstart, low, hi)` as the recursive call, *with an appropriate assignment to newstart* on the line above it, and an appropriate use of `rest_of_steps` in the return value below it.... .

**Recursion limit exceeded?** You can get more memory for recursion by adding these lines to the top of your file:

```
import sys
sys.setrecursionlimit(50000)
```

This provides 50000 function calls in the recursive stack.

**Want to slow down your sleepwalker?** You can also slow down the simulation by adding these lines to the top of your file:

```
import time
import sys
```

Then, in your `rwsteps` or `rwpow` functions, you can include the lines

```
sys.stdout.flush() # forces Python to print everything _now_
time.sleep(0.1) # and then sleep for 0.1 seconds
```

Adjust as you see fit!

## Part 4: Create Simulations to Analyze Your Random Walks

To analyze random walks, we need two terms:

1. The "*signed displacement*" is the number of steps *away from the start* that the random walker has reached. It is signed, because displacements

to the right are considered positive and displacements to the left are considered negative. This is natural: to find the signed displacement, simply subtract; it's the ending position of the random walker minus the starting position of the random walker. To do this, you will write a variation of `rwpos`, not `rwsteps`.

2. The "*squared displacement*" is the *square* of the number of steps away from the start that the random walker has reached. That is, it is the square of the signed displacement.

With these two terms in mind, here are the two questions we ask you to investigate:

- What is the average final *signed displacement* for a random walker after making 100 random steps? What about after  $N$  random steps? As described above, the signed displacement is just the result of `rwpos` minus the `start` location. Do **not** use `abs`.
- What is the average *squared displacement* for a random walker after making 100 random steps? What about after  $N$  random steps, in terms of  $N$ ? Be sure you square the signed displacements **before** you sum the values in order to average them!

You should adapt the random-walk functions you wrote to investigate these two questions. In particular, you should

- **To-do item #1** Write a version of `rwpos` that does **not** print any debugging or explanatory information. Rather, it should simply return the final position. Call this new version `rwposPlain`. **Be careful!** the recursive call(s) will need to change so that they call `rwposPlain`, not `rwpos`!
- **To-do item #2** Come up with a plan for how you will answer these questions. This plan should include a list comprehension similar to the following:

```
LC = [rwposPlain(0, 100) for x in range(142)]
```

Not surprisingly, the 142 will probably be replaced by a variable in your final implementation. To find the *average* of the values created, you will use `sum(LC)`, along with `len(LC)`...

- **To-do item #3** To build intuition, run the above list comprehension at the Python In [42]: prompt. (For some value other than 42...) Look

at the resulting value of LC (there will be 142 elements). Also, find the average of LC. Don't use a calculator; figure out how to do it in Python!

- **To-do item #4** Write two more functions:

- `ave_signed_displacement(numtrials)`, which should run `rwposPlain(0, 100)` for `numtrials` times and return the average of the result. Use the above list comprehension as the first line of your function! (You'll want to replace 142 with a variable...)
- `ave_squared_displacement(numtrials)`, which should run `rwposPlain(0, 100)` for `numtrials` times and return the average of the *squares* of the results! One way to do this is to create a slightly different list comprehension. Remember that `x**2` is Python's way of squaring `x`.

- Then, use your functions and reflect on the results you find from these computational tests. To do this, place your answers inside your python program file by either making them comments (using the `#` symbol) OR, *even easier*, including them in triple-quoted strings (since those can include newlines). For example,

```
"""
To compute the average signed displacement for
a random walker after 100 random steps, I ...
(briefly explain what you did and your results)

Be sure to copy the data and average from at least
one of your runs of ave_signed_displacement and
at least one of your runs of ave_squared_displacement
"""
```

Thus, your file should include

- (1) answers to these two questions and how you approached them and
- (2) the above Python functions, including `ave_signed_displacement(numtrials)` and `ave_squared_displacement(numtrials)`

Make sure to include explanatory docstrings and comments for each function you write!

Please include any references you might have used—you're welcome to read all about random walks online, if you like. However, you should also feel free not to bother—whether your answers and analyses are correct or not will have *no effect* on the grading of this Part 4 of this problem! Rather, it will be graded on

whether your functions work as they should, whether they *would be helpful* in answering those questions, and in the clarity and effectiveness of your write-up.

### **Extra: Optional Extra-Credit Variations**

For extra-credit points (optional), feel free to make variations in the ASCII rendering of your sleepwalker(s)...

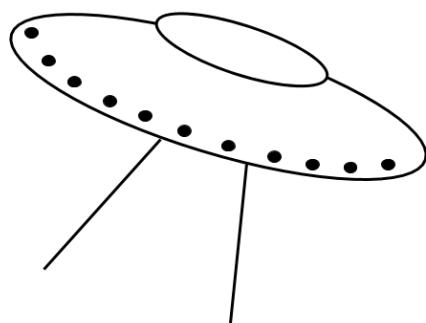
- For example, a particularly creative ASCII emoji or boundary
- A character that changes depending on whether it's moving left or right
- A separate `rwsteps` function that has more than one wanderer (perhaps interacting with each other)
- Combining all of these or doing something totally crazy (2d, anyone?)

Be sure to add a clear and obvious comment bragging about your extras—we don't want to miss them!

Website design by Madeleine Masser-Frye and Allen Wu

## CSforAll - MoreTurtle

### CS for All



CSforAll Web > Chapter2 > MoreTurtle

This website accompanies the textbook *CS For All* by Alvarado, Dodds, Kuennen, and Libeskind-Hadas

#### More Turtle

##### **Your Custom *Recursive Design***

Here, create one or more recursive functions that uses turtle graphics to create a recursive artwork...

You may create your own functions, or add new parameters or capabilities to the functions you wrote in lab, or both.

We're "pining" for some interesting variants. If you do add extra features to your tree—or create any other work—make sure to "fir"nish us with a comment within your file explaining how to run your code.

Any composition is welcome. For full extra credit, there needs to be *some* recursive function of your own design that contributes to your art, but it does not have to be entirely recursive!

### **Other Turtle Capabilities**

Feel free to explore Python's turtle library reference for ideas on turtle's capabilities.

The `fill` and `color` settings, in particular, are fun to try...

**Have fun with turtle!**

Website design by Madeleine Masser-Frye and Allen Wu

# CSforAll - ReadItandWeep

## CS for All

CSforAll Web > Chapter2 > ReadItandWeep

This website accompanies the textbook *CS For All* by Alvarado, Dodds, Kuenning, and Libeskind-Hadas

### The Read-it-and-Weep Sequence

In this problem, you will implement the mathematically interesting Look-and-Say sequence. This is a strange mathematical sequence that starts like this:

1, 11, 21, 1211, 111221, 312211, 13112221, ...

What's the pattern? It starts with 1. When we see that 1, we say "I see one one". So, the next number in the sequence is "one one" or 11. When we see 11, we say "I see two ones". So, the next number in the sequence is 21 (two one). Now, we look at that number and say "I see one two and one one", so the next number in the sequence is 1211 (one two one one). And so forth!

Later, your task in this problem is to write a program that prints out the `next` number in this sequence, *starting with any initial term!*

What makes this problem different than many others is that *the design is entirely up to you!* You should consider

- Writing a helper function (or more than one!)
- What types of loops you will need (`for` or `while`)
- Or, feel free to use recursion (it can be done using recursion equally well...)

Finally, you'll use `next` to generate as many terms as you might like...

#### Writing `next`

So, the main function to write is named `next(term)`. The details:

- `term` should be any integer
- the `next` function should return the next "read" term, which needs to be an `int`, based on the input `term`

Again, the design is up to you. Here are some examples we will test (and there will be a few more, too!)

```
In [1]: next(21)
Out[1]: 1211
```

```
In [2]: next(2222) # notice this won't happen in the real sequence
Out[2]: 42
```

```
In [3]: next(312211)
Out[3]: 13112221
```

**Warning!** Your `next` function needs to output an int! (The autograder will be upset otherwise!) Use `int(s)` if you need to!!

Hint

- You can convert from an integer `x` to a string, e.g., with `str(x)`
- You can convert from a string `s` to an integer, e.g., with `int(s)`
- `next` should return an integer (`int`), but remember that you can convert back and forth from strings using the two functions above!

### Read-it-and-Weep

Now, write a function called `readit(n)` that prints the first `n` terms in the read-it-and-weep sequence: one per line, starting with a 1. This `readit` function is unusual in that it does *not* need to return any value at all: it is just being used to print things.

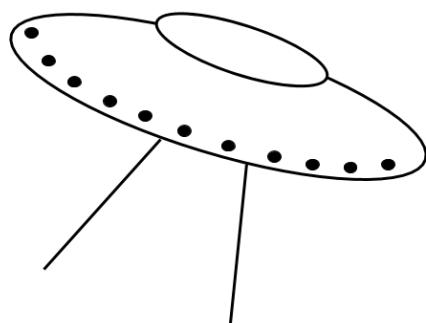
Here's an example of `readit(n)` in action...

```
In [1]: readit(6)
1
11
21
1211
111221
312211
```

Website design by Madeleine Masser-Frye and Allen Wu

## CSforAll - RecursionMuscles

### CS for All



CSforAll Web > Chapter2 > RecursionMuscles

This website accompanies the textbook *CS For All* by Alvarado, Dodds, Kuennen, and Libeskind-Hadas

### Recursion Muscles

This problem asks you to write the following Python functions \*using recursion\* (not loops!) and to test these functions carefully.

You may use recursion, conditional statements (`if`, `else`, `elif`), and list or string

indexing and slicing. Note that some of these problems can be written without using recursion, e.g. using `map`, `filter`, `reduce`, or other "mass-processing" structures. However, the objective here is to build your recursion muscles, so please stick to recursion. **DO NOT** use any loop structures (`while` or `for`). Loops will make your recursion muscles weak and flabby.

Try to keep your functions as "lean and clean" as possible. That is, keep your functions short and elegant.

Do not use built-in functions (e.g. `len`, `sum`, etc.). However, your functions may call other functions that you write yourself. Calling another function for help will mostly be unnecessary, but it may be handy in a few places.

Be sure to include a docstring under the signature line of each function. The docstring should indicate what the function computes (returns) and what its arguments are or what they mean.

- `dot(L, K)` should return the dot product of the lists L and K. Recall that the dot product of two vectors or lists is the sum of the products of the elements in the same position in the two vectors. You may assume that the two lists are of equal length. If they are of different lengths, it's up to you what result is returned. If the two lists are both empty, `dot` should return 0.0. Assume that the argument lists contain only numeric values.

```
In [1]: dot([5,3], [6,4]) <-- Note that 5*6 + 3*4 = 42
Out[1]: 42
```

Besides the example above, try `dot([1], [])` and `dot([], [42])` and a few others of your own devising.

- `explode(S)` should take a string S and return a list of the characters (each of which is a string of length 1) in that string. For example:

```
In [1]: explode("spam")
Out[1]: ['s', 'p', 'a', 'm']
```

```
In [2]: explode("")
Out[2]: []
```

Note that Python is happy to use either single quotes or double quotes to delimit strings—they are interchangeable. But if you use a single quote at the start of a string you must use one at its end (and similarly for double quotes). For example:

```
In [1]: "spam" == 'spam'
Out[1]: True
```

- `ind(e, L)` accepts an element e and a sequence L, where by "sequence" we mean either a list or a string (fortunately indexing and slicing work the same for both lists and strings, so your `ind` function should be able to handle both types of arguments!). Then `ind` should return the index at

which `e` is **first** found in `L`. Counting begins at 0, as is usual with lists.

If `e` is NOT an element of `L`, then `ind(e, L)` should return an integer that is **exactly** the length of `L`.

Remember, don't use the `len` function explicitly though! Your recursive implementation can find the length by itself.

```
In [1]: ind(42, [55, 77, 42, 12, 42, 100])
Out[1]: 2
```

```
In [2]: ind(42, list(range(0,100)))
Out[2]: 42
```

```
In [3]: ind('hi', ['hello', 42, True])
Out[3]: 3
```

```
In [4]: ind('hi', ['well', 'hi', 'there'])
Out[4]: 1
```

```
In [5]: ind('i', 'team')
Out[5]: 4
```

```
In [6]: ind(' ', 'outer exploration')
Out[6]: 5
```

- `removeAll(e, L)` accepts an element `e` and a `list` `L`. Then `removeAll` should return another list that is the same as `L` except that all elements identical to `e` have been removed. Notice that `e` has to be a top-level element to be removed, as the examples illustrate:

```
In [1]: removeAll(42, [55, 77, 42, 11, 42, 88])
Out[1]: [55, 77, 11, 88]
```

```
Below, 42 is NOT top-level!
In [2]: removeAll(42, [55, [77, 42], [11, 42], 88])
Out[2]: [55, [77, 42], [11, 42], 88]
```

```
Below, [77,42] IS top-level!
In [3]: removeAll([77, 42], [55, [77, 42], [11, 42], 88])
Out[3]: [55, [11, 42], 88]
```

Aside: It's possible to write `removeAll` so that it works even if the second argument is a string instead of a list, but you do not need to do so here.

- `deepReverse(L)` accepts a list of elements, where some of those elements may be lists themselves. `deepReverse` returns the reversal of the list where, additionally, any element that is a list is also deepReversed. Here are some examples:

```
In [1]: deepReverse([1, 2, 3])
Out[1]: [3, 2, 1]

In [2]: deepReverse([1, [2, 3], 4])
Out[2]: [4, [3, 2], 1]

In [3]: deepReverse([1, [2, [3, 4], [5, [6, 7], 8]]])
Out[3]: [[[8, [7, 6], 5], [4, 3], 2], 1]
```

For this problem, you will need the ability to test whether or not an element in the list is a list itself. To this end, you can use the following line of code, which tests whether or not `x` is a list:

```
if type(x) == list:
 # if True you will end up here
else:
 # if False you will end up here
```

Just FYI, you can similarly test if an item is a string (use `str` rather than `list` in the above code) or an integer (use `int`) or any other Python datatype. Nifty!

- `deepRemoveAll(item, L)` removes the given item from the given list `L`, no matter how deeply the item is nested in the list. To clarify, recall that the `removeAll` function that you wrote above works like this:

```
In[1]: removeAll(42, [42, 67, 42, [42, 42, 43], 47])
Out[1]: [67, [42, 42, 43], 47]
```

In other words, in this example, it removes all of the 42's from the list, but it does not remove 42's that are embedded in lists within that list. Your job here is to write a function called `deepRemoveAll` that scours its input and removes not only the desired item, but also removes that item deep inside other lists. Here are a few examples:

```
In[2]: deepRemoveAll(42, [42, 67, 42, [41, 42, 43], 47])
Out[2]: [67, [41, 43], 47]
```

```
In[3]: deepRemoveAll(47, [42, 47, [1, 2, [47, 48, 49], 50, 47, 51], 52])
Out[3]: [42, [1, 2, [48, 49], 50, 51], 52]
```

- `letterScore(letter, scorelist)` accepts a single letter string called `letter` and a list, where each element in that list is itself a list of the form `[character, value]`. In those inner lists, `character` is a single letter and `value` is a number associated with that letter (e.g., its Scrabble score). The `letterScore` function then returns a single number, namely the value associated with the given `letter`. For example, you can cut and paste the following Scrabble score list into your `hw1pr1.py` file:

```
scrabbleScores = [["a", 1], ["b", 3], ["c", 3], ["d", 2], ["e", 1],
```

```
["f", 4], ["g", 2], ["h", 4], ["i", 1], ["j", 8],
["k", 5], ["l", 1], ["m", 3], ["n", 1], ["o", 1],
["p", 3], ["q", 10], ["r", 1], ["s", 1], ["t", 1],
["u", 1], ["v", 4], ["w", 4], ["x", 8], ["y", 4],
["z", 10]]
```

If you include this in your file (outside of any function you define—for example right after the header comments in your file)—then `scrabbleScores` is a "global variable"; it can be referred to by any function defined in that file and, more importantly for this example, it can be used once we load in that file.

```
In [1]: letterScore("c", scrabbleScores)
Out[1]: 3
```

```
In [2]: letterScore("a", scrabbleScores)
Out[2]: 1
```

If the `letter` is not in the `scorelist`, `letterScore` should not crash. Instead, it should return something sensible (such as 0). This is an example of *input validation*—making sure your program behaves well even if it is misused. (Bad input validation is the number-one cause of computer security problems!)

- `wordScore(S, scorelist)` should accept a string `S` and a `scorelist` in the format described above, and should return the Scrabble score of that string. Again, `wordScore` should behave well if `S` contains letters not found in `scoreList`. However, you are allowed to crash badly if `scoreList` is in the wrong format (such as not being a list at all); that's because we haven't yet learned the way to protect against that kind of crash.

Here are some examples:

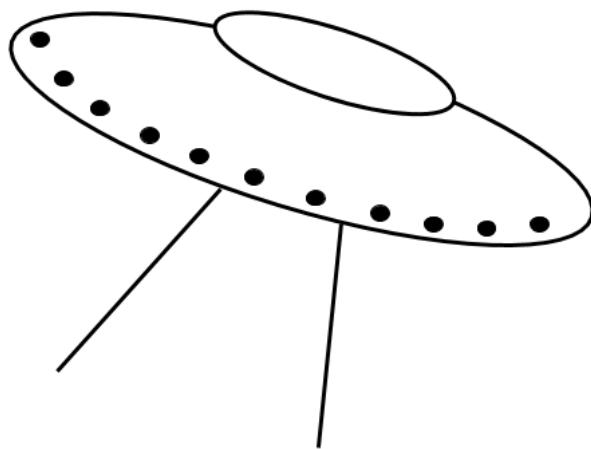
```
In [1]: wordScore('spam', scrabbleScores)
Out[1]: 8
```

```
In [2]: wordScore("wow", [['o', 10], ['w', 42]])
Out[2]: 94
```

Website design by Madeleine Masser-Frye and Allen Wu

# CSforAll - GooglesSecret

## CS for All



CSforAll Web > Chapter2 > GooglesSecret

This website accompanies the textbook *CS For All* by Alvarado, Dodds, Kuenning, and Libeskind-Hadas

### Google's "Secret"

In this problem you will write a few interesting Python functions using Python's `map` and `reduce` functions. These functions are at the heart of the software in Google. You can Google "mapreduce" to see evidence of this.

#### Introduction to `map`

Consider the `dbl` function:

```
def dbl(x):
 """ dbl returns twice its argument x
 x: a number (int or float)"""
 return 2*x
```

Now, take a look at this example:

```
In [1]: mylist = list(range(1, 5))
```

```
In [2]: mylist
```

```
Out[2]: [1, 2, 3, 4]
```

```
In [3]: newlist = list(map(dbl, mylist))
```

```
In [4]: newlist
```

```
Out[4]: [2, 4, 6, 8]
```

```
In [5]: mylist
```

```
Out[5]: [1, 2, 3, 4]
```

Notice that the `map` function (which is built in to Python) took two arguments: The first is the name of a function (in this case `dbl`) and the second is a list (in this case the list `[1, 2, 3, 4]` that we created with `range`). The result of `map(dbl, mylist)` was a new list which has the same number of elements as the original list but every element got doubled! What actually happens here is that `map` causes each element in `mylist` to get "plopped" in to the `dbl` function. The `dbl` function then returns a new number (the double of its argument) and that new number goes into the new iterable which `map` then returns to us. We then cast that iterable as a list so that we can see it and further manipulate it.

Recall that the built-in `sum` function takes a list of numbers as its argument and returns the sum of the numbers in the list. For example:

```
In [1]: sum(list(range(1, 5)))
```

```
Out[1]: 10
```

Remember that `range` generates an iterable that does not include the endpoint, so `list(range(1, 5))` returns the list `[1, 2, 3, 4]`.

Here is a function that accepts an integer argument `n` and returns the sum  $0 + 2 + 4 + \dots + 2n$ .

```
def doublesum(n):
 """returns the sum 0 + 2 + ... + 2n """
 list1 = list(range(1, n+1))
 list2 = list(map(dbl, list1))
 answer = sum(list2)
 return answer
```

Of course, this could also have been written this way:

```
def doublesum1(n):
 return sum(list(map(dbl, (range(1, n+1)))))
```

And then we could have been sneaky and factored out the 2 to compute  $2(0 + 1 + \dots + n)$  and written it this way:

```
def doublesum2(n):
 return 2 * sum(list(range(1, n+1)))
```

## Introduction to reduce

Note that you will need to use `from functools import reduce` to get `reduce` into Python 3. Put this at the top of your `hw0pr2.py` file.

Another way to add things up is to note that  $a + b + c + d + \dots$  can also be calculated as  $((a + b) + c) + d \dots$  The `reduce` function can do this for us:

```
In [1]: from functools import reduce
```

```
In [2]: def add2(x, y):
```

```
... return x + y
```

```
...
```

```
In [3]: def sumlist(L):
```

```
... return reduce(add2, L)
```

**Note:** In the above inputs, we advise you to type in the function manually, as copy pasting the example inputs above is a hassle. Make sure to declare your function properly, so that IPython gives you a new line within the same input.

Here, `reduce` applies `add2` to the first two elements of `L`. Then it takes that result and applies `add2` again, using the previous result and the next element of `L`. So for a three-element list we get:

```
add2(add2(L[0], L[1]), L[2])
```

which is precisely the same as `sum(L)`.

Try this in your Python shell:

```
In [1]: from functools import reduce
```

```
In [2]: def sub2(x, y):
```

```
... return x - y
```

```
...
```

```
In [3]: reduce(sub2, [1, 2, 3, 4, 5])
```

```
Out[3]: -13
```

Why does it return -13?

### Function #1: `inverse(n)`

First, write a very short and simple function called `inverse(n)` that takes a number `n` as its argument and returns its reciprocal. This function should always return a floating point number, even if the argument is an integer. For example:

```
In [1]: inverse(3)
```

```
Out[1]: 0.3333333333333333
```

### Function #2: `e(n)`

Next, write a function called `e(n)` that approximates the mathematical value `e` using a Taylor expansion.

You may know that `e` can be expressed as the sum  $1 + 1/1! + 1/2! + 1/3! + \dots$

This function will approximate `e` by adding up the first `n` terms of this sequence (after the leading 1), where `n` is some positive integer provided by the user. For example:

```
In [1]: e(1)
```

```
Out[1]: 2
```

```
In [2]: e(2)
```

```
Out[2]: 2.5
```

```
In [3]: e(3)
```

```
Out[3]: 2.6666666666666667
```

```
In [4]: e(10)
```

```
Out[4]: 2.718281801146385
```

To this end, it will help to use the `factorial` function in the `math` module. You can import the `math` module with `import math`

at the top of your file. After that, you can use `math.factorial(n)` to compute the factorial of `n`.

You'll also need to use your `inverse` function and `map` (possibly more than once)!

### Function #3: `error(n)`

Next, write a function called `error(n)` that returns the absolute value of the difference between the "actual" value of `e` (you can get this using `math.e`) and the approximation in your `e(n)` function, again assuming that `n` terms (beyond the leading 1) are used.

Note that the absolute value function is built-in to Python and is called `abs`. Here are some examples of `error(n)` in action:

```
In [1]: error(1)
Out[1]: 0.7182818284590451
```

```
In [2]: error(2)
Out[2]: 0.2182818284590451
```

```
In [3]: error(3)
Out[3]: 0.05161516179237813
```

```
In [4]: error(10)
Out[4]: 2.7312660133560485e-08
```

The last error is on the order of  $10^{-8}$ : pretty good!

#### **Function #4: factorial(n), Writing Your Own Factorial Function**

In the problem above, we used the factorial function in the math module. Here, you'll write your own factorial function. First, we start with a simple function that returns the product of its two arguments:

```
def mult(x, y):
 """Returns the product of x and y"""
 return x * y
```

Nothing too surprising here. Now, take a look at this:

```
In [1]: reduce(mult, [2, 3])
Out[1]: 6
```

```
In [2]: reduce(mult, [2, 3, 4])
Out[2]: 24
```

```
In [3]: reduce(mult, [1, 2, 3, 4])
Out[3]: 24
```

As we saw above, `reduce` takes two arguments—a function and a list—and applies that function to "compress" the list into a single value. In this case, it multiplied all of the values together.

Now, write a function `factorial(n)` that takes a positive integer `n` and uses `reduce` and `mult` to return  $n!$ .

#### **Function #5: mean(L)— This Is, Indeed, Mean...**

Next, write a function called `mean(L)` that takes a list argument and returns the mean (average) value in that list.

Using `reduce` will be handy here, especially if you define an `add` function that returns the sum of two numbers, similar in spirit to the `mult` function above.

Also, you'll need to know the number of elements in the list. This can be found using the built-in function `len`. For example:

```
In [1]: len([1, 3, 5])
Out[1]: 3
```

```
In [2]: len(list(range(1,10)))
Out[2]: 9
```

Note that technically in In [2]: you could type `len(range(1,10))`, but that's taking the length of `range()` which is itself a generator. For our purposes we will only ever be taking the lengths of lists.

Here is the `mean` function in action:

```
In [1]: mean([1, 2, 3])
Out[1]: 2
```

```
In [2]: mean([1, 1, 1])
Out[1]: 1
```

```
In [3]: mean([1, 2, 3, 4])
Out[1]: 2.5
```

Now make sure your `mean` function gets that last answer correct, i.e., 2.5.

### Extra Credit: Writing `prime(n)`

This optional problem is worth up to +4 extra-credit points.

First, take a look at this friendly little function:

```
def div(k):
 return 42 % k == 0
```

This function takes an integer argument `k` and then returns the result of evaluating the expression

```
42 % k == 0
```

The left-hand side of that expression basically computes the remainder when 42 is divided by `k`. (`k` need not be an integer, but then computing the remainder modulo `k` is a bit weird!) Next, that remainder is tested to see if it is equal to 0 (that's what the double equal sign is doing). The result is a Boolean value—either `True` or `False`. Try this function out.

Next, take a look at this strange Python function called `divides`:

```
def divides(n):
 def div(k):
 return n % k == 0
 return div
```

Notice that this function has *another function*, `div`, that is defined inside it. Moreover, `divides` returns `div`. Weird! We are returning a function rather than a number! This is a lovely feature of Python and many so-called “functional” programming languages (e.g. Scheme, Haskell, ML, among others). Play with `divides` and make sure that you feel comfortable with what is going on here. Try these examples, where use `divides` to create to handy functions named `d9` and `d10`.

```
d9 = divides(9)
d10 = divides(10)
d9(3)
d9(4)
d10(2)
d10(3)
d10(5)
```

Now, here's your challenge. Write a function called `prime(n)` that takes a positive integer argument `n` ( $n \geq 2$ ) and returns `True` or `False` depending on whether `n` is prime or composite. *You should not use any loop structures or recursion here*. Instead, you may use `map`, you may call the `divides` function above, and you may wish to use `sum`, which accepts a list of numbers and returns the sum of the numbers in that list. Aside from the `def prime(n)` line, your program should be at most three lines long. (With an advanced construct named `lambda` it could be done in only one line!)

# CSforAll - MakingChange

## CS for All

CSforAll Web > Chapter2 > MakingChange

This website accompanies the textbook *CS For All* by Alvarado, Dodds, Kuenning, and Libeskind-Hadas

### Making Change

In class, we discussed the **subset** problem and the use-it-or-lose-it paradigm that we used to solve these kinds of problems. In this problem, you will solve a different problem using the same powerful use-it-or-lose-it strategy. Start by reviewing how **subset** worked. This problem is closely related!

Imagine that you just got a job working for Cash Register Advanced Products. (The public-relations division has suggested that the company avoid using an acronym for the company name.) The company builds electronic cash registers and the software that controls them. The cash registers are used all around the world, in countries with many different coin systems.

Next, imagine that we are given a list of the coin types in a given country. For example, in the U.S. the coin types are:

[1, 5, 10, 25, 50, 100]

and in Europe they are:

[1, 2, 5, 10, 20, 50]

But in the Kingdom of Shmorbodia, the coin types are:

[1, 7, 24, 42]

In general, the coin system could be anything, except that there is **always a 1-unit coin (penny)**. In these examples, the denominations were given smallest to largest, but that's not always necessarily the case. There's nothing about use-it-or-lose-it that depends on the order of the coins.

Here's the problem. Given an amount of money and a list of coin types, we would like to find the **least number of coins** that makes up that amount of

money. For example, in the U.S. system, if we want to make 48 cents, we give out 1 quarter, 2 dimes, and 3 pennies. That solution uses 6 coins, which is the best we can do for this case. Making 48 cents in the Shmorbodian system, however, is different. Giving out a 42-cent coin—albeit tempting—will force us to give the remaining balance with 6 pennies, using a total of 7 coins. We could do better by simply giving two 24-cent coins.

Your first task is to write a function called `change(amount, coins)`, where `amount` is a non-negative integer indicating the amount of change to be made and `coins` is a list of coin values. The function should return a non-negative integer indicating the minimum number of coins required to make up the given `amount`.

Here is an example of this function in action:

```
In [1]: change(48, [1, 5, 10, 25, 50])
Out[1]: 6
```

```
In [2]: change(48, [1, 7, 24, 42])
Out[2]: 2
```

```
In [3]: change(35, [1, 3, 16, 30, 50])
Out[3]: 3
```

```
In [4]: change(6, [4, 5, 9])
Out[4]: inf
```

In the last case, the function returns the special number "inf", meaning infinity, to indicate that change can't be made for that amount (because there is no 1-unit coin). See below for more on this case.

### A Few Notes and Tips...

Not surprisingly, the secret to all happiness is to use the *use-it-or-lose-it* recursion strategy.

Second, you may want to use the built-in function `min(x, y)`, which returns the smaller of its two arguments.

Third, in the event that `change` is confronted with a problem for which there is no solution, returning an infinite value is an appropriate way to indicate that there is no number of coins that would work. This happens, for example, when we are asked to make change for some positive amount of money but there are no coins in the list. What is infinity in Python? In class, we saw that one way to do this is this... First, import the `math` package:

```
from math import *
```

Now, you have access to the value `inf`. Alternatively, if you import the `math` package this way...

```
import math
```

... then you would access infinity with `math.inf`. Finally, if you don't import the `math` package at all, you can still get infinity in this slightly weird way using `float('inf')`.

### Giving Change

Just knowing the minimum number of coins is not as useful as getting the actual list of coins. Next, write another version of the `change` function called `giveChange`, which takes the same arguments as `change` but returns a list whose first member is the minimum number of coins and whose second member is a list of the coins in that optimal solution. Here's an example:

```
In [1]: giveChange(48, [1, 5, 10, 25, 50])
Out[1]: [6, [25, 10, 10, 1, 1]]
```

```
In [2]: giveChange(48, [1, 7, 24, 42])
Out[2]: [2, [24, 24]]
```

```
In [3]: giveChange(35, [1, 3, 16, 30, 50])
Out[3]: [3, [16, 16, 3]]
```

```
In [4]: giveChange(6, [4, 5, 9])
Out[4]: [inf, []]
```

The order in which the coin values are presented in the original list doesn't matter, and similarly, the order in which your solution reports the coins to use is also unimportant: In other words the solution `[3, [16, 16, 3]]` is the same to us as `[3, [3, 16, 16]]` or `[3, [16, 3, 16]]`. After all, all of these solutions use the same three coins!

Here are a few observations to keep in mind. First, while it may be tempting to have `giveChange` call your `change` function, this is actually not so helpful! Instead, write an all new function called `giveChange` that calls itself recursively but doesn't call any other function for help! Your `giveChange` function will be structured very similarly to your `change` function.

Not that `giveChange` will always return a list of the form `[numberOfCoins, listOfCoins]`. So, if your original `change` function returned 0, for example, then your new `giveChange` function would probably return `[0, []]` instead to indicate that there are zero coins of change and the list of coins is the empty list!

Now, use your `change` function as a guide to write your `giveChange` function, but keep in mind that everytime that `change` would have returned a number (a

number of coins), your new `giveChange` function will return a list of the form `[numberOfCoins, listOfCoins]`.

Website design by Madeleine Masser-Frye and Allen Wu

# Chapter 3: Functional Programming, Part Deux — cs5book 1 documentation

## Navigation

- [index](#)
- [next |](#)
- [previous |](#)
- [cs5book 1 documentation »](#)

## Chapter 3: Functional Programming, Part Deux

*Whoever said that pleasure wasn't functional?*

—Charles Eames

### 3.1 Cryptography and Prime Numbers



*I'm headed home soon, but there's just a bit more shopping to be done first!*

Imagine that our alien is sitting at a cafe, surfing the Internet, sipping a triple mochaccino, and is now about to purchase the Harry Potter Complete DVD Collection (movies 1 through 17) from the massive online store, Nile.com. As it types in its credit card number to make its purchase, it suddenly pauses to wonder how its financial details will be kept secure as they are transmitted over the cafe's Wi-Fi and then over the vast reaches of the Internet. That's a valid concern. The good news is that many online stores use cryptography to keep such transactions secure.



*Rivest, Shamir, and Adleman received the Turing Award—the computer science equivalent of the Nobel Prize.*

One of the most famous and widely-used cryptography schemes is called RSA, after the three computer scientists who invented it: Ron Rivest, Adi Shamir, and Leonard Adleman. Here's how it works: The online store has its own mathematical function that all customers use to encrypt their data before transmitting it over the network. This mathematical function is made publicly available. The hope is that while anyone can easily use this function to encrypt data, only the online store can "undo" (or, more technically, "invert") the function to decrypt the data and recover the original number.

For example, imagine that Nile.com tells customers to use the function  $\lfloor(f(x)=2x)\rfloor$ . It's certainly easy to encrypt any number we wish to send—we simply double it. Unfortunately, any first grader with a calculator can decrypt the message by simply dividing it by 2, so that encryption function is not secure.

The RSA scheme uses a slightly more complicated function. If  $\lfloor(x)\rfloor$  is our credit-card number, we encrypt it using the function  $\lfloor(f(x) = x^e \text{ mod } n)\rfloor$ , where  $\lfloor(e)\rfloor$  and  $\lfloor(n)\rfloor$  are carefully chosen numbers. (Remember from the previous chapter that  $\lfloor(x^e \text{ mod } n)\rfloor$  means the remainder when  $\lfloor(x^e)\rfloor$  is divided by  $\lfloor(n)\rfloor$ ; it can be easily computed in Python using the expression  $(x**e) % n$ .)

It turns out that if  $\langle e \rangle$  and  $\langle n \rangle$  are chosen appropriately, the online store will be able to decrypt the number to retrieve the credit card number  $\langle x \rangle$  but it will be nearly impossible for anyone else to do so—even though everyone knows  $\langle e \rangle$  and  $\langle n \rangle$ .

That's pretty interesting, but how do we choose  $\langle e \rangle$  and  $\langle n \rangle$  and how will the store later decrypt the number that it receives? Well, we first choose two different large prime numbers  $\langle p \rangle$  and  $\langle q \rangle$  at random. Next,  $\langle n \rangle$  is just  $\langle pq \rangle$ . Now, to get the number  $\langle e \rangle$ , we have to perform two steps: first we let  $\langle m = (p-1)(q-1) \rangle$ , and then we choose our exponent  $\langle e \rangle$  to be a random prime number less than  $\langle m \rangle$  that is also not a divisor of  $\langle m \rangle$ . That's it!

Now, any number  $\langle x \rangle$  less than  $\langle n \rangle$  can be encrypted by computing  $\langle x^e \text{ mod } n \rangle$ . Once we have selected  $\langle e \rangle$  and  $\langle n \rangle$ , we can share those values with anyone wishing to send us encrypted information. In a typical Internet shopping transaction, your web browser would get the publicly available values of  $\langle e \rangle$  and  $\langle n \rangle$  from the online store and use them to encrypt your credit card number,  $\langle x \rangle$ . Together, the values  $\langle e \rangle$  and  $\langle n \rangle$  are called the *public key* for this store. (In cryptology, a public key is simply a key that can be safely published without giving away the corresponding *secret key*.)

For example, let  $\langle p=3 \rangle$  and  $\langle q=5 \rangle$ . They're certainly prime (although they are way too small to be secure in practice). Now,  $\langle n=3 \times 5=15 \rangle$  and  $\langle m = (3-1) \times (5-1) = 8 \rangle$ . For our encryption exponent  $\langle e \rangle$ , we could choose the prime number 3 because it's less than 8 and also doesn't divide 8. Now, we can encrypt any number less than n. Let's encrypt the number 13, for example. We can compute  $\langle 13^3 \text{ mod } 15 \rangle$  in Python as  $(13**3) \% 15$ ; the result is 7. So 7 is our encrypted number, which we send over the Internet to the online store.



*I think the mathematics of cryptography should be called “discreet” math.*

How does the store decrypt that 7 and discover that the original number was actually 13? At the same time that the encryption exponent  $e$  was computed, we should have also computed a decryption exponent  $\langle d \rangle$ , which has two properties: it is between 1 and  $\langle m - 1 \rangle$ , and  $\langle ed \text{ mod } m = 1 \rangle$ . It's not hard to show that, because of the way  $\langle e \rangle$  and  $\langle m \rangle$  were chosen, there is exactly one value that has these properties; we call  $d$  the *multiplicative inverse of  $e$  modulo  $m$* . In our example,  $\langle e = 3 \rangle$  and  $\langle m = 8 \rangle$ , and  $\langle d \rangle$  is also 3 (it's a coincidence that  $\langle e \rangle$  and  $\langle d \rangle$  are equal; that's not normally the case—if it were, the decryption key wouldn't exactly be a secret!). Notice that  $\langle ed \text{ mod } 8 = 9 \text{ mod } 8 = 1 \rangle$ . Now, the online store can decrypt any number  $\langle y \rangle$  that it receives by simply computing  $\langle y^d \text{ mod } n \rangle$ . In our case, we received the encrypted number  $\langle y = 7 \rangle$ . We compute  $\langle 7^3 \text{ mod } 15 \rangle$  using Python  $((7**3) \% 15)$  and get the answer 13. Indeed, that's the value that we encrypted a moment ago! Keep in mind that while the encryption key  $\langle e \rangle$  and  $\langle n \rangle$  are public, the online store must keep the *decryption key*  $\langle d \rangle$  private. Anyone with access to  $\langle d \rangle$  can decrypt any message sent with the encryption key.

Exactly *why* this works is not too hard to show and is often taught in an introductory discrete math or algorithms course. But you may be wondering why we are so confident that the scheme is secure. This, too, requires a bit more time to explain than we have here. We will point out, however, that since the the values  $\langle e \rangle$  and  $\langle n \rangle$  are public, if a malicious person could find the two primes  $\langle p \rangle$  and  $\langle q \rangle$  that we originally selected, then they could figure out  $\langle m \rangle$  and then  $\langle d \rangle$  and they could crack the code. The good news is that “factoring” a number  $\langle n \rangle$  into its prime divisors is known to be a “computationally hard” problem—*very* hard. (*Computationally hard* means a problem that takes a long time to compute the answer for.) For example, the U.S. National Institutes of Standards and Technology estimates that if we encrypt a message today using public keys that are about 600 digits long, it would take until about the year 2030 to crack the code—even if a very large number of very fast computers were used for the attack.

## 3.2 First-Class Functions



*I prefer to do everything first-class.*

Recall that in the previous chapter we learned about Python functions and explored the power of recursion. The style of programming that we examined—programs constructed from functions that call one another (and possibly themselves)—is called *functional programming*. Interestingly, in functional programming languages like Python, functions are actually data just like numbers, lists, and strings. We say that functions are “first-class citizens” of the language. In particular, we can write functions that take *other functions* as arguments and return *other functions* as results! In this chapter we’ll explore these ideas, first using them to write a short program that efficiently generates long lists of primes, and ultimately writing a function that generates both the encryption and decryption functions for RSA cryptography. By the end of this chapter, we’ll have written Python programs that will allow you to securely send data to your friends.

### 3.3 Generating Primes

Motivated by RSA cryptography, our first mission is to find a way to generate a list of primes. One reasonable way to do this is to first write a function that determines whether or not its argument is prime. Once we have such a function, we could use it to test a sequence of consecutive numbers for primality, keeping those that are prime in a list.

But how do we test if a single positive integer  $\backslash(n\backslash)$  is prime? One idea is to simply test whether any number between 2 and  $\backslash(n - 1\backslash)$  divides it. If so, the number is not prime. Otherwise the number is prime. (In fact, it suffices to test just the numbers between 2 and  $\backslash(\sqrt{n}\backslash)$ , since if  $\backslash(n\backslash)$  is not prime, at least one of its divisors must be less than or equal to  $\backslash(\sqrt{n}\backslash)$ . But for now, let’s simply test all the possible divisors between 2 and  $\backslash(n-1\backslash)$ .)

To that end, it would be useful to have a function `divisors(n)` that accepts our number  $n$  (which we wish to test for primality) and returns `True` if  $n$  has any divisors (other than 1 and itself), and `False` otherwise. Actually, it will turn out to be handy if `divisors` accepts two additional numbers, `low` and `high`, that give a range of divisors to test. That will let us reduce the amount of work that has to be done.

For example, `divisors(15, 2, 14)` should return `True` because 15 has a divisor between 2 and 14 (and therefore is not prime), but `divisors(11, 2, 10)` should return `False` because 11 has no divisors between 2 and 10 (and therefore is prime). Also, note that `divisors(35, 2, 4)` should return `False` even though 35 is *not* prime.

We can write the `divisors(n, low, high)` function using recursion! To simplify matters, we’ll assume that the arguments are all positive integers. If `low` is higher than `high`, then the answer is `False` because  $n$  cannot have any divisors in the specified range (since there are no numbers in increasing order between `low` and `high`). So, if `low > high` we must return `False`—which is a base case.

But what if `low` is less than or equal to `high`? In this case, we can test whether  $n$  has a divisor between `low` and `high` like this: If  $n$  is divisible by `low`, then we’ve found a divisor in that range and we must return `True`. Otherwise, the answer to the question: “Does  $n$  have a divisor between `low` and `high`?” now becomes the same as the answer to the question “Does  $n$  have a divisor between `low+1` and `high`?” But that’s a version of the original question, and thus one that we can solve recursively! Here’s our solution:

```
def divisors (n, low, high):
 """Returns True if n has a divisor in the range from low to high.
 Otherwise returns False."""
 if low > high:
 return False
 elif n % low == 0: # Is n divisible by low?
 return True
 else:
 return divisors (n , low + 1, high)
```

Now we can test if  $n$  is prime by checking whether it has any divisors between 2 and  $n-1$ :

```
def isPrime (n):
 """For any n greater than or equal to 2,
 Returns True if n is prime. False if not."""
 if divisors (n, 2, n-1):
 return False
 else :
 return True
```

We can do this even more elegantly this way:

```
def isPrime (n):
```

```

'''For any n greater than or equal to 2,
Returns True if n is prime. False if not.'''
return not divisors (n, 2, n-1)

```

Recall from Chapter 2, that not “negates” a Boolean, so if `divisors(n, 2, n-1)` is `True` then `not divisors(n, 2, n-1)` is `False`, and if `divisors(n, 2, n-1)` is `False` then `not divisors(n, 2, n-1)` is `True`.

Now we can use `isPrime` to generate lists of primes, again using recursion. Imagine that we want to know all of the primes from 2 to some limit, for example 100. For each number in that range, we could test if it’s prime and, if so, add it to our growing list of primes. Before you look at the code below, see if you can determine the base case and the recursive step for such a function.

Now, here is the Python implementation:

```

def listPrimes (n, limit):
 '''Returns a list of prime numbers between n and limit.'''
 if n == limit:
 return []
 elif isPrime (n):
 return [n] + listPrimes (n+1, limit)
 else:
 return listPrimes (n+1, limit)

```

Notice that in the second return statement, we returned `[n] + listPrimes(n+1, limit)` rather than `n + listPrimes(n+1, limit)`. Why? Well, in this case the plus sign means that two lists should be concatenated, so the expressions on its left and right have to be lists. Indeed, the result of calling `listPrimes` will be a list, since by definition a list is what this function returns (and notice that in the base case it returns the empty list). However, `n` is a number, not a list! To make it a list, we place it inside square brackets. That way, we are concatenating two lists and life is good.

The above strategy for generating primes works, but it’s quite slow—particularly when attempting to generate large primes. The problem is that it repeats a lot of work. For example, if you call `listPrimes(51, 2, 50)` it will test 2 as a divisor and fail—but it will still insist on testing 4,6,8,...,50 even though we’ve already proven that 51 isn’t even!



*It's not entirely clear that Eratosthenes actually discovered this idea.*

A much faster algorithm for generating primes is the so-called *sieve of Eratosthenes*. This method is named after Eratosthenes, an ancient Greek mathematician who lived around 2200 years ago. Here’s the idea: To find all of the primes from 2 to 1000, for example, we first write down all of the integers in that range. Then we start with 2; it’s the first prime. Now, we remove (or “sift”) all multiples of 2 from this list since they are definitely not prime. When we’re done, we come *back* to the beginning of our remaining list. The number 3 survived the sifting of numbers that was performed by 2, so 3 is prime. We now cancel out all remaining numbers that are multiples of 3, since they too cannot be prime. When we’re done, we look at the first number in the remaining list. It’s not 4 (it got sifted out earlier when we looked at 2), but 5 is there. So 5 is prime and we let it sift all of its multiples that remain in the list. We continue this process until every remaining number has had a chance to sift the numbers above it.

|     | 2   | 3   | 4   | 5   | 6   | 7   | 8   | 9   | 10  | Prime numbers |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|---------------|
| 11  | 12  | 13  | 14  | 15  | 16  | 17  | 18  | 19  | 20  |               |
| 21  | 22  | 23  | 24  | 25  | 26  | 27  | 28  | 29  | 30  |               |
| 31  | 32  | 33  | 34  | 35  | 36  | 37  | 38  | 39  | 40  |               |
| 41  | 42  | 43  | 44  | 45  | 46  | 47  | 48  | 49  | 50  |               |
| 51  | 52  | 53  | 54  | 55  | 56  | 57  | 58  | 59  | 60  |               |
| 61  | 62  | 63  | 64  | 65  | 66  | 67  | 68  | 69  | 70  |               |
| 71  | 72  | 73  | 74  | 75  | 76  | 77  | 78  | 79  | 80  |               |
| 81  | 82  | 83  | 84  | 85  | 86  | 87  | 88  | 89  | 90  |               |
| 91  | 92  | 93  | 94  | 95  | 96  | 97  | 98  | 99  | 100 |               |
| 101 | 102 | 103 | 104 | 105 | 106 | 107 | 108 | 109 | 110 |               |
| 111 | 112 | 113 | 114 | 115 | 116 | 117 | 118 | 119 | 120 |               |

Figure 3.1: Sieve of Eratosthenes

([http://commons.wikimedia.org/wiki/File:Sieve\\_of\\_Eratosthenes\\_animation.gif](http://commons.wikimedia.org/wiki/File:Sieve_of_Eratosthenes_animation.gif))

We'll implement a recursive algorithm motivated by Eratosthenes' algorithm as a Python function called... `primeSieve`. (In the spirit of full disclosure, our implementation will take a few liberties with Eratosthenes' algorithm.) It will take a list of numbers 2,3,... up to the largest number that we're interested in, and will return a list of all the primes in the original list. Fortunately, Python has a built-in function that allows us to obtain the list of all integers from a starting point to an ending point. It's called `range`, and in Python 2 it works like this:

```
>>> range(0,5)
[0, 1, 2, 3, 4]
>>> range(3,7)
[3, 4, 5, 6]
```

In Python 3 `range` works almost the same, but it needs a little nudge to turn the result into a list:

```
>>> list(range(0,5))
[0, 1, 2, 3, 4]
>>> list(range(3,7))
[3, 4, 5, 6]
```

Notice that the list that we get back from `range` seems to stop one number too soon. That may seem weird, but as we'll see later, it turns out to be useful. So getting *all* the integers from 2 to 1000, for example, is easy: `range(2, 1001)`. We can then pass that list into the `primeSieve` function.

Before writing `primeSieve` let's just do a small thought experiment to better understand how it will work. Imagine that we start it with the list `range(2, 11)`, which is:

```
[2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Passing this list to `primeSieve` is basically saying "Could you please find me all of the primes in this list?" To accommodate your polite request, the `primeSieve` function should grab the 2 and hold on to it because it's a prime. It should then sift all of the multiples of 2 from this list, resulting in a new list:

[3, 5, 7, 9]

Now what? Well, primeSieve would like to do as little work as possible and instead send that list to some function and ask it, “Could you please find me all of the primes in *this* list?” Aha! We can send that list *back* to primeSieve because its job is to find the primes in a given list. That’s just recursion!

So, continuing with our example, the first time we called primeSieve it found the 2, sifted out the multiples of 2 to get the list [3, 5, 7, 9], and called primeSieve on that list. Whatever comes back from the recursive call will be tacked on to the 2 that we’re currently holding—and then we’ll have the whole list of primes from 2 to 10.

The recursive call to primeSieve with the argument list [3, 5, 7, 9] will similarly grab the 3 from the front of that list, sift out all of the multiples of 3 to get list [5, 7], and will ask for help finding the primes in that list. Whatever primes are returned will be tacked on to the 3 that we’re holding, and that will be all of the primes in the list [3, 5, 7, 9].



*I think that this approach should be called the “wishful thinking” method because we just wish for a helper function whenever we need one.*

In the next recursive call, we’ll grab 5 and recur on the list [7]. That recursive call will grab the 7 and recur on the empty list. We see here that since the list is getting shorter each time, the base case arises when we have an empty list. In that case, primeSieve must report that “the list of all the primes in my argument is empty”—that is, it should return the empty list.

We still need a function that will do the actual sifting. We can imagine a function called sift that takes two arguments—a number toRemove and a list of numbers numList—and returns all of the numbers in numList that are *not* multiples of toRemove. Let’s come back to that in a moment, and write our primeSieve under the assumption that we have sift. This approach for writing programs is called *top-down design* because we start at the “top” (what we want) and then work our way down to the details that we need.

```
def primeSieve(numberList):
 """Returns the list of all primes in numberList, using a prime sieve algorithm."""
 if numberList == []:
 # if the list is empty,
 return []
 # ...we're done
 else:
 prime = numberList[0] # The first element is prime!
 return [prime] + primeSieve(sift(prime, numberList[1:]))
```

Now, we need to sift. The next section will introduce a tool that will help us do exactly that, so read on...

### 3.4 Filtering

Fortunately for us, Python (like most functional programming languages) has a built-in function named filter that does (almost) exactly the sifting that we would like to do.

We’ll eventually get back to the problem of general list sifting that we started above, but for the moment let’s just focus on the problem of sifting a list by removing numbers divisible by two. To demonstrate filter in action, let’s first define a function called isNotDivisibleBy2 that takes a number n as an argument and returns a Boolean value: True if the number is not divisible by 2 (i.e., it is odd) and False otherwise (i.e. it is even).

```
def isNotDivisibleBy2(n):
 """Returns True if n is not divisible by 2,
 else returns False."""

 return n % 2 != 0
```

Now back to filtering. Having isNotDivisibleBy2, here’s how we can use it with Python’s filter function. In Python 2 it looks like this:

```
>>> filter(isNotDivisibleBy2, range(3, 10))
[3, 5, 7, 9]
```

In Python 3, filter doesn’t eagerly produce a list, so we need to send its result to the list function, as follows:

```
>>> list(filter(isNotDivisibleBy2, range(3, 10)))
[3, 5, 7, 9]
```



*I couldn't function without a filter to remove the noxious oxygen from Earth's air!*

You may have inferred what `filter` is doing: its first argument is a *function* and its second argument is a list. The function is a special one that takes a single argument and returns a Boolean result. A function that returns a Boolean is called a *predicate*; you can think of the predicate as telling us whether or not it “likes” its argument. Then, `filter` gives us back all of the elements in the list that the predicate likes. In our example, `isNotDivisibleBy2` is a predicate that likes odd numbers. So we got back the list of all the odd numbers in the original list.

A function that takes other functions as arguments?! Strange, but definitely allowed, and even encouraged! This idea is central to functional programming: functions can be passed into and returned from other functions just like any other kind of data. In fact, Chapter 4 will explain why this idea is not so strange after all.



*Four letr wrds rock!*

Lists of numbers are not all we can filter. Here’s another example of `filter`, this time using lists of strings. In preparation for its next visit to Earth, our alien is attempting to master English and a number of other languages. The alien has a list of words to learn, but it’s particularly keen on learning the four-letter words first. For example, if it’s given the list of words `['aardvark', 'darn', 'heck', 'spam', 'zyzzyva']` it would like to have a way of filtering that list to just be the “bad” words `['darn', 'heck', 'spam']`.

Again, we define a predicate function first: a function called `isBad` that takes a string named `word` as an argument and returns a Boolean value: `True` if the word is of length four and `False` otherwise.

```
def isBad(word):
 """Returns True if the length of "word" is 4, else returns False."""

 return len(word) == 4
```



\*In Python 2, we can omit the call to `list`, so we simply have `filter(isBad, ...)`

Now that we have `isBad`, here’s how we can use it with Python’s `filter` function:

```
>>> list(filter(isBad, ['ugh', 'darn', 'heck', 'spam', 'zyzzyva']))
['darn', 'heck', 'spam']
```

### 3.5 Lambda

`Filter` helps us sift lists of numbers, but so far all we’ve seen how to do is to sift them to remove even numbers. If we wanted to remove multiples of 3 we would need another helper function:

```
def isNotDivisibleBy3(n):
 """Returns True if n is not divisible by 3, else returns False."""

 return n % 3 != 0
```

And then we’d need another to remove multiples of 5, and one for 7, and 11, and so on. Obviously this is not going to work.

The idea behind our `primeSieve` function is that we want to change what we are filtering for “on the fly,” depending on which number is currently first in the list. You might imagine that we could add a second argument to our predicate function, as follows:

```
def isNotDivisibleBy(n, d):
 """Returns True if n is not divisible by d, else returns False."""
 return n % d != 0
```

Generally, this would solve the problem, but in this case we have a problem: `filter` requires that the predicate function we pass it takes only one argument. So this new definition will not work either.



*Is a disposable function environmentally friendly?!*

What we really need is a sort of “disposable” function that we can define just when we need it—using the number we currently care about sifting—and then immediately throw it away after we are done using it.

In fact, this function will be so temporary that we won’t even bother to give it a name! Such a function is called an anonymous function.

Here’s what an anonymous function definition looks like for our `isNotDivisibleBy2` example:

```
>>> filter(lambda n: n % 2 != 0, range(0, 1001))
```



“Lambda” derives from a branch of mathematics called the lambda calculus (see section 3.7), which influenced the first functional language, LISP.

The word “lambda” indicates that we are defining an anonymous function—a short-lived function that we need right here and nowhere else. Then comes the names of the arguments to the function (in this case, one argument named `n`), then a colon, and then the value that this function should return. So, this anonymous function `lambda n: n % 2 != 0` is exactly equivalent to our `isNotDivisibleBy2` function above.

The anonymous function syntax in Python is odd; in particular, we don’t put parentheses around the function’s arguments, and there is no `return` statement. Instead, anonymous functions in Python implicitly return whatever value comes after the colon.

Just to make the point that anonymous functions really are full-fledged functions, take a look at this:

```
(ch03_lambda)
```

Whoa—that’s really weird! In the first line, we’re defining a variable named `double` and giving it the value `lambda x: 2 * x`. But in a functional programming language functions are truly *first-class citizens*; they are data just like numbers, strings, and lists. So `double` is a function with one argument, and if we want to use it we need to pass it a value. In the second line, that’s exactly what we’re doing. In fact, when we define a function in the “normal” way, that is by starting with the line `def double(n)`, we are *really* just saying `double` is a variable whose value is the function that I’m going to define in the lines following the `def` statement.”



\*If using Python 2, we can omit the call to `list`.

Finally, we can use anonymous functions to finish writing `sift`, as follows:

```
def sift(toRemove, numList):
 """Takes a number, toRemove, and a list of numbers, numList.
 Returns the list of those numbers in numList that are not multiples of toRemove."""
 return list(filter(lambda n: n % toRemove != 0, numList))
```

```
return list(filter(lambda x: x % toRemove != 0, numList))
```

The anonymous function we pass into `filter` uses `toRemove` in its body without having to pass it in as an argument. It can do this because this function is defined in an environment where `toRemove` *already exists* and has a value.

Of course we could also write `sift` using the list-comprehension syntax (see the sidebar in section 3.6.2); then it would look like this:

```
def sift(toRemove, numList):
 return [x for x in numList if x % toRemove != 0]
```

Finally, as a reminder, here is our `primeSieve` function again, using `sift`:

```
def primeSieve(numberList):
 """Returns the list of all primes in numberList using a prime sieve algorithm."""
 if numberList == []:
 # if the list is empty,
 return []
 else:
 prime = numberList[0] # The first element is prime!
 return [prime] + primeSieve(sift(prime, numberList[1:]))
```

It's surprising how much entertainment value there is in running `primeSieve` to generate long lists of primes. However, Python got angry with us when we tried something like this:



\*If using Python 2, we don't need to call `list`.

```
>>> primeSieve(list(range(2, 10000)))
```

We got a long and unfriendly error message. The reason is that this created a lot of recursion, and Python is trained to believe that if there is a lot of recursion going on, there must be an error in your program (usually because you forgot to put in a base case). Exactly how much recursion Python thinks is too much depends on the Python version and your computer's operating system. However, you can ask Python to allow you more recursion by including the following two lines at the top of your file:

```
import sys
sys.setrecursionlimit(20000) # Allow 20000 levels of recursion
```

We asked for \((20,000)\) levels of recursion here. Some operating systems may allow you more or less than this. (Most modern machines will allow much, much more.)



*Shameless sales pitch!*

This brings us to one last point: while the prime sieve is quite efficient, most good implementations of the RSA scheme use even more efficient methods to generate large prime numbers. In practice, the primes used by RSA when encoding Internet transactions are generally a few hundred digits long! A CS course on algorithms or cryptography may well show you some of the more sophisticated and efficient algorithms for generating primes.

## 3.6 Putting Google on the Map!

In the rest of this chapter we will continue to build on the idea of functions as first-class citizens that can be passed into or returned from other functions. We'll look at two more examples: Google's MapReduce approach to processing data, and taking derivatives of functions.

Imagine that you work for Nile.com. Your company maintains a list of product prices, and periodically the prices are systematically increased. Your boss comes into your office one morning and asks you to write a function called `increment` that takes a list of numbers as an argument and returns a new list in which each number is replaced by a value one greater. So, with the argument `[10, 20, 30]`, the result should be `[11, 21, 31]`.

No problem! We can write increment recursively. For the base case, if the argument is an empty list, we'll also return an empty list. (Remember, the result is always a list of the same length as the argument, so returning anything other than the empty list in this case would be violating our contract!). For the recursive case, we observe that we can easily increment the first number in the list: We simply find that element and add one to it. We can then slice that first element off the list and increment the remainder. The phrase “increment the remainder” is essentially saying “use recursion on the remaining list.” In other words, `increment([10, 20, 30])` will first find the 10, add one to make it 11, and then recursively call `increment([20, 30])` to get the result [21, 31]. Once we have that, we just concatenate the 11 to the front of that list, resulting in [11, 21, 31]. Here's the Python program:

```
def incrementList(numberList):
 """Takes a list of numbers as an argument and returns
 a new list with each number incremented by one."""

 if numberList == []:
 return []
 else:
 newFirst = numberList[0] + 1 # increment 1st element
 # Next, increment the remaining list
 incrementedList = incrementList(numberList[1:])
 # Now return the new first element and the
 # incremented remaining list
 return [newFirst] + incrementedList
```

### 3.6.1 Map



*This boss is straight out of Dilbert!*

Your boss is pleased, but now tells you “I need a very similar function that takes a list as an argument and adds 2 to each element in that list.” Obviously, we could modify `increment` very slightly to do this. Then your boss tells you, “I need yet another function that takes a list as an argument and triples each element.” We can do that too, but this is getting old.

We see that your boss frequently needs us to write functions that take a list of numbers as an argument and return a new list in which *some function* is applied to every element in that list.



*This is known more succinctly as the principle of generalization. We're trying to make our function more general and less specific.*

An important principle in computer science is “if you are building lots of very similar things, try instead to build one thing that can do it all.”

We've seen this principle before when we started writing functions, but here we'll take it one step further. The “do it all” thing in this case is a function called `map`, which is built in to Python and many other functional programming languages. We'll show it to you and then explain. But first, it will be handy to have two simple functions to help with our example. One function, called `increment`, takes a number as an argument and returns that number plus 1. The second, called `triple`, takes a number as an argument and returns three times that number:

```
def increment(x):
 """Takes a number x as an argument and returns x + 1."""
 return x+1

def triple(x):
 """Takes a number x as an argument and returns 3 * x."""
```

```
return 3 * x
```

If you're using Python 2, you can now do the following:

```
>>> map(increment, [10, 20, 30])
[11, 21, 31]
>>> map(triple, [1, 2, 3, 4])
[3, 6, 9, 12]
```

If you're using Python 3, `map` is a bit lazy and requires some cajoling to produce the list, just like `filter` did. The magic incantation required in Python 3 looks like this:

```
>>> list(map(increment, [10, 20, 30]))
[11, 21, 31]
>>> list(map(triple, [1, 2, 3, 4]))
[3, 6, 9, 12]
```

This is passing the result of `map` to a built-in function called `list`, which forces Python 3 to actually produce the list.

Notice that `map` takes two arguments. The first is a function; the second is a list. The function that is given to `map` (e.g., `increment` and `triple` in our examples) **must** be a function that takes one argument and produces a single result. Then, `map` applies that function one-by-one to each element in the list to build a new list of elements.

Sometimes, we don't need to create our own functions to give to `map`. For example, take a look at this example:

```
>>> map(len, ['I', 'like', 'spam'])
[1, 4, 4]
```

In Python 3, this would be

```
>>> list(map(len, ['I', 'like', 'spam']))
[1, 4, 4]
```

In this case, the built-in function `len` is applied to each of the strings in the given list. First, `len` is applied to the string '`I`'. The length of the string is 1 and that's the first thing to go in the result list. Then the `len` function is applied to the string '`like`'; the result is 4, which is the next thing to go in the result. Finally, the length of '`spam`' is 4, which is the last value placed in the result list.

Notice how our abstraction theme comes into play with `map`. The details of how the function we pass to `map` is applied to the list are hidden away. We no longer have to worry specifically about how to step through the list to modify each element: `map` does this for us.

Note

**List Comprehensions** The `map`, `reduce`, and `filter` functions, and the `lambda` notation for anonymous functions, are not unique to Python; they are found in most functional programming languages. However, Python offers an alternative syntax to `map` and `filter` called *list comprehensions*, which allow us to avoid using `lambda`. Sometimes that's both conceptually easier and faster to type!

Let's start with the example from Section 3.6.1, where we wanted to increment every element in the list `[10, 20, 30]`. Whereas earlier we used `map` and an `increment` function to do this, the list comprehension syntax looks like this:

```
>>> [x + 1 for x in [10, 20, 30]]
[11, 21, 31]
```

Similarly, we can triple every element in the list `[1, 2, 3, 4]` using the syntax:

```
>>> [3 * x for x in [1, 2, 3, 4]]
[3, 6, 9, 12]
```

In general, the syntax is:

```
[f(x) for x in L]
```

...where `f` is some function and `L` is some list. Notice that this is really just like `map` in that we are mapping the function `f` to every element `x` in the list `L`. By the way, there's nothing special about the name `x` for the variable; we could use a different variable name as in:

```
>>> [len(myString) for myString in ['I', 'like', 'spam']]
[1, 4, 4]
```

A very similar syntax can be used to do the job of `filter`. Rather than using `filter` to obtain the four-letter words in a list as we saw earlier, we can use list comprehensions this way:

```
>>> words = ['aardvark', 'darn', 'heck', 'spam', 'zyzzyva']
>>> [x for x in words if len(x) == 4]
['darn', 'heck', 'spam']
```

In this case, we defined the list `words` before using the list comprehension, just to make the point that we can do that. Of course, we could have also done this as:

```
>>> [x for x in ['aardvark', 'darn', 'heck', 'spam', 'zyzzyva']
... if len(x) == 4]
```

Similarly, we could get the multiples of 42 between 0 and 1 million (another example where we used `filter`) this way:

```
>>> [x for x in range(0, 1000000) if x % 42 == 0]
```

In general, the format for using list comprehensions to filter looks like this:

```
[x for x in L if ...]
```

...where `L` is a list and the `...` represents some Boolean expression; that is, an expression that evaluates to either `True` or `False`. We get back the list of all values of `x` in the list `L` for which that Boolean expression (recall that it's called a predicate) is `True`. That is, we get all of the values of `x` that the predicate "likes." Finally, we can write list comprehensions that combine both `map` and `filter` into one. Here's an example where we produce the square of every even number between 0 and 10:

```
>>> [x**2 for x in range(0, 11) if x % 2 == 0]
[0, 4, 16, 36, 64, 100]
```

This list comprehension is mapping the function  $\lambda(f(x) = x^2)$  to every value of `x` in the list of integers from 0 to 10 subject to filtering that list through a predicate that only likes even numbers. Thus, we get the squares of even numbers. In general, the syntax is:

```
[f(x) for x in L if ...]
```

where `f` is a function, `L` is a list, and what comes after the `\(\dots\)` is a predicate. We get back the list of  $\lambda(f(x))$  values for those values of `x` for which the predicate is `True`.

By the way, this syntax works just the same in Python 2 and 3. Python 3 doesn't require any special prodding to produce a list when list comprehensions are used.

### 3.6.2 Reduce



Ten mochaccinos, a laptop, the Harry Potter DVD collection,...

Our alien has just learned about `map` and is quite excited. During its visit to Earth, the alien purchased a number of items and it plans to request reimbursement from its employer.

For example, here is a list of the costs, in U.S. dollars, of several items purchased by the alien: [14, 10, 12, 5]. The alien would like to convert these costs to its own currency and then add up the total. The exchange rate is 1 U.S. dollar equals 3 alien dollars. So, the values of the four items in alien dollars are [42, 30, 36, 15] and the total request for reimbursement in alien dollars will be  $\lambda(42 + 30 + 36 + 15 = 123)$ .

Converting a list from U.S. dollars to alien dollars is no problem—we did that above using `map` and our triple function. Now, though, we want to add up the elements in the resulting list. We could write a recursive function for that task. However, it turns out that there are many very similar tasks that the alien needs to perform as well.

Here's one: Imagine that the alien converted between a variety of currencies while shopping on Earth. For example, the alien first got some dollars, then changed dollars into Euros, then changed Euros into rubles, and then rubles into yen. If the alien started with 1 U.S. dollar, how many yen is that? This is a matter of computing the products of exchange rates. If one dollar is 0.7 Euros, one Euro is 42 rubles, and one ruble is 3 yen, then

one dollar is  $(0.7 \times 42 \times 3 = 88.2)$  yen. So now the alien needs to compute the *product* of a list of numbers.

Again, rather than writing a separate function to add up the numbers in a list and another to multiply the numbers in a list (and possibly others to do other things to elements in a list), Python provides a general-purpose function called `reduce` that reduces all of the elements in a list to a single element (e.g., their sum or product or something else).

Let's first define a function called `add` that takes *two* arguments and returns their sum, and another called `multiply` that takes *two* arguments and returns their product.

```
def add(x, y):
 """Returns the sum of the two arguments."""
 return x + y

def multiply(x, y):
 """Takes two numbers and returns their product."""
 return x * y
```

Now, here's `reduce` in action:

```
>>> from functools import *
>>> reduce(add, [1, 2, 3, 4])
10
>>> reduce(multiply, [1, 2, 3, 4])
24
```

In the first case `reduce` reduced our list to the sum of its elements, and in the second case to the product of its elements. Notice that the first argument to `reduce` is a function and the second is a list. A subtle point here is that the function that we give as an argument to `reduce` must take *two* arguments and return a single result. Then, `reduce` takes the first two elements from the list and applies the given function to reduce them to one element, takes that result and the next element from the list and applies the function to them, and repeats that process until all of the elements are reduced to a single value.



*While `reduce` is built in to Python 2, in Python 3 you'll need to include the line `from functools import *` before using `reduce`. You can include that line at the Python prompt or at the top of any file that uses `reduce`.*

### 3.6.3 Composition and MapReduce



*Remember that in Python 3 you'll need to include the line `from functools import *` since we're using `reduce` here.*

Finally, imagine that we plan to frequently take a list, map some function (e.g., `triple`) to each element in that list to produce a new list (e.g., the list of 3 times each element), and then apply some other function to reduce that new list to a single value (e.g., using `add` to get the sum of these values). This is a combination of `map` and `reduce`, or more accurately the *composition* of `map` and `reduce`. Let's write such a function and call it `mapReduce`. It will take two functions and a list as arguments—the first function for `map` and the second for `reduce`:

```
(ch03_mapreduce)
```

Alternatively, this can be done more succinctly in one line as follows:

```
def mapReduce(mapFunction, reduceFunction, myList):
 """Applies mapFunction to myList to construct a new list
 and then applies reduceFunction to the new list
 and returns that value."""
```

```
return reduce(reduceFunction, map(mapFunction, myList))
```

Now, for example, we can use `mapReduce` to convert prices from U.S. dollars to alien dollars and add up the total, like this:

```
>>> mapReduce(triple, add, [14, 10, 12, 5])
123
```

The second version of `mapReduce` above is particularly elegant in the way that `map` and `reduce` are composed or “glued” in one line. Notice that in order for Python to evaluate the expression `reduce(reduceFunction, map(mapFunction, myList))` it must first evaluate `map(mapFunction, myList)`, since that is one of the arguments to `reduce`. So, the `map` function is applied to `mapFunction` and list `myList` to get the resulting list. Now, `reduce` has both the function and the list that it needs to do its thing.

The `mapReduce` function is so broadly useful that it inspired a major part of Google’s internal software design! To read more about it, you can (of course) just Google “`mapReduce`.”

Note

### Functional Programming Meets Psychotherapy

The ideas behind functional programming actually predate the computer. A mathematical framework called the *lambda calculus* was developed by the logician Alonzo Church in the 1930’s to examine the limits of what is theoretically computable. (We address some of these issues in Chapter 7.) The lambda calculus served as the basis of actual programming languages, such as the early language LISP developed at MIT in the 1950’s. The “lambda” in lambda calculus is the reason for its appearance in Python’s anonymous functions.

LISP and related functional programming languages have been of particular interest to researchers in Artificial Intelligence (AI) because they are well suited for analyzing text. We can take a human’s typed input and write functions that process the words as strings and sentences as lists of strings. This is called *natural language processing*. (The “natural” here refers to human languages in contrast to programming languages.)

One of the early and controversial examples of natural language processing and artificial intelligence was the ELIZA program, first written by Professor Joseph Weizenbaum at MIT in the mid-1960’s. Eliza impersonated a psychologist and allowed the human user to enter text that ELIZA would process and return a simple response that—while not actually aware of the user’s emotions—seemed generally appropriate. Eliza was first written by Weizenbaum in a language called SLIP, but was later reprogrammed in LISP.

Here’s the beginning of an actual transcript of an interaction with ELIZA from an online version of the program (available [here](#)).

Eliza: Hello. I am ELIZA. How can I help you?

You: I am learning about functional programming.

Eliza: How long have you been learning about functional programming?

You: I guess it has been a few weeks now.

Eliza: Can you elaborate on that?

Responses such as “Can you elaborate on that?” or even “Tell me about your mother” are commonly used and quickly reveal that ELIZA is actually rather clueless. While Weizenbaum evidently only intended ELIZA to serve as a demonstration of the power of natural language processing, some people actually took an interest in using ELIZA as their therapist! A book and a documentary have been made about Weizenbaum and ELIZA, and a number of free and commercial versions of the program are available, including some that have a very bad attitude and deliberately berate their “patient.”

## 3.7: Functions as Results

If functions can be the arguments to other functions, it must also be possible for them to be the results of functions, right? Take a look at this `scale` function and try to figure out what it does:

The `scale` function takes the number 42 as an argument and returns a function (the `lambda` indicates that we’re giving back a function) that we are then assigning to the variable `f`. So `f` is a function, but what does that function do? Well, for any argument `x` it returns `42 * x`—as we see here:

```
>>> f(2)
84
>>> f(10)
```

There's something odd here! In the definition of the `scale` function, the line:

```
return lambda x: n * x
```

makes it look like the function that we're returning has *two variables*, `n` and `x`, whereas the `lambda x` indicates that this is a function of just *one* variable, `x`. Indeed, in the examples above, we defined `f = scale(42)` and then we gave `f` just *one* argument as in `f(2)` and `f(10)`.

When we said `f = scale(42)`, the `42` was passed into `scale`'s argument `n`, and `n` was replaced with `42`. Therefore, `scale(42)` actually returned



*Maybe I can write a program that will write all my future CS programming assignments for me!*

```
lambda x: 42 * x
```

That's clearly a function of just *one* variable and that is the function that we then assigned to our variable `f`.

Although the `scale` function is admittedly not super-useful, the amazing thing here is that we wrote a program that can write other programs—a powerful concept that is widely used by computer scientists. Indeed, many complex programs are, at least in part, written by other programs.

By the way, rather than using an anonymous function inside our `scale` function, we could have given the function a name and returned that function. We do this as follows:

```
def scale(n):
 def multiply(x): # Here we are defining a new function,
 return n * x # but it's INSIDE scale!
 return multiply # Here we are returning that function
```

Notice that the indented `def multiply(x)` indicates that `multiply` is being defined inside `scale`, so this is defining `multiply` to be one of `scale`'s own variables, just like defining any variable within a function. The only difference is that `multiply` is a *function* rather than a number, list, or string. Then, once we've defined that variable, we return it. Now, calling `scale(42)`, for example, gives us back a function and we can do exactly what we did a moment ago with the first version of `scale`:

```
>>> f = scale(42)
>>> f(2)
84
>>> f(10)
420
```

Functions that take other functions as arguments or return a function as a result are called *higher-order functions*.

### 3.7.1: Python Does Calculus!



*I'd rather impersonate them and alienate them!*

In addition to learning English and other languages, our alien has been advised to study calculus before returning to Earth, so that it can better impersonate a college student.

You probably recall the idea of taking the derivative of a function. For example, the derivative of  $f(x) = x^2$  is a new function  $f'(x) = 2x$ . For our purposes, the key observation is that the derivative of a function is another function. You may also recall that there are many rules for computing derivatives, and yet some functions are very hard to differentiate and others are downright impossible. So in some cases we may want to

approximate the derivative—and we can do this computationally! In fact, let's write a Python function that differentiates *any* function (approximately).

We looked back at our own calculus notes and found that the derivative of a function  $\langle(f(x))\rangle$  is a function  $\langle(f'(x))\rangle$  defined as follows:

$$\langle(\lim_{h \rightarrow 0} \frac{f(x+h) - f(x)}{h})\rangle$$

The derivative is defined as “the limit as  $\langle(h)\rangle$  goes to zero.” However, for small positive values of  $\langle(h)\rangle$ , like  $\langle(h = 0.0001)\rangle$ , we get a good approximation of the limit. We can make the approximation as good as we want by making  $\langle(h)\rangle$  as small as we want. Our objective, then, is to write a function called `derivative` that takes a function and a value of  $\langle(h)\rangle$  as an argument, and returns another function that is the approximation of the derivative for that value of  $\langle(h)\rangle$ . Here we go:

```
def derivative(f, h):
 """Returns a new function that is the approximation of
 the derivative of f with respect to h."""

 return lambda x: (f(x+h) - f(x)) / h
```

Notice that we're returning a function, as we're supposed to. That function takes an argument  $x$  and returns the approximate value of the derivative at that value  $x$  for the given  $h$ .

Wait a second! It seems that the function that we're returning is using a function `f` and two numeric variables,  $x$  and  $h$ —so why do we see just the variable  $x$  after the `lambda`? This is the same issue that we saw earlier with the `scale` function. Remember that when we call the `derivative` function, we pass it both a function `f` and a number `h`. Then, that function and that number are “plugged in” wherever we see a `f` and a `h` in the program. In the expression `lambda x: (f(x+h) - f(x)) / h`, both the `f` and the `h` are no longer variables—they are actual values (albeit the `f` is a value that happens to be a function). So, the `lambda` anonymous function truly only has one argument, and that's  $x$ .

Let's try it out. First, let's define a function  $\langle(x^2)\rangle$  and name it `square`.

```
def square(x):
 return x**2
```

Now, let's use our `derivative` function:

```
>>> g = derivative(square, 0.0001)
>>> g(10)
20.00009999890608
```



*The second derivative is “accelerated” material!*

The actual derivative of  $\langle(x^2)\rangle$  is  $\langle(2x)\rangle$ , so the derivative at  $\langle(x=10)\rangle$  is actually  $\langle(20)\rangle$ . We're getting a pretty good approximation. That's good, but here's something better: if we now want to find the second derivative of  $\langle(x^2)\rangle$ , we can differentiate our first derivative. The actual second derivative of  $\langle(x^2)\rangle$  is simply  $\langle(2)\rangle$  (for all values of  $\langle(x)\rangle$ ). Let's see what our `derivative` function says.

```
>>> h = derivative(g, 0.0001)
>>> h(10)
2.0000015865662135
```

### 3.7.2: Higher Derivatives

Speaking of second derivatives, it sure would be handy to have a more general function that could find us the first, second, third, or generally, the  $\langle(k^{\text{th}})\rangle$  derivative of a function. Let's write a function called `kthDerivative(f, h, k)` to do this. It will take arguments that include a function `f`, a small number `h`, and a positive integer `k` and will return an approximation of the  $\langle(k^{\text{th}})\rangle$  derivative of  $\langle(f)\rangle$ . We could do this “from scratch”, not relying on the existing `derivative` function that we just wrote, but since we have it already let's use `derivative`.

What's the base case? The simplest case appears to be when  $\langle(k=1)\rangle$ , in which case we just return the derivative of the given function. (Although it would be equally defensible to have a base case at  $\langle(k=0)\rangle$ , in which case we

would just return  $\lfloor f \rfloor$  itself.) Otherwise,  $\lfloor k > 1 \rfloor$  and we need to find the recursive substructure. Well, the  $\lfloor k \cdot \{th\} \rfloor$  derivative of  $\lfloor f \rfloor$  is, by definition, just the derivative of the  $\lfloor ((k-1) \cdot \{st\}) \rfloor$  derivative of  $\lfloor f \rfloor$ . So we can ask the `kthDerivative` function to find the  $\lfloor ((k-1) \cdot \{st\}) \rfloor$  derivative of  $f$  and then apply our derivative function to it. This should be fun and easy.



*A function of degree 4 is “quartic”, one of degree 9 is “nonic”. Believe it or not, one of degree 100 is called “hectic.”*

```
def kthDerivative(f, h, k):
 """Returns a new function that is the approximation of
 the kth derivative of f with respect to h."""
 if k == 1:
 return derivative(f, h)
 else:
 return derivative(kthDerivative(f, h, k-1), h)
```

Let's take this out for a spin by defining the “quartic” function  $\lfloor (x^4) \rfloor$  and then taking the third derivative, which we know to be  $\lfloor (24x) \rfloor$ .

```
def quartic(x):
 """Returns x**4."""
 return x**4
```



*Please! Let's not go any further down the slippery slope of bad math puns.*

```
>>> g = kthDerivative(quartic, 0.0001, 3)
>>> g(10)
241.9255906715989
```

The actual answer should have been 240; we got an approximation due to the value we used for  $\lfloor (h) \rfloor$ .

That was a high-velocity discussion of derivatives, but it demonstrated some of the integral features of functional programming.

### 3.8: RSA Cryptography Revisited

We began this chapter with a discussion of RSA cryptography. Recall that if Alice wants to be able to receive secure messages, then she can use the RSA algorithm to generate a public and a private key. The public key is entirely public—anyone can use it to encrypt a message to Alice. The private key, however, belongs exclusively to her; she uses it to decrypt messages encrypted using her public key. Of course, if Alice's friend, Bob, wants to receive encrypted messages too, he can use RSA to generate his own public and private keys. He then shares his public key and keeps his private key secure. When you want to send an encrypted message to Alice, you use her public key to encrypt the message. When you want to send a message to Bob, you use his public key.

Our objective is to write a function called `makeEncoderDecoder()`, which takes no arguments, constructs the RSA encryption and decryption keys, and returns *two* functions: The first encrypts data using the encryption key (which is built into the function, so we don't need to keep track of it—convenient when a key is hundreds of digits long!) The second function decrypts encrypted data using the decryption key (again built-in). Alice, Bob, you, and your friends will all be able to use the `makeEncoderDecoder()` function to construct encryption and decryption functions; each encryption function will be made public and each decryption function will be kept by its owner. Here's an example of Alice using `makeEncoderDecoder`.

```
>>> AliceEncrypt, AliceDecrypt = makeEncoderDecoder()
Maximum number that can be encrypted is 34
>>> AliceEncrypt(5)
```

```

10
>>> AliceDecrypt(10)
5
>>> AliceEncrypt(31)
26
>>> AliceDecrypt(26)
31

```

Notice here that `makeEncoderDecoder` returned two functions (we'll see in a moment how we can return two things), which we've named `AliceEncrypt` and `AliceDecrypt`. Then, we tested those two functions by encrypting them using Alice's encryption function (which contains the encryption key inside of it) and then decrypting them with Alice's decryption function (which incorporates the decryption key).

Let's first remind ourselves how the RSA scheme works. We begin by choosing two different random prime numbers  $(p)$  and  $(q)$  (preferably large, but we'll be able to decide how large later). We compute  $(n = pq)$  and  $(m = (p-1)(q-1))$ . We then choose the public encryption key  $(e)$  to be a random prime number between 2 and  $(m-1)$  such that  $(e)$  does not divide  $(m)$ . Next, we construct the decryption key  $(d)$  to be the multiplicative inverse of  $(e)$  modulo  $(m)$ —that is, the unique number  $(d)$  such that  $(d \leq m-1)$  and  $(ed \bmod m == 1)$ . Now, we can encrypt a number  $(x)$  between 1 and  $(n-1)$  by computing  $(y = x^e \bmod n)$ . The value  $(y)$  is the encrypted message. We can decrypt  $(y)$  by computing  $(y^d \bmod n)$ .

We've already written functions that produce lists of prime numbers. We'll use our efficient `primeSieve` function for that purpose. Recall that this function takes a list of consecutive integers from 2 to some largest value and returns the list of all primes in that range. Our first task will be to call the `primeSieve` function and choose two different prime numbers from that list. For now, let's restrict the range of the prime numbers to be between 2 and 10. This will be useful for testing purposes, and later you can change the maximum value from 10 to something much larger.

To choose items at random, we can use Python's `random` package. We tell Python that we want to use that package by including the line `import random` at the top of the file. Now, we have access to a variety of functions in that package. You can find out what's in the package by looking at the Python documentation Web site or by typing `import random` at the Python prompt and then typing `help(random)`, which will describe all of the functions in that package.

One of the most useful functions in the `random` package is `random.choice`. It takes a list as an argument and returns a randomly selected element of that list. Here's an example of that function in action:

```

>>> import random
>>> random.choice([1, 2, 3, 4])
3
>>> random.choice([1, 2, 3, 4])
2

```

We need to choose two different prime numbers in the range from 2 to 10. We could use `random.choice` twice, one to choose each of the two prime numbers, but there is a chance that we'll get the same prime number twice, which doesn't work for RSA. So, we can use another function in the `random` package, `random.sample`. It takes two arguments: a list and the number of items that we want to choose randomly—but uniquely—from that list. The function returns a list of the randomly selected items. Here's an example of `random.sample` in action:

```

>>> import random
>>> random.sample([1, 2, 3, 4], 2) # Pick 2 items at random
[4, 3]
>>> random.sample(range(2, 100), 3) # Pick 3 different items
[17, 42, 23]

```

When a function returns a list of several items and we know how many items will be in that list, we can give names to those items like this:

```

>>> import random
>>> a, b = random.sample([1, 2, 3, 4], 2)
>>> a
4
>>> b
3

```

In this case, we knew that the `random.sample` function was going to return a list of two items (since we asked it to!) and we assigned the first of those items to the variable `\(a\)` and the second to `\(b\)`. This technique is called *multiple assignment*.

Now let's assume that we've already written a function called `inverse(e, m)` that returns the multiplicative inverse of `\(e\)` modulo `\(m\)`—that is, the unique number `\(d < m\)` such that `\(ed \mod m = 1\)`. We'll write that function in a moment, but assuming it exists, we have the ingredients for the `makeEncoderDecoder` function:

```
def makeEncoderDecoder():
 """Returns two functions: An RSA encryption function
 and an RSA decryption function."""
 #
 # Choose 2 primes:
 #
 p, q = random.sample(primeSieve(list(range(2, 10))), 2)
 n = p*q # compute n
 m = (p-1)*(q-1) # compute m
 print ("Maximum number that can be encrypted is ", n-1)

 #
 # Choose a random prime for e:
 #
 e = random.choice(primeSieve(list(range(2, m))))
 if m % e == 0: # If e divides m, it won't work!
 print ("Please try again")
 return
 else:
 d = inverse(e, m) # compute d
 encoder = lambda x: (x**e) % n # encryption function
 decoder = lambda y: (y**d) % n # decryption function
 return [encoder, decoder]
```



Notice that this function is returning a list of two functions: the encryption and decryption functions! The encryption and decryption keys `\(e\)` and `\(d\)` are actual numbers that are embedded in those two functions, but `\(x\)` and `\(y\)` are the names of variables—the arguments that the user will ask to have encrypted or decrypted.

Finally, we need the `inverse(e, m)` function. We can implement it very simply by using `filter`, which we saw in Section 3.4. We're looking for the single number `\(d\)` such that `\(d < m\)` and `\(ed \mod m = 1\)`. So we can generate all of the integers between 1 and `\(m-1\)` (which is done with `range(1, m)`) and then filter all of the values `\(d\)` such that `\(ed \mod m == 1\)`. Mathematicians tell us that there will be exactly one value of those. The `filter` function returns a list, so we use a sneaky Python trick and treat the call as if it were itself a list: we put `[0]` after it to pull out the first element and return it.

```
from functools import *
def inverse(e, m):
 """Returns the inverse of e mod m"""
 return filter(lambda d: e*d % m == 1, range(1, m))[0]
```

You have to admit that this is amazing! However, we must admit that there are a few places where this function could be improved and extended. First, the prime numbers that we're choosing from for `\(p\)` and `\(q\)` are in the range from 2 to 10. We could easily change the 10 to something much larger, but we almost certainly don't want primes as small as 2. On the other hand, the `primeSieve` function expects to get a list of numbers that begins with 2 (do you see why?). To limit it to large prime numbers, we'd first need to generate primes beginning from 2 and then slice off the small ones. In addition, currently the `makeEncoderDecoder` might choose an encryption key `\(e\)` that is a divisor of `\(m\)`. In this case, we're told to try running the function again. That's not a very nice solution. You might try to think of ways to fix this so that `makeEncoderDecoder` would keep choosing values of `\(e\)` until it finds one that is not a divisor of `\(m\)`. You can do this with recursion, or you can do it with something called a `while` loop which we'll see in Chapter 5.

You might also rightfully object that encrypting numbers is not as interesting as encrypting strings (i.e., text). We'd agree. The good news is that strings can be converted into numbers and vice versa, as we'll see in the next chapter. Thus, it would not be difficult to encrypt strings by first converting them to numbers and then encrypting those numbers as we've done here. Decryption would first decrypt the number and then convert that back into the original string. In fact, that's *exactly* how real encryption programs work!

### 3.9: Conclusion



*I've been wondering what's going on inside my laptop.*

In this chapter we've seen a beautiful idea: Functions are “first-class citizens” of the language, allowing us to treat them just like any other kind of data. In particular, functions can be the arguments and results of other functions. As a result, it's possible to have general-purpose higher-order functions like `map`, `reduce`, and `filter` that can be used in many different ways by simply providing them with appropriate functions of their own. Moreover, we can write other higher-order functions that produce functions as results, allowing us to easily write programs that write other programs.

Now that we have explored the foundations of programming, we will turn our attention to a new question: “How exactly does a computer actually manage to do all of this amazing stuff?” To answer that question, in the next chapter we're going to open the hood of a computer and see just how it works.

#### Table Of Contents

- Chapter 3: Functional Programming, Part Deux
  - 3.1 Cryptography and Prime Numbers
  - 3.2 First-Class Functions
  - 3.3 Generating Primes
  - 3.4 Filtering
  - 3.5 Lambda
  - 3.6 Putting Google on the Map!
    - \* 3.6.1 Map
    - \* 3.6.2 Reduce
    - \* 3.6.3 Composition and MapReduce
  - 3.7: Functions as Results
    - \* 3.7.1: Python Does Calculus!
    - \* 3.7.2: Higher Derivatives
  - 3.8: RSA Cryptography Revisited
  - 3.9: Conclusion

**Previous topic** Chapter 2 : Functional Programming

**Next topic** Chapter 4: Computer Organization

#### Quick search

Enter search terms or a module, class or function name.

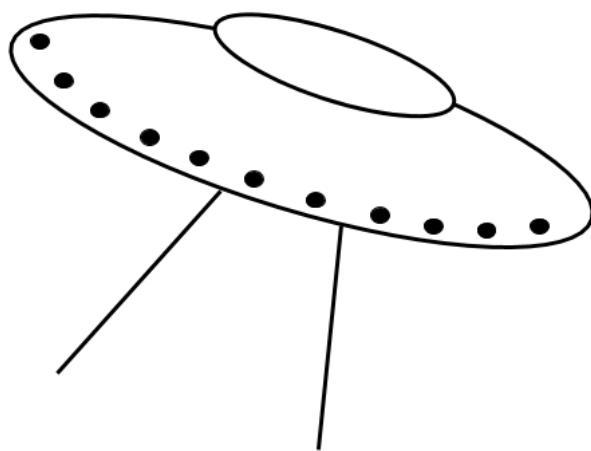
#### Navigation

- index
- next |
- previous |
- cs5book 1 documentation »

© Copyright 2013, hmc. Created using Sphinx 1.2b1.

# CSforAll - CaesarSorting

## CS for All



CSforAll Web > Chapter3 > CaesarSorting

This website accompanies the textbook *CS For All* by Alvarado, Dodds, Kuenning, and Libeskind-Hadas

### Sorting out Caesar!

This problem asks you to write several functions using *functional programming*, i.e., conditionals, recursion, and/or list comprehensions.

For each one, be sure to

- Name the function as specified ***including capitalization***—this helps us test them smoothly
- Include a docstring that briefly explains the function's arguments and what it does

#### Function to write #1: `encipher(S, n)`

Write the function `encipher(S, n)` whose first argument `S` is a string `S` and whose second argument `n` is a non-negative integer between 0 and 25. Then, `encipher` should return a new string in which the letters in `S` have been “rotated” by `n` characters forward in the alphabet, wrapping around as needed.

For this problem, you should assume that upper-case letters are "rotated" to upper-case letters, lower-case letters are "rotated" to lower-case letters, and all non-alphabetic characters are left *unchanged*. For example, if we were to shift the letter 'y' by 3, we would get 'b' and if we were to shift the letter 'Y' by 3 we would get 'B'. (In python, you can use the test if 'a' <= c <= 'z': to determine if a character c is between 'a' and 'z' in the alphabet.)

You can write encipher any way you like as long as you use functional programming—that is, feel free to use any combination of conditionals, recursion, and list comprehensions.

You might use the class suggestion of writing a helper function that "rotates" a single character by n spots, wrapping around the alphabet as appropriate. In lecture we looked at how that might work. Then, you could use this helper function to encipher your string. It's up to you how you do this!

That said, for rotating, keep in mind that the built-in functions `ord` and `chr` convert from single-character strings to integers and back:

- For example, `ord('a')` returns 97
- ...and `chr(97)` returns 'a'.

Remember that

- Uppercase letters wrap around the alphabet to uppercase letters.
- Lowercase letters wrap always to lowercase letters.
- Non-letters do not wrap or change at all!

Hint

- Write a function `rot(c, n)` that rotates c, a single character, forward by n spots in the alphabet.
- We wrote `rot13(c)` in class—it's very close to `rot(c, n)`!
- Remember that you'll need to wrap the alphabet (as `rot13` did) and *leave non-alphabetic characters unchanged*
- Test out your `rot(c, n)` function to make sure it works:

```
rot('a', 2) --> 'c'
rot('y', 2) --> 'a'
rot('A', 3) --> 'D'
rot('Y', 3) --> 'B'
rot(' ', 4) --> ''
```

Hint

If you have `rot(c, n)`, you're nearly there!

- With `rot(c, n)`, this problem is identical to the `dna_to_rna` (transcribe) problem!
- That is, you can handle one letter at a time—recursively—using `rot(c, n)`, with a base case (an empty-string `s`)...
- Alternatively, you can use a list comprehension to apply `rot(c, n)` many times.
  - `['H', 'e', 'l', 'p', '!']`
- If you do use a list comprehension, then use `list_to_str` (here) to get back to a string!

```
def list_to_str(L):
 """L must be a list of characters;
 this function returns a single string made from them.
 """
 if len(L) == 0:
 return ""
 return L[0] + list_to_str(L[1:])

assert list_to_str(['c', 's', '5', '!']) == 'cs5!'
```

However, you implement `encipher`, be sure to test! Here's a start:

```
assert encipher('xyza', 1) == 'yzab'
assert encipher('Z A', 1) == 'A B'
```

Here are some more examples; we encourage you to turn them into `asserts`:

```
In [1]: encipher('xyza', 1)
Out[1]: 'yzab'
```

```
In [2]: encipher('Z A', 1)
Out[2]: 'A B'
```

```
In [3]: encipher('*ab?', 1)
Out[3]: '*bc?'
```

```
In [4]: encipher('This is a string!', 1)
Out[4]: 'Uijt jt b tusjoh!'
```

```
In [5]: encipher('Caesar cipher? I prefer Caesar salad.', 25)
Out[5]: 'Bzdrq bhogdq? H oqdedq Bzdrq rzkzc.'
```

### Function to write #2: `decipher(S)`

On the other hand, `decipher(S)` will be given a string of English text already shifted by some amount. Then, `decipher` should return, to the best of its ability, the *original* English string, which will be some rotation (possibly 0) of the argument `S`.

**Note:** some strings have more than one English "deciphering." What's more, it is difficult or impossible to handle very short strings correctly. Thus, your `decipher` function *does not have to be perfect*. However, it should work almost all of the time on long stretches of English text, e.g., sentences of 8+ words. On a single word or short phrase, you will not lose any credit for not getting the correct deciphering!

#### Hint

- A good place to start is to create a line ***with every possible ENCODING***, something like this:  
`L = [ _____ for n in range(26) ]`
- Then, you will want to use the `LoL` "list of lists" technique in which each element of `L` gets a score. You might want to look back at how that worked...  
`LoL = [ _____ for x in L ]`
- It's entirely up to you how you might want to score "Englishness." See below for some starting points... .
- To be specific, take a look at the `bestWord` example that found the word with the greatest scrabble-score in a list of words. That's not so far from what you want here!
- Then, go back and take a look at the min/max lecture to see how to handle the `LoL` "list of lists"

One approach you could try is to use letter frequencies—a function providing those frequencies is provided below. Feel free to cut-and-paste it into your file. Scrabble scores have also been suggested in the past! You're welcome to use some additional "heuristics" (rules of thumb) of your own design. Also, you are welcome to write one or more small "helper" functions that will assist in writing `decipher`.

However you approach it, **be sure** to describe whatever strategies you used in writing your decipher function in a short comment above your decipher function.

Some decipher examples:

```
In [1]: decipher('Bzdrzq bhogdq? H oqdedq Bzdrzq rzkzc.')
Out[1]: 'Caesar cipher? I prefer Caesar salad.'
```

```
In [2]: decipher('Hu lkjhapvu pz doha ylthpuz hmaly dl mvynla \
'Iclyfaopun dl ohcl slhyulk.')
Out[2]: 'An education is what remains after we forget everything we have learned.'
```

```
In [3]: decipher('Onyx balks')
Out[3]: 'Edon rqbai'
mine is wrong! This is OK here...
```

Again, we encourage you to turn these examples (at least the first two!) into asserts.

Note that the last example shows that our decipherer gets some short phrases wrong—***this is completely OK!*** As phrases get longer, your decipherer should get more and more of them correct, but it does not have to get single words or short phrases—after all, for short strings, there are likely to be rotations that have more “English-y” letters than the original!

Here is a letter-probability function and its source:

```
table of probabilities for each letter...
def letProb(c):
 """If c is the space character or an alphabetic character,
 we return its monogram probability (for english),
 otherwise we return 1.0. We ignore capitalization.
 Adapted from
 http://www.cs.chalmers.se/Cs/Grundutb/Kurser/krypto/en_stat.html
 """
 if c == ' ': return 0.1904
 if c == 'e' or c == 'E': return 0.1017
 if c == 't' or c == 'T': return 0.0737
 if c == 'a' or c == 'A': return 0.0661
 if c == 'o' or c == 'O': return 0.0610
 if c == 'i' or c == 'I': return 0.0562
 if c == 'n' or c == 'N': return 0.0557
 if c == 'h' or c == 'H': return 0.0542
 if c == 's' or c == 'S': return 0.0508
 if c == 'r' or c == 'R': return 0.0458
 if c == 'd' or c == 'D': return 0.0369
 if c == 'l' or c == 'L': return 0.0325
 if c == 'u' or c == 'U': return 0.0228
 if c == 'm' or c == 'M': return 0.0205
 if c == 'c' or c == 'C': return 0.0192
 if c == 'w' or c == 'W': return 0.0190
 if c == 'f' or c == 'F': return 0.0175
 if c == 'y' or c == 'Y': return 0.0165
 if c == 'g' or c == 'G': return 0.0161
 if c == 'p' or c == 'P': return 0.0131
 if c == 'b' or c == 'B': return 0.0115
 if c == 'v' or c == 'V': return 0.0088
 if c == 'k' or c == 'K': return 0.0066
```

```

if c == 'x' or c == 'X': return 0.0014
if c == 'j' or c == 'J': return 0.0008
if c == 'q' or c == 'Q': return 0.0008
if c == 'z' or c == 'Z': return 0.0005
return 1.0

```

### Hint

For decipher you might have your program "look at" all 26 possible rotations of the argument string and then decide on which is "best"...

### Function to write #3: `bisort(L)` - Binary-List Sorting

Design and write a function named `bisort(L)`, which will accept a list `L` and should return a list with the same elements as `L`, but in ascending order. (Note: the second character is an "ell" for "list", not a 1 or an "i"!) `bisort` \*ONLY NEEDS TO HANDLE LISTS OF BINARY DIGITS\*, that is, this function can and should assume that `L` will always be a list containing only 0s and 1s.

You may not call Python's built-in `sort` function to solve this problem! Also, you should not use your own `sort` (asked in a question below), but you may use any other technique to implement `bisort`. In particular, you might want to think about how to take advantage of the constraint that the argument will be a binary list—this is a considerable restriction!

One function that some have found helpful is `count(e, L)`, one of the helper functions we used in an earlier class. Grab it from there (or try rewriting it, perhaps...here's the crucial piece: `LC = [1 for x in L if x == e] !`)

You would need to include `count(e, L)` in your file, and then you could use it to return the number of times that `e` appears in `L`... .

Here are some examples:

```
In [1]: bisort([1, 0, 1])
Out[1]: [0, 1, 1]
```

```
In [2]: L = [1, 0, 1, 0, 1, 0, 1]
```

```
In [3]: bisort(L)
Out[3]: [0, 0, 0, 1, 1, 1, 1]
```

In the end, this problem is much *easier* than ordinary sorting!

### Function to write #4: `gensort(L)` - General-Purpose Sorting

Use recursion to write a general-purpose sorting function `gensort(L)`, which accepts a list `L` and returns a list with the same elements as `L`, but in ascending order. Feel free to use the `max` function built into Python (or `min` if you prefer) and the `remOne` function we discussed in class. Recursion—that is, sorting the *rest* of the list—will help, too.

Here are some examples:

```
In [1]: gensort([42, 1, 3.14])
Out[1]: [1, 3.14, 42]
```

```
In [2]: L = [7, 9, 4, 3, 0, 5, 2, 6, 1, 8]
```

```
In [3]: gensort(L)
```

```
Out[3]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

For this problem, you should **not** use any of Python's built-in implementations of sorting—for example, `sorted(L)` or `L.sort()`. Rather, you're designing and implementing your own approach from scratch!

Note that `gensort(L)` should work for *lists* `L`. It does **not** have to work for string arguments.

### Function to write #5: `jscore(S, T)`— Jotto Scoring

Write a function named `jscore(S, T)`, which will accept two strings, `S` and `T`. Then, `jscore` returns the "jotto score" of `S` compared with `T`.

This jotto score is the number of characters in `S` that are shared by `T`. Repeated letters are counted multiple times, as long as they appear multiple times in both strings. The examples below will make this clear. Note that, in contrast to the traditional game of 5-letter jotto, we are not constraining the lengths of the argument strings here!

There are several ways to accomplish this, many of which use small helper functions—feel free to add any such helper functions you might like. (We may have written what you need in class...)

Note that if either `S` or `T` is the empty string, the jotto score should be zero!

Hint

This line turns out to be a useful test: `if S[0] in T:`

Some examples:

```
In [1]: jscore('diner', 'syrup')
just the 'r'
Out[1]: 1
```

```
In [2]: jscore('geese', 'elate')
two 'e's are shared
Out[2]: 2
```

```
In [3]: jscore('gattaca', 'aggccaggcgc') # 2 'a's, 1 't', 1 'c', 1 'g'
Out[3]: 5
```

```
In [4]: jscore('gattaca', "") # if empty, return 0
Out[4]: 0
```

### Function to write #6: `exact_change(target_amount, L)`

**Making change!** Use recursion to write a Python function `exact_change` with the following signature:

```
def exact_change(target_amount, L):
```

where the argument `target_amount` is a single non-negative integer and the argument `L` is a list of positive integers. Then, `exact_change` should return either `True` or `False`: it should return `True` if it's possible to create `target_amount` by adding up some—or all—of the values in `L`. It should return `False` if it's **not** possible to create `target_amount` by adding up some or all of the values in `L`.

For example, `L` could represent the coins you have in your pocket and `target_amount` could represent the price of an item—in this case, `exact_change` would tell you whether or not you can pay for the item *exactly*.

Here are a few examples of `exact_change` in action. Notice that you can *always* make change for the target value of 0, and you can *never* make change for a negative target value: these are two, but not all, of the base cases!

```
In [1]: exact_change(42, [25, 1, 25, 10, 5, 1])
Out[1]: True
```

```
In [2]: exact_change(42, [25, 1, 25, 10, 5])
Out[2]: False
```

```
In [3]: exact_change(42, [23, 1, 23, 100])
Out[3]: False
```

```
In [4]: exact_change(42, [23, 17, 2, 100])
Out[4]: True
```

```
In [5]: exact_change(42, [25, 16, 2, 15])
Out[5]: True
needs to be able to "skip" the 16...
```

```
In [6]: exact_change(0, [4, 5, 6])
Out[6]: True
```

```
In [7]: exact_change(-47, [4, 5, 6])
Out[7]: False
```

```
In [8]: exact_change(0, [])
Out[8]: True
```

```
In [9]: exact_change(42, [])
Out[9]: False
```

#### Hint

Similar to `LCS`, below, this problem can be handled by recurring *twice* and giving a name to each of the two results.

- For the first, try solving the problem *without* the first coin. (This is the *lose-it* case!)
  - You might even use the variable name `loseit`, as in `loseit = exact_change(...)`
- For the second, try solving it *with* the first coin. (This is the *use-it* case!)
  - You might continue by using the variable name `useit`, as in `useit = exact_change(...)`
- Then, have your code figure out the appropriate boolean value to return, depending on the results it gets!

#### Hint

For the last part of the last hint: This problem puts the `or` into *use it or lose it*—literally!

## Function to write #7: `LCS(S, T)`- DNA Matching

This week's final algorithmic challenge is to write a function named `LCS(S, T)`, which will accept two strings, `S` and `T`. `LCS` should return the longest common subsequence (LCS) that `S` and `T` share. The LCS will be a string whose letters are a subsequence of `S` and a subsequence of `T` (they must appear in the same order, though not necessarily consecutively, in those argument strings).

Note that if either `S` or `T` are the empty string, then the result should be the empty string!

Some examples:

```
In [1]: LCS('human', 'chimp')
Out[1]: 'hm'
```

```
In [2]: LCS('gattaca', 'tacgaacta')
Out[2]: 'gaaca'
```

```
In [3]: LCS('wow', 'whew')
Out[3]: 'ww'

In [4]: LCS("", 'whew')
first argument is the empty string
Out[4]: ""

In [5]: LCS('abcdefgh', 'efghabcd')
Out[5]: 'abcd'
```

Note that if there are ties, any one of the ties is OK: in the last example above, 'efgh' would be an equally acceptable result.

### Hint

Consider the following strategy:

- if the first two characters match, use them!
- If the first two characters don't match, recur *twice*: you could call this *use it or lose it or lose it!*
- For the first "lose it," recur to toss out one argument's initial letter:  
`result1 = LCS(S[1:], T)`
- For the second "lose it," recur to toss out the other argument's initial letter:  
`result2 = LCS(___, ___)`
- – A couple of details still need to be filled in...
- Finally, return the *better* of those two results—you'll have to remind yourself what "better" means for this problem!
- Good luck!

### Extra!

Are you saying to yourself, *never enough algorithms!* ?

Here is an optional extra-credit algorithm-design challenge that builds from `exact_change`. It's more difficult because:

- It returns the actual coins for making change, and
- It can also return `False`, so there are several cases to handle *after* the recursion...

### Extra-credit option #1 (up to +7ec pts): `make_change(target_amount, L)`

For up to +7 e.c. points, write a second change-handling function named `make_change(target_amount, L)`.

This function should actually determine which values (from `L`) could be returned to total the `target_amount`.

That is, instead of simply returning `True` or `False`, your `make_change` function should return *a list* of coins taken from `L` that sum up to `target_amount`. If there is no such list, then `make_change` should simply return `False`. If there can be more than one possible list of values from `L`, then your function may return any one of the valid answers.

The *order* of the values returned does not matter, though it's natural to have them in the same order as they appear in the original list (our tests will do this...).

You do not have to, but you are welcome to use `exact_change` as a subroutine (helper function) here!

The examples below show how `make_change` should work; these are the same arguments as in the `exact_change` function above.

In addition, `sorted` has been called, at least on the non-empty feasible cases, so that the results have a well-defined order:

```
In [1]: sorted(make_change(42, [25, 1, 25, 10, 5, 1]))
Out[1]: [1, 1, 5, 10, 25]
```

```
In [2]: make_change(42, [25, 1, 25, 10, 5])
Out[2]: False
```

```
In [3]: make_change(42, [23, 1, 23, 100])
Out[3]: False
```

```
In [4]: sorted(make_change(42, [23, 17, 2, 100]))
Out[4]: [2, 17, 23]
```

```
In [5]: sorted(make_change(42, [25, 16, 2, 15]))
Out[5]: [2, 15, 25]
```

```
In [6]: make_change(0, [4, 5, 6])
Out[6]: []
```

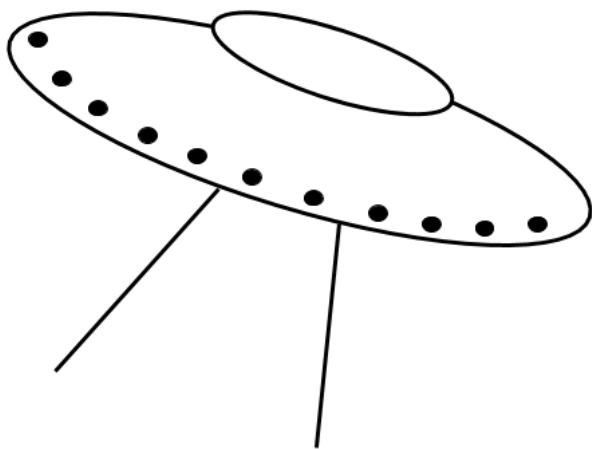
```
In [7]: make_change(-47, [4, 5, 6])
Out[7]: False
```

```
In [8]: make_change(0, [])
Out[8]: []
```

```
In [9]: make_change(42, [])
Out[9]: False
```

# CSforAll - Integrals

## CS for All



CSforAll Web > Chapter3 > Integrals

This website accompanies the textbook *CS For All* by Alvarado, Dodds, Kuenning, and Libeskind-Hadas

### Numeric Integration

#### Starter Code

In this lab you will build a Python program that can compute the areas underneath arbitrary mathematical functions (mathematicians call these *integrals*).

Doing this, you'll

- Practice writing Python functions.
- Gain experience with *list comprehensions*.
- Write helper functions and compose large functions from smaller ones.

In addition, you'll calculate answers that mathematicians don't have formulas for...and you'll never have to use anything more complicated than multiplication!

Start by creating a new file and pasting this code into it:

```
this gives us functions like sin and cos...
from math import *

two more functions (not in the math library above)

def dbl(x):
 """Doubler! argument: x, a number"""
 return 2*x

def sq(x):
 """Squarer! argument: x, a number"""
 return x**2

examples for getting used to list comprehensions...

def lc_mult(N):
 """This example accepts an integer N
 and returns a list of integers
 from 0 to N-1, **each multiplied by 2**
 """
 return [2*x for x in range(N)]

def lc_idiv(N):
 """This example accepts an integer N
 and returns a list of integers
 from 0 to N-1, **each divided by 2**
 WARNING: this is INTEGER division...!
 """
 return [x//2 for x in range(N)]

def lc_fdiv(N):
 """This example accepts an integer N
 and returns a list of integers
 from 0 to N-1, **each divided by 2**
 NOTE: this is floating-point division...!
 """
 return [x/2 for x in range(N)]

assert lc_mult(4) == [0, 2, 4, 6]
assert lc_idiv(4) == [0, 0, 1, 1]
assert lc_fdiv(4) == [0.0, 0.5, 1.0, 1.5]

Here is where your functions start for the lab:

Step 1, part 1
def unitfracs(N):
 """Be sure to improve this docstring!
 """
 pass
replace this line (pass is Python's empty statement)
```

### Trying Out *List Comprehensions*:

To begin, try out *list comprehensions* at the Python prompt...these examples should help build intuition about how list comprehensions work:

```
In [1]: lc_mult(10)
multiplication example
Out[1]: [0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

```
In [2]: lc_mult(5)
a smaller example
Out[2]: [0, 2, 4, 6, 8]
```

```
In [3]: lc_idiv(10)
integer division
Out[3]: [0, 0, 1, 1, 2, 2, 3, 3, 4, 4]
```

```
In [4]: lc_fdiv(10)
floating-point division
Out[4]: [0.0, 0.5, 1.0, 1.5, 2.0, 2.5, 3.0, 3.5, 4.0, 4.5]
```

Read over those three one-line functions; be sure you have internalized how they work!

**Note** that the calls `lc_idiv(10)` and `lc_fdiv(10)` return *different* lists; the first uses integer division (rounding down); the second uses floating-point division.

**To do** Change the `lc_idiv` function so that it becomes

```
return [float(x//2) for x in range(N)]
```

Before running it, decide if you think `lc_idiv(10)` will now be the same or different than before. Also, will the `assert` statement above succeed or fail?

Then, type `run your_file_name` in ipython and then and try `lc_idiv(10)`. Did it match your expectations? Nothing to write here, but be sure you're happy with why this new output is what it is!

**Optional** The [ListComprehension?](#) page has more examples to try, if you'd like...

### Integration: *why?* and *what?*

Integration is sometimes described as finding the area between a mathematical function and the horizontal ( $x$ ) axis.

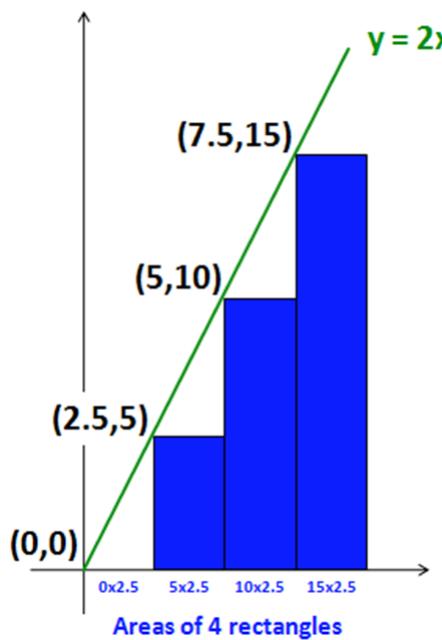
More generally, integration is simply the *sum* of a numeric function's results, i.e., y-values across a chosen span.

It is important because it provides a one-number summary of what the function is "doing" over an interval. It's also used to determine how fields of forces act on a particular point, surface, or object over time; it is also essential in defining a function's "average value" on an interval or region. Integrals can be used to design an airplane wing, predict the weather, or calculate how much a bank account will be worth in ten years even if interest rates are changing.

As an example, recall the `dbl` function from above (with a better docstring):

```
def dbl(x):
 """Argument: a number x (int or float)
 Return value: twice the argument
 """
 return 2*x
```

Suppose you wanted to estimate the integral of `dbl`, on the interval between 0.0 and 10.0. You could create the following (rough) approximation with rectangles:



1. Here, the interval is divided into 4 parts. The list  $[0, 2.5, 5, 7.5]$  represents the  $x$  value of the *left-hand endpoint* of each of the four subintervals.
2. We next find the return values from  $\text{dbl}(x)$  for each possible  $x$  in the list above. Here, we name these return values  $Y$ :  $Y = [0, 5, 10, 15]$ .
3. Now we add up the areas of the rectangles whose upper-left-hand corner (height) is at these  $Y$  values. Each rectangle extends down to the  $x$ -axis. Each rectangle's individual width is 2.5, because there are four equal-width rectangles spanning a total of 10 units of width.
4. We find the rectangles' areas in the usual way. Their heights are contained in the list  $Y$  and their widths are 2.5, so the total area is  

$$0*2.5 + 5*2.5 + 10*2.5 + 15*2.5$$
or  

$$(0 + 5 + 10 + 15)*2.5$$
which is  $(30)*2.5$ , or 75.

Yes, this is a very rough approximation for the "area under the curve," i.e., the integral. But if we make the width of the rectangles smaller, their sum will approximate that area as closely as we'd like!

More importantly, this example suggests a general way to approximate a function's integral over a known interval:

1. Divide the interval into  $N$  equal parts and create a list with the corresponding  $x$  (argument) values.
2. Find the  $y$  values (results) of the function for each value of  $x$  calculated in step 1
3. Calculate the area of each rectangle under the curve. Their heights are the  $y$  values; their widths are the separation between the  $x$  values.
4. Sum the rectangles' areas and return the result: that's the integral, or an approximation that can be made arbitrarily close.

The rest of this lab involves writing functions for each of these four steps; then you'll use those functions to answer questions about the results.

### Step 1: Calculating the $x$ Values to Be Evaluated

First, you will write a function named `unitfracs(N)` and then one named `scaledfracs(low, hi, N)`.

**Writing `unitfracs` ...**

Take a look at how `unitfracs(N)` works:

In [1]: `unitfracs(2)`

Out[1]: [0.0, 0.5]

In [2]: `unitfracs(4)`

Out[2]: [0.0, 0.25, 0.5, 0.75]

In [3]: `unitfracs(5)`

Out[3]: [0.0, 0.2, 0.4, 0.6, 0.8]

In [4]: `unitfracs(3)`

Out[4]: [0.0, 0.333333333333333, 0.6666666666666666]

In [5]: `unitfracs(10)`

Out[5]: [0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9]

**To do** Write `unitfracs(N)`.

As its name suggests, it returns a list of evenly-spaced left-hand endpoints (fractions) in the unit interval [0, 1).

Hint

Copy, paste, and alter the example function `lc_fdiv` in order to write `unitfracs`. *You will only need to change a single character from that code!*

Be sure to fix the docstring of `unitfracs` to reflect what it does, as well.

And finally, write at least three `assert` statements that implement tests -- feel free to use some of the examples above.

## Writing `scaledfracs...`

Next, take a look at how `scaledfracs(low, hi, N)` works:

In [1]: `scaledfracs(10, 30, 5)`

Out[1]: [10.0, 14.0, 18.0, 22.0, 26.0]

In [2]: `scaledfracs(41, 43, 8)`

Out[2]: [41.0, 41.25, 41.5, 41.75, 42.0, 42.25, 42.5, 42.75]

In [3]: `scaledfracs(0, 10, 4)`

Out[3]: [0.0, 2.5, 5.0, 7.5]

**To do** Write `scaledfracs(low, hi, N)`. It creates `N` left endpoints uniformly through the interval [low, hi].

Hint

This is tricky ... here is some additional explanation:

*Use* `unitfracs`. For example, use this line as a starting point:

```
return [for x in unitfracs(N)]
```

This way, you won't have to redo the work of `unitfracs`!

You might feel that this is closely related to the `interp` function you wrote in Lab 1—you're right!

You don't need to use that `interp` function, but you will want to use the ideas from it! Here it is, for reference:

```
def interp(low, hi, frac):
 """Returns a value frac of the way from low to hi"""
 return low + (hi-low)*frac
```

Note that the role of `frac` above is as the "interpolating fraction," which is exactly what `x` is doing in the list comprehension!

Do include a docstring that reflects what `scaledfracs` does and again, write `assert` statements to test the examples

above.

### Step 2: Calculating the Y Values

Your `scaledfracs` function can produce arbitrary lists of evenly-spaced  $x$  values.

Next, you'll need to calculate the  $y$  values (results) of a function at each of these  $x$  positions. Again, you'll use list comprehensions to make this process simple and fast!

Although the goal is to handle arbitrary functions, we'll start with a concrete function and build up.

#### Writing `sqfracs`...

**To do** Write a function `sqfracs(low, hi, N)` that works as follows:

```
In [1]: sqfracs(4, 10, 6)
Out[1]: [16.0, 25.0, 36.0, 49.0, 64.0, 81.0]
```

```
In [2]: sqfracs(0, 10, 5)
Out[2]: [0.0, 4.0, 16.0, 36.0, 64.0]
```

Here, `sqfracs` is very similar to `scaledfracs` *except that each value is squared*.

Hint

Use `scaledfracs` here. In the same way that `scaledfracs` used `unitfracs`, `sqfracs` can use `scaledfracs`! Consider the snippet:

```
for x in scaledfracs(low, hi, N)
```

As usual, write some `asserts` to test `sqfracs`. For extra fun, concoct some additional examples of your own!

#### Writing `f_of_fracs`...

**To do** Write a function `f_of_fracs(f, low, hi, N)` that works as follows:

```
In [1]: f_of_fracs(dbl, 10, 20, 5)
Out[1]: [20.0, 24.0, 28.0, 32.0, 36.0]
```

```
In [2]: f_of_fracs(sq, 4, 10, 6)
Out[2]: [16.0, 25.0, 36.0, 49.0, 64.0, 81.0]
```

```
In [3]: f_of_fracs(sin, 0, pi, 2)
the sine function
Out[3]: [0.0, 1.0]
the above values are rounded versions of what
will actually be displayed...
```

Note that `f_of_fracs` takes a *function* as its first argument—this is no problem in Python.

You might copy and paste `sqfracs` as a *model*: only a few characters need to be changed!

As in the previous parts of this problem, use the examples given above to create at least three assert statements for `f_of_fracs`.

### Step 3: Calculate the Area and Put It All Together

You now have functions that calculate both the  $x$  and, more importantly, the  $y$  values of a function at regularly-spaced intervals.

**To do** Next, you'll write `integrate(f, low, hi, N)` which will return the final, desired value: the integral of `f` from `low` to `hi` using `N` steps.

Take a look at a few examples and hints below.

As these examples highlight, `integrate` does not return a list; it returns a single, floating-point value:

```
In [1]: import math
to use math.pi and math.sin
```

```
In [2]: integrate(dbl, 0, 10, 4)
Out[2]: 75.0
the example from the top of this lab
```

```
In [3]: integrate(dbl, 0, 10, 1000)
Out[3]: 99.9
rounded from 99.8999... (precise value: 100)
```

```
In [4]: integrate(sq, 0, 3, 1000000) # a million steps will give Python pause
Out[4]: 8.9999865000044963
close! (precise value: 9.0)
```

```
In [5]: integrate(math.sin, 0, math.pi, 1000)
Out[5]: 1.9999983550656628
pretty good! (precise value: 2.0)
```

Don't worry about small errors in the rightmost (least-significant) digits. They occur from roundoff errors -- these are unavoidable when using a finite floating-point accuracy.

To get started, here is the "signature" (the def line) and docstring for `integrate`. Feel free to use this:

```
def integrate(f, low, hi, N):
 """Integrate returns an estimate of the definite integral
 of the function f (the first argument)
 with lower limit low (the second argument)
 and upper limit hi (the third argument)
 where N steps are taken (the fourth argument)

 integrate simply returns the sum of the areas of rectangles
 under f, drawn at the left endpoints of N uniform steps
 from low to hi
 """
```

```
assert integrate(dbl, 0, 10, 4) == 75
assert integrate(sq, 0, 10, 4) == 2.5 * sum([0, 2.5*2.5, 5*5, 7.5*7.5])
```

This time, we'll try "test-driven development", in which you write the tests `first` and then make your function satisfy the tests. That's why we included the `asserts` above! (Note that the second `assert` is an example of how to test even if you don't know the numerical answer!)

Hint

- **Do NOT use a list comprehension:** no list is being created here
- Rather, use `f_of_fracs` to generate the list of heights you need
- Remember that the result of `f_of_fracs` is a big list of y-values!
- You want to multiply those y-values (heights) by the rectangles' width. **But all the widths are the same!**
- So, you can sum the heights **then** multiply by the width!
- Use Python's built-in `sum`. `sum(L)` returns the sum of the elements in the list `L`.

Some examples of how `sum` works:

```
In [1]: sum([10, 4, 1])
Out[1]: 15
```

```
In [2]: sum(range(0, 101))
Out[2]: 5050
```

If your integrate function works, congratulations! You've built a general-purpose routine that can provide integrals for any computable function (including those for which there is no closed-form integral—i.e., ones where the best mathematicians in the world don't know how to compute an exact result).

Next, you'll put integrate to use...

### Questions To Answer in Your File

You should put your answers either within comments or within triple-quoted strings in your file. Strings are easier because you don't need to use the comment character, `#`, on multiple lines.

***Do not answer this "zeroth" question;*** we have placed the answer here, just as an example of how to answer the other two:

#### Question 0. (example only)

Explain why `integrate(dbl, 0, 10, N)` converges to 100 as `N` increases.

The answer in the file might look like (note that it's actually a pretty poor answer because it doesn't address the question):

```
"""Q0.
```

The value of `integrate(dbl, 0, 10, N)` as `N` increases is equal to  
the area under the dbl function, the line  
 $y = 2x$ , between  $x = 0.0$  to  $x = 10.0$ .

This value is the area of the triangle in the image at the top of the problem's page.

That area is 100, because the triangle's height is 20 and its width is 10.  
For a triangle,  $A = 0.5 \cdot h \cdot w$ .

```
"""
```

#### Question 1.

As noted, the exact value of the integral of the `dbl` function,  $y = 2x$ , from  $x = 0.0$  to  $x = 10.0$ , is 100, which is the area of the triangle in the image at the top of this page. That area is 100, because the triangle's height is 20 and its width is 10.

The calls to `integrate(dbl, 0, 10, 4)` and `integrate(dbl, 0, 10, 1000)`, shown above, produce a little *less* than 100.

In a sentence explain why `integrate` will always *underestimate* the correct value of this particular integral.

As a follow up, what is a function whose integral would always be *overestimated* on the same interval, from 0 to 10? (If you're not sure about this, answer the next question first.)

#### Question 2.

The following function, `c`, traces a part of a circular arc with radius 2.

```
def c(x):
 """c is a semicircular function of radius two"""
 return (4 - x**2)**0.5
```

Place this function into your file and confirm the values of

```
In [1]: integrate(c, 0, 2, 2)
Out[1]: 3.732
rounded...
```

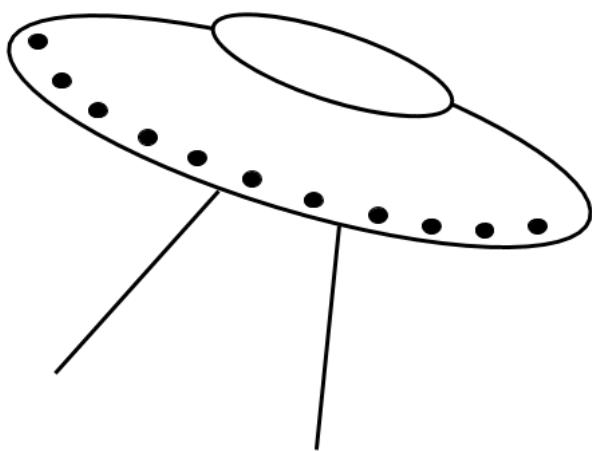
```
In [2]: integrate(c, 0, 2, 20)
Out[2]: 3.228
rounded...
```

Next, find the values of `integrate(c, 0, 2, 200)` and `integrate(c, 0, 2, 2000)` and make a note of them in your answer,

As  $N$  goes to infinity (i.e., becomes larger and larger), what does the value of this integral become? Why?

## CSforAll - ScrabbleWords

### CS for All



CSforAll Web > Chapter3 > ScrabbleWords

This website accompanies the textbook *CS For All* by Alvarado, Dodds, Kuenning, and Libeskind-Hadas

#### Scrabble Words

In chapter 2, we did a bit of rudimentary Scrabble scoring. Your ultimate task here is to write a function that takes as an argument a "rack"—a **list** of letters—and returns the highest-scoring word that can be made with those letters (remember that each letter in the rack can only be used once). This is the key ingredient in a computerized Scrabble game!

In this problem, you may use recursion as well as the built-in higher-order functions `map`, `filter`, and `reduce`. **In fact**, if you write a function that needs to process a long list (e.g. a dictionary) with more than a few hundred items in it, you are much better off letting `map`, `filter`, or `reduce` cruise through those lists since they have been optimized to work very fast.

There will be a few places here where using anonymous functions (`lambda`) is probably the cleanest and nicest way to do business. Use it when it's the cleanest way to proceed.

One of the objectives of this problem is to have you think about designing a more complicated program that involves several functions. You have complete autonomy in deciding what functions to write in order to reach the ultimate goal (see more on the two required functions below). Try to think carefully about which functions you will need, and try to implement those functions so that they are as simple and clean as possible. Part of your score on this problem will be based on the elegance of your overall design and individual functions.

Our sample solution has fewer than 9 functions, two of which we copied from our solution to Problem 1. The remaining 7 (or so) functions range from one to four lines of code per function. While you are not required to have the same number of functions and you may have a few slightly longer functions, this is intended to indicate that there is not much code that needs to be written here!

All of your functions should be in one file. Be sure to have a comment at the top of the file with your name(s), the filename, and the date.

Be sure to include a docstring for every function that you write.

You may wish to include—by copying—some of the functions that you wrote in Chapter 2's Function Frenzy problem.

A bit later in this problem, we'll give you a fairly large dictionary of English words to use. For now, we recommend that you use the following tiny dictionary during the course of testing and development. Include these lines near the top of your file. That way, they will be global variables that can be used when we—and you—test your functions.

Place these two variables at the top of your file:

```
scrabbleScores = [["a", 1], ["b", 3], ["c", 3], ["d", 2], ["e", 1],
 ["f", 4], ["g", 2], ["h", 4], ["i", 1], ["j", 8],
 ["k", 5], ["l", 1], ["m", 3], ["n", 1], ["o", 1],
 ["p", 3], ["q", 10], ["r", 1], ["s", 1], ["t", 1],
 ["u", 1], ["v", 4], ["w", 4], ["x", 8], ["y", 4],
 ["z", 10]]
```

```
nanoDictionary = ["a", "am", "at", "apple", "bat", "bar", "babble",
 "can", "foo", "spam", "spammy", "zzyzva"]
```

### Do not change the values of these global variables!

#### The Details...

Ultimately, there are two functions that we will be testing.

- `scoreList(rack, dictionary)` has two arguments: a `rack`, which is a list of lower-case letters, and `dictionary`, which is a list of legal words.

The `scoreList` function returns a list of all of the words in the `Dictionary` argument that can be made from those letters, together with the score for each one. Specifically, this function returns a list of lists, each of which contains a string that can be made from the `Rack` and its Scrabble score. Here are some examples using `nanoDictionary` above:

```
In [1]: scoreList(["a", "s", "m", "t", "p"], nanoDictionary)
Out[1]: [['a', 1], ['am', 4], ['at', 2], ['spam', 8]]
```

```
In [2]: scoreList(["a", "s", "m", "o", "f", "o"], nanoDictionary)
Out[2]: [['a', 1], ['am', 4], ['foo', 6]]
```

The order in which the words are presented is not important.

- `bestWord(rack, dictionary)` accepts a `Rack` and `Dictionary` as above and returns a list with two elements: the highest-scoring possible word from that `Rack`, followed by its score. If there are ties, they can be broken arbitrarily. Here is an example, again using `nanoDictionary` above:

```
In [1]: bestWord(["a", "s", "m", "t", "p"], nanoDictionary)
Out[1]: ['spam', 8]
```

Aside from these two functions, all of the other helper functions are up to you! Some of those helper functions may be ones that you wrote in Problem 1, or slight variants of those functions.

You might find it useful to use a strategy in which you write a function that determines whether or not a given string can be made from the given `Rack` list. Then, another function can use that function to determine the list of *all* strings in the dictionary that can be made from the given `Rack`. Finally, another function might score those words.

Remember to use `map`, `reduce`, or `filter` where appropriate—they are powerful and they are optimized to be very fast.

**Test each function carefully** with small test arguments before you proceed to write the next function. This will save you a lot of time and aggravation!

#### A reminder about Python's `in` operator:

Imagine that you are writing a function that attempts to determine if a given string `S` can be made from the letters in the `Rack` list. You might be tempted to scramble ("permute," to use a technical term) the `Rack` in every possible way as part of this process, but that is more work than necessary. Instead, you can test if the first symbol in your string, `S[0]`, appears in the rack with the statement:

```
if S[0] in Rack:
```

```
if True you will end up here
else:
```

```
if False you will end up here
```

This `in` operator is very handy. From here, recursion will let you do the rest of the work without much effort on your part!

Although we don't expect that you will end up doing a huge amount of recursion, you should know that Python can get snippy when there are a lot of recursive calls.

By default, Python generally complains when there are 1000 or more recursive calls of the same function. To convince it to be friendlier, you can use the following at the top of your file:

```
import sys

sys.setrecursionlimit(10000)
Allows up to 10000 recursive calls; the maximum allowable varies from system to system
```

## Trying It With a Bigger Dictionary

Finally, if you want to test out a big dictionary (it's more fun than the little one above), you can download the file `dict.txt`. It contains a large list of words (a bit over 4000). The list is simply called `Dictionary`. Place this file in the same directory as your file and rename it as `dict.py`.

After you have downloaded the file and placed it in the same directory as your file, you can import that file at your shell in order to use `Dictionary` in your testing.

Here are some examples showing how to use this larger `Dictionary`

```
In [1]: from dict import *
this imports the large Dictionary variable
```

```
In [2]: scoreList(['w', 'y', 'l', 'e', 'l', 'o'], Dictionary)
Out[2]: [['leo', 3], ['low', 6], ['lowly', 11], ['ow', 5], ['owe', 6], ['owl', 6], ['we', 5], ['well', 7], ['woe', 6], ['yell', 7], ['yo', 5]]
```

Notice that "yellow" is not in this dictionary. We "cropped" off words of length 6 or more to keep the dictionary from getting too large. Finally, here are examples of `bestWord` in action (remember that tie scores are OK, too):

```
In [3]: bestWord(['w', 'y', 'l', 'e', 'l', 'o'], Dictionary)
```

```
Out[3]: ['lowly', 11]
```

```
In [4]: bestWord(["s", "p", "a", "m", "y"], Dictionary)
```

```
Out[4]: ['may', 8]
```

```
In [5]: bestWord(["s", "p", "a", "m", "y", "z"], Dictionary)
```

```
Out[5]: ['zap', 14]
```

## And An Even Bigger Dictionary!

Want to try this with an even bigger dictionary? Two Harvey Mudd CS students a couple of years ago constructed one with over 100,000 words. You should be able to import it into your program without difficulty:

Start by downloading bigD.txt. Place this file in the same directory as your file and rename it as bigD.py.

Then, you can use the same technique as above:

```
In [1]: from bigD import *
```

```
this imports the very large Dictionary variable
```

```
In [2]: scoreList(['a', 'b', 'v', 'x', 'y', 'y', 'z', 'z', 'z'], Dictionary)
```

```
Out[2]: [['ab', 4], ['aby', 8], ['ax', 9], ['ay', 5], ['ba', 4], ['bay', 8], ['by', 7],
['ya', 5], ['yay', 9], ['za', 11], ['zax', 19], ['zyzzyva', 43], ['zzz', 30]]
```

***Please do not*** include any of the special dictionary-import statements in your code.

Good luck—have fun!

# CSforAll - SoundsGood

## CS for All

CSforAll Web > Chapter3 > SoundsGood

This website accompanies the textbook *CS For All* by Alvarado, Dodds, Kuenning, and Libeskind-Hadas

### ***Sounds Good!***

#### **Starting Files to Download**

***Starter file for this problem:***

Download pythonSounds.zip from this link.

Be sure to unzip that folder somewhere. It has several files ***all of which need to stay in that same folder:***

- soundstarter.py (the file to edit and run!)
- swfaith.wav
- swnotry.wav
- spam.wav
- csaudio.py
- play (A small MacOS application for playing sounds...)

#### ***You Need to Work From Within This Folder!***

- To do this, be sure to cd into the pythonSounds folder.
- For example, you could first move the *whole* pythonSounds folder to your desktop... be sure to do this *before* you open your python code in VSCode!
- Then cd Desktop and cd pythonSounds
- From there, you can run the usual ipython and then run soundstarter.py
- Please do keep the files inside pythonSounds all together!

**If you move the soundstarter.py file without the others..., things won't work!**

## Warm-Up: Helper Functions Using List Comprehensions (*L*Cs)

Take a moment to remind yourself how *list comprehensions* work... .

Look over the three\_ize function near the top of the soundstarter.py file:

```
def three_ize(L):
 """three_ize is the motto of the green CS 5 alien.
 It's also a function that accepts a list and
 returns a list of elements each three times as large.
 """
 # this is an example of a list comprehension
```

```
LC = [3 * x for x in L]
return LC
```

This function "maps" the expression  $3*x$  over the values  $x$  in the list  $L$ .

Try it out with

#### Example(s):

```
In [1]: three_ize([13, 14, 15])
Out[1]: [39, 42, 45]
```

List comprehensions are a versatile syntax for mapping a function (or expression) across all elements of a list.

If you feel good about list comprehensions, onward! (If you think more explanation/practice would be worthwhile, try our ListComprehension<sup>7</sup> page.)

#### Function to Write #1: scale

With the above function as your model, write a function `scale` with the following signature:

```
def scale(L, scale_factor):
```

where `scale` returns a list similar to  $L$ , except that each element has been multiplied by `scale_factor`.

#### Example(s):

```
In [1]: scale([70, 80, 420], 0.1)
Out[1]: [7.0, 8.0, 42.0]
```

*Use a list comprehension here.*

#### Going Further: *Index-Based* List Comprehensions

Next, make sure this `three_ize_by_index` function is in your `soundstarter.py` file.

Look it over:

```
def three_ize_by_index(L):
 """three_ize_by_index has the same behavior as three_ize
 but it uses the INDEX of each element, instead of
 using the elements themselves -- this is much more flexible!
 """

another example of a list comprehension
```

```
N = len(L)
LC = [3 * L[i] for i in range(N)]
return LC
```

This function does *exactly the same thing* as `three_ize`—it simply uses the index of each element to do so. That is, now the **location** of each element, named  $i$ , is changing

This index-based use of list comprehensions is even more flexible than the element-based style, as the next couple of questions will show.

#### Functions to Write #2 and #3: add\_2 and add\_3

With the above ***index-based*** functions as a guide, write a function `add_2` with the following signature:

```
def add_2(L, M):
```

such that `add_2` accepts two lists and returns a single list that is an element-by-element sum of the two arguments. If the arguments are different lengths, your `add_2` should return a list that is as long as the *shorter* of the two. Just ignore or drop the extra elements from the longer list.

Using `min` and `len(L)` and `len(M)` together is one way to do this. For example, the line

```
N = min(len(L), len(M))
```

will assign the smaller of the lengths of `L` and `M` to `N`.

You will want to use the *index-based* approach for this `add_2` function. You might use `three_ize_by_index` as a starting point.... Also, consider how this LC might help:

```
LC = [L[i] + M[i] for ...]
```

Here are two examples of `add_2` in action:

```
In [1]: add_2([10, 11, 12], [20, 25, 30])
```

```
Out[1]: [30, 36, 42]
```

```
In [2]: add_2([10, 11], [20, 25, 30])
```

```
Out[2]: [30, 36]
```

Then, write the analogous three-argument function `add_3` with the following signature:

```
def add_3(L, M, P):
```

where `L`, `M`, and `P` are all lists and `add_3` returns the sum of all of them, but with only as many elements as the shortest among them has.

The strategy will be very similar to `add_2`.

#### Function to Write #4: `add_scale_2`

Next, write a function `add_scale_2` with the following signature:

```
def add_scale_2(L, M, L_scale, M_scale):
```

such that `add_scale_2` accepts two lists `L` and `M` and two floating-point numbers `L_scale` and `M_scale`. These stand for *scale for L* and *scale for M*, respectively.

Then, `add_scale_2` should return a single list that is an element-by-element sum of the two argument lists, *each scaled by its respective floating-point value*. If the argument lists are different lengths, your `add_scale_2` should return a list that is as long as the *shorter* of the two. Again, just drop any extra elements.

#### Example(s):

```
In [1]: add_scale_2([10, 20, 30], [7, 8, 9], 0.1, 10)
```

```
Out[1]: [71.0, 82.0, 93.0]
```

```
In [2]: add_scale_2([10, 20, 30], [7, 8], 0.1, 10)
```

```
Out[2]: [71.0, 82.0]
```

This will not be too different from the previous examples!

#### A Helper Function: `randomize`

Next, take a look at this function in your `soundstarter.py` file:

```
def randomize(x, chance_of_replacing):
 """randomize accepts an original value, x
 and a fraction named chance_of_replacing.
```

With the "chance\_of\_replacing" chance, it should return a random float from -32767 to 32767.

Otherwise, it should return x (not replacing it).

```
"""
r = random.uniform(0, 1)
if r < chance_of_replacing:
 return random.uniform(-32768, 32767)
else:
 return x
```

Read over the docstring and try it out.

Nothing to do here except build an understanding of what this function is doing: how often it returns the original argument and how often it returns a random value. That random value happens to always be within the amplitude of a sound's pressure samples.

Though it's random, here is a set of five real runs:

```
In [1]: randomize(42, .5)
Out[1]: 42
```

```
In [2]: randomize(42, .5)
Out[2]: 42
```

```
In [3]: randomize(42, .5)
Out[3]: 29209.30669767395
```

```
In [4]: randomize(42, .5)
Out[4]: 42
```

```
In [5]: randomize(42, .5)
Out[5]: 17751.221299744262
```

### Function to Write #5: replace\_some

Next, write a function `replace_some` with the following signature:

```
def replace_some(L, chance_of_replacing):
```

such that `replace_some` accepts a list L and a floating-point value `chance_of_replacing`.

Then, `replace_some` should independently replace—or not replace—each element in L, using the helper function `randomize`.

Since this function is random, the runs below won't be replicated on your system, but try yours out to make sure it's working in a similar fashion.

Hint

use `randomize` in a list comprehension: that's it! Consider how to complete this thought (and don't forget to return LC):

```
LC = [randomize(___, ____) for x in L]
```

### Example(s):

```
In [1]: replace_some(range(40, 50), .5)
replace about half (hopefully the 42 remains!)
Out[1]: [40, 41, 42, -17461.09350529409, 44, -13989.513742241645, 46, -26247.774200304026, 48, 49]
```

```
In [2]: replace_some(range(20, 30), .1)
replace about a tenth (but it's random: here 2 of them get replaced)
Out[2]: [20, 21, 16774.26240973895, 23, 24, 25, -18184.919872079583, 27, 28, 29]
```

To help you test, here are two assertions to paste into your file. Note that second one says that the result is DIFFERENT from 42!

```
assert replace_some(range(40, 50), 0) == list(range(40, 50))
assert replace_some([42], 1.0) != 42
```

In addition to providing practice with data and functions, the above examples will be helpful in creating functions that handle audio data in various ways...

The `replace_some` function will allow you to add "static" (random values) to *some* of any sound, e.g., to make it sound "crackly."

## Sound Coding...

First things first: try out this function, which should already be in your `soundstarter.py` file.

You can run it with `test()`:

```
a function to make sure everything is working
def test():
 """A test function that plays swfaith.wav
 You'll need swfaith.wav in this folder.
 """
 play('swfaith.wav')
```

For this to work, your Python will need to support sound (every version we've tested does). If yours does not—no problem, simply work with a partner from here on during this lab.

Also, you'll need the `swfaith.wav` file in the folder in which `soundstarter.py` is located. As long as you're in the original folder, all of this should be the case. If not, go grab all of those files that came with `soundstarter.py` and copy them over to whichever folder you're working in.

Before we go on, you'll need a bit of background information on audio data. Then you'll have a chance to write a number of audio-processing functions.

## Background on Representing Audio Information

What is inside an audio file?

Depending on the format, the actual audio data might be encoded in many different ways. One of the most basic is known as pulse code modulation (PCM), in which the sound waves are sampled every so often and given values in the range -128 to 127 (if 1 byte per sound sample is used) or -32768 to 32767 (if there are 2 bytes for each sample). Wikipedia explains it [here](#).

The `.wav` file format encodes audio in basically this way, and the cross-platform program Audacity is an excellent tool for visualizing the individual PCM samples of an audio file. You don't need Audacity for this problem, but it runs on Windows and Macs and is fun to play around with if you'd like to. Audacity can also convert to `.wav` from `.mp3` and many other formats. Last but not least, Audacity was created by Dominic Mazzoni, an HMC alum!

## Getting Started with Sound

We present two examples to start acquiring and manipulating sound data. Try these:

### Sound Example #1: changeSpeed

This function should already be in your file, but if not, it's here for easy copy-and-paste:

```
The example changeSpeed function
def changeSpeed(filename, newsr):
 """changeSpeed allows the user to change an audio file's speed.
 Arguments: filename, the name of the original file
 newsr, the new sampling rate in samples per second
 Result: no return value, but
 this creates the sound file 'out.wav'
 and plays it
 """
 print("Playing the original sound...")
 play(filename)

 sound_data = [0, 0]
 # an "empty" list
 read_wav(filename, sound_data) # get data INTO sound_data

 samps = sound_data[0]
 # the raw pressure samples

 print("The first 10 sound-pressure samples are\n", samps[:10])
 sr = sound_data[1]
 # the sampling rate, sr

 print("The number of samples per second is", sr)

 # we don't really need this line, but for consistency...
 newsamps = samps
 # same samples as before
 new_sound_data = [newsamps, newsr]
 # new sound data pair
 write_wav(new_sound_data, "out.wav") # write data to out.wav
 print("\nPlaying new sound...")
 play('out.wav')
 # play the new file, 'out.wav'
```

Read over this example and try it out on the three sound files provided:

```
In [1]: changeSpeed("swfaith.wav", 44100) # fast Vader
... some printing ...
```

```
In [2]: changeSpeed("spam.wav", 11025)
slow Monty Python
... some printing ...
```

```
In [3]: changeSpeed("swnotry.wav", 22050) # regular-speed Yoda
... some printing ...
```

### What's Inside the Sound-Processing Code?

1. The sound data is returned *in two pieces* by the call to `read_wav`, using the lines

```
read_wav(filename, sound_data)
samps = sound_data[0]
sr = sound_data[1]
```

Note that the lines are separated and that `sound_data` starts out as `[0, 0]`. What is happening is that `read_wav`

changes the `sound_data` list so that its 0-index element is the list of samples and the 1-index element is the sampling rate (`sr`).

2. After that call, the variable `samps` is a large list of raw pressure samples (floats). **Don't print this list**—it can be too big and can slow down or completely choke the ipython shell!
3. Also, after that call, the variable `sr` is an integer that represents the *sampling rate*, i.e., the number of samples that should be played per second for normal-speed playback.
4. Some printing happens, so that you can see a little bit of the data.
5. We already have the new sampling rate—that was the `newsr` argument. For consistency, we use the variable `newsamps` to label the new sound data samples. In this case they're not changing at all, but in some later programs `newsamps` will be different from `samps`.
6. The code then writes `newsamps` and `newsr` out to a file named `out.wav`, which will appear in the folder you're working in (replacing an old one, if present).
7. To finish, the function plays that new file, which is now at the sampling rate of `newsr`.

Variations of these steps will be in all of the sound functions.

The next example shows how to create a new sound by changing the samples themselves. That is, `newsamps` will be different from `samps` (the old samples). Remember that `samps` will be a very large list of pressure values (~50,000 elements).

### Sound Example #2: flipflop

This function should already be in your file, but if not, it's also here for easy reference and copy-and-paste:

```
def flipflop(filename):
 """flipflop swaps the halves of an audio file
 Argument: filename, the name of the original file
 Result: no return value, but
 this creates the sound file 'out.wav'
 and plays it
 """
 print("Playing the original sound...")
 play(filename)

 print("Reading in the sound data...")
 sound_data = [0, 0]
 read_wav(filename, sound_data)
 samps = sound_data[0]
 sr = sound_data[1]

 print("Computing new sound...")

 # this gets the midpoint and calls it x
 x = len(samps)//2
 newsamps = samps[x:] + samps[:x]
 newsr = sr
 new_sound_data = [newsamps, newsr]

 print("Writing out the new sound data...")
 write_wav(new_sound_data, "out.wav") # write data to out.wav

 print("Playing new sound...")
```

```
play('out.wav')
```

Take a look at the middle part of this code, where the new sound samples are created from the old ones. In this case, the `newsamps` are a "flipflopped" version of the old `samps`. Note that this code is EXACTLY the same as the `flipside` problem from Lab 2<sup>7</sup>.

As a result, the sound's second half is placed before its first half.

In building your audio-processing functions, use `flipflop` as a starting point.

### Sound Function to Write #1: reverse

Next, write a sound-handling function `reverse` with the following signature:

```
def reverse(filename):
```

such that `reverse` accepts a `filename` as did `flipflop`.

Copy-and-paste `flipflop` to get started!

Like `flipflop`, the sampling rate should not change, but the function should create a *reversed* set of sound samples and then handle them in the same way as the two examples above. That is, you'll want to write them to the file `out.wav` and then play that file.

**Remember** that to reverse the list `samps`, you can write `samps[::-1]` in Python!

#### Example(s):

```
In [1]: reverse('swfaith.wav')
redaV htraD sounds eerier but less intimidating
... lots of printing ...
```

Note that this `reverse` function *won't* need to use any of the helper functions you wrote above—but the next few will!

### Sound Function to Write #2: volume

Now, write a sound-handling function `volume` with the following signature:

```
def volume(filename, scale_factor):
```

such that `volume` accepts a `filename` as usual and a floating-point value `scale_factor`. Then, `volume` should handle the sound in the usual way, with the output file and played sound being scaled in amplitude (volume) by the scaling factor `scale_factor`. In other words, each sample should be multiplied by `scale_factor`.

Hint

Here, use the helper function `scale` you wrote at the beginning of the lab!

All you'll need is

```
newsamps = scale(_____, _____)
```

(This is quite typical—only a very small amount of the sound code needs to be altered for each function.)

#### Example(s):

```
In [1]: volume('swfaith.wav', .1)
A calmer Darth...
... lots of printing ...
```

```
In [2]: volume('swfaith.wav', 10.0)
A caffeinated Darth!
... lots of printing ...
```

You'll notice that your hearing adjusts remarkably well to this function's changes in absolute volume, making the perceived effect considerably less than you might expect.

You will also find that if you *increase* the volume too much, the sound becomes distorted, just as when an amplifier is turned up to 11.

### Sound Function to Write #3: static

Now, write a sound-handling function `static` with the following signature:

```
def static(filename, probability_of_static):
```

such that `static` accepts a `filename` (as usual) and a floating-point value `probability_of_static`, which you can assume will be between 0 and 1.

Then, `static` should handle the sound in the usual way, with the output samples being replaced with a probability of `probability_of_static`. When they're replaced, the samples should simply be random values, uniformly chosen in the valid range from -32768 to 32767.

Here, you should use the helper function `replace_some` that you wrote earlier in the lab. You won't need `randomize`, because `replace_some` already uses it!

#### Example(s):

```
In [1]: static('swfaith.wav', .05)
Vader, driving into a tunnel
... lots of printing ...
```

```
In [2]: static('swfaith.wav', .25)
Vader on dial-up from a galaxy far, far away
... lots of printing ...
```

You might see how high you can increase the percentage of static until the original is no longer discernable. People adapt less well to this than to volume changes.

### Sound Function to Write #4: overlay

Now, write a sound-handling function `overlay` with the following signature:

```
def overlay(filename1, filename2):
```

such that `overlay` accepts *two* filenames, and creates a new sound that overlays (combines) the two. The result should be as long as the shorter of the two. (Drop any extra samples, just as in `add_scale_2`.)

Use your `add_scale_2` helper function to assist with this! That way, you can adjust the relative loudness of the two input files. You are welcome, but certainly not required, to add more arguments to your `overlay` function so that you can change the relative volumes on the fly (or crop the sounds on the fly, which is a bit more ambitious).

**Remember** that `add_scale_2(samps1, samps2, 0.5, 0.5)` *must* take lists (`samps`) as arguments—not filenames, which are simply strings! The `samps` are lists of the raw sound data.

#### Example(s):

```
In [1]: overlay('swfaith.wav', 'swnotry.wav')
Vader vs. Yoda
... lots of printing ...
```

Extra: how could you modify `overlay` so that it doesn't truncate the longer sound? Instead of truncating it, you could let it continue against silence, or you could repeat the shorter sound...

The next function overlays a file with a shifted version of itself.

### Sound Function to Write #5: echo

This one is more of a challenge... .

Try writing a sound-handling function `echo` with the following signature:

```
def echo(filename, time_delay):
```

such that `echo` accepts a `filename` as usual and a floating-point value `time_delay`, which represents a number of seconds.

Then, `echo` should handle the sound in the usual way, with the original sound being overlaid by a copy of itself shifted later in time by `time_delay`.

To do the overlaying, you'll want to use `add_scale_2`, as before.

To handle the time-shifting, notice that you can use the sampling rate to convert between the number of samples and time in seconds:

- For example, if `time_delay` is 0.1 and the sampling rate is 22050, then the number of samples to wait is 2205
- Similarly, if `time_delay` is 0.25 and the sampling rate is 44100, then the number of samples to wait is 11025

Hint

How to "add wait time" to samples: the easiest way to add "blank space" or "blank sound" in front of `samps` is to concatenate (add a list of) zeros to the front of the list `samps`. For example,

```
samps2 = [0]*42 + samps
```

would "wait" 42 samples, by including 42 blank-sound samples, at the start of the sound data `samps`.

You'll probably want a value other than 42—in fact, the challenge is to compute the *correct* value there!

How could you figure out what integer you need *instead of 42*?

- Remember that you know the time you'd like (in seconds) and the sampling rate (in samples per second).
- Be sure that you use an integer—remember that, if you have a floating-point value `f`, then `int(f)` is an integer.
- By the way, there are other approaches that work for `echo`, as well! with thanks to Sophie Harris for inventing one of the alternatives!

### Example(s):

```
In [1]: echo('swfaith.wav', .1)
How many zeros would be needed in front?
... lots of printing ...
```

### Sound Example #3: Generating Pure Tones

The final examples of provided functions generate a pure sine-wave tone. Here is the code, though it should also be in the file:

```
Helper function for generating pure tones
def gen_pure_tone(freq, seconds, sound_data):
 """pure_tone returns the y-values of a cosine wave
 whose frequency is freq Hertz.
 It returns nsamples values, taken once every 1/44100 of a second.
 Thus, the sampling rate is 44100 hertz.
 0.5 second (22050 samples) is probably enough.
```

```

"""
if sound_data != [0, 0]:
 print("Please input a value of [0, 0] for sound_data.")
 return
sampling_rate = 22050

how many data samples to create
nsamples = int(seconds*sampling_rate) # rounds down

our frequency-scaling coefficient, f
f = 2*math.pi/sampling_rate
converts from samples to Hz

our amplitude-scaling coefficient, a
a = 32767.0
sound_data[0] = [a*math.sin(f*n*freq) for n in range(nsamples)]
sound_data[1] = sampling_rate
return sound_data

def pure_tone(freq, time_in_seconds):
 """Generates and plays a pure tone of the given frequency."""
 print("Generating tone...")
 sound_data = [0, 0]
 gen_pure_tone(freq, time_in_seconds, sound_data)

 print("Writing out the sound data...")
 write_wav(sound_data, "out.wav") # write data to out.wav

 print("Playing new sound...")
 play('out.wav')

```

Look over this code and try it out to get a feel for what it does, though the math of the sine wave is not crucial. Rather, the important details are that the function `pure_tone` takes a desired frequency `freq` and the span `time_in_seconds`. The mathematical details are then delegated to `gen_pure_tone`.

### Example(s):

```
In [1]: pure_tone(440, 0.5)
0.5 seconds of the concert-tuning A
... lots of printing ...
```

You can look up frequencies for other notes at this Wikipedia page, among many others. Here's a small chart, as well:

It's interesting to note that C0 is below the range of normal human hearing (we can only hear down to about 20 Hz) but B8 leaves plenty of room (most people below the age of 40 can hear to 20,000 Hz or higher). Also, most pianos only go to A0 (28 Hz), but the Bösendorfer Imperial Concert Grand has extra keys (colored black) that go all the way down to C0. Just in case you need extra bass!

### Sound Function to Write #6: `chord`

The final lab problem is to build on the above example to write a chord-creation function named `chord`, with the following signature:

```
def chord(f1, f2, f3, time_in_seconds):
```

such that `chord` accepts three floating-point frequencies `f1`, `f2`, and `f3`, along with a floating-point `time_in_seconds`.

In the end, your `chord` function should create and play a three-note chord from those frequencies.

You will want to get *three* sets of `samps` and `sr` from `gen_pure_tone`, e.g.,

```
samps1, sr1 = gen_pure_tone(f1, time_in_seconds, [0, 0])
samps2, sr2 = gen_pure_tone(f2, time_in_seconds, [0, 0])
samps3, sr3 = gen_pure_tone(f3, time_in_seconds, [0, 0])
```

**Then**, you really need an `add_scale_3` function, though we don't have that yet. But you can create it! (You could use `add_scale2` and `add_3` as starting points, but we'd recommend writing `add_scale_3` on its own—not calling those other functions.)

Finally, you'll need to take the resulting samples-list (perhaps name it `newsamps`) and process it using code borrowed from the previous functions:

```
new_sound_data = [newsamps, newsr]

print("Writing out the new sound data...")
write_wav(new_sound_data, "out.wav") # write data to out.wav

print("Playing new sound...")
play('out.wav')
```

### Example(s):

```
In [1]: chord(440.000, 523.251, 659.255, 1.0)
A minor chord
... lots of printing ...
```

**Does your chord sound awful?** Remember what happened when you used `volume` to increase the volume too much? `gen_pure_tone` produces a tone that is at maximum volume. When you combine two (or three) such tones, their peak volumes add together—and the result is too loud for the computer to handle, producing distortion. Think about how you could adjust for this phenomenon without simply making the chord *too quiet*. In other words:

- You'll want to keep the overall amplitude at 1.0.
- Since the amplitude of each original is 1.0, you'll need to use *fractional* scale values to make sure the overall amplitude of the summed waves stays at 1.0 or less.
- If the wave exceeds 1.0 in amplitude, it will be "clipped" by the speakers, which will sound like loud static overlaying the sound (or just sound horrible).

*Challenge:* Use the table of frequencies above to change that chord from an A-minor to an A-major chord. Or build your own...

*But what about creating a C minor 7th (or augmented) chord?*

Indeed, you might want to create larger chords with arbitrarily many notes...or other unusual/odd/interesting/inspired/disturbing algorithmically-generated sound effects. We certainly encourage you to try things out! It's not terribly hard to write something that plays real music!

Hint

It can help to have functions such as `Aflat(duration)` or `Cm7(duration)`, and then write something that builds on those.

### What if you could see sound?

You can hear the products of the functions you've written, now let's visualize them.

In `csaudio.py` we've included a function called `graph` which you will be able to use. This function uses `matplotlib` to graph the amplitude over time and a heat map of frequency over time. It can handle any number of filenames (in theory) but please limit yourself to a few at a time for your computer's sake. Give it a try by running `graph('swfaith.wav')`. In order to see your sound functions in action, add the following line to the end of the function, run your file, then try the function again.

```
graph(filename, 'out.wav')
```

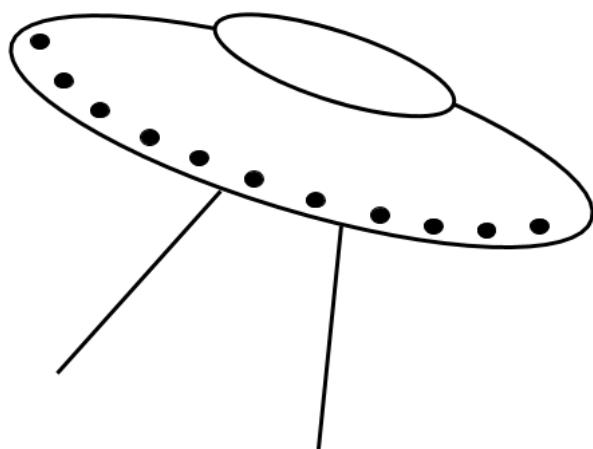
Note: This is for when the sound function has a single filename as input. For functions that don't fall in that category, adjust the inputs to graph as fit.

Consider: How would the graphs changed if you...

- lowered the volume?
- sped up the track?
- used the flipflop function?
- overlayed another sound file?

# CSforAll - HigherOrderFunctions

## CS for All



CSforAll Web > Chapter3 > HigherOrderFunctions

This website accompanies the textbook *CS For All* by Alvarado, Dodds, Kuenning, and Libeskind-Hadas

### Higher-Order Functions

This problem has two short functions to practice writing higher-order functions.

`compose`

The `compose(funcList)` function takes a list of zero or more functions as input, where each function in that list takes one input and produces one output. `compose` returns a single function of one input and one output that represents the *composition* of functions in `funcList`. Specifically, if `funcList` is a list of functions `[f1, f2, ..., fk]` then the function returned by `compose` will be the function that on input `X` returns `=f1(f2(f3(... fk(X) )) )`. Here are some examples...

```
def double(X):
 return 2*X
```

```
def square(X):
 return X**2

L1 = [double, square]
L2 = [square, double]
```

```
In [1]: foo = compose(L1)
In [2]: foo(10)
Out[2]: 200
In [3]: bar = compose(L2)
In [4]: bar(10)
Out[4]: 400
```

Notice that in these examples, there were only two functions in the lists. But, in general, there can be *any* number of functions in the list, including zero functions! Think carefully about what the base case should be for an empty list. (It *must* still return a function since `compose` is "contractually obligated" to always return a function.) Please also develop a few tests of your own with other lists of functions (e.g., with zero functions, one function, and three functions).

Aside from the `def compose(funcList)` and docstring lines, this function should be only two lines long.

### makePoly

Here's one Python function for computing a specific polynomial:

```
def poly(X):
 """ compute the polynomial f(x) = 3X**2 + 7X + 1 """
 return 3 * X**2 + 7* X + 1
```

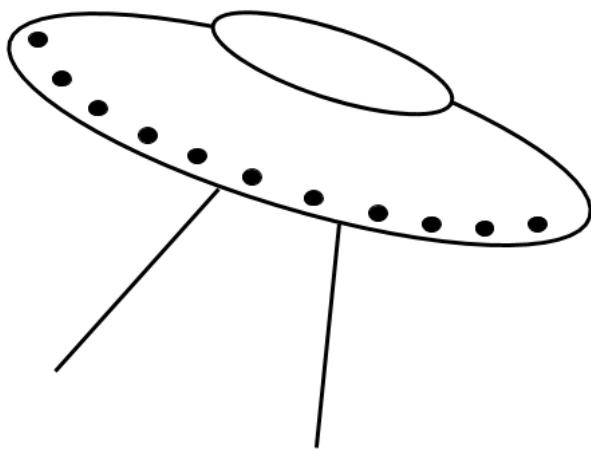
After writing *many* different Python functions like this one for many different polynomials, you can imagine it would be better to write a single Python function called `makePoly(coeffList)` that will take a list of coefficients as input and return a *function* that computes the polynomial defined by those coefficients. For example, here's `makePoly(coeffList)` in action:

```
In [1]: f = makePoly([3, 7, 1])
In [2]: f(10)
Out[2]: 371
In [3]: g = makePoly([5, 0, 0, 42])
In [4]: g(2)
Out[4]: 82
```

Aside from the `def makePoly(coeffList)` and docstring lines, this function need only be three lines long. Write it!

## CSforAll - 42andMe

### CS for All



CSforAll Web > Chapter3 > 42andMe

This website accompanies the textbook *CS For All* by Alvarado, Dodds, Kuenning, and Libeskind-Hadas

### 42andme

#### First, The Gratuitous Backstory...

You've been hired as a summer intern at the genetic testing company 42andme. Here's how 42andme works... A customer sends 42andme a saliva sample (and 42 dollars). The saliva is processed and the DNA is extracted (some of you have done this extraction in Biology 23 already!). 42andme then compares regions of the client's DNA to various "reference sequences". For example, if a certain region of your DNA is similar to a given reference sequence, that could indicate something about your ancestry, immunity or propensity to a certain disease, etc. The comparison between a region of your DNA and a reference sequence is done using the longest common subsequence (LCS) algorithm that we saw in class. (Although we mentioned that edit distance is sometimes a better measure of distance than LCS, if the two sequences being compared are of the same or similar length - which they will usually be in this case - LCS works very well!)

## A Reminder of LCS

In class we wrote the `LCS` function, which takes two strings as arguments and finds the length of their longest common subsequence. Here is the code that we developed:

```
def LCS(S1, S2):
 """Returns the length of the longest common subsequence of S1 and S1."""
 if S1 == "" or S2 == "":
 return 0
 else:
 if S1[0] == S2[0]:
 # Match...
 return 1 + LCS(S1[1:], S2[1:]) # ... so we get 1 match point and then recurse
 else:
 # No match...
 option1 = LCS(S1, S2[1:])
 option2 = LCS(S1[1:], S2)
 return max(option1, option2)
```

## Global Sequence Alignment, Part 1

While this `LCS` function computes the *length* of the longest common subsequence, 42andme needs more than that - they want to show their clients the *actual* longest common subsequence. To that end, you'll write an all new function called `fancyLCS(S1, S2)` which will take two strings `S1` and `S2` as arguments and return a list with three items: A number indicating the length of the longest common subsequence of these two strings, a copy of `S1`, and a copy of `S2`—but in the copies, the symbols that are not used in the longest common subsequence will be replaced by `#` symbols.

```
In [1]: fancyLCS("x", "y")
Out[1]: [0, '#', '#']
the LCS has length 0. Neither the "x" nor the "y" are used in the LCS.

In [2]: fancyLCS("spam", "")
Out[2]: [0, #####, ""]
the LCS has length 0. Not used: the 4 letters in "spam" or the (zero) letters in ""

In [3]: fancyLCS("spa", "m")
Out[3]: [0, ####, "#"]
nothing is used in either string, so we stamp everything out!

In [4]: fancyLCS("cat", "car")
Out[4]: [2, "ca#", "ca#"]
the "ca" is common but the "t" and "r" don't match

In [5]: fancyLCS("cat", "lca")
Out[5]: [2, "ca#", "#ca"]
the "ca" is still the longest common part

In [6]: fancyLCS("human", "chimpanzee")
Out[6]: [4, 'h#man', '#h#m#an##']
"human" and "chimpanzee" have an LCS of length 4.
The "u" in "human" is not used and many letters in "chimpanzee" are not used.
```

Build your `fancyLCS` function by simply modifying the `LCS` function that we wrote in class. Keep it as simple as possible, and do not have `fancyLCS` call any helper functions. Calling the built-in `len` function is fine, but `fancyLCS` should not call `LCS`; this would make it much more complicated than necessary.

By the way, if you want a string made of 3 copies of the string "spam" you can do `"spam" + "spam" + "spam"` or, equivalently, `"spam"*3`. While you might not want 3 copies of "spam", you probably will need some number of "#" symbols!

## Global Sequence Alignment, Part 2

Now, you'll create a very slightly different version of fancyLCS, called align, to do sequence *alignment*. Start by copying fancyLCS and modifying it slightly. This function will again return a list of three items: the length of the longest common subsequence, and two strings. These two returned strings will be essentially identical to the two original strings except that they will both have the same length and will contain a - (hyphen) symbol at any location where they mismatch.

Here are some examples of align in action:

```
In [1]: align("x", "y")
Out[1]: [0, 'x-', 'y']
The solution [0, '-x', 'y-'] is also fine

In [2]: align("spam", "")
Take a close look here -- this is a base case...
Out[2]: [0, 'spam', '----']
... notice that the second string is all ---- to have the same length as the first

In [3]: align("cat", "car")
Out[3]: [2, 'cat-', 'ca-r']
The solution [2, 'ca-t', 'car-'] is also fine

In [4]: align("hi", "ship")
Out[4]: [2, '-hi-', 'ship']
```

Notice that the two output strings always have the same length! This allows us to write one above the other and see how they align.

```
-hi-
ship
```

Notice that a longest common subsequence is found wherever the two strings agree in the same position. If two strings disagree at a given position, exactly one of those strings has a - in that position. Notice also that the entirety of the two supplied strings appears in the output, but with some added - symbols where there is disagreement.

Here is one more example using two DNA fragments:

```
In [1]: align("ATTGC", "GATC")
Out[1]: [3, '-ATTGC', 'GAT--C']
```

Just to help see what's going on here, let's line these two output strings up one above the other:

```
-ATTGC
GAT--C
```

You can see here that ATC is a longest common subsequence. In general, there may be several different equal-length longest common subsequences, which will result in several possible correct solutions. Your align function can report any single valid solution.

### Finally...

Your align function is finding what's called a "global alignment" of the two DNA strings. In the last example above, the solution

```
-ATTGC
GAT--C
```

indicates that these two DNA strings may have evolved from a common ancestor that had ATC at indices 1, 2, and 5 and other unknown nucleotides elsewhere. The two strings "ATTGC" and "GATC" may have evolved from this common ancestor but had some extra nucleotides inserted and others deleted.

Finally, use your align function to align Prof. Ran's spam42 gene against the reference sequence.

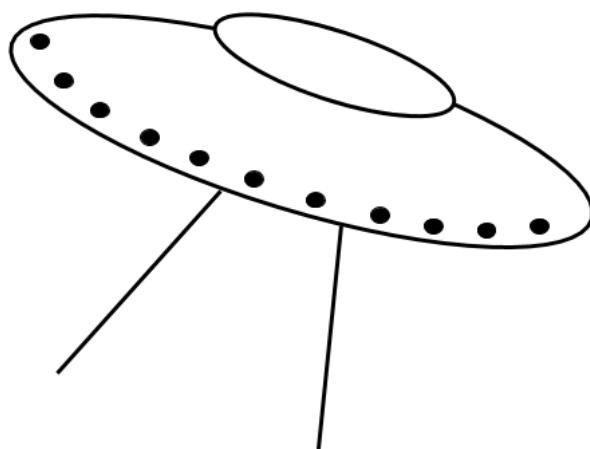
```
ran = "GTACGTCGATAACTG"
```

```
reference = "TGATCGTCATAACGT"
```

Place your alignment (the output strings from your `align` function, written one above the other) in a comment at the very top of your file.

# CSforAll - RNAFolding

## CS for All



CSforAll Web > Chapter3 > RNAFolding

This website accompanies the textbook *CS For All* by Alvarado, Dodds, Kuenning, and Libeskind-Hadas

### RNA Folding

#### Part 1: RNA Folding...

42andme has decided to enter a new market: Analyzing RNA to help predict immunity to certain diseases. Like DNA, RNA is a single-stranded polymer of nucleotides comprising nucleotides 'A', 'U', 'C', and 'G' (unlike DNA, which has a 'T' for thymine, RNA has a 'U' for uracil).

RNA folds to form complex structures, which have a variety of functions in the cell. The way that RNA folds can help predict its function. Mutations in the RNA can cause different folds which can help predict immunity or propensity to certain diseases. Your next job at 42andme is to write software that predicts how an RNA string will fold.

- RNA folds in a way that, in general, puts it into the most stable, low-energy state. A good first-order approximation, and the one that we will use here, is to assume that RNA folds in a way that maximizes the total number of paired bases.

- Any 'A' (adenine) on the RNA strand can pair with any 'U' (uracil) on the same strand, even if they are adjacent to each-other. Similarly, any 'G' (guanine) can pair with any 'C' (cytosine). When a base pairs with another, it cannot then pair a second time—it becomes inert.
- When two nucleotides pair to each other, they pinch together and any bases in between them are scrunched up into a loop. Within this loop, pairing can take place. Outside of the loop on any continuous strand, pairing can also take place. Pairing *cannot* take place between a nucleotide inside a loop and one outside the loop! (This is called the "no pseudoknots" assumption. In fact, pseudoknots do occur in some cases in folded RNA, but most RNA folding algorithms exclude them from consideration because they are believed to be best examined in the tertiary structure of RNA rather than at this secondary level.)

First, copy this little function into your file. It takes two nucleotides as input and returns `True` if those nucleotides are complementary (and thus can be matched to one another).

```
def complement(base1,base2):
 """Returns boolean indicating if two RNA bases are complementary."""
 if base1=="A" and base2=="U":
 return True
 elif base1=="U" and base2=="A":
 return True
 elif base1=="C" and base2=="G":
 return True
 elif base1=="G" and base2=="C":
 return True
 else:
 return False
```

Now, your task is to write a function called `fold(RNA)` that accepts an RNA nucleotide sequence—that is a string made up of the letters 'A', 'U', 'G', and 'C'—as its sole argument. The function then returns a single number, which is the maximum number of matches possible in a valid folding of this RNA string.

You'll need to use `map` and `filter`. (Please do not use `for` loops here. The reason is that 42andme has special hardware, purchased from Giigle, that allows `map` and `filter` to be parallelized and thus code with `map` and `filter` will run much faster on their computers than code using `for` loops.) You'll also want to use some `lambda` expressions within those `map` and `filter` statements to keep your code succinct and elegant!

Why `map`? Recall that like in the Giigle Maps problem from class, `map` allows us to explore many options (more than just the two options that use-it-or-lose-it would consider). Why `filter`? Consider the nucleotide at index 0 in the string. One option is to lose it. The other option is to use it. But, how should we use it? We can potentially match it with any complementary nucleotide that appears to its right in the string. The `filter` function can be used to filter (that is, to keep) the indices where complementary nucleotides (complementary to `RNA[0]`) are found.

*Here's one more very cool thing that you should know before you start.* Imagine that you have a list like this:

```
In [1]: myList = [[42, ["foo", "bar", "spam"]], [32, ["hi", "mom"]], [24, ["this", "is", "weird"]]]
In [2]: max(myList)
Out[2]: [42, ['foo', 'bar', 'spam']]
In [3]: min(myList)
Out[3]: [24, ['this', 'is', 'weird']]
```

Notice that each element of `myList` is itself a list. And, each of those lists has a number in the zeroeth position. When we use `max` or `min`, those functions compute the maximum or minimum based on the zeroeth elements in the lists. That's very useful in general and will be particularly useful to you here!

You're ready to write your `fold` function. It should be less than 10 lines long. (It's OK to be a bit longer, but this approximate line count is intended to have you avoid writing more complicated code than necessary.)

Test your function and make sure it produces the correct result on the following strings:

```
In [1]: fold("ACCCCCU")
Out[1]: 1

In [2]: fold("ACCCCGU")
Out[2]: 2
```

```
In [3]: fold("AAUUGCGC")
```

```
Out[3]: 4
```

```
In [4]: fold("ACUGAGCCU")
```

```
Out[4]: 3
```

```
In [5]: fold("ACUGAGCCCUGUUAGCUA")
```

```
Out[5]: 8
```

You may find the built-in `max` function useful. This function can be invoked with a pair of numbers as arguments, or it can take a whole list, as demonstrated here:

```
In [1]: max(42, 10)
```

```
Out[1]: 42
```

```
In [2]: max([13, 42, 18, 0, 7])
```

```
Out[2]: 42
```

Congratulations! When you've finished this, you've achieved complete proficiency with this week's material and you can submit your code and be done. If you'd like to pursue a deeper level of mastery of this material, you're encouraged to explore one or both of the bonus problems below!

### Optional Bonus Part A

Your `fold` function found the maximum number of matches possible in an RNA folding. In this optional problem, you'll write a new version of this function called `getFold` that returns a "care package" list in which the first element is the maximum number of matches and the second element is a list. Each element of that list is itself a list of two elements of the form  $[x, y]$ , where  $x$  and  $y$  are the indices of two nucleotides that are matched in an optimal folding. In other words, `getFold` lets us actually see the pairs that are matched in an optimal solution.

Here are some examples of `getFold` in action. **Keep in mind that in many cases there will be more than one possible solution (folding) for a given RNA sequence.** Therefore, if your solutions look different than the ones below, they may still be correct! However, since all optimal solutions will have the same number of matches, make sure that your solutions have the same number of matches as ours. Then, check that your solutions are valid by checking to see that the pairs of indices that are found are really matching pairs!

```
In [1]: getFold("ACCCCCU")
```

```
Out[1]: [1, [[0, 6]]]
```

```
In [2]: getFold("ACCCCGU")
```

```
Out[2]: [2, [[0, 6], [4, 5]]]
```

```
In [3]: getFold("AAUUGCGC")
```

```
Out[3]: [4, [[0, 3], [1, 2], [4, 5], [6, 7]]]
```

```
In [4]: getFold("ACUGAGCCU")
```

```
Out[4]: [3, [[1, 3], [4, 9], [5, 6]]]
```

```
In [5]: getFold("ACUGAGCCCUGUUAGCUA")
```

```
Out[5]: [8, [[0, 2], [3, 7], [5, 6], [8, 10], [11, 18], [12, 13], [14, 15], [16, 17]]]
```

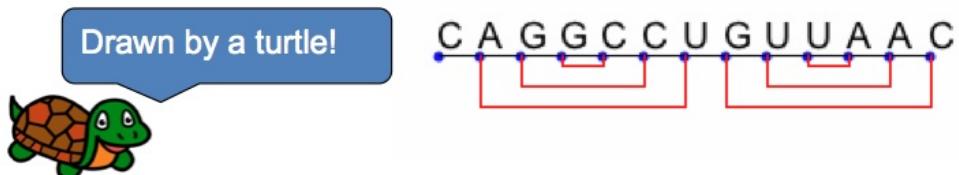
**Warning:** We've deliberately avoided using `for` loops and you may be tempted to use them here. However, `for` loops are not necessary because `map` and `filter` can do everything that you might want a `for` loop to do and using `map` and `filter` will result in cleaner code and much faster code if it's run on a high-performance computer that has special hardware for mapping and filtering in parallel!.

### Optional Bonus Part B

Your boss at 42andme points out that it would be *really* nice to actually see how the RNA folds! This optional bonus problem asks you to use the `turtle` package to render the solution found by `getFold`.

Exactly how you choose to draw the folding is up to you, but one possible solution is suggested below.

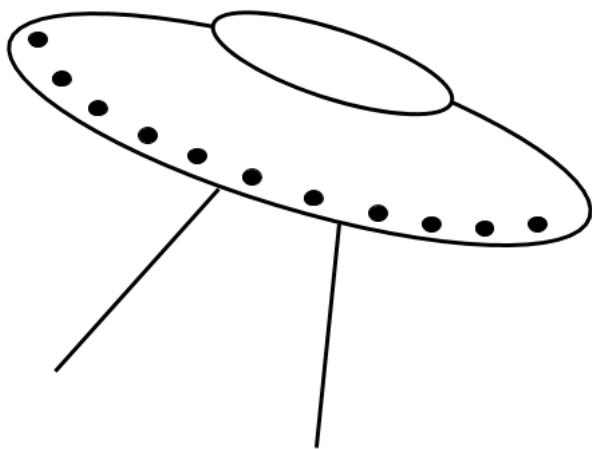
```
>>> getFold("CAGGCCUGUUUAC")
[6, [[1, 6], [2, 5], [3, 4], [7, 12], [8, 11], [9, 10]]]
```



You should write a function called `drawFold(RNA)` that takes an RNA string as input. It then calls `getFold(RNA)` to get the care package and then uses the `turtle` package to render (a fancy word for "draw") that folding in some clear way. You can get more details on the drawing tools available in the `turtle` package by Googling `python turtle`. You'll find documentation there on everything that the turtle can do (e.g., drawing in different colors, writing letters on the screen, drawing dots of different sizes, etc.).

# CSforAll - WordBreak

## CS for All



CSforAll Web > Chapter3 > WordBreak

This website accompanies the textbook *CS For All* by Alvarado, Dodds, Kuenning, and Libeskind-Hadas

### Word Break!

In this problem, you'll write a short and (hopefully) fun word game called Word Break! Here's the big idea. You'll have a list of valid words and a function that scores words (like the scrabble scoring function). For the sake of example though, let's use a short word list like this...

```
miniWordList = ["a", "am", "amp", "ample", "as", "asp", "at", "ate", "sat",
 "spa", "spam", "tea", "was", "wasp"]
```

Copy-and-paste that `miniWordList` into your file. It's now a global value that can be used by any functions you define in that file. This will be used for testing your code, but you're welcome later to use a more interesting list of valid words!

For now, we'll use Python's built-in `len` function to score words (a word comes in and a score comes out, where the score in this example is just the length of the word). But, that too can be changed as you'll see!

The game, called `wordBreak(string)` then takes a string as input. Let's see a few examples with explanations on the right.

```
wordBreak("waspxxxampleyyteazz") <-- We run the function with some string as input
Enter your best solution: was tea <-- Here, the program asked us to enter a string and we entered was tea
Your score was 6 <-- "was" and "tea" are found in the input string which begins with "was" and ends with "tea".
 <-- We got 6 points because we're using string length at the moment to score and "was" and "tea" have total length of 6
Best solution is [12, ['wasp', 'ample', 'tea']] <-- But, the program found a better solution with score of 12
```

Notice that a valid solution for the input (in this case the input was "waspampletea") is a sequence of strings, each of which is in our list of valid words (in this case `miniWordList`) and those strings are found left-to-right in the input string without sharing any letters. For example, for the string "waspampletea" , another valid solution would have been "spam" followed by "tea". Using length as the scoring function, that solution has a score of 7.

Here's a case where the player's solution wasn't valid!

```
>>> wordBreak("waspampletea")
Enter your best solution: foo bar
Your solution wasn't valid!
```

The problem here is that "foo" (and also "bar") are not found in the list of valid words. Moreover, they aren't found in the input string. Double whammy! No good.

Here's another case where the player's solution wasn't valid.

```
>>> wordBreak("waxxsp")
Enter your best solution: wasp
Your solution wasn't valid!
```

The problem here is that "wasp" isn't found in the string "waxxsp". The x's break things up!

Here's one more case where the player's solution wasn't valid.

```
>>> wordBreak("waspampletea")
Enter your best solution: waspam etea
Your solution wasn't valid!
```

Here the problem is that "waspam" and "etea" are not in the list of valid words (our `miniWordList` above) even though they are found left-to-right in the input string.

Here's the `wordBreak(string)` function. You can copy-and-paste it into your file. Don't worry! We haven't spoiled the fun. This isn't yet the interesting part of writing the game!

```
from functools import reduce

def wordBreak(string):
 scoreFunction = len
 # We're setting the scoring function. We can change that function to something else!
 wordList = miniWordList
 # We're setting the list of valid words. It can be changed to something else!
 userInput = input("Enter your best solution: ")
 # Get user input
 userList = userInput.split() # split the input string into a list of strings
 if check(userList, string, wordList):
 userScore = reduce(lambda X, Y: X+Y, map(scoreFunction, userList))
 print("Your score was ", userScore)
 best = showStringScore(string, scoreFunction, wordList, {})
 print("Best solution is ", best)
 else:
 print("Your solution wasn't valid!")
```

Notice that we can use different scoring functions and different word lists by simply changing the first two lines. For example, here we are changing the first line of the above function to use a scoring function that scores each word as the square of its length (thereby preferring even a few long words to many short words)...

```
scoreFunction = lambda X: len(X) ** 2
```

You can imagine other more interesting scoring functions. Similarly, the second line of code could be modified to use a different list of valid words.

## How Does User Input Work Again?

To better understand how user input works in Python, try running this code:

```
def getInput():
 userInput = input("Enter some words: ")
 print(userInput)
prints the userInput string
 print(userInput.split())
splits the userInput into a list of strings and prints that list
```

## Checking User Input

Notice that the game calls a function called `check(playerList, string, wordList)` as input. That function takes the list of words entered by the player, a string, and a list of valid words and returns `True` if the player's list of words is valid in the sense that they are all in the `wordList` and that they can be found left-to-right in the `string`. Here are some examples:

```
>>> check(["amp", "spa", "tea"], "xxxampyyyspazzteaxx", miniWordList)
```

```
True <- "amp", "spa", and "tea" were found left-to-right in the string that follows AND are each in the miniWordList
```

```
>>> check(["amp", "spa", "tea"], "xxxharveymuddyyspazzteaxx", miniWordList)
```

```
False <- "amp" wasn't found in the string
```

```
>>> check(["chocolate", "spa", "tea"], "chocolatexxxspaxxxtea", miniWordList)
```

```
False <- "chocolate" wasn't found in the miniWordList
```

The `check` function will be short (ours is under 10 lines long). You may need to find the index of one string in another string. You can use the `ind` function that you wrote in Homework 0 (or a small variant of it for strings) or you can use the built-in `index` method that works like this. (The syntax is weird, but we'll explain that later in the term):

```
>>> myString = "hello there!"
```

```
>>> myString.index("the") <- What is the starting index of "the" in myString?
```

```
6
```

Notice that `myString` is just a name that we used for this example. In this case, the leftmost occurrence of "the" in `myString` begins at index (i.e. position) 6.

## stringScore

Notice that the `wordBreak` game calls a function called `showStringStore(string, scoreFunction, wordList, memo)` which finds the best solution for the given string. We'll start with something slightly more modest - we'll write a function called `stringStore(string, scoreFunction, wordList, memo)` that takes as input the `string`, the scoring function, the list of valid words, and a memo (initially an empty dictionary) and only returns the score, not the actual words that make up that score. Here are some examples.

```
>>> stringScore("waspampletea", len, miniWordList, {})
```

```
12
```

This is telling us that the best solution, using length to score solutions, has length 12. We saw that earlier. It's "wasp", "ample" and "tea" of total length 12. Here are three more examples. Take a look at what each is doing to make sure that you're totally happy with what `stringScore` should do.

```
>>> stringScore("spamxxxspamxxx", len, ["spam", "chocolate"], {})
```

```
8
```

```
>>> stringScore("spamxxxspamxxx", lambda x: 1, ["spam", "chocolate"], {})
```

```
2
```

```
>>> stringScore("spamp", lambda x: len(x)**2, ["sp", "amp", "spam"], {})
```

```
16
```

To write `stringScore`, you'll use the lose-it-or-use-it technique. More accurately, you'll use the lose-it-or-use-it-or-use-it... technique. Huh? Notice that we can either use or lose the first letter of the input string. If we lose it, great! But, if we use it, then it must be the first letter of a consecutive run of letters that make up a string in the given list of valid words. So, there are potentially many different ways to use that first letter. This is very

similar in spirit to the RNA folding problem, where we either didn't use the first base or, if we did use it, we could use it in many different possible ways. As in RNA folding, filtering and mapping are super helpful here!

Our `stringScore` function is about 10 lines long. Don't write too much more code than that (other than the docstring, of course, which isn't part of the line count). If a line gets too long, you can continue it on the next line. That will make your code easier to read. Our 10 lines look like 13 or 14, because we broke up a single line of code over 3 or 4 lines for clarity.

### showStringScore

Your final task is to implement `showStringScore` that finds "care packages" comprising both the score and the solution itself.

Here's `showStringScore` in action, recapitulating the examples above:

```
>>> showStringScore("waspampletea", len, miniWordList, {})
[12, ['wasp', 'ample', 'tea']]
>>> showStringScore("spamxxxspamxxx", len, ["spam", "chocolate"], {})
[8, ['spam', 'spam']]
>>> showStringScore("spamxxxspamxxx", lambda x: 1, ["spam", "chocolate"], {})
[2, ['spam', 'spam']]
>>> showStringScore("spamp", lambda x: len(x)**2, ["sp", "amp", "spam"], {})
[16, ['spam']]
```

Our code is about 15 lines long (not including docstrings or breaking a single line of code over multiple lines, which is a good idea and brings ours to around 20 lines).

Now, play the `wordBreak` game and make sure that it works well. Hours (or at least minutes) of cheap entertainment!

### Want Even More?

If you want to do more with this, try using the `3esl.txt` file for a more comprehensive list of words, change the game so that it creates a random string of some length (rather than having the user enter their own string), and any other features that you think would be fun. Be creative! If you do this, please put a comment at the top of your file explaining what features you added and how to run your program.

If you import the `3esl.txt` file in your program, there's no need to submit it. We have it and we'll be ready for the import if you choose to use it.

# Chapter 4: Computer Organization — cs5book 1 documentation

## Navigation

- [index](#)
- [next |](#)
- [previous |](#)
- [cs5book 1 documentation »](#)

## Chapter 4: Computer Organization

*Computers are useless. They can only give you answers.*

—Pablo Picasso

### 4.1 Introduction to Computer Organization



*Hey!*

When we run a Python program, what's actually going on inside the computer? While we hope that recursion is feeling less like magic to you now, the fact that an electronic device can actually interpret and execute something as complicated as a recursive program may seem - what's the right word here? - *alien*. The goal of this chapter is to metaphorically pry the lid off a computer and peer inside to understand what's really going on there.

As you can imagine, a computer is a complicated thing. A modern computer has on the order of billions of transistors. It would be impossible to keep track of how all of those components interact. Consequently, computer scientists and engineers design and think about computers using what are called *multiple levels of abstraction*. At the lowest level are components like transistors - the fundamental building blocks of modern electronic devices. Using transistors, we can build higher level devices called *logic gates* - they are the next level of abstraction. From logic gates we can build electronic devices that add, multiply, and do other basic operations - that's yet another level of abstraction. We keep moving up levels of abstraction, building more complex devices from more basic ones.

As a result, a computer can be designed by multiple people, each thinking about their specific level of abstraction. One type of expert might work on designing smaller, faster, and more efficient transistors. Another might work on using those transistors - never mind precisely how they work - to design better components that are based on transistors. Yet another expert will work on deciding how to organize these components into even more complex units that perform key computational functions. Each expert is grateful to be standing (metaphorically) on the shoulders of another person's work at the next lower level of abstraction.

By analogy, a builder thinks about building walls out of wood, nails, and sheet rock. An architect designs houses using walls without worrying too much about how they are built. A city planner thinks about designing cities out of houses, without thinking too much how they are built, and so forth. This idea is called “abstraction” because it allows us to think about a particular level of design using the lower level ideas abstractly; that is, without having to keep in mind all of the specific details of what happens at that lower level.

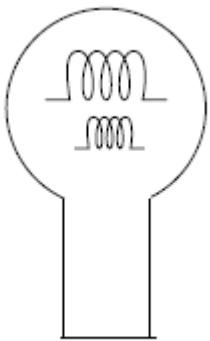


Figure 4.1: A three-way light bulb.

This seemingly simple idea of abstraction is one of the most important ideas in computer science. Not only are computers designed this way, but software is as well. Once we have basic workhorse functions like `map`, `reduce`, and others, we can use those to build more complex functions. We can use those more complex functions in many places to build even more complex software. In essence, we modularize so we can reuse good things to build bigger good things.

In this spirit, we'll start by looking at how data is represented in a computer. Next, we'll move up the levels of abstraction from transistors all the way up to a full-blown computer. We'll program that computer in its own "native language" and talk about how your Python program ultimately gets translated to that language. By the end of this chapter, we'll have a view of what happens when we run a program on our computer.

## 4.2 Representing Information

At the most fundamental level, a computer doesn't really know math or have any notion of what it means to compute. Even when a machine adds  $1+1$  to get  $2$ , it isn't *really* dealing with numbers. Instead, it's manipulating electricity according to specific rules.

To make those rules produce something that is useful to us, we need to associate the electrical signals inside the machine with the numbers and symbols that we, as humans, like to use.

### 4.2.1 Integers



*That's a shocking idea!*

The obvious way to relate electricity to numbers would be to assign a direct correspondence between voltage (or current) and numbers. For example, we could let zero volts represent the number 0, 1 volt be 1, 10 volts be 10, and so forth. There was a time when things were done this way, in so-called analog computers. But there are several problems with this approach, not the least of which would be the need to have a million-volt computer!



*Whose bright idea was this?*

Here's another approach. Imagine that we use a light bulb to represent numbers. If the bulb is off, the number is 0. If the bulb is on, the number is 1. That's fine, but it only allows us to represent two numbers.

All right then, let's upgrade to a "three-way" lamp. A three-way lamp really has four switch positions: off, and three increasing brightness levels. Internally, a three-way bulb has two filaments (Figure 4.1), one dim and one bright. For example, one might be 50 watts, and the other 100. By choosing neither of them, one, the other, or

both, we can get 0, 50, 100, or 150 watts worth of light. We could use that bulb to represent the numbers 0, 50, 100, and 150 or we could decide that the four levels represent the numbers 0, 1, 2, and 3.

Internally, a computer uses this same idea to represent integers. Instead of using 50, 100, and 200, as in our illuminating example above, computers use combinations of numbers that are powers of 2. Let's imagine that we have a bulb with a 20-watt filament, a 21-watt filament, and a 22-watt filament. Then we could make the number 0 by turning none of the filaments on, we could make 1 by turning on only the 20-watt filament, we could make 2 by turning on the 21-watt filament, and so forth, up to  $(2^0 + 2^1 + 2^2 = 7)$  using all three filaments.

Imagine now that we had the following four consecutive powers of 2 available to us:  $(2^0, 2^1, 2^2, 2^3)$ . Take a moment to try to write the numbers 0, 1, 2, and so forth as high as you can go by using zero or one of each of these powers of 2. Stop reading. We'll wait while you try this.

If all went well, you discovered that you could make all of the integers from 0 to 15 using 0 or 1 of each of these four powers of 2. For example, 13 can be represented as  $(2^0 + 2^2 + 2^3)$ . Written another way, this is:

$$[13 = 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0]$$



*There are 10 kinds of people in this world: Those who understand binary and those who don't.*

Notice that we've written the larger powers of 2 on the left and the smaller powers of two on the right. This convention is useful, as we'll see shortly. The 0's and 1's in the equation above - the *coefficients* on the powers of 2 - indicate whether or not the particular power of 2 is being used. These 0 and 1 coefficients are called *bits*, which stands for *binary digits*. *Binary* means "using two values"- here, the 0 and the 1 are the two values.

It's convenient to use the sequence of bits to represent a number, without explicitly showing the powers of two. For example, we would use the bit sequence 1101 to represent the number 13 since that is the order in which the bits appear in the equation above. Similarly 0011 represents the number 3. We often leave off the leading (that is, the leftmost) 0's, so we might just write this as 11 instead.

The representation that we've been using here is called base 2 because it is built on powers of 2. Are other bases possible? Sure! You use base 10 every day. In base 10, numbers are made out of powers of 10 and rather than just using 0 and 1 as the coefficients, we use 0 through 9. For example, the sequence 603 really means

$$[6 \cdot 10^2 + 0 \cdot 10^1 + 3 \cdot 10^0]$$

Other bases are also useful. For example, the Yuki Native American tribe, who lived in Northern California, used base 8. In base 8 we use powers of 8 and the coefficients that we use are 0 through 7. So, for example, the sequence 207 in base 8 means

$$[2 \cdot 8^2 + 0 \cdot 8^1 + 7 \cdot 8^0]$$

which is 135 (in base 10). It is believed that the Yukis used base 8 because they counted using the eight slots *between* their fingers.



*In Star Wars, the Hutts have 8 fingers and therefore also count in base 8.*

Notice that when we choose some base  $(b)$  (where  $(b)$  is some integer greater than or equal to 2), the digits that we use as coefficients are 0 through  $(b - 1)$ . Why? It's not hard to prove mathematically that when we use this convention, every positive integer between 0 and  $(bd - 1)$  can be represented using  $(d)$  digits. Moreover, every integer in this range has a *unique* representation, which is handy since it avoids the headache of having multiple representations of the same number. For example, just as 42 has no other representation in base 10, the number 1101 in base 2 (which we saw a moment ago is 13 in base 10) has no other representation in base 2.

Many older or smaller computers use 32 bits to represent a number in base 2; sensibly enough, we call them "32-bit computers." Therefore, we can uniquely represent all of the positive integers between 0 and  $2^{32} - 1$ , or

4, 294, 967, 295. A powerful modern computer that uses 64 bits to represent each number can represent integers up to  $2^{64} - 1$ , which is roughly 18 *quadrillion*.

#### 4.2.2 Arithmetic



*We'll have sum fun with addition.*

Arithmetic in base 2, base 8, or base 42 is analogous to arithmetic in base 10. For example, let's consider addition. In base 10, we simply start at the rightmost column, where we add those digits and “carry” to the next column if necessary. For example, when adding  $(17+25)$ ,  $(5+7 = 12)$  so we write down a 2 in the rightmost position and carry the 1. That 1 represents a 10 and is therefore propagated, or carried, to the next column, which represents the “10’s place.”



*Although I find coming to agreement with myself to be easier in general.*

Addition in base 2 is nearly identical. Let's add 111 (which, you'll recall is 7 in base 10) and 110 (which is 6 in base 10). We start at the rightmost (or “least significant”) column and add 1 to 0, giving 1. Now we move to the next column, which is the  $(2^1)$  position or “2’s place”. Each of our two numbers has a 1 in this position.  $1 + 1 = 2$  but we only use 0’s and 1’s in base 2, so in base 2 we would have  $1 + 1 = 10$ . This is analogous to adding 7 + 3 in base 10: rather than writing 10, we write a 0 and carry the 1 to the next column. Similarly, in base 2, for  $1 + 1$  we write 0 and carry the 1 to the next column. Do you see why this works? Having a 2 in the “2’s place” is the same thing as having a 1 in the “4’s place”. In general having a 2 in the column corresponding to  $(2^i)$  is the same as having a 1 in the next column, the one corresponding to  $(2^{i+1})$  since  $(2 \cdot 2^i = 2^{i+1})$ .

**Takeaway message:** *Addition, subtraction, multiplication, and division in your favorite base are all analogous to those operations in base 10!*



*We'll try not to get carried away with these examples, but you should try adding a few numbers in base 2 to make sure that it makes sense to you.*

#### 4.2.3 Letters and Strings

As you know, computers don't only manipulate numbers; they also work with symbols, words, and documents. Now that we have ways to represent numbers as bits, we can use those numbers to represent other symbols.



*In ASCII, the number 42 represents the asterisk (\*)*

It's fairly easy to represent the alphabet numerically; we just need to come to an agreement, or “convention,” on the encoding. For example, we might decide that 1 should mean “A”, 2 should mean “B”, and so forth. Or we could represent “A” with 42 and “B” with 97; as long as we are working entirely within our own computer system it doesn't really matter.

But if we want to send a document to a friend, it helps to have an agreement with more than just ourselves. Long ago, the American National Standards Institute (ANSI) published such an agreement, called ASCII (pronounced

“as key” and standing for the American Standard Code for Information Interchange). It defined encodings for the upper- and lower-case letters, the numbers, and a selected set of special characters - which, not coincidentally, happen to be precisely the symbols printed on the keys of a standard U.S. keyboard.

Note

### Negative Thinking

We’ve successfully represented numbers in base 2 and done arithmetic with them. But all of our numbers were positive. How about representing negative numbers? And what about fractions?

Let’s start with negative numbers. One fairly obvious approach is to reserve one bit to indicate whether the number is positive or negative; for example, in a 32-bit computer we might use the leftmost bit for this purpose: Setting that bit to 0 could mean that the number represented by the remaining 31 bits is positive. If that leftmost bit is a 1 then the remaining number would be considered to be negative. This is called a *sign-magnitude* representation. The price we pay is that we lose half of our range (since we now have only 31 bits, in our example, to represent the magnitude of the number). While we don’t mean to be too negative here, a bigger problem is that it’s tricky to build computer circuits to manipulate sign-magnitude numbers. Instead, we use a system called two’s complement.

The idea behind two’s complement is this: It would be very convenient if the representation of a number plus the representation of its negation added up to 0. For example, since 3 added to -3 is 0, it would be nice if the binary representation of 3 plus the binary representation of -3 added up to 0. We already know what the binary representation of 3 is 11. Let’s imagine that we have an 8-bit computer (rather than 32- or 64-bit), just to make this example easier. Then, including the leading 0’s, 3 would be represented as 00000011. Now, how could we represent -3 so that its representation added to 00000011 would be 0, that is 00000000?

Notice that if we “flip” all of the bits in the representation of 3, we get 11111100. Moreover,  $00000011 + 11111100 = 11111111$ . If we add one more to this we get  $11111111 + 00000001$  and when we do the addition with carries we get 100000000; a 1 followed by eight 0’s. If the computer is only using eight bits to represent each number then that leftmost ninth bit will not be recorded! So, what will be saved is just the lower eight bits, 00000000, which is 0. So, to represent -3, we can simply take the representation of 3, flip the bits, and then add 1 to that. (Try it out to make sure that you see how this works.) In general, the representation of a negative number in the two’s complement system involves flipping the bits of the positive number and then adding 1.



“*ord*” stands for “ordinal”. You can think of this as asking for the “ordering number” of the symbol

You can look up the ASCII encoding on the web. Alternatively, you can use the Python function *ord* to find the numerical representation of any symbol. For example:

```
>>> ord('*')
42
>>> ord('9')
57
```



4 bits are sometimes called a “nybble”; we take no responsibility for this pathetically nerdy pun.

Why is the ordinal value of ‘9’ reported as 57? Keep in mind that the 9, in quotes, is just a character like the asterisk, a letter, or a punctuation symbol. It appears as character 57 in the ASCII convention. Incidentally, the inverse of *ord* is *chr*. Typing *chr*(42) will return an asterisk symbol and *chr*(57) will return the symbol ‘9’.

Each character in ASCII can be represented by 8 bits, a chunk commonly referred to as a “byte.” Unfortunately, with only 8 bits ASCII can only represent 256 different symbols. (You might find it entertaining to pause here and write a short program that counts from 0 to 255 and, for each of these numbers, prints out the ASCII symbol corresponding to that number. You’ll find that some of the symbols printed are “weird” or even invisible. Snoop on the Web to learn more about why this is so.)

Note

### Piecing It Together

How about representing fractions? One approach (often used in video and music players) is to establish a convention that everything is measured in units of some convenient fraction (just as our three-way bulb works in units of 50 watts). For example, we might decide that everything is in units of 0.01, so that the number 100111010 doesn't represent 314 but rather represents 3.14.

However, scientific computation often requires both more precision and a wider range of numbers than this strategy affords. For example, chemists often work with values as on the order of  $\backslash(10^{23}\backslash)$  or more (Avogadro's number is approximately  $\backslash(6.02 \times 10^{23}\backslash)$ ), while a nuclear physicist might use values as small as  $\backslash(10^{-12}\backslash)$  or even smaller.

Imagine that we are operating in base 10 and we have only eight digits to represent our numbers. We might use the first six digits to represent a number, with the convention that there is an implicit 0 followed by a decimal point, just before the first digit. For example, the six digits 123456 would represent the number 0.123456. Then, the last two digits could be used to represent the exponent on the power of 10. So, 12345678 would represent  $\backslash(0.123456 \times 10^{78}\backslash)$ . Computers use a similar idea to represent fractional numbers, except that base 2 is used instead of base 10.

It may seem that 256 symbols is a lot, but it doesn't provide for accented characters used in languages like French (Français), let alone the Cyrillic or Sanskrit alphabets or the many thousands of symbols used in Chinese and Japanese.



*There are even unofficial Unicode symbols for Klingon!*

To address that oversight, the International Standards Organization (ISO) eventually devised a system called Unicode, which can represent every character in every known language, with room for future growth. Because Unicode is somewhat wasteful of space for English documents, ISO also defined several “Unicode Transformation Formats” (UTF), the most popular of which is *UTF-8*. You may already be using UTF-8 on your computer, but we won't go into the gory details here.

Of course, individual letters aren't very interesting. Humans normally like to string letters together, and we've seen that Python uses a data type called “strings” to do this. It's easy to do that with a sequence of numbers; for example, in ASCII the sequence 99, 104, 111, 99, 111, 108, 97, 116, 101 translates to “chocolate”. The only detail missing is that when you are given a long string of numbers, you have to know when to stop; a common convention is to include a “length field” at the very beginning of the sequence. This number tells us how many characters are in the string. (Python uses a length field, but hides it from us to keep the string from appearing cluttered.)

#### 4.2.4 Structured Information

Using the same concepts, we can represent almost any information as a sequence of numbers. For example, a picture can be represented as a sequence of colored dots, arranged in rows. Each colored dot (also known as a “picture element” or pixel) can be represented as three numbers giving the amount of red, green, and blue at that pixel. Similarly, a sound is a time sequence of “sound pressure levels” in the air. A movie is a more complex time sequence of single pictures, usually 24 or 30 per second, along with a matching sound sequence.



*That would be more than a bit annoying!*

That's the level of abstraction thing again! Bits make up numbers, numbers make up pixels, pixels make up pictures, pictures make up movies. A two-hour movie can require several billion bits, but nobody who is making or watching a movie wants to think about all of those bits!

## 4.3 Logic Circuitry



*My favorite food chain sells donuts.*

Now that we have adopted some conventions on the representation of data it's time to build devices that manipulate data. We'll start at a low level of abstraction of transistors and move up the metaphorical "food chain" to more complex devices, then units that can perform addition and other basic operations, and finally to a full-blown computer.

### 4.3.1 Boolean Algebra

In Chapter 2 we talked about Boolean variables - variables that take the value `True` or `False`. It turns out that Booleans are at the very heart of how a computer works.

As we noted in the last section, it's convenient to represent our data in base 2, also known as binary. The binary system has two digits, 0 and 1 just as Boolean variables have two values, `False` and `True`. In fact, we can think of 0 as corresponding to `False` and 1 as `True` as corresponding to `True`. The truth is, that Python thinks about it this way too. One funny way to see this is as follows:

```
>>> False + 42
42
>>> True + 2
3
```



*We are writing these in upper-case letters to indicate that we are talking about operations on bits-0's and 1's - rather than Python's built-in and, or, and not.*

Weird - but there it is: in Python `False` is really 0 and `True` is really 1. By the way, in many programming languages this is not the case. In fact, programming language designers have interesting debates about whether it's a good idea or not to have `False` and `True` be so directly associated with the numbers 0 and 1. On the one hand, that's how we often think about `False` and `True`. On the other hand, it can result in confusing expressions like `False + 42` which are hard to read and prone to introducing programmer errors.

With the Booleans `True` and `False` we saw that we could use the operations `and`, `or`, and `not` to build up more interesting Boolean expressions. For example, `True and True` is the same as `True` while `True or False` is `True` and `not True` is `False`. Of course, we can now emulate these three operations for 0 and 1.  $1 \text{ AND } 1 = 1$ ,  $1 \text{ OR } 0 = 1$ , and  $\text{NOT } 1 = 0$ .

Although your intuition of `AND`, `OR`, and `NOT` is probably fine, we can be very precise about these three operations by defining them with a *truth table*: a listing of all possible combinations of values of the input variables, together with the result produced by the function. For example, the truth table for `AND` is:

| <i>x</i> | <i>y</i> | <i>x AND y</i> |
|----------|----------|----------------|
| 0        | 0        | 0              |
| 0        | 1        | 0              |
| 1        | 0        | 0              |
| 1        | 1        | 1              |

In Boolean notation, `AND` is normally represented as multiplication; an examination of the above table shows that as long as *x* and *y* are either 0 or 1, *x AND y* is in fact identical to multiplication. Therefore, we will often write *xy* to represent *x AND y*.

**Takeaway message:** `AND` is 1 if and only if both of its arguments are 1.

`OR` is a two-argument function that is 1 if either of its arguments are 1. The truth table for `OR` is:

| $x$ | $y$ | $x \text{ OR } y$ |
|-----|-----|-------------------|
| 0   | 0   | 0                 |
| 0   | 1   | 1                 |
| 1   | 0   | 1                 |
| 1   | 1   | 1                 |

OR is normally written using the plus sign:  $\backslash(x + y)$ . The first three lines of the above table are indeed identical to addition, but note that the fourth is different.

**Takeaway message:** OR is 1 if either of its arguments is 1.

Finally, NOT is a one-argument function that produces the opposite of its argument. The truth table is:

| $x$ | NOT $x$ |
|-----|---------|
| 0   | 1       |
| 1   | 0       |

NOT is normally written using an overbar, e.g.  $\backslash(\bar{x})$

#### 4.3.2 Making Other Boolean Functions

Amazingly, any function of Boolean variables, no matter how complex, can be expressed in terms of AND, OR, and NOT. No other operations are required because, as we'll see, any other operation could be made out of AND, OR, and NOTs. In this section we'll show how to do that. This fundamental result will allow us to build circuits to do things like arithmetic and, ultimately, a computer. For example, consider a function described by the truth table below. This function is known as "implication" and is written  $\backslash(x \rightarrow y)$ .

| $x$ | $y$ | $x \rightarrow y$ |
|-----|-----|-------------------|
| 0   | 0   | 1                 |
| 0   | 1   | 1                 |
| 1   | 0   | 0                 |
| 1   | 1   | 1                 |

This function can be expressed as  $\backslash(\bar{x} + xy)$ . To see why, try building the truth table for  $\backslash(\bar{x} + xy)$ . That is, for each of the four possible combinations of  $x$  and  $y$ , evaluate  $\backslash(\bar{x} + xy)$ . For example, when  $\backslash(x = 0)$  and  $\backslash(y = 0)$ , notice that  $\backslash(\bar{x} + xy)$  is 1. Since the OR of 1 and anything else is always 1, we see that  $\backslash(\bar{x} + xy)$  evaluates to 1 in this case. Aha! This is exactly the value that we got in the truth table above for  $\backslash(x = 0)$  and  $\backslash(y = 0)$ . If you continue doing this for the next three rows, you'll see that values of  $\backslash(x \rightarrow y)$  and  $\backslash(\bar{x} + xy)$  always match. In other words, they are identical. This method of enumerating the output for each possible input is a fool-proof way of proving that two functions are identical, even if it is a bit laborious.

For simple Boolean functions, it's often possible to invent an expression for the function by just inspecting the truth table. However, it's not always so easy to do this, particularly when we have Boolean functions with more than two inputs. So, it would be nice to have a systematic approach for building expressions from truth tables. The *minterm expansion principle* provides us with just such an approach.



*Perhaps "minterms" should be called "happyterms".*

We'll see how the minterm expansion principle works through an example. Specifically, let's try it out for the truth table for the implication function above. Notice that when the inputs are  $\backslash(x = 1)$  and  $\backslash(y = 0)$ , the truth table tells us that the output is 0. However, for all three of the other rows (that is pairs of inputs), the output is more "interesting" - it's 1. We'll build a custom-made logical expression for each of these rows with a 1 as the output. First, consider the row  $\backslash(x = 0) ; \backslash(y = 0)$ . Note that the expression  $\backslash(\bar{x} \cdot \bar{y})$  evaluates to 1 for this particular pair of inputs because the NOT of 0 is 1 and the AND of 1 and 1 is 1. Moreover, notice that for every other possible pair of values for  $x$  and  $y$  this term  $\backslash(\bar{x} \cdot \bar{y})$  evaluates to 0. Do

you see why? The only way that  $\bar{x}\bar{y}$  can evaluate to 1 is for  $x$  to be 0 (and thus for  $y$  to be 0) and for  $\bar{y}$  to be 1 (since we are computing the AND here and AND outputs 1 only if both of its inputs are 1). The term  $\bar{x}\bar{y}$  is called a minterm. You can think of it as being custom-made to make the inputs  $(x = 0); (y = 0)$  “happy” (evaluate to 1) and does nothing for every other pair of inputs.

We’re not done yet! We now want a minterm that is custom-made to evaluate to 1 for the input  $(x = 0); (y = 1)$  and evaluates to 0 for every other pair of input values. Take a moment to try to write such a minterm. It’s  $\bar{x}y$ . This term evaluates to 1 if and only if  $(x = 0)$  and  $(y = 1)$ . Similarly, a minterm for  $(x = 1); (y = 1)$  is  $xy$ . Now that we have these minterms, one for each row in the truth table that contains a 1 as output, what do we do next? Notice that in our example, our function should output a 1 if the first minterm evaluates to 1 or the second minterm evaluates to 1 or the third minterm evaluates to 1. Also notice the words “or” in that sentence. We want to OR the values of these three minterms together. This gives us the expression  $\bar{x}\bar{y} + \bar{x}y + xy$ . This expression evaluates to 1 for the first, second, and fourth rows of the truth table as it should. How about the third row, the “uninteresting” case where  $(x = 1); (y = 1)$  should output 0. Recall that each of the minterms in our expression was custom-made to make exactly one pattern “happy”. So, none of these terms will make the  $(x = 1); (y = 1)$  “happy” and thus, for that pair of inputs, our newly minted expression outputs a 0 as it should!

It’s not hard to see that this minterm expansion principle works for every truth table. Here is the precise description of the process:

1. Write down the truth table for the Boolean function that you are considering.
2. Delete all rows from the truth table where the value of the function is 0.
3. For each remaining row we will create something called a “minterm” as follows:
  - (a) For each variable that has a 1 in that row, write the name of the variable. If the input variable is 0 in that row, write the variable with a negation symbol to NOT it.
  - (b) Now AND all of these variables together.
4. Combine all of the minterms for the rows using OR .

You might have noticed that this general algorithm for converting truth tables to logic expressions only uses AND, OR, and NOT operations. It uses NOT and AND to construct each minterm and then it uses OR to “glue” these minterms together. This effectively proves that AND, OR, and NOT suffice to represent any Boolean function!

The minterm expansion principle is a recipe - it’s an *algorithm*. In fact, it can be implemented on a computer to automatically construct a logical expression for any truth table. In practice, this process is generally done by computers. Notice, however that this algorithm doesn’t necessarily give us the simplest expression possible. For example, for the implication function, we saw that the expression  $\bar{x}+xy$  is correct. However, the minterm expansion principle produced the expression  $\bar{x}\bar{y} + \bar{x}y + xy$ . These expressions are logically equivalent, but the first one is undeniably shorter. Regrettably, the so-called “minimum equivalent expressions” problem of finding the shortest expression for a Boolean function is very hard. In fact, a Harvey Mudd College graduate, David Buchfuhrer, recently showed that the minimum equivalent expressions problem is provably as hard as some of the hardest (unsolved) problems in mathematics and computer science. Amazing but true!



*That means that computers are helping design other computers! That seems profoundly amazing to me.*

### 4.3.3 Logic Using Electrical Circuits

Next, we’d like to be able to perform Boolean functions in hardware. Let’s imagine that our basic “building block” is an electromagnetic switch as shown in Figure 4.2. There is *always* power supplied to the switch (as shown in the upper left) and there is a spring that holds a movable “arm” in the up position. So, normally, there is no power going to the wire labeled “output”. The “user’s” input is indicated by the wire labeled “input.” When the input power is off (or “low”), the electromagnet is not activated and the movable arm remains up, and output is 0. When the input is on (or “high”), the electromagnet is activated, causing the movable arm to swing downwards and power to flow to the output wire. Let’s agree that a “low” electrical signal corresponds to the number 0 and a “high” electrical signal corresponds to the number 1. Now, let’s build a device for computing the AND function using switches. We can do this as shown in Figure 4.3 where there are two switches in series.

In this figure, we use a simple representation of the switch without the power or the ground. The inputs are  $\langle x \rangle$  and  $\langle y \rangle$ , so when  $\langle x \rangle$  is 1, the arm of the first switch swings down and closes the switch, allowing power to flow from the left to the right. Similarly, when  $\langle y \rangle$  is 1, that arm of the second switch swings down, allowing power to flow from left to right. Notice that when either or both of the input  $\langle x, y \rangle$  are 0, at least one switch remains open and there is no electrical signal flowing from the power source to the output. However, when both  $\langle x \rangle$  and  $\langle y \rangle$  are 1 both switches close and there is a signal, that is a 1, flowing to the output. This is a device for computing  $\langle x \rangle \text{ AND } \langle y \rangle$ . We call this an AND gate.

Similarly, the circuit in Figure 4.4 computes  $\langle x \rangle \text{ OR } \langle y \rangle$  and is called an OR gate. The function NOT  $\langle x \rangle$  can be implemented by constructing a switch that conducts if and only  $\langle x \rangle$  is 0.

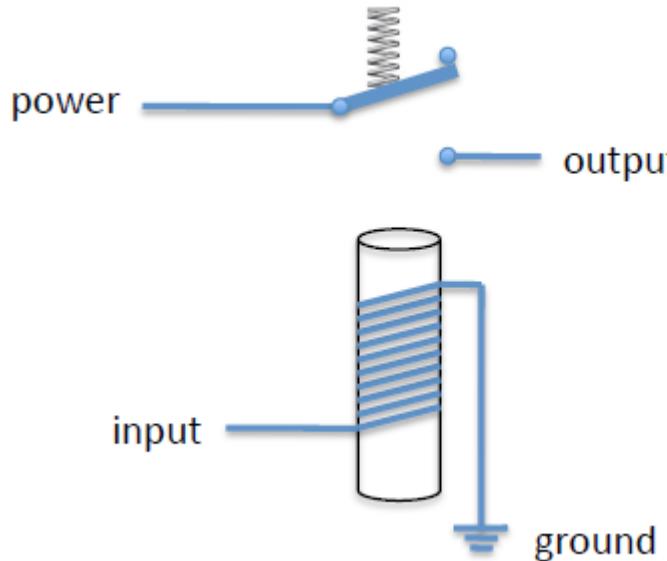


Figure 4.2: An electromagnetic switch.

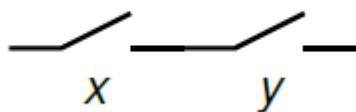


Figure 4.3: An AND gate constructed with switches.

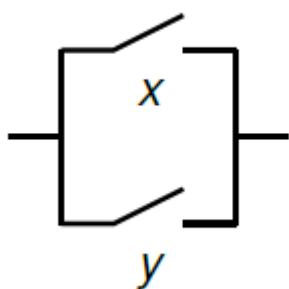


Figure 4.4: An OR gate constructed with switches.



*Those gate shapes are weird. I'm on the fence about whether I like them or not.*

While computers based on electromechanical switches were state-of-the-art in the 1930's, computers today are built with transistorized switches that use the same principles but are much smaller, much faster, more reliable, and more efficient. Since the details of the switches aren't terribly important at this level of abstraction, we represent, or "abstract", the gates using symbols as shown in Figure 4.5

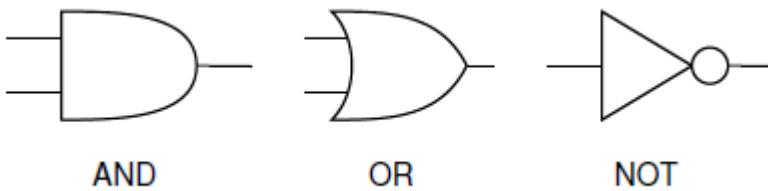


Figure 4.5: Symbols used for AND, OR, and NOT gates.

We can now build circuits for any Boolean function! Starting with the truth table, we use the minterm expansion principle to find an expression for the function. For example, we used the minterm expansion principle to construct the expression  $(\bar{x}\bar{y} + \bar{x}y + xy)$  for the implication function. We can convert this into a circuit using AND, OR, and NOT gates as shown in Figure 4.6.

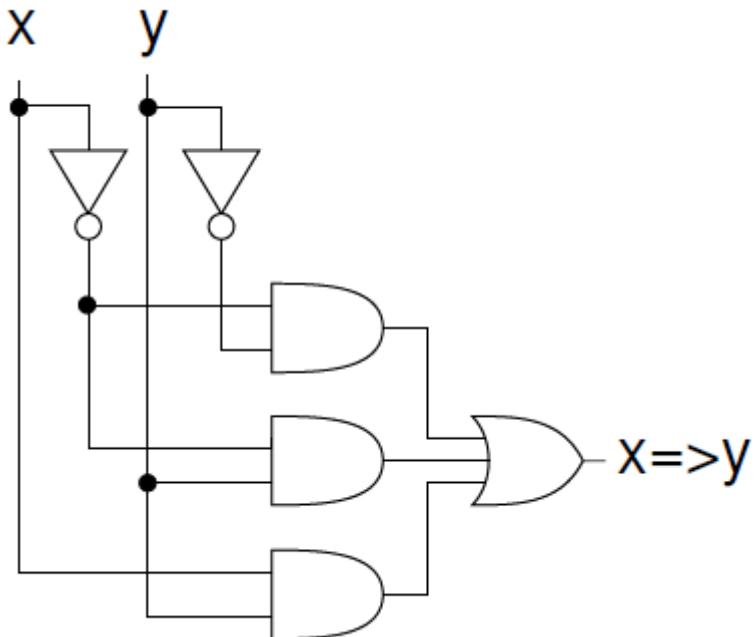


Figure 4.6: A circuit for the implication function.

#### 4.3.4 Computing With Logic

Now that we know how to implement functions of Boolean variables, let's move up one more level of abstraction and try to build some units that do arithmetic. In binary, the numbers from 0 to 3 are represented as 00, 01, 10, and 11. We can use a simple truth table to describe how to add two two-bit numbers to get a three-bit result:

| $x$ | $y$ | $x + y$ |
|-----|-----|---------|
| 00  | 00  | 000     |
| 00  | 01  | 001     |
| 00  | 10  | 010     |
| :   | :   | :       |
| 01  | 10  | 011     |
| 01  | 11  | 100     |
| :   | :   | :       |
| 11  | 11  | 110     |

In all, this truth table contains sixteen rows. But how can we apply the minterm expansion principle to it? The trick is to view it as three tables, one for each bit of the output. We can write down the table for the rightmost output bit separately, and then create a circuit to compute that output bit. Next, we can repeat this process for the middle output bit. Finally, we can do this one more time for the leftmost output bit. While this works, it is much more complicated than we would like! If we use this technique to add two 16-bit numbers, there will be  $\backslash(2^{\{32\}}\backslash)$  rows in our truth table resulting in several *billion* gates.



*Ouch!*

Fortunately, there is a much better way of doing business. Remember that two numbers are added (in any base) by first adding the digits in the rightmost column. Then, we add the digits in the next column and so forth, proceeding from one column to the next, until we're done. Of course, we may also need to carry a digit from one column to the next as we add.

We can exploit this addition algorithm by building a relatively simple circuit that does just one column of addition. Such a device is called a *full adder*, (admittedly a funny name given that it's only doing one column of addition!). Then we can "chain" 16 full adders together to add two 16-bit numbers or chain 64 copies of this device together if we want to add two 64-bit numbers. The resulting circuit, called a *ripple-carry adder*, will be much simpler and smaller than the first approach that we suggested above. This modular approach allows us to first design a component of intermediate complexity (e.g. the full adder) and use that design to design a more complex device (e.g. a 16-bit adder). Aha! Abstraction again!

The full adder takes three inputs: The two digits being added in this column (we'll call them  $\backslash(x\backslash)$  and  $\backslash(y\backslash)$ ) and the "carry in" value that was propagated from the previous column (we'll call that  $\backslash(c_{\{\backslash mbox{in}\}}\backslash)$ ). There will be two outputs: The sum (we'll call that  $\backslash(z\backslash)$ ) and the "carry out" to be propagated to the next column (we'll call that  $\backslash(c_{\{\backslash mbox{out}\}}\backslash)$ ). We suggest that you pause here and build the truth table for this function. Since there are three inputs, there will be  $\backslash(2^{\{3} = 8\backslash)$  rows in the truth table. There will be two columns of output. Now, treat each of these two output columns as a separate function. Starting with the first column of output, the sum  $\backslash(z\backslash)$ , use the minterm expansion principle to write a logic expression for  $\backslash(z\backslash)$ . Then, convert this expression into a circuit using AND, OR, and NOT gates. Then, repeat this process for the second column of output,  $\backslash(c_{\{\backslash mbox{out}\}}\backslash)$ . Now you have a full adder! Count the gates in this adder - it's not a very big number.

Finally, we can represent this full adder abstractly with a box that has the three inputs on top and the two outputs on the bottom. We now chain these together to build our ripple-carry adder. A 2-bit ripple-carry adder is shown in Figure 4.7. How many gates would be used in total for a 16-bit ripple-carry adder? It's in the hundreds rather than the billions required in our first approach!

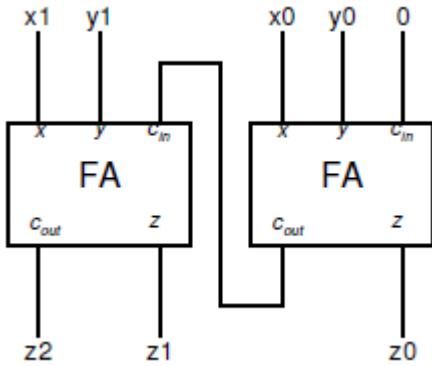


Figure 4.7: A 2-bit ripple-carry adder. Each box labeled “FA” is a full adder that accepts two input bits plus a carry, and produces a single output bit along with a carry into the next FA.

Now that we’ve built a ripple-carry adder, it’s not a big stretch to build up many of the other kinds of basic features of a computer. For example, consider building a multiplication circuit. We can observe that multiplication involves a number of addition steps. Now that we have an addition module, that is an abstraction that we can use to build a multiplier!

**Take Away** *Using the minterm expansion principle and modular design, we can now build virtually all of the major parts of a computer.*

#### 4.3.5 Memory



*I'm glad that you didn't forget this part!*

There is one important aspect of a computer that we haven’t seen how to design: memory! A computer can store data and then fetch those data for later use. (“Data” is the plural of “datum”. Therefore, we say “those data” rather than “that data.”) In this section we’ll see how to build a circuit that stores a single bit (a 0 or 1). This device is called a *latch* because it allows us to “lock” a bit and retrieve it later. Once we’ve built a latch, we can abstract that into a “black box” and use the principle of modular design to assemble many latches into a device that stores a lot of data.

A latch can be created from two interconnected NOR gates: NOR is just OR followed by NOT. That is, its truth table is exactly the opposite of the truth table for OR as shown below.

| x | y | x NOR y |
|---|---|---------|
| 0 | 0 | 1       |
| 0 | 1 | 0       |
| 1 | 0 | 0       |
| 1 | 1 | 0       |

A NOR gate is represented symbolically as an OR gate with a little circle at its output (representing negation).

Now, a latch can be constructed from two NOR gates as shown in Figure 4.8. The input  $\setminus(S\setminus)$  is known as “set” while the input  $\setminus(R\setminus)$  is known as “reset”. The appropriateness of these names will become evident in a moment.

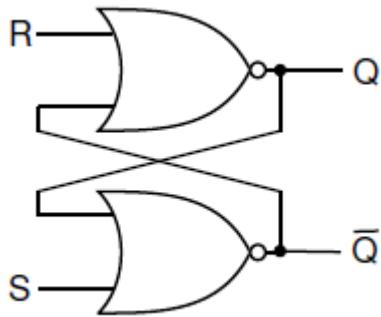


Figure 4.8: A latch built from two NOR gates.

What in the world is going on in this circuit!? To begin with, suppose that all of  $\langle R \rangle$ ,  $\langle S \rangle$ , and  $\langle Q \rangle$  are 0. Since both  $\langle Q \rangle$  and  $\langle S \rangle$  are 0, the output of the bottom NOR gate,  $\langle (\bar{Q}) \rangle$ , is 1. But since  $\langle (\bar{Q}) \rangle$  is 1, the top NOR gate is forced to produce a 0. Thus, the latch is in a stable state: the fact that  $\langle (\bar{Q}) \rangle$  is 1 holds  $\langle Q \rangle$  at 0.

Now, consider what happens if we change  $\langle S \rangle$  (remember, it's called "set") to 1, while holding R at 0. This change forces  $\langle (\bar{Q}) \rangle$  to 0; for a moment, both  $\langle Q \rangle$  and  $\langle (\bar{Q}) \rangle$  are zero. But the fact that  $\langle (\bar{Q}) \rangle$  is zero means that both inputs to the top NOR gate are zero, so its output,  $\langle Q \rangle$ , must become 1. After that happens, we can return S to 0, and the latch will remain stable. We can think of the effect of changing  $\langle S \rangle$  to 1 for a moment as "setting" the latch to store the value 1. The value is stored at  $\langle Q \rangle$ . ( $\langle (\bar{Q}) \rangle$  is just used to make this circuit do its job, but it's the value of  $\langle Q \rangle$  that we will be interested in.)

An identical argument will show that  $\langle R \rangle$  (remember, it's called "reset") will cause  $\langle Q \rangle$  to return to zero, and thus  $\langle (\bar{Q}) \rangle$  will become 1 again. That is, the value of  $\langle Q \rangle$  is reset to 0! This circuit is commonly called the *S-R latch*.

What happens if both  $\langle S \rangle$  and  $\langle R \rangle$  become 1 at the same time? Try this out through a thought experiment. We'll pause here and wait for you.

Did you notice how this latch will misbehave? When  $\langle S \rangle$  and  $\langle R \rangle$  are both set to 1 (this is trying to set and reset the circuit simultaneously—very naughty) both  $\langle Q \rangle$  and  $\langle (\bar{Q}) \rangle$  will become 0. Now, if we let  $\langle S \rangle$  and  $\langle R \rangle$  return back to 0, the inputs to both NOR gates are 0 and their outputs both become 1. Now each NOR gate gets a 1 back as input and its output becomes 0. In other words, the NOR gates are rapidly "flickering" between 0 and 1 and not storing anything! In fact, other weird and unpredictable things can happen if the two NOR gates compute their outputs at slightly different speeds. Circuit designers have found ways to avoid this problem by building some "protective" circuitry that ensures that  $\langle S \rangle$  and  $\langle R \rangle$  can never be simultaneously set to 1.



*I'm sheepish about sharing my RAM puns because you've probably herd them already.*

So, a latch is a one-bit memory. If you want to remember a 1, turn  $\langle S \rangle$  to 1 for a moment; if you want to remember a 0, turn  $\langle R \rangle$  to 1 for a moment. If you aggregate 8 latches together, you can remember an 8-bit byte. If you aggregate millions of bits, organized in groups of 8, you have the *Random Access Memory (RAM)* that forms the memory of a computer.

#### 4.4 Building a Complete Computer



*"Balancing" the club's budget?! We promise that wheel have no more unicycle jokes!*

Imagine that you've been elected treasurer of your school's unicycling club and its time to do the books and balance the budget. You have a large notebook with all of the club's finances. As you work, you copy a few

numbers from the binder onto a scratch sheet, do some computations using a calculator, and jot those results down on your scratch sheet. Occasionally, you might copy some of those results back into the big notebook to save for the future and then jot some more numbers from the notebook onto your scratch sheet.



*We spoke too soon about no more unicycle jokes.*

A modern computer operates on the same principle. Your calculator and the scratch sheet correspond to the *CPU* of the computer. The CPU is where computation is performed but there's not enough memory there to store all of the data that you will need. The big notebook corresponds to the computer's memory. These two parts of the computer are physically separate components that are connected by wires on your computer's circuit board.

What's in the CPU? There are devices like ripple-carry adders, multipliers, and their ilk for doing arithmetic. These devices can all be built using the concepts that we saw earlier in this chapter, namely the minterm expansion principle and modular design. The CPU also has a small amount of memory, corresponding to the scratch sheet. This memory comprises a small number of *registers* where we can store data. These registers could be built out of latches or other related devices. Computers typically have on the order of 16 to 32 of these registers, each of which can store 32 or 64 bits. All of the CPU's arithmetic is done using values in the registers. That is, the adders, multipliers, and so forth expect to get their inputs from registers and to save the results to a register, just as you would expect to use your scratch pad for the input and output of your computations.



*See! We didn't tire you with any more unicycle puns.*

The memory, corresponding to the big notebook, can store a lot of data - probably billions of bits! When the CPU needs data that is not currently stored in a register (scratch pad), it requests that data from memory. Similarly, when the CPU needs to store the contents of a register (perhaps because it needs to use that register to store some other values), it can ship it off to be stored in memory.

What's the point of having separate registers and memory? Why not just have all the memory in the CPU? The answer is multifaceted, but here is part of it: The CPU needs to be small in order to be fast. Transmitting a bit of data along a millimeter of wire slows the computer down considerably! On the other hand, the memory needs to be large in order to store lots of data. Putting a large memory in the CPU would make the CPU slow. In addition, there are other considerations that necessitate separating the CPU from the memory. For example, CPUs are built using different (and generally much more expensive) manufacturing processes than memories.



*Memory is slow. If the CPU can read from or write to a register in one unit of time, it will take approximately 100 units of time to read from or write to memory!*

Now, let's complicate the picture slightly. Imagine that the process of balancing the unicycle club's budget is complicated. The steps required to do the finances involve making decisions along the way (e.g. "If we spent more than \$500 on unicycle seats this year, we are eligible for a rebate.") and other complications. So, there is a long sequence of instructions that is written in the first few pages of our club notebook. This set of instructions is a program! Since the program is too long and complicated for you to remember, you copy the instructions one-by-one from the notebook onto your scratch sheet. You follow that instruction, which might tell you, for example, to add some numbers and store them some place. Then, you fetch the next instruction from the notebook.

How do you remember which instruction to fetch next and how do you remember the instruction itself? The CPU of a computer has two special registers just for this purpose. A register called the *program counter* keeps track of the location in memory where it will find the next instruction. That instruction is then fetched from

memory and stored in a special register called the *instruction register*. The computer examines the contents of the instruction register, executes that instruction, and then increments the program counter so that it will now fetch the next instruction.

Note

### John von Neumann (1903-1957)

One of the great pioneers of computing was John von Neumann (pronounced “NOY-mahn”), a Hungarian-born mathematician who contributed to fields as diverse as set theory and nuclear physics. He invented the Monte Carlo method (which we used to calculate  $\pi$  in Section 1.1.2), cellular automata, the *merge sort* method for sorting, and of course the von Neumann architecture for computers.

Although von Neumann was famous for wearing a three-piece suit everywhere—including on a mule trip in the Grand Canyon and even on the tennis court—he was not a boring person. His parties were always popular (although he sometimes sneaked away from his guests to work) and he loved to quote from his voluminous memory of off-color limericks. Despite his brilliance, he was a notoriously bad driver, which might explain why he bought a new car every year.

Von Neumann died of cancer, perhaps caused by radiation from atomic-bomb tests. But his legacy lives on in every computer built today.

This way of organizing computation was invented by the famous mathematician and physicist, Dr. John von Neumann, and is known as the *von Neumann architecture*. While computers differ in all kinds of ways, they all use this fundamental principle. In the next subsection we’ll look more closely at how this principle is used in a real computer.



*One of von Neumann’s colleagues was Dr. Claude Shannon—*inventor of the minterm expansion principle*. Shannon, it turns out, was also a very good unicyclist.*

#### 4.4.1 The von Neumann Architecture

We mentioned that the computer’s memory stores both instructions and data. We know that data can be encoded as numbers, and numbers can be encoded in binary. But what about the instructions? Good news! Instructions too can be stored as numbers by simply adopting some convention that maps instructions to numbers.

For example, let’s assume that our computer is based on 8-bit numbers and let’s assume that our computer only has four instructions: add, subtract, multiply, and divide. (That’s very few instructions, but let’s start there for now and then expand later). Each of these instructions will need a number, called an *operation code* (or *opcode*), to represent it. Since there are four opcodes, we’ll need four numbers, which means two bits per number. For example, we might choose the following opcodes for our instructions:

| Operation | Meaning  |
|-----------|----------|
| 00        | Add      |
| 01        | Subtract |
| 10        | Multiply |
| 11        | Divide   |

Next, let’s assume that our computer has four registers number 0 through 3. Imagine that we want to add two numbers. We must specify the two registers whose values we wish to add and the register where we wish to store the result. If we want to add the contents of register 2 with the contents of register 0 and store the result in register 3, we could adopt the convention that we’ll write “add 3, 0, 2”. The last two numbers are the registers where we’ll get our inputs and the first number is the register where we’ll store our result. In binary, “add 3, 0, 2” would be represented as “00 11 00 10”. We’ve added the spaces to help you see the numbers 00 (indicating “add”), 11 (indicating the register where the result will be stored), 00 (indicating register 0 as the first register that we will add), and 10 (indicating register 2 as the second register that we will add).



Computer scientists often start counting from 0.

In general, we can establish the convention that an instruction will be encoded using 8 bits as follows: The first two bits (which we'll call I<sub>0</sub> and I<sub>1</sub>) represent the instruction, the next two bits (D<sub>0</sub> and D<sub>1</sub>) encode the “destination register” where our result will be stored, the next two bits (S<sub>0</sub> and S<sub>1</sub>) encode the first register that we will add, and the last two bits (T<sub>0</sub> and T<sub>1</sub>) encode the second register that we will add. This representation is shown below.

|                               |                               |                               |                               |
|-------------------------------|-------------------------------|-------------------------------|-------------------------------|
| I <sub>0</sub> I <sub>1</sub> | D <sub>0</sub> D <sub>1</sub> | S <sub>0</sub> S <sub>1</sub> | T <sub>0</sub> T <sub>1</sub> |
|-------------------------------|-------------------------------|-------------------------------|-------------------------------|

Recall that we assume that our computer is based on 8-bit numbers. That is, each register stores 8 bits and each number in memory is 8 bits long. Figure 4.9 shows what our computer might look like. Notice the program counter at the top of the CPU. Recall that this register contains a number that tells us where in memory to fetch the next instruction. At the moment, this program counter is 00000000, indicating address 0 in memory.



So a computer is kind of like a dog?

The computer begins by going to this memory location and fetching the data that resides there. Now look at the memory, shown on the right side of the figure. The memory addresses are given both in binary (base 2) and in base 10. Memory location 0 contains the data 00100010. This 8-bit sequence is now brought into the CPU and stored in the instruction register. The CPU's logic gates decode this instruction. The leading 00 indicates it's an addition instruction. The following 10 indicates that the result of the addition that we're about to perform will be stored in register 2. The next 00 and 10 mean that we'll get the data to add from registers 0 and 2, respectively. These values are then sent to the CPU's ripple-carry adder where they are added. Since registers 0 and 2 contain 00000101 and 00001010, respectively, before the operation, after the operation register 2 will contain the value 00001111.

In general, our computer operates by repeatedly performing the following procedure:

1. Send the address in the program counter (commonly called the *PC* ) to the memory, asking it to read that location.
2. Load the value from memory into the instruction register.
3. *Decode* the instruction register to determine what instruction to execute and which registers to use.
4. *Execute* the requested instruction. This step often involves reading operands from registers, performing arithmetic, and sending the results back to the destination register. Doing so usually involves several sub-steps.
5. Increment the PC (Program Counter) so that it contains the address of the next instruction in memory. (It is this step that gives the PC its name, because it *counts* its way through the addresses in the program.)

|                      |                                                      |          |
|----------------------|------------------------------------------------------|----------|
| Program Counter      | <table border="1"><tr><td>00000000</td></tr></table> | 00000000 |
| 00000000             |                                                      |          |
| Instruction Register | <table border="1"><tr><td>00000000</td></tr></table> | 00000000 |
| 00000000             |                                                      |          |
| Register 0           | <table border="1"><tr><td>00000101</td></tr></table> | 00000101 |
| 00000101             |                                                      |          |
| Register 1           | <table border="1"><tr><td>00000000</td></tr></table> | 00000000 |
| 00000000             |                                                      |          |
| Register 2           | <table border="1"><tr><td>00001010</td></tr></table> | 00001010 |
| 00001010             |                                                      |          |
| Register 3           | <table border="1"><tr><td>00000000</td></tr></table> | 00000000 |
| 00000000             |                                                      |          |

Central Processing Unit (CPU)

| Location<br>(Binary) | Location<br>(Base 10) | Contents |
|----------------------|-----------------------|----------|
| 00000000             | 0                     | 00100010 |
| 00000001             | 1                     | 00011010 |
| 00000010             | 2                     | 10001100 |
| 00000011             | 3                     |          |
| 11111111             | 255                   | ...      |

Memory

Figure 4.9: A computer with instructions stored in memory. The program counter tells the computer where to get the next instruction.

“Wait!” we hear you scream. “The memory is storing *both* instructions *and* data! How can it tell which is which?!?” That’s a great question and we’re glad you asked. The truth is that the computer *can’t tell* which is which. If we’re not careful, the computer might fetch something from memory into its instruction register and try to execute it when, in fact, that 8-bit number represents the number of pizzas that the unicycle club purchased and not an instruction! One way to deal with this is to have an additional special instruction called “halt” that tells the computer to stop fetching instructions. In the next subsections we’ll expand our computer to have more instructions (including “halt”) and more registers and we’ll write some real programs in the language of the computer.

**Takeaway message:** *A computer uses a simple process of repeatedly fetching its next instruction from memory, decoding that instruction, executing that instruction, and incrementing the program counter. All of these steps are implemented with digital circuits that, ultimately, can be built using the processes that we’ve described earlier in this chapter.*

## 4.5 Hmmmm

The computer we discussed in Section 4.4 is simple to understand, but its simplicity means it’s not very useful. In particular, a real computer also needs (at a minimum) ways to:

1. Move information between the registers and a large memory,
2. Get data from the outside world,
3. Print results, and
4. Make decisions.

To illustrate how these features can be included, we have designed the Harvey Mudd Miniature Machine, or Hmmmm. Just like our earlier 4-instruction computer, Hmmmm has a program counter, an instruction register, a set of data registers, and a memory. These are organized as follows:



*Hmmm is music to my ears!*



*Different computers have different word sizes. Most machines sold today use 64-bit words; older ones use 32 bits. 16-bit, 8-bit, and even 4-bit computers are still used for special applications. Your digital watch probably contains a 4-bit computer.*

1. While our simple computer used 8-bit instructions, in Hmmmm, both instructions and data are 16 bits wide. The set of bits representing an instruction is called a *word*. That allows us to represent a reasonable range

of numbers, and lets the instructions be more complicated.

2. In addition to the program counter and instruction register, Hmmm has 16 registers, named R0 through R15. R0 is special: it always contains zero, and anything you try to store there is thrown away.
3. Hmmm's memory contains 256 locations. Although the memory can be used for either instructions or data, programs are prevented from reading or writing the instruction section of memory. (Some modern computers offer a similar feature.)



Hmmm's instruction set is large, but not hmmmungous.

It's very inconvenient to program Hmmm by writing down the bits corresponding to the instructions. Instead, we will use *assembly language*, which is a programming language where each machine instruction receives a friendlier symbolic representation. For example, to compute  $R3 = R1 + R2$ , we would write:

```
add r3, r1, r2
```

A very simple process is used to convert this assembly language into the 0's and 1's - the "machine language" - that the computer can execute.



A complete list of Hmmm instructions, including their binary encoding, is given in Figure 4.10.

#### 4.5.1 A Simple Hmmm Program



Not the least of which is that writing even a short program in assembly language can be a hmmmbling experience!

To begin with, let's look at a program that will calculate the approximate area of a triangle. It's admittedly mundane, but it will help us move on to more interesing Hmmm-ing shortly. (We suggest that you follow along by downloading Hmmm from <http://www.cs.hmc.edu/~cs5grad/cs5/hmmm/documentation/documentation.html> and trying these examples out with us.)

Let's begin by using our favorite editor to create a file named triangle1.hmmm with the following contents:

```

Calculate the approximate area of a triangle.

First input: base
Second input: height
Output: area

0 read r1 # Get base
1 read r2 # Get height
2 mul r1 r1 r2 # b times h into r1
3 setn r2 2
4 div r1 r1 r2 # Divide by 2
5 write r1
6 halt
```

## How Does It Work?

What does all of this mean? First, anything starting with a pound sign (“#”) is a comment; Hmmm ignores it. Second, you’ll note that each line is numbered, starting with zero. This number indicates the memory location where the instruction will be stored.

You may have also noticed that this program doesn’t use any commas, unlike the example `add` instruction above. Hmmm is very lenient about notation; all of the following instructions mean exactly the same thing:

```
add r1,r2,r3
```

```
ADD R1 R2 R3
```

```
ADD R1,r2, R3
```

```
aDd R1,,R2, ,R3
```

Needless to say, we don’t recommend the last two options!

So what do all of these lines actually do? The first two (0 and 1) read in the base and height of the triangle. When Hmmm executes a `read` instruction, it pauses and prompts the user to enter a number, converts the user’s number into binary, and stores it into the named register. So the first-typed number will go into register R1, and the second into R2.

The `MULTIPLY` instruction then finds  $(b \times h)$  by calculating  $R1 = R1 \times R2$ . This instruction illustrates three important principles of hmmm programming:

1. Most arithmetic instructions accept three registers: two *sources* and a *destination*.
2. The destination register is always listed first, so that the instruction can be read somewhat like a Python assignment statement.
3. A source and destination can be the same.

After multiplying, we need to divide  $(b \times h)$  by 2. But where can we get the constant 2? One option would be to ask the user to provide it via a `read` instruction, but that seems clumsy. Instead, a special instruction, `setn` (*set to number*), lets us insert a small constant into a register. As with `mul`, the destination is given first; this is equivalent to the Python statement `R2 = 2`.

The `DIVIDE` instruction finishes the calculation, and `WRITE` displays the result on the screen. There is one thing left to do, though: after the `WRITE` is finished, the computer will happily try to execute the instruction at the following memory location. Since there isn’t a valid instruction there, the computer will fetch a collection of bits there that are likely to be invalid as an instruction, causing the computer to crash. So we need to tell it to `HALT` after it’s done with its work.

That’s it! But will our program work



*It would be hmmmurous if we got this wrong*

## Trying It Out

We can *assemble* the program by running `hmmmAssembler.py` from the command line: [1] User-typed input is shown in blue and the prompt is shown using the symbol %. The prompt on your computer may look different.

```
% ./hmmmAssembler.py
Enter input file name: triangle1.hmmm
Enter output file name: triangle1.hb

ASSEMBLY SUCCESSFUL

0 : 0000 0001 0000 0001 0 read r1 # Get base
1 : 0000 0010 0000 0001 1 read r2 # Get height
```

```

2 : 1000 0001 0001 0010 2 mul r1 r1 r2 # b times h into r1
3 : 0001 0010 0000 0010 3 setn r2 2
4 : 1001 0001 0001 0010 4 div r1 r1 r2 # Divide by 2
5 : 0000 0001 0000 0010 5 write r1
6 : 0000 0000 0000 0000 6 halt

```

If you have errors in the program, `hmmmAssembler.py` will tell you; otherwise it will produce the output file `triangle1.hb` (“hb” stands for “Hmmm binary”). We can then test our program by running the Hmmm simulator:

```

% ./hmmmSimulator.py
Enter binary input file name: triangle1.hb
Enter debugging mode? n
Enter number: 4
Enter number: 5
10
% ./hmmmSimulator.py
Enter binary input file name: triangle1.hb
Enter debugging mode? n
Enter number: 5
Enter number: 5
12

```

We can see that the program produced the correct answer for the first test case, but not for the second. That’s because Hmmm only works with integers; division rounds fractional values down to the next smaller integer just as integer division does in Python.

#### 4.5.2 Looping

If you want to calculate the areas of a lot of triangles, it’s a nuisance to have to run the program over and over. Hmmm offers a solution in the *unconditional jump* instruction, which says “instead of executing the next sequential instruction, start reading instructions beginning at address  $n$ .” If we simply replace the `halt` with a `jump`:

```

#
Calculate the approximate areas of many triangles.
#
First input: base
Second input: height
Output: area
#
0 read r1 # Get base
1 read r2 # Get height
2 mul r1 r1 r2 # b times h into r1
3 setn r2 2
4 div r1 r1 r2 # Divide by 2
5 write r1
6 jumpn 0

```

then our program will calculate triangle areas forever.

What’s that `jumpn 0` instruction doing? The short explanation is that it’s telling the computer to jump back to location 0 and continue executing the program there. The better explanation is that this instruction simply puts the number 0 in the program counter. Remember, the computer mindlessly checks its program counter to determine where to fetch its next instruction from memory. By placing a 0 in the program counter, we are ensuring that the next time the computer goes to fetch an instruction it will fetch it from memory location 0.

Since there will come a time when we want to stop, we can *force* the program to end by holding down the Ctrl (“Control”) key and typing C (this is commonly written “Ctrl-C” or just “ $\wedge$ C”):

```

% ./hmmSimulator.py
Enter binary input file name: triangle2.hb
Enter debugging mode? n
Enter number: 4
Enter number: 5

```

```
10
Enter number: 5
Enter number: 5
12
Enter number: ^C
```

Interrupted by user, halting program execution...

That works, but it produces a somewhat ugly message at the end. A nicer approach might be to automatically halt if the user inputs a zero for either the base or the height. We can do that with a *conditional jump* instruction, which works like `jmpn` if some *condition* is true, and otherwise does nothing.



*I believe that I learned that when someone jeqzn's I should say "gesundheit"*

There are several varieties of conditional jump statements and the one we'll use here is called `jeqzn` which is pronounced "jump to *n* if equal to zero" or just "jump if equal to zero." This conditional jump takes a register and a number as input. If the specified register contains the value zero then we will jump to the instruction specified by the number in the second argument. That is, if the register contains zero then we will place the number in the second argument in the program counter so that the computer will continue computing using that number as its next instruction.

```
Calculate the approximate areas of many triangles.
Stop when a base or height of zero is given.
#
First input: base
Second input: height
Output: area
#
0 read r1 # Get base
1 jeqzn r1 9 # Jump to halt if base is zero
2 read r2 # Get height
3 jeqzn r2 9 # Jump to halt if height is zero
4 mul r1 r1 r2 # b times h into r1
5 setn r2 2
6 div r1 r1 r2 # Divide by 2
7 write r1
8 jmpn 0
9 halt
```

Now, our program behaves politely:

```
% ./hmmmSimulator.py
Enter binary input file name: triangle3.hb
Enter debugging mode? n
Enter number: 4
Enter number: 5
10
Enter number: 5
Enter number: 5
12
Enter number: 0
```

The nice thing about conditional jumps is that you aren't limited to just terminating loops; you can also use them to make decisions. For example, you should now be able to write a Hmmm program that prints the absolute value of a number. The other conditional jump statements in Hmmm are included in the listing of all Hmmm instructions at the end of this chapter.

### 4.5.3 Functions

Here is a program that computes factorials:

```

Calculate N factorial.

Input: N
Output: N!

Register usage:

r1 N
r2 Running product

0 read r1 # Get N
1 setn r2,1
2 jeqzn r1,6 # Quit if N has reached zero
3 mul r2,r1,r2 # Update product
4 addn r1,-1 # Decrement N
5 jumpn 2 # Back for more

6 write r2
7 halt
```

(If you give this program a negative number, it will crash unpleasantly; how could you fix that problem?) The `addn` instruction at line 4 simply adds a constant to a register, replacing its contents with the result. We could achieve the same effect by using `setn` and a normal `add`, but computer programs add constants so frequently that Hmmm provides a special instruction to make the job easier.

But suppose you need to write a program that computes  $\binom{n}{k} = \frac{n!}{k!(n-k)!}$ . (If you haven't seen this formula before, it counts the number of different ways to choose  $k$  items from a set of  $n$  distinct items and it's pronounced "  $n$  choose  $k$  ".)



*I wouldn't choose to write that program.*

Since we need to compute three different factorials, we would like to avoid having to copy the above loop three different times. Instead, we'd prefer to have a *function* that computes factorials, just like Python has.

Creating a function is just a bit tricky. It can't read its input from the user, and it must return the value that it computed to whatever code called that function.

It's convenient to adopt a few simple conventions to make this all work smoothly. One such convention is to decide on special registers to be used for *parameter passing*, i.e., getting information into and out of a function. For example, we could decide that `r1` will contain  $n$  when the factorial function starts, and `r2` will contain the result. (As we'll see later, this approach is problematic in the general case, but it's adequate for now.)

Our new program, with the factorial function built in, is:

```

Calculate C(n,k) = n!/k!(n-k)!.

First input: N
Second input: K
Output: C(N,K)

Register usage:

r1 Input to factorial function
```

```

r2 r1 factorial
r3 N
r4 K
r5 C(N,K)
#
Factorial function starts at address 15
#
0 read r3 # Get N
1 read r4 # Get K

2 copy r1,r3 # Calculate N!
3 calln r14,15 # ...
4 copy r5,r2 # Save N! as C(N,K)

5 copy r1,r4 # Calculate K!
6 calln r14,15 # ...
7 div r5,r5,r2 # N!/K!

8 sub r1,r3,r4 # Calculate (N-K)!
9 calln r14,15 # ...
10 div r5,r5,r2 # C(N,K)

11 write r5 # Write answer
12 halt

13 nop # Waste some space
14 nop

Factorial function. N is in R1. Result is R2.
Return address is in R14.
15 setn r2,1 # Initial product
16 jeqzn r1,20 # Quit if N has reached zero
17 mul r2,r1,r2 # Update product
18 addn r1,-1 # Decrement N
19 jumpn 16 # Back for more

20 jumpr r14 # Done; return to caller

```

As you can see, the program introduces a number of new instructions. The simplest is `nop`, the *no-operation* instruction, at lines 13 and 14. When executed, it does absolutely nothing.

Why would we want such an instruction? If you've already written some small Hmmm programs, you've probably discovered the inconvenience of renumbering lines. By including a few `nops` as *padding*, we make it easy to insert a new instruction in the sequence from 0-15 without having to change where the factorial function begins.



*One of the hmmmdingers of programming in assembly language!*

Far more interesting is the `calln` instruction, which appears at lines 3, 6, and 9. `Calln` works similarly to `jumpn`: it causes Hmmm to start executing instructions at a given address, in this case 15. But if we had just used a `jumpn`, after the factorial function calculated its result, it wouldn't know whether to jump back to line 4, line 7, or line 10! To solve the problem, the `calln` uses register R14 to save the address of the instruction immediately *following* the call. [2]

We're not done, though: the factorial function itself faces the other end of the same dilemma. R14 contains 4, 7, or 10, but it would be clumsy to write code equivalent to "If R14 is 4, jump to 4; otherwise if R14 is 7, jump to 7; ..." Instead, the `jumpr` (jump to address in register) instruction solves the problem neatly, if somewhat confusingly. Rather than jumping to a fixed address given in the instruction (as is done in line 19), `jumpr` goes to

a *variable* address taken from a register. In other words, if R14 is 4, `jumpr` will jump to address 4; if it's 7 it will jump to 7, and so forth.

#### 4.5.4 Recursion

In Chapter 2 we learned to appreciate the power and elegance of recursion. But how can you do recursion in Hmmm assembly language? There must be a way; after all, Python implements recursion as a series of machine instructions on your computer. To understand the secret, we need to talk about the *stack*.

#### Stacks

You may recall that in Chapter 2 we talked about stacks (remember those stacks of storage boxes)? Now we're going to see precisely how they work.

A stack is something we are all familiar with in the physical world: it's just a pile where you can only get at the top thing. Make a tall stack of books; you can only see the cover of the top one, you can't remove books from the middle (at least not without risking a collapse!), and you can't add books anywhere except at the top.



*The sheer weight of that book could crush your stack!*

The reason a stack is useful is because it remembers things and gives them back to you in reverse order. Suppose you're reading *Gone With the Wind* and you start wondering whether it presents the American Civil War fairly. You might put *GWTW* on a stack, pick up a history book, and start researching. As you're reading the history book, you run into an unfamiliar word. You place the history book on the stack and pick up a dictionary. When you're done with the dictionary, you return it to the shelf, and the top of the stack has the history book, right where you left off. After you finish your research, you put that book aside, and voilá! *GWTW* is waiting for you.

This technique of putting things aside and then returning to them is precisely what we need for recursion. To calculate  $\lfloor(5!) \rfloor$ , you need to find  $\lfloor(4!) \rfloor$  for which you need  $\lfloor(3!) \rfloor$  and so forth. A stack can remember where we were in the calculation of  $\lfloor(4!) \rfloor$  and help us to return to it later.



*Again, this is just a convention.*

To implement a stack on a computer, we need a *stack pointer*, which is a register that holds the memory address of the top item on the stack. Traditionally, the stack pointer is kept in the highest-numbered register, so on Hmmm we use R15.

Suppose R15 currently contains 102, [3] and the stack contains 42 (on top), 56, and 12. Then we would draw the stack like this:

|       | Address | Contents |
|-------|---------|----------|
| R15 → | 102     | 42       |
|       | 101     | 56       |
|       | 100     | 12       |

To *push* a value, say 23, onto the stack, we must increment R15 to contain the address of a new location (103) and then store 23 into that spot in memory.

Later, to *pop* the top value off, we must ask R15 where the top is [4] (103) and recover the value in that location. Then we decrement R15 so that it points to the new stack top, location 102. Everything is now back where we started.

The code to push something, say the contents of R4, on the stack looks like this:

```
addn r15,1 # Point to a new location
stor r4,r15 # Store r4 on the stack
```

The storr instruction stores the contents of R4 into the memory location *addressed* by register 15. In other words, if R15 contains 103, the value in R4 will be copied into memory location 103.

Just as storr can put things onto the stack, loadr will recover them:

```
loadr r3,r15 # Load r3 from the stack
addn r15,-1 # Point to new top of stack
```

**Important note:** in the first example above, the stack pointer is incremented *before* the storr; in the second, it is decremented *after* the loadr. This ordering is necessary to make the stack work properly; you should be sure you understand it before proceeding further.

### Saving Precious Possessions

When we wrote the `\((\{n \atop k}\))` program in Section 4.5.3, we took advantage of our knowledge of how the factorial function on line 15 worked. In particular, we knew that it only modified registers R1, R2, and R14, so that we could use R3 and R4 for our own purposes. In more complex programs, however, it may not be possible to partition the registers so neatly. This is especially true in recursive functions, which by their nature tend to re-use the same registers that their callers use.

The only way to be sure that a called function won't clobber your data is to save it somewhere, and the stack is a perfect place to use for that purpose. When writing a Hmmm program, the convention is that you must save all of your “precious possessions” before calling a function, and restore them afterwards. Because of how a stack works, you have to restore things in reverse order.

But what's a precious possession? The quick answer is that it's any register that you plan to use, *except* R0 and R15. In particular, if you are calling a function from inside another function, R14 is a precious possession.



*It's a common mistake, but to err is hmman.*

Many newcomers to Hmmm try to take shortcuts with stack saving and restoring. That is to reason, “I know that I'm calling two functions in a row, so it's silly to restore my precious possessions and save them again right away.” Although you can get away with that trick in certain situations, it's very difficult to get it right, and you are much more likely to cause yourself trouble by trying to save time. We strongly suggest that you follow the suggested *stack discipline* rigorously to avoid problems.

Let's look at a Hmmm program that uses the stack. We'll use the recursive algorithm to calculate factorials:

```
Calculate N factorial, recursively
#
Input: N
Output: N!
#
Register usage:
#
r1 N! (returned by called function)
r2 N

0 setn r15,100 # Initialize stack pointer
1 read r2 # Get N
2 calln r14,5 # Recursive function finds N!
3 write r1 # Write result
4 halt

Function to compute N factorial recursively
#
Inputs:
#
```

```

r2 N
#
Outputs:
#
r1 N!
#
Register usage:
#
r1 N! (from recursive call)
r2 N (for multiplication)

5 jeqzn r2,18 # Test for base case (0!)

6 addn r15,1 # Save precious possessions
7 storer r2,r15 # ...
8 addn r15,1 # ...
9 storer r14,r15 # ...

10 addn r2,-1 # Calculate N-1
11 calln r14,5 # Call ourselves recursively to get (N-1)!

12 loadr r14,r15 # Recover precious possessions
13 addn r15,-1 # ...
14 loadr r2,r15 # ...
15 addn r15,-1 # ...

16 mul r1,r1,r2 # (N-1)! times N
17 jumpr r14 # Return to caller

Base case: 0! is always 1
18 setn r1,1
19 jumpr r14 # Return to caller

```

The main program is quite simple (lines 0 - 4): we read a number from the user, call the recursive factorial function, and write the answer, and then halt.

The factorial function is only a bit more complex. After testing for the base case of  $\backslash(0!\backslash)$ , we save our “precious possessions” in preparation for the recursive call (lines 6 - 9). But what is precious to us? The function uses registers R1, R2, R14, and R15. We don’t need to save R15 because it’s the stack pointer, and R1 doesn’t yet have anything valuable in it. So we only have to save R2 and R14. We follow the stack discipline, placing R2 on the stack (line 7) and then R14 (line 9).

After saving our precious possessions, we call ourselves recursively to calculate  $\backslash((N-1)!\backslash)$  (lines 10–11) and then recover registers R14 (line 12) and R2 (line 14) in reverse order, again following the stack discipline. Then we multiply  $\backslash((N-1)!\backslash)$  by  $\backslash(N\backslash)$ , and we’re done.

It is worth spending a bit of time studying this example to be sure that you understand how it operates. Draw the stack on a piece of paper, and work out how values get pushed onto the stack and popped back off when the program calculates  $\backslash(3!\backslash)$ .

#### 4.5.5 The Complete Hmmm Instruction Set



*Note that sub can be combined with jltzn to evaluate expressions like  $\backslash(a < b\backslash)$ .*

This finishes our discussion of Hmmm. We have covered all of the instructions except sub, mod, jnezn, jgtzn, and jltzn; we trust that those don’t need separate explanations.

For convenience, Figure 4.10 at the bottom of the page summarizes the entire instruction set, and also gives the binary encoding of each instruction.

As a final note, you may find it instructive to compare the encodings of certain pairs of instructions. In particular, what is the difference between `add`, `mov`, and `nop`? How does `calln` relate to `jumpn`? We will leave you with these puzzles as we move on to imperative programming.

#### 4.5.6 A Few Last Words

What actually happens when we run a program that we've written in a language such as Python? In some systems, the entire program is first automatically translated or *compiled* into machine language (the binary equivalent of assembly language) using another program called a *compiler*. The resulting compiled program looks like the Hmmm code that we've written and is executed on the computer. Another approach is to use a program called an *interpreter* that translates the instructions one line at a time into something that can be executed by the computer's hardware.

It's important to keep in mind that exactly when and how the program gets translated - all at once as in the case of a compiler or one line at a time in the case of an interpreter - is an issue separate from the language itself. In fact, some languages have both compilers and interpreters, so that we can use one or the other.

Why would someone even care whether their program was compiled or interpreted? In the compiled approach, the entire program is converted into machine language before it is executed. The program runs very fast but if there is an error when the program runs, we probably won't get very much information from the computer other than seeing that the program "crashed". In the interpreted version, the interpreter serves as a sort of "middle-man" between our program and the computer. The interpreter can examine our instruction before translating it into machine language. Many bugs can be detected and reported by the interpreter before the instruction is ever actually executed by the computer. The "middle-man" slows down the execution of the program but provides an opportunity to detect potential problems. In any case, ultimately every program that we write is executed as code in machine language. This machine language code is decoded by digital circuits that execute the code on other circuits.

## 4.6 Conclusion

Aye caramba! That was a lot. We've climbed the levels of abstraction from transistors, to logic gates, to a ripple-carry adder, and ultimately saw the general idea of how a computer works. Finally, we've programmed that computer in its native language.



*I think a mochaccino and a donut would be quite meaningful right about now.*

Now that we've seen the foundations of how a computer works, we're going back to programming and problem-solving. In the next couple of chapters, some of the programming concepts that we'll see will be directly tied to the issues that we examined in this chapter. We hope that the understanding that you have for the internal workings of a computer will help the concepts that we're about to see be even more meaningful than they would be otherwise.

Instruction

Description

Aliases

**System instructions**

**halt**

Stop!

None

**read rX**

Place user input in register rX

None

**write rX**

Print contents of register rX

None

**nop**

Do nothing

None

### **Setting register data**

**setn** rX N

Set register rX equal to the integer N (-128 to +127)

None

**addn** rX N

Add integer N (-128 to 127) to register rX

None

**copy** rX rY

Set rX = rY

**mov**

### **Arithmetic**

**add** rX rY rZ

Set rX = rY + rZ

None

**sub** rX rY rZ

Set rX = rY - rZ

None

**neg** rX rY

Set rX = -rY

None

**mul** rX rY rZ

Set rX = rY \* rZ

None

**div** rX rY rZ

Set rX = rY / rZ (integer division; no remainder)

None

**mod** rX rY rZ

Set rX = rY % rZ (returns the remainder of integer division)

None

**Jumps!**

**jumpn** N

Set program counter to address N

None

**jumpr** rX

Set program counter to address in rX

**jump**

**jeqzn rX N**  
If  $rX == 0$ , then jump to line N

**jeqz**

**jnezn rX N**  
If  $rX != 0$ , then jump to line N

**jnez**

**jgtzn rX N**  
If  $rX > 0$ , then jump to line N

**jgtz**

**jltzn rX N**

If  $rX < 0$ , then jump to line N

**jltz**

**calln rX N**

Copy the next address into  $rX$  and then jump to mem. addr. N

**call**

#### Interacting with memory (RAM)

**loadn rX N**

Load register  $rX$  with the contents of memory address N

None

**storen rX N**

Store contents of register  $rX$  into memory address N

None

**loadr rX rY**

Load register  $rX$  with data from the address location held in reg.  $rY$

**loadi, load**

**storer rX rY**

Store contents of register  $rX$  into memory address held in reg.  $rY$

**storei, store**

Figure 4.10: The Hmmm instruction set.

Footnotes

---

[1] On Windows, you can navigate to the command line from the Start menu. On a Macintosh, bring up the Terminal application.

Table 2:

---

[2] We could have chosen any register except R0 for this purpose, but by convention Hmmm programs use R14.

Table 3:

---

[3] We say that R15 *points to* location 102.

Table 4:

|     |                                             |
|-----|---------------------------------------------|
| [4] | We say we <i>follow the pointer</i> in R15. |
|-----|---------------------------------------------|

## Table Of Contents

- Chapter 4: Computer Organization
  - 4.1 Introduction to Computer Organization
  - 4.2 Representing Information
    - \* 4.2.1 Integers
    - \* 4.2.2 Arithmetic
    - \* 4.2.3 Letters and Strings
    - \* 4.2.4 Structured Information
  - 4.3 Logic Circuitry
    - \* 4.3.1 Boolean Algebra
    - \* 4.3.2 Making Other Boolean Functions
    - \* 4.3.3 Logic Using Electrical Circuits
    - \* 4.3.4 Computing With Logic
    - \* 4.3.5 Memory
  - 4.4 Building a Complete Computer
    - \* 4.4.1 The von Neumann Architecture
  - 4.5 Hmmm
    - \* 4.5.1 A Simple Hmmm Program
    - \* How Does It Work?
    - \* Trying It Out
    - \* 4.5.2 Looping
    - \* 4.5.3 Functions
    - \* 4.5.4 Recursion
    - \* Stacks
    - \* Saving Precious Possessions
    - \* 4.5.5 The Complete Hmmm Instruction Set
    - \* 4.5.6 A Few Last Words
  - 4.6 Conclusion

**Previous topic** Chapter 3: Functional Programming, Part Deux

**Next topic** Chapter 5: Imperative Programming

## Quick search

Enter search terms or a module, class or function name.

## Navigation

- index
- next |
- previous |
- cs5book 1 documentation »

© Copyright 2013, hmc. Created using Sphinx 1.2b1.

# CSforAll - Binary

## CS for All

CSforAll Web > Chapter4 > Binary

This website accompanies the textbook *CS For All* by Alvarado, Dodds, Kuenning, and Libeskind-Hadas

### From Decimal to Binary and Beyond...

This lab problem is all about converting numbers to and from base 10 (where *most* humans operate) from and to base 2 (where virtually all computers operate).

At the end of the lab, you'll extend this to *ternary*, or base 3, where fewer computers and humans, but many more aliens, operate!

#### Function #1    `isOdd(n)`    Warm-Up Function!

As background, we'll recall how to determine whether values are even or odd in Python:

- First, open up Python and create a new blank file
- Then, just to get started, **write a Python function** called `isOdd(n)` that accepts a single argument, an integer `n`, and returns `True` if `n` is odd and `False` if `n` is even. Be sure to return these values, not strings! The evenness or oddness of a value is considered its *parity*. You should use the `%` operator (the "mod" operator). Remember that in Python `n % d` returns the remainder when `n` is divided by `d` (assuming `n >= 0`). Here are two examples of `isOdd` in action:

```
In [1]: isOdd(42)
Out[1]: False
```

```
In [2]: isOdd(43)
Out[2]: True
```

...and here are some tests:

```
assert isOdd(42) == False
assert isOdd(43) == True
```

If you prefer shorter versions of those tests, these also work:

```
assert not isOdd(42)
assert isOdd(43)
```

Sweet!

### Function #2 numToBinary(N)

This part of the lab motivates converting decimal numbers into binary form *one bit at a time*, which may seem "*odd*" at first...!

**Quick overview:** you will end up writing a function `numToBinary(N)` that works as follows:

```
In [1]: numToBinary(5)
Out[1]: '101'
```

```
In [2]: numToBinary(12)
Out[2]: '1100'
```

**Starter code:** If you'd like, we provide one starting point for your `numToBinary` function here. A useful first task is to write a docstring!

```
def numToBinary(N):
 """
 if N == 0:
 return ''
 elif N%2 == 1:
 return ----- + '1'
 else:
 return ----- + '0'

assert numToBinary(0) == ''
assert numToBinary(42) == '101010'
```

### Thoughts:

- Notice that this function is, indeed, handling only one "bit" (zero or one) at a time.
- We didn't use `isOdd`—this is OK. (This way is a bit more flexible for when we switch to base 3!)
- Since we don't want leading zeros, if the argument `N` is zero, it returns the empty string.
- *This means that `numToBinary(0)` will be the empty string.* This is both required and OK!
- If the argument `N` is odd, the function adds 1
- If the argument `N` is even (`else`), the function adds 0
- *What recursive calls to `numToBinary`—and other computations—are needed in the blank spaces above?*

Hint

- You'll want to recur by calling `numToBinary` on a smaller value.
- What *value* of `N` results when one bit (the rightmost bit) of `N` is removed? That's what you'll want!
- Remember that the `//` operator is integer division (with rounding down)
- Stuck? Check this week's class notes for additional details...
- *tfw* binary is *fleek*? Check out the gold notes' *fleek* version (which can easily extend to *any* base...)

### More Examples!:

```
In [1]: numToBinary(0)
Out[1]: ''

In [2]: numToBinary(1)
Out[2]: '1'

In [3]: numToBinary(4)
Out[3]: '100'

In [4]: numToBinary(10)
Out[4]: '1010'

In [5]: numToBinary(42)
Out[5]: '101010'

In [6]: numToBinary(100)
Out[6]: '1100100'
```

### Function #3 binaryToNum(S)

Next, you'll tackle the more challenging task of converting from base 2 to base 10, *again from right to left*. We'll represent a base-2 number as a string of 0's and 1's (bits).

**Quick overview:** you will end up writing a function `binaryToNum(S)` that works as follows:

```
In [1]: binaryToNum('101')
Out[1]: 5
```

```
In [2]: binaryToNum('101010')
Out[2]: 42
```

**Starter code:** If you'd like, we provide one starting point for your `binaryToNum` function here. Again, a useful first task is to write a docstring!

```
def binaryToNum(S):
 """
 if S == '':
 return 0

 # if the last digit is a '1'...
 elif S[-1] == '1':
 return _____ + 1

 else: # last digit must be '0'
 return _____ + 0

assert binaryToNum('') == 0
assert binaryToNum('101010') == 42
```

### Thoughts:

- Remember that the argument is a *string* named `S`.
- Notice that this function is, again, handling only one "bit" (zero or one) at a time, right to left.
- Reversing the action of the prior function, if the argument is an empty string, the function returns 0. This is both required and OK!

- If the *last digit* of S is '1', the function adds the value 1 to the result.
- If the *last digit* of S is '0', the function adds the value 0 to the result. (Not strictly required, but OK.)
- *What recursive calls to binaryToNum—and other computations—are needed in the blank spaces above?*

### Hint

- You'll want to recur by calling `binaryToNum` on a smaller string.
- How do you get the string of everything *except* the last digit?! (Use slicing!)
- When you recur, the recursive call will return the **value** of the smaller string—*This will be too small a value*, but...
- What computation can and should you perform to make the value correct?
- Remember that the recursion is returning the value of a binary string *one bit shorter* (shifted to the right by one spot)—remember how that right-shift changes the overall value. *Then undo that effect!*
  - You can either multiply or shift, but *be sure to do so outside & after the recursive call to binaryToNum*
- That's all you'll need (it's just one operation after the recursive call)!

### More examples!:

```
In [1]: binaryToNum("100")
Out[1]: 4

In [2]: binaryToNum("1011")
Out[1]: 11

In [3]: binaryToNum("00001011")
Out[1]: 11

In [4]: binaryToNum("")
Out[1]: 0

In [5]: binaryToNum("0")
Out[1]: 0

In [6]: binaryToNum("1100100")
Out[1]: 100

In [7]: binaryToNum("101010")
Out[1]: 42
```

## Functions #4 and #5 `increment(S)` and `count(S, n)`

### Binary Counting!

In this problem we'll write several functions to do count in binary—*use the two functions you wrote above for this!*

**Quick overview:** you'll write `increment(S)`, which accepts an 8-character string `S` of 0's and 1's and returns the *next largest* number in base 2. Here are some sample calls and their results:

```
In [1]: increment('00000000')
Out[1]: '00000001'
```

```
In [2]: increment('00000001')
Out[2]: '00000010'
```

```
In [3]: increment('00000111')
Out[3]: '00001000'
```

```
In [4]: increment('11111111')
Out[4]: '00000000'
```

### Thoughts:

- Notice that `increment('11111111')` should wrap around to the all-zeros string. This can be a special case (`if`).
- You don't need recursion here!
- Instead, use both of the conversion functions you wrote earlier in the lab! Here is pseudocode:
  - Let `n` = the numeric value of the argument `S`
  - Let `x = n + 1` (this is the increment!)
  - Convert `x` back into a binary string with your other converter!
  - Give a name, say `y`, to that newly created binary string...
  - At this point, you're almost finished!

### Hint

- The tricky part is ensuring you have enough leading zeros (in front of `y`, if you used that name...)
- You could add 42 zeros in front of `y` with `'0'*42 + y`

- Now, consider the *correct* number of zeros to add... it will involve the `len` function!

**Next function:** here, you'll use the above function to write `count(S, n)`, which accepts an 8-character binary string as its argument and then counts `n` times upward from `S`, printing as it goes. Here are some examples:

```
In [1]: count("00000000", 4)
00000000
00000001
00000010
00000011
00000100
```

```
In [2]: count("11111110", 5)
11111110
11111111
00000000
00000001
00000010
00000011
```

### Thoughts:

- This means your function will print a total of `n+1` binary strings.
- You should use the Python `print` command, since nothing is being returned. We're only printing to the screen.
- You *do* need recursion here. What are the base case and the recursive case?

### Hint

- Use the `increment` function!
- Your base case involves `n` (what's the "simplest" value of `n`?)
- Your recursive case will involve `n-1`.

## Base-3: Ternary and Balanced Ternary

There are 10 types of people in the world: those who know ternary, those who don't, and those who think this is a binary joke.

**Functions #6 and #7    `numToTernary(N)` and `ternaryToNum(S)`**

### "Ordinary" Ternary

For this part of the lab, we extend these representational ideas from base 2 (binary) to base 3 (ternary). Just as binary numbers use the two digits 0 and 1, ternary numbers use the digits 0, 1, and 2. Consecutive columns in the ternary numbers represent consecutive powers of *three*. For example, the ternary number

1120

when evaluated from right to left, evaluates as 1 twenty-seven, 1 nine, 2 threes, and 0 ones. Or, to summarize, it is  $1*27 + 1*9 + 2*3 + 0*1 == 42$ .

**In a comment or triple-quoted string, explain what the ternary representation is for the value 59, and why it is so.**

Use the thought processes behind the conversion functions you have already written to create the following two functions:

- `numToTernary(N)`, which should return a ternary string representing the value of the argument N (just as `numToBinary` does)
- `ternaryToNum(S)`, which should return the value equivalent to the argument string S, when S is interpreted in ternary.

### Hint

- We're not providing starter code here, but...
- Base your solution from the corresponding functions `numToBinary` and `binaryToNum`!
- In fact, copying and pasting those functions (and then changing them as needed) is a great strategy here.

### Examples:

```
In [1]: numToTernary(42)
Out[1]: '1120'

In [2]: numToTernary(4242)
Out[2]: '12211010'

In [3]: ternaryToNum('1120')
Out[3]: 42

In [4]: ternaryToNum('12211010')
Out[4]: 4242
```

**Finale! Functions #8 and #9    balancedTernaryToNum(S) and numToBalancedTernary(N)**

### Balanced Ternary

It turns out that the use of *positive* digits is common, but not at all necessary. A variation on ternary numbers, called *balanced ternary*, uses three digits:

- + (the plus sign) represents +1
- 0 represents zero, as usual
- - (the minus sign) represents -1

This leads to an unambiguous representation using the same power-of-three columns as ordinary ternary numbers. For example,

+0-+

can be evaluated, from right to left, as +1 in the ones column, -1 in the threes column, 0 in the nines column, and +1 in the twenty-sevens column, for a total value of  $1*27 + 0*9 -1*3 + 1*1 == 25$ .

For this problem, write functions that convert to and from balanced ternary analogous to the base-conversions above:

- `balancedTernaryToNum(S)`, which should return the decimal value equivalent to the balanced ternary string S
- `numToBalancedTernary(N)`, which should return a balanced ternary string representing the value of the argument N

Again, a good strategy here is to start with copies of your `numToTernary` and `ternaryToNum` functions, and then alter them to handle balanced ternary instead.

Here are some examples with which to check your functions:

```
In [1]: balancedTernaryToNum('----0')
Out[1]: 42
```

```
In [2]: balancedTernaryToNum('++-0+')
Out[2]: 100
```

```
In [3]: numToBalancedTernary(42)
Out[3]: '----0'
```

```
In [4]: numToBalancedTernary(100)
Out[4]: '++-0+'
```

Hint

Consider that switching from a digit of value 2 to a digit of value -1 *actually decreases the value of N by 3!* To avoid changing the overall value of N, you'll have to get that three back—by adding it back in!

Though binary is the representation underlying all of today's digital machines, it was not always so—and who knows how long binary's predominance will continue? Qubits are lurking!

Website design by Madeleine Masser-Frye and Allen Wu

# CSforAll - AdditionCircuits

## CS for All

CSforAll Web > Chapter4 > AdditionCircuits

This website accompanies the textbook *CS For All* by Alvarado, Dodds, Kuenning, and Libeskind-Hadas

### Addition Circuits!

#### Getting Started With Logisim and the Starter File

This week you will design several circuits using a digital circuit design tool called Logisim.

Here are the two files you'll need:

- The Java application file: `logisim-evolution.jar`
- The (zipped) starter file (extract + use this!): `evolution.circ.zip`

This year we are using ***Logisim Evolution***

- There is an old version whose name is plain-old "Logisim"—its files are incompatible (and some machines don't run it)
- *Don't use the old version, just named Logisim. Instead, be sure to use the newer version named ***Logisim Evolution***.*

#### Need Java? USE JAVA 8!

If running Logisim requires you to install Java / Java's runtime environment (JRE), that is totally ok.

- Mac version of Java 8: `jre-8u181-macosx-x64.dmg`
- Windows version of Java 8: `jre-8u181-windows-x64.exe`
- Link to Java SE 8, all versions
- If you have a newer version of Java and need to remove it, this link has instructions for Macs; for Windows, use the built-in *Uninstaller*

## Start *Logisim Evolution*

If you've installed Java, double-clicking `logisim-evolution.jar` should start the Logisim Evolution program.

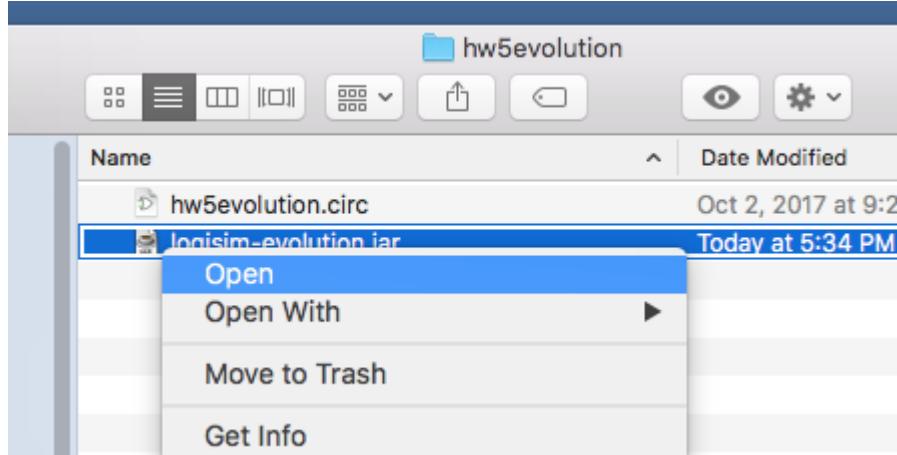
### Mac? No permission?

If you're on a Mac and it complains that you don't have permission (because it's not by a recognized developer...), not to worry. It's true that the software isn't by a formal developer (it's an open-source project). But not harmful... . To start it on a Mac, right-click or control-click the icon, and choose **Open**. It will provide an option that lets you start the application:

- Permission complaints? Some Macs will not allow you to open Logisim Evolution, saying it's not by a registered developer. You can get around this by:
  - Right-clicking (control-clicking) and choosing *Open...*:



or



- Enabling the application via your Security and Privacy settings (under System Preferences) is also possible.

*Not working?* A failsafe is to run at the command line where `logisim-evolution.jar` is located:

```
java -jar logisim-evolution.jar
```

If you find yourself needing any of these, please ask!

### ***File ... Open* The Starter Circuits**

All of your circuits will be in a *single* file:

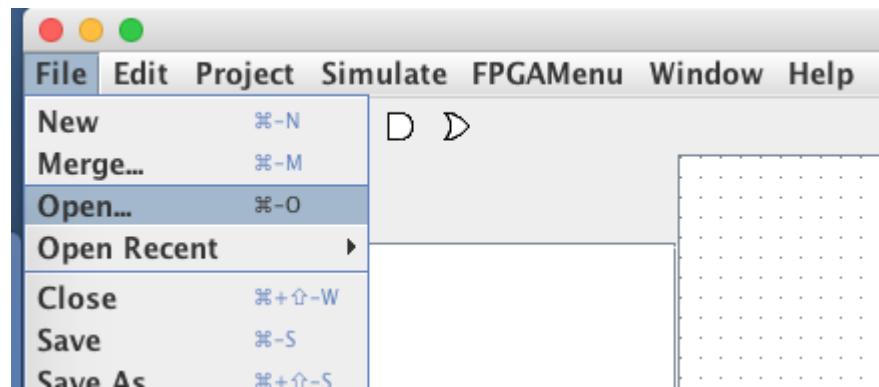
- *Don't start from scratch—please!*
- *Do* use the `evolution.circ` file

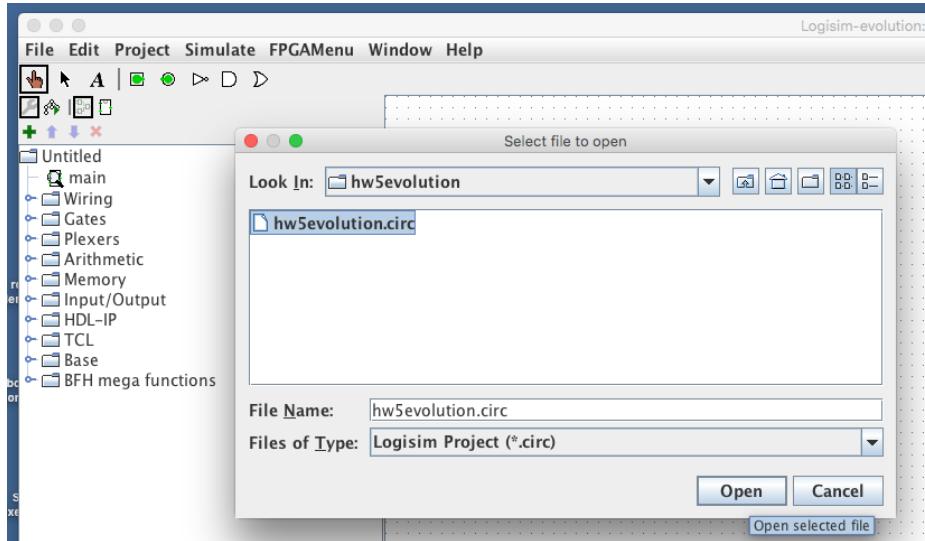
Double-clicking does NOT work.

Instead,

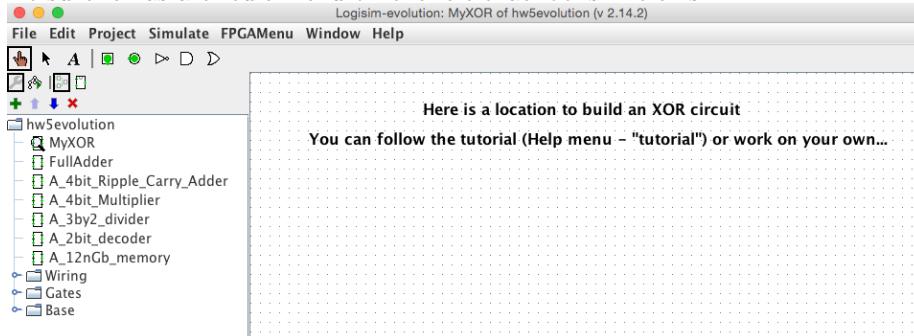
- Start *Logisim Evolution* (see above)
- Use *File—Open* from Logisim Evolution's menus, then
- Navigate to the `evolution.circ` file, and open it.

In pictures (on a Mac):





Be sure it has a circuit menu on the left that looks like this:



## Circuit Addition!

### Circuit Design!

- The `evolution.circ` file is where you will write and save all of your circuits for this assignment.
- Notice** that in your Logisim explorer pane (the upper left part of the Logisim screen) there are icons labelled MyXOR, FullAdder, A\_4bit\_Ripple\_Carry\_Adder, A\_4bit\_Multiplier, A\_3by2\_Divider, and perhaps some others. These are all of the parts ("subcircuits") of this week's homework. This lab covers only the first three of those subcircuits.

## Part 1: Tutorial and XOR

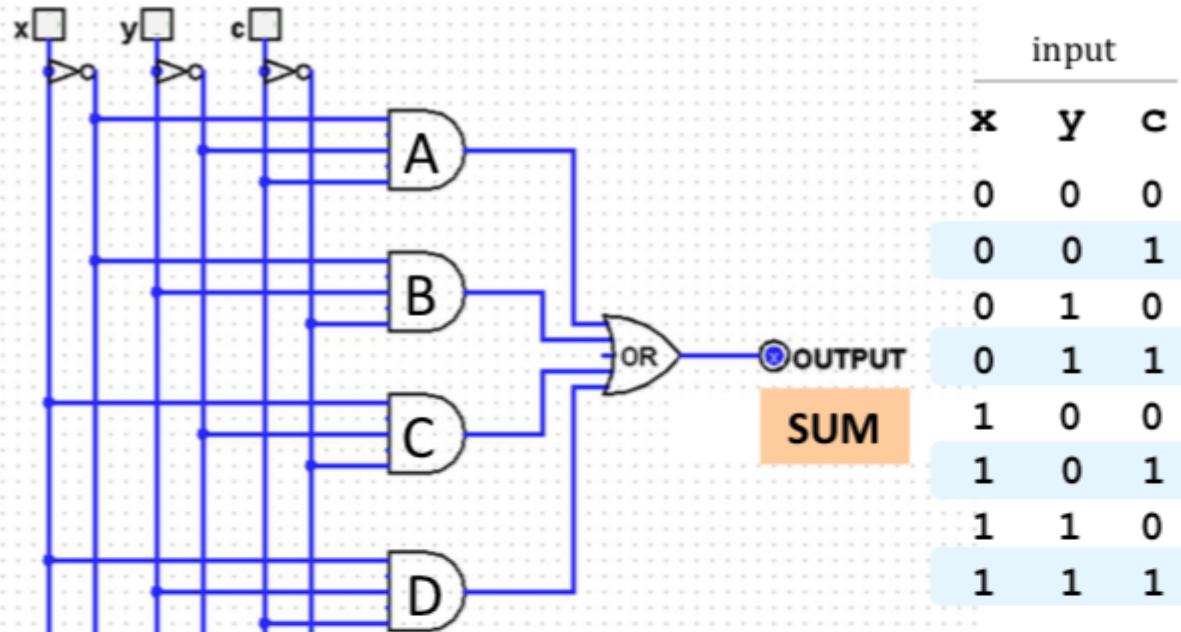
- With `evolution.circ` open, **double-click** on the MyXOR icon in the explorer pane. For the moment, you'll be working on that subcircuit. Your job here is to build a circuit to compute the XOR function using **only** AND, OR, and NOT gates.
- Go to the "Help" menu and select "Tutorial". Read through the beginner's tutorial up to and including Step 4, *testing your circuit*. As you do so, create the XOR circuit the tutorial is describing. Do this in the "MyXOR" tab of your Logisim file. This is pretty short reading, and don't worry if you don't digest every last "bit"!
  - At least you'll know what's in the tutorial so that you can go back and find the details later if you need them.
- Be sure to test your XOR circuit by changing the inputs with the "poke tool" (the little hand in the upper left of the menu bar). Check to see if you are getting the right outputs for your inputs. Save the file by going to the "File" menu and clicking on "Save".

## Part 2: Designing and Building a Full Adder

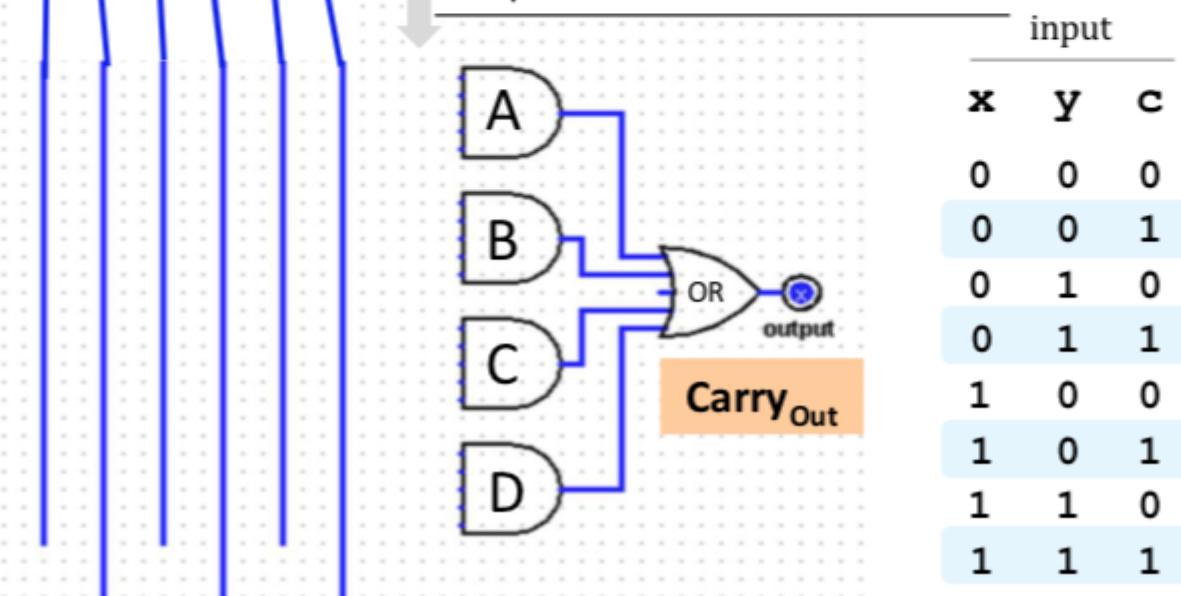
- Double-click** on the **FullAdder** subcircuit tab in your `evolution.circ` file.
- Now, you should be in the *Full Adder* circuit's explorer pane. You'll notice that we've provided the labeled inputs and outputs. **Please use those inputs and outputs!**—they help us grade the files:
  - Inputs X and Y are the two bits to be added, and **CarryIn** is the carry from the previous column.
  - Similarly, the two outputs are already placed at the bottom. You can move these if you need to, but please keep their relative positions the same so that we can easily test your circuit!
  - Notice that we have rotated the input and output pins from their normal default orientation. This was done by clicking on the item and selecting the desired rotation in the attribute pane in the lower left of the screen.
- Your task is to build a full adder (FA), which is the circuit you paper-designed in class on the *second* full-page exercise (at least in gold...).

Recall that a full adder is a device that takes **three** inputs:

- two bits to be added, and the "carry in" from the previous column of addition.
  - The FA then computes **two** outputs: the "sum" bit, and a "carry out" bit
- Thus, the first step is to look up, for example, from the class notes, the full truth table for a FA (full adder). The image below has most of it:



(2) Draw the upstream wires that will implement this function as a circuit.



- Your full adder should use the minterm expansion principle to create four AND gates for the "sum" bit and four AND gates for the "carry out" bit. You designed (on paper) a full adder in class. Now is the chance to implement it in a *verifiable* fashion!
  - You will want to create the two "rails" for the input bit named `Y` and two more "rails" for the input bit named `CarryIn`
  - There will be four AND gates whose outputs are connected to one OR gate and then the `Sum` output bit.
  - There will be four AND gates whose outputs are connected to one OR gate and then the `CarryOut` output bit.
  - Each of the two outputs demonstrates a separate application of the minterm expansion principle.
  - Please don't try to simplify your circuit and please don't use gates other than AND, OR, and NOT.
  - The purpose of this problem is to practice using the minterm expansion principle—not to optimize!
- Be sure to thoroughly test your FA before moving on. *Your FA needs to work*, because you'll use it to build the four-bit ripple-carry adder, next!

### Part 3: Building a 4-Bit Ripple-Carry Adder

- **Double-click** on the ripple-carry adder subcircuit in your `evolution.circ` file...
- Now that you have a full adder (FA), you will build a 4-bit ripple-carry adder. Recall that this device takes two 4-bit numbers and adds them up. Let's call the digits of the first number  $X_3, X_2, X_1, X_0$  where  $X_0$  is the least significant bit (the rightmost bit) and  $X_3$  is the most significant bit. Similarly, let's call the digits of the second number  $Y_3, Y_2, Y_1, Y_0$ . We wish to compute:

$$\begin{array}{r} X_3 \ X_2 \ X_1 \ X_0 \\ + \ Y_3 \ Y_2 \ Y_1 \ Y_0 \\ \hline \end{array}$$

- Notice that although there are only 4 bits in each of the two numbers, the sum can have 5 bits due to a carry!

After you've double-clicked on the "4-bit Ripple-Carry Adder" icon in the explorer pane, you can "plop" full-adders into your 4-bit ripple-carry adder by *single-clicking* on the "FA" icon from the explorer pane (since you've already designed the FA). This will create a "box" that represents your FA. After all, now that you've designed the FA, you don't particularly want to see all of its details each time you use it! If you move your mouse ("hover")

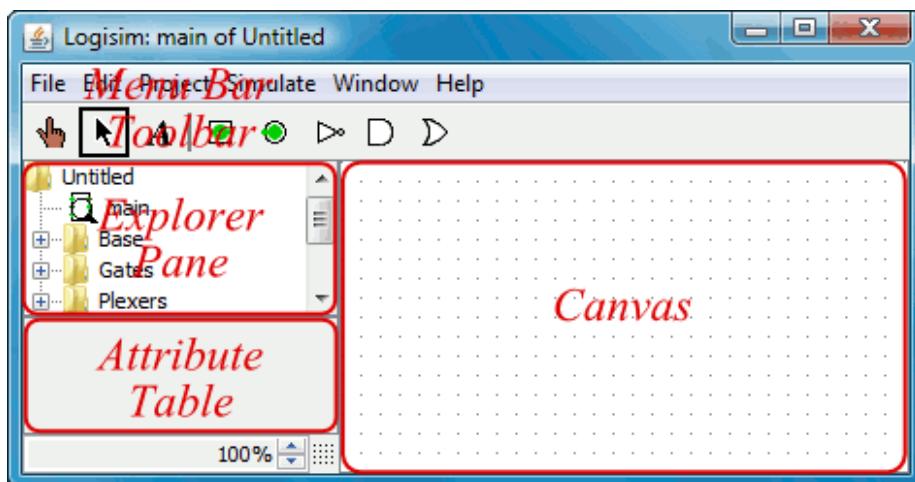
over this box, you'll see that it says "FA". If you "hover" over the "pins" (the little input and output markers on the box), it will show you what those pins are. This is because we labelled those pins when we designed the FA.

You can move the inputs and outputs that we've provided, but please keep their relative positions so that we can find them easily.

One last thing! You'll need to have a carry-in of 0 on the far right of your ripple-carry adder. One way to do this is to add an extra input and set its value to be 0. This is not a great solution, since we really don't want the user of our ripple-carry adder to be able to change that carry-in value. A better choice is to go the explorer pane, click on the "Wiring" folder, and then look down that list to the item labeled "Constant". Plop one of these down in your circuit. Then click on it and you can alter the value of that constant in the attributes pane. This is now a constant value that your circuit can use!

#### Side Panel / Circuit Menus Missing in Mac OS?

If you accidentally minimize the Explorer and Attributes Pane from the left side:



It can be frustrating to get it back! One command that we have tried (on the Mac) that has worked is (from the command-line)

```
defaults delete com.cburch.logisim
```

to see the defaults domains use

```
defaults domains | tr , "\n" | grep -v com.apple
```

Alternatively, *sometimes* you can restart the program and they'll come back. If your problem persists, ask one of us!

Website design by Madeleine Masser-Frye and Allen Wu

# CSforAll - ChangingBases

## CS for All

CSforAll Web > Chapter4 > ChangingBases

This website accompanies the textbook *CS For All* by Alvarado, Dodds, Kuenning, and Libeskind-Hadas

### Changing Bases

These problems are all about the base—of numbers' representation—and converting or computing with them.

To begin, this problem will ask you to convert from base 10 to other bases and vice versa. Because bases higher than base 10 require that new "digits" be introduced, we'll stick to bases that are between 2 and 10.

#### Function #1: `numToBaseB(N, B)`

Write a Python function called `numToBaseB(N, B)`, whose arguments are a non-negative integer N and a base B (between 2 and 10 inclusive); it should return a string representing the number N in base B.

You don't need to check to be sure that B is between 2 and 10—we will ensure those are the only values we test.

Also, remember that *integer division* (no fractional portions) is what's needed here. In Python, integer division rounds down and uses `//`, e.g.,

```
18//7 == 2
```

Here are some sample runs—be sure the first two work! Several thoughts are below:

```
In [1]: numToBaseB(3116, 9)
```

```

Out[1]: '4242'

In [2]: numToBaseB(141474, 8)
Out[2]: '424242'

In [3]: numToBaseB(42, 8)
Out[3]: '52'

In [4]: numToBaseB(42, 5)
Out[4]: '132'

In [5]: numToBaseB(42, 10)
Out[5]: '42'

In [6]: numToBaseB(42, 2)
Out[6]: '101010'

In [7]: numToBaseB(4, 2)
Out[7]: '100'

In [8]: numToBaseB(4, 3)
Out[8]: '11'

In [9]: numToBaseB(4, 4)
Out[9]: '10'

In [10]: numToBaseB(0, 4)
Out[10]: '' # notice the empty string for an argument N of 0!!

In [11]: numToBaseB(0, 2)
Out[11]: '' # notice the empty string for an argument N of 0!!

```

### Thoughts:

- Remember, that your function is returning a **string**, and not a numeric value!
- The `numToBin` example from Thursday's slides is probably the best place to start...
- Ask yourself: what has to change in order to output base B instead of base 2?

Here are some tests to paste into your file:

```

assert numToBaseB(0, 4) == ''
assert numToBaseB(42, 5) == '132'

```

Feel free to add a few more!

### Hint

- Your code **must** return the empty string when value of N is 0. (This avoids leading zeros!)
- Here are the Python functions for converting back and forth between strings and numbers. You'll find them most useful for dealing with single characters or digits!
  - `str(x)` returns the string representation of the number (integer) `x`.
  - `int(s)` returns the integer value of the string `s`. If `s` doesn't represent an int, Python stops with an error.

### Function #2: `baseBToNum(S, B)`

Naturally, we'd like to do the opposite conversion as well!

To that end, write a Python function called `baseBToNum(S, B)` that accepts as arguments a string S and a base B where S represents a number in base B where B is between 2 and 10 inclusive. `baseBToNum` should then return an integer in base 10 representing the same number as S.

You don't need to check to be sure that B is between 2 and 10—we will ensure those are the only values we test.

Here are some sample runs—be sure the first two work (they're the largest examples we'll use):

```
In [1]: baseBToNum('5733', 9)
Out[1]: 4242

In [2]: baseBToNum('1474462', 8)
Out[2]: 424242

In [3]: baseBToNum('222', 4)
Out[3]: 42

In [4]: baseBToNum("101010", 2)
Out[4]: 42

In [5]: baseBToNum("101010", 3)
Out[5]: 273

In [6]: baseBToNum("101010", 10)
Out[6]: 101010
```

```

In [7]: baseBToNum("11", 2)
Out[7]: 3

In [8]: baseBToNum("11", 3)
Out[8]: 4

In [9]: baseBToNum("11", 10)
Out[9]: 11

In [10]: baseBToNum("", 10)
Out[10]: 0 # the empty string should return 0

```

**Thoughts:**

- Remember, that your function is returning a **number**, and taking a string as its argument **S**!
- The `binToNum` example from Thursday's slides is probably the best place to start... .
- Again, the key is to ask yourself: What has to change in order to output base **B** instead of base 2?

**Hint**

- Your `baseBToNum` function should output 0 when **S** is the empty string.
- As usual, the rightmost character of the string will be the "ones' column," the *least* significant digit of the number in base **B**.
- ...but this means that the value of the rightmost character is simply `int(S[-1])`!
- Again, here are the Python functions for converting back and forth between strings and numbers:
  - `str(x)` returns the string representation of the number (integer) **x**.
  - `int(s)` returns the integer value of the string **s**. If **s** doesn't represent an int, Python stops with an error.

**Function #3: `baseToBase(B1, B2, s_in_B1)`**

Now, we can assemble what we've written to create a function called called `baseToBase(B1, B2, s_in_B1)` that takes three arguments: a base **B1**, a base **B2** (both of which are between 2 and 10, inclusive) and **s\_in\_B1**, which is a string representing a number in base **B1**.

Then, your `baseToBase` function should return a string representing the same number in base **B2**.

Here are some examples:

```
In [1]: baseToBase(2, 10, "11") # 11 in base 2 is 3 in base 10...
Out[1]: '3'
```

```
In [2]: baseToBase(10, 2, "3") # 3 in base 10 is 11 in base 2...
Out[2]: '11'
```

```
In [3]: baseToBase(3, 5, "11") # 11 in base 3 is 4 in base 5...
Out[3]: '4'
```

```
In [4]: baseToBase(2, 3, '101010')
Out[4]: '1120'
```

```
In [5]: baseBToNum('1120', 3)
Out[5]: 42
```

```
In [6]: baseToBase(2, 4, '101010')
Out[6]: '222'
```

```
In [7]: baseToBase(2, 10, '101010')
Out[7]: '42'
```

```
In [8]: baseToBase(5, 2, '4321')
Out[8]: '1001001010'
```

```
In [9]: baseToBase(2, 5, '1001001010')
Out[9]: '4321'
```

Here are some tests:

```
assert baseToBase(2, 4, '101010') == '222'
assert baseToBase(2, 5, '1001001010') == '4321'
```

### Thoughts:

- Don't rewrite any conversions at all! Instead, convert to decimal and then back to the desired final base!

### Hint

- First *use* `baseBToNum` to get the ordinary, decimal number for `s_in_B1`. Give it a name.
- Then, use the `numToBaseB` function to convert that value to the appropriate base!

#### **Function #4:    add(S, T)**

Here's a short problem that puts what you've written to use!

Write a function, called `add(S, T)` , that takes two binary strings `S` and `T` and returns their sum, *also in binary*.

We encourage you to do this by converting the two binary strings to two base-10 numbers, adding the two numbers, and then converting the resulting sum back into base 2!

Here are some sample invocations:

```
In [1]: add("11", "1")
Out[1]: '100'

In [2]: add("11", "100")
Out[2]: '111'

In [3]: add("110", "11")
Out[3]: '1001'

In [4]: add("11100", "11110")
Out[4]: '111010'

In [5]: add("10101", "10101")
Out[5]: '101010'
```

#### **Function #5:    addB(S, T)**

Above, `add` shows one way of adding two binary numbers: first convert them to base 10, add those two base 10 numbers, and then convert the result back to binary.

In this problem, however, you will implement a different, more direct, method for adding two binary numbers, using the "elementary-school" algorithm we used in class:

$$\begin{array}{r} 101110 \\ 100111 \\ \hline \end{array}$$

which, after the addition would look like this:

$$\begin{array}{r} 111 \\ 101110 \\ 100111 \\ \hline \end{array}$$

-----  
1010101

Here the "carry" bits are in blue.

For this problem, write a Python function called `addB(S, T)` that takes two string arguments. These strings are the representations of binary numbers.

Your `addB` function should return a new string representing the sum of the two argument strings.

The sum needs to be computed using the "elementary-school" binary addition algorithm, shown above and in class, and ***not using base conversions***. That is—this is *purely syntactic* addition!

Hint

- You'll need at least two base cases. A base-case question to consider is *what if S had no characters, but T still did have characters*—what would be the correct string to return?
- The "carry" case is another tricky part of this problem.
- When you need to carry, you will have THREE strings to add: (1) the rest of S (`S[:-1]`), (2) the rest of T, and the additional carry string, which is simply '1'.
- The key is to use `addB twice`: once to add the carry string with one of the two "rests," and a second time to add the first result to the other of the two "rests"!

Here's a starting point:

```
def addB(S, T):
 """ docstring - please include """
 # base cases!

 # There will be four recursive cases--here is the first one:
 if S[-1] == '0' and T[-1] == '0':
 return addB(S[:-1], T[:-1]) + '0' # 0 + 0 == 0

 # three more recursive cases still to specify...
```

Here are some samples:

```
In [1]: addB("11100", "11110")
Out[1]: '111010'
```

```
In [2]: addB("10101", "10101")
Out[2]: '101010'
```

```
In [3]: addB("11", "1")
Out[3]: '100'
```

```
In [4]: addB("11", "100")
Out[4]: '111'

...and some tests

assert addB('11', '100') == '111'
assert addB("11100", "11110") == '111010'
assert addB('110','11') == '1001'
assert addB('110101010','11111111') == '1010101001'
assert addB('1','1') == '10'
```

Website design by Madeleine Masser-Frye and Allen Wu

# CSforAll - Division

## CS for All

CSforAll Web > Chapter4 > Division

This website accompanies the textbook *CS For All* by Alvarado, Dodds, Kuenning, and Libeskind-Hadas

### Desperate for Division!

As with the previous parts of this assignment, this one should be done in your `evolution.circ` file, specifically in the "A\_3by2\_divider" subcircuit.

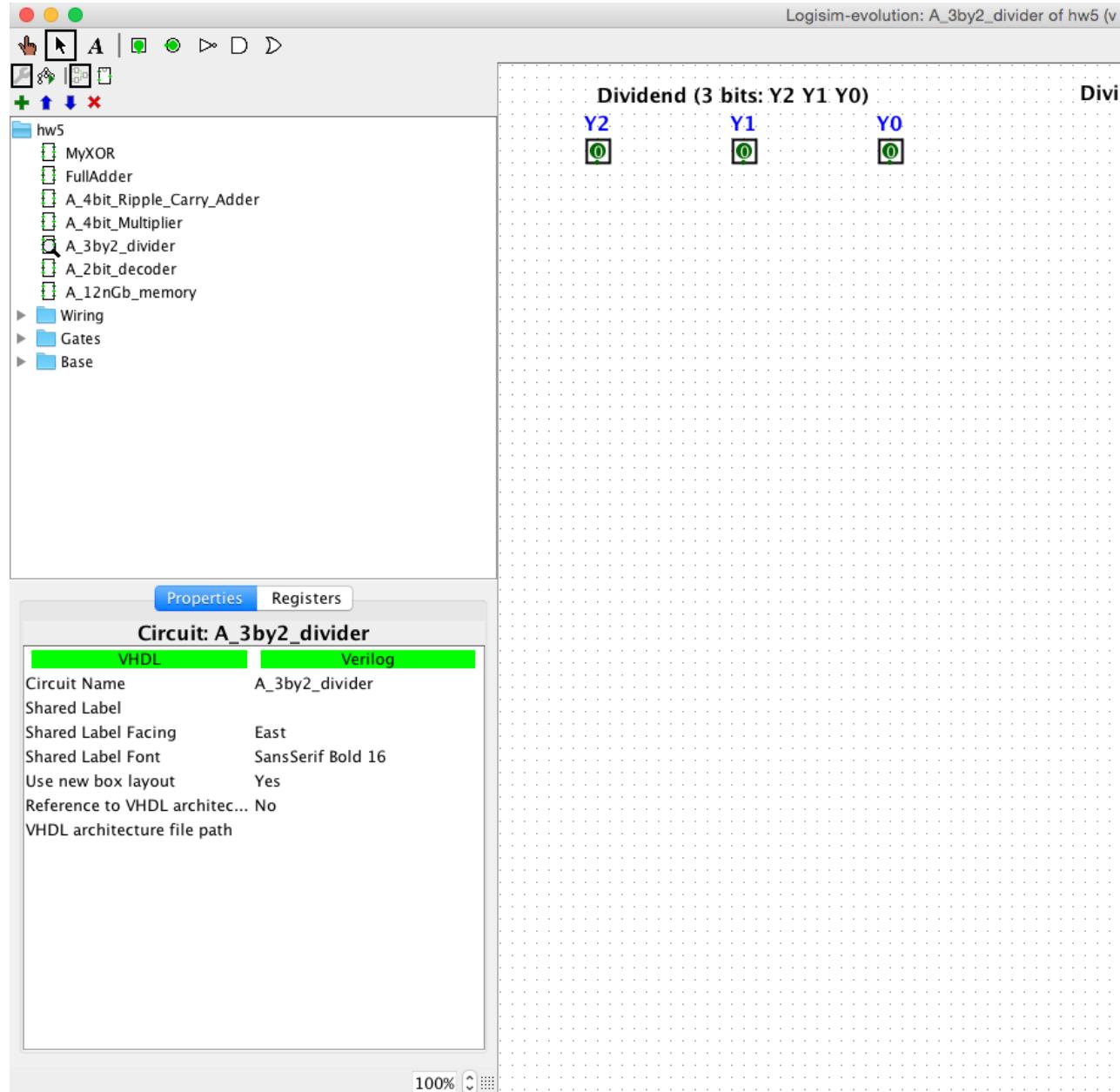
#### Setting up Your Division Circuit

For this part, use the input and output pins already provided in the *A\_3by2 divider* circuit tab within `evolution.circ`.

The inputs to the circuit are the following:

- **The dividend** can be any integer from 0 to 7 (inclusive) and so should consist of three input bits, named Y2 Y1 Y0, with Y2 the most significant bit and Y0 the least significant.
- **The divisor** can be any integer from 0 to 3 (inclusive) and so should consist of two input bits, named X1 X0, with X1 the more significant bit and X0 the less significant one.
- **The quotient** consists of four bits:
  - There are three output bits representing the quotient, named Z2 Z1 Z0, with Z2 the most significant bit and Z0 the least significant.
  - There is also one **error** bit, named **error**. This bit should be zero if the quotient is valid, one if it is invalid—which happens if and only if the divisor's value is 0.

Here is a screenshot of the "A\_3by2 divider" circuit set-up in Logisim:



### *Integer Division...*

You are free to approach this circuit using the minterm expansion principle or another combination of techniques of your choosing—see the hints below for

some additional ideas.

Keep in mind that your circuit will implement *integer* division, so that the following results should hold:

- The **error** bit should be on if and only if the divisor's value is zero. If the error bit is on, it does not matter whether any other bits in the quotient are on or off.
- The quotient should *round down* and ignore any remainders. Thus, for example,
  - 7 divided by 2 should be 3,
  - 7 divided by 3 should be 2,
  - 5 divided by 2 should be 2,
  - 5 divided by 3 should be 1, and so on...

Hint

### Idea #1: plain minterm expansion

This puts the "expansive" into minterm expansion...

This possibility is simply to create a large number of AND gates, one for each 1 in the truth table for the division circuit (see the lecture notes). You can have an AND gate for each 1 or an AND gate for each row containing a 1 (and then reuse them...).

This would implement the full truth table via the basic minterm expansion principle, which is the name for "one AND gate per 1."

- If you do this, you will need one OR gate for each output bit (to "OR" together all of the ways in which that bit might turn on)
- The nice thing about Logisim is that *you can change the number of inputs for the OR gates!*
- To do this, grab an OR gate and bring it into the canvas.
- Then, in the *attributes* panel in the lower left, find the *number of inputs* attribute
- You will be able choose a large number of inputs from there.

### Idea #2: building a subcircuit for each divisor value

Alternatively, you might break the problem into subpieces by considering each case:

- (**divisor is 0**): dividing by 0 yields an error...
- (**divisor is 1**): any dividend divided by 1 will result in passing that dividend directly to the output (quotient).
- (**divisor is 2**): division by 2 is equivalent to shifting the dividend "to the right" by one bit on its way to the quotient

- (**divisor is 3**): and dividing by three is simply tricky...

Each of these four pieces could be a subcircuit, built and tested on its own. (Many people take this path but also decide that building a subcircuit for dividing by zero may be overkill.)

To build the subcircuits, we would recommend:

- First, create each of these as new circuits.
- Copy the five inputs and (at least) the Z2, Z1, and Z0 outputs from the original divider circuit.
- Paste those five inputs and your chosen outputs into each subcircuit.
- Then, connect them appropriately to implement each specific piece of the overall division...
- Be sure that each subcircuit outputs zeros when it is not active (because the divisor value is something different).
- This latter property is what **AND** gates do!
  - For example, the divide-by-one's circuit Z0 output would be the AND of (NOT X1), (X0), and (Y0).
  - The divide-by-one circuit's Z1 output would be the AND of (NOT X1), (X0), and (Y1).
  - The divide-by-one circuit's Z0 output would be the AND of (NOT X1), (X0), and (Y2).
  - You can work in a similar manner for each other subcircuit *changing the values as appropriate!*

### **Putting the pieces together**

The final question is how each of these four subcircuits can be **combined** to produce the overall correct bits of output!

The answer is that you want to OR each of those subcircuits' outputs (Z2, Z1, Z0) together. Put those OR'ed values into the overall outputs (Z2, Z1, Z0).

- This works *as long as* each subcircuit outputs all zeros when the divisor is *different* than the one it seeks to handle.

Good luck with the divider!

Website design by Madeleine Masser-Frye and Allen Wu

# CSforAll - MultiplicationMadness

## CS for All

CSforAll Web > Chapter4 > MultiplicationMadness

This website accompanies the textbook *CS For All* by Alvarado, Dodds, Kuenning, and Libeskind-Hadas

### Multiplication Madness!

#### A Multiplication Circuit

In this problem you will build a multiplication circuit in Logisim Evolution. To check your answers, you'll need access to a calculator that can multiply in binary. The Windows calculator can do this, and there are several online calculators that can do it too—alternatively, you could write a function in Python that uses last week's base-conversion functions!

Recall that multiplying base-2 numbers is really just the same algorithm that we all learned in grade school for multiplying two base-10 numbers! Your task here is to build a circuit to compute the product of two 4-bit numbers. In the `evolution.circ` file that you downloaded for the addition circuit, there is a subcircuit named "A\_4bit\_Multiplier". Design your circuit there.

***Please use the inputs and outputs provided:*** they help to make it easier to grade the assignments, since they're standardized! (Thanks!!)

#### Notes and Hints on the 4-Bit Binary Multiplier

This problem does ***not*** ask you to use the minterm expansion principle. Instead, you'll "glue" together components that you have already built, with not-too-many additional logic gates (but lots of wire!).

Let the two numbers being multiplied be labeled X3 X2 X1 X0 and Y3 Y2 Y1 Y0, where X0 and Y0 are the least significant bits of the two numbers.

To compute the product, we must first compute the product of the number  $X_3 X_2 X_1 X_0$  with the bit  $Y_0$ . This result is called a *partial product*. This partial product can be computed very simply, using exactly four gates.

Hint

Build a 4x1 multiplier as a "helper circuit"!

Many people have found it useful to build a "helper circuit" that multiplies a 4-bit number by a 1-bit number. From there, the multiplier for this problem is simplified considerably. Feel free to do this as a design strategy, if you'd like.

To create a new circuit in the left-hand column, go to the "Project" menu and choose "Add Circuit." Perhaps name it "A\_4x1\_multiplier" so that the graders know you've taken this route. (Note that in Logisim Evolution you cannot start a circuit name with a number.)

Be sure to label the inputs to the 4x1 multiplier with the label tool (not the text tool), so that you'll be able to use them in your 4x4 multiplier!

Hint

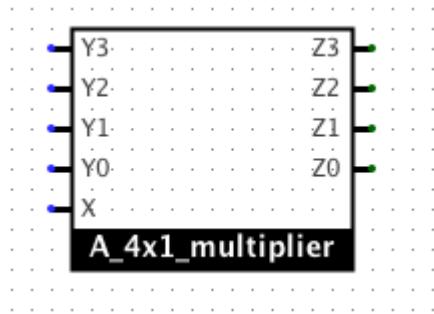
Adding the four partial products next...

Now, we need to find the remaining partial products and add up the results. You will find that it is easiest to add up the partial products as you compute them rather than waiting to add up all four partial products at the end!

Your circuit should have four inputs labeled  $X_3, X_2, X_1, X_0$  and four inputs labeled  $Y_3, Y_2, Y_1, Y_0$ . Use the 4-bit ripple-carry adders that you have already built, a small number of additional logic gates, and a lot of wire to build your multiplier.

**Note** There are two shapes to choose from for your 4x1 multipliers...

- The default layout is "caterpillar"-like:



- The smaller, old-box layout is also available (change the **Use new box layout** attribute to **No**)—it's totally up to you.

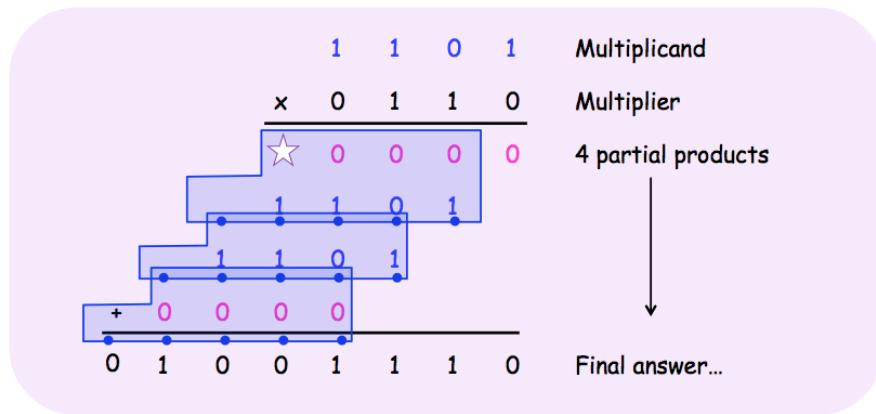
Overall, you'll need four of the 4x1 multipliers and three of the 4-bit ripple carry adders to build the multiplier.

Hint

*How does the ripple-carry-adder help?*

This image shows a graphical overlay of the multiplication:

### hw5pr2: A 4-bit multiplier



(A3) Remember that the AND gate is **single-bit** multiplication.

(A2) Use a 4x1-bit helper circuit to find the four *partial products*...

(A1) You need three (3) ripple-carry adders to finish: *see above...*



In particular, note that the blue "schoolbus" subcircuit is exactly the four-bit ripple-carry-adder that you built earlier in this homework.

Be careful! Here, ensuring the wires are connected to the correct spot is crucial!

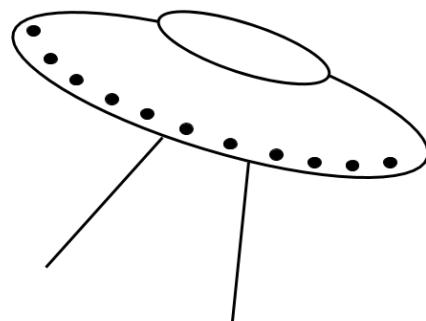
### **Design Strategies!**

You'll want to lay out your circuit neatly and carefully to make it readable and easy to build. To this end, you may find it useful to have the X inputs in a row and the Y inputs in a column, and build a 2-dimensional grid of wires. However, you're welcome to do it any way you like. If you're careful you'll find that the circuit is actually quite nice in its geometric structure and quite easy to reason about. If you're not careful, it can turn into a gnarly tangle of wires!

Website design by Madeleine Masser-Frye and Allen Wu

## CSforAll - RecursivePower

### CS for All



CSforAll Web > Chapter4 > RecursivePower

This website accompanies the textbook *CS For All* by Alvarado, Dodds, Kuennen, and Libeskind-Hadas

#### Recursive Power!

In a previous problem, you implemented a program to compute "x to the power y" using iteration.

In this problem, you'll implement the recursive program that computes x raised to the power y.

Your program only needs to handle nonnegative inputs.

Consider the following recursive Python program, in which `main()` gets input from the user and then calls a recursive "power" function that computes " $x$  to the power  $y$ " and returns it to "main" to be printed.

```
def main():
 read x from user # this isn't real python syntax, but the
 read y from user # idea here is to get input from the user.
 print(power(x, y))

def power(x, y):
 if y == 0:
 return 1
 else:
 return x * power(x, y-1)
```

### Example Run

Here is an example of the Hmmm code running to compute  $3^{**}5 == 243$

Admittedly, it's not too exciting, but that's only because all of the excitement is inside the Hmmm program itself!

```
Enter number: 3
Enter number: 5
243
```

(Be sure this doesn't accidentally compute  $5^{**}3$ , which is 125.)

### Your Task

Your job is to implement this recursive "power" code into Hmmm assembly language.

To start, you should use the provided `pr4.hmmm` file—from the downloaded folder you grabbed during the first Hmmm problem problem.

Then, remove the Hmmm program that is standing in as a placeholder there. Replace it with the recursive-factorial example from class, which is shown below.

Then, in your file, alter the recursive-factorial example to create a Hmmm implementation of a recursive `power` function as "faithfully" as possible.

By "faithful" we mean that you will have a section of Hmmm code that corresponds to "main" and a section that corresponds to "power". The "main" part will read in input  $x$  and  $y$  from the user and then jump to "power". Your "power" code will, in the general case, call itself recursively on "power( $x, y-1$ )" and then

return from that recursion to multiply its result by x and return that value. You'll need to use a function-call stack, as we experimented with during class.

Notice that, in Python, "power" does not print anything—it simply returns its value to "main", which does the printing. In Hmmm, you will want to print out the output.

```
This is the recursive factorial (from class):
```

```
00 read r1 # Read user input into r1
01 setn r15 42 # Initialize the stack pointer
02 nop # space for expansion (see pr4!)
03 calln r14 7 # Call the factorial function (at line 7)
04 nop # space for expansion (or printing)
05 write r13 # Write result to the screen
06 halt # All done!

+++ Factorial function +++
Base Case (lines 7-10)
07 jnezn r1 11 # the base-case test: is the input r1 != 0 ?
08 setn r13 1 # If r1 is 0, then the return value, r13, is 1
09 nop # room for expansion (or printing!)
10 jumpr r14 # Return the value, r13, to the line # in r14

Recursive Case (lines 11-20)
11 pushr r1 r15 # Save (push) r1 to the stack (at loc. r15)
12 pushr r14 r15 # Save (push) r14 to the stack
13 addn r1 -1 # Find N-1 and put it into r1
14 nop # room for expansion (or printing)
15 calln r14 7 # Then, ask for factorial of N-1 (Wow!)
16 nop # room for expansion (or printing)
17 popr r14 r15 # Recover (pop) r14 from the stack
18 popr r1 r15 # Recover (pop) r1 from the stack
19 mul r13 r1 r13 # Calculate r13 = N * fac(N-1) (Wow!)
20 jumpr r14 # All done! return r13 to the caller at r14
```

**Note!** Factorial and power are VERY close to each other! You will only have to change a few lines (perhaps surprisingly few) to create your recursive power if you start with the code above.

Hint

Consider making the first line

```
00 read r2
```

and then continuing with the factorial code. This will allow the base to live in `r2` and (after pushing other lines downward), will allow the power to live in `r1`. This is convenient!

In addition, if you do this, the `nop` will help you avoid renumbering the whole thing!

As with each Hmmm problem, do include a comment for each line of code, or almost every line, as shown above. Also, be sure to test your code thoroughly, and submit your file in the usual way.

Website design by Madeleine Masser-Frye and Allen Wu

# CSforAll - MemoryMadness

## CS for All

CSforAll Web > Chapter4 > MemoryMadness

This website accompanies the textbook *CS For All* by Alvarado, Dodds, Kuenning, and Libeskind-Hadas

### Memory Madness!

If you are in desperate need of more circuit design, here's a fun and rewarding challenge. Here, you'll implement the foundational circuits that enable such memory —and then build the 12 bits of memory itself.

This problem should be another circuit in your `evolution.circ` file...

### The 2-Bit Decoder

A 2-bit decoder is a device that takes two bits as input and has four outputs. The input bits are labeled `Input0` and `Input1`, and the outputs are labeled `Output0`, `Output1`, `Output2`, `Output3`. If the two input bits are 00 then `Output0` is a 1 and all the other three outputs are 0. If the two input bits are 01 then `Output1` is a 1 and all the other three outputs are 0. If the two input bits are 10 then `Output2` is a 1 and all other three outputs are 0. Finally, if the two input bits are 11 then `Output3` is a 1 and all the others are 0.

**Using the minterm expansion principle**, build a 2-bit decoder in Logisim Evolution. Your solution should use only AND, OR, and NOT gates and should be derived from the minterm expansion principle. We've already provided a subcircuit named `A_2bit_decoder` in the `evolution.circ` file that you downloaded for the first circuit problem. Place your circuit there, making sure to label your inputs and outputs as we've named them in this problem.

### The SR Latch

You can create a space for a new circuit from the "Project"—"Add circuit" menus. Name your circuit "SR\_Latch"

Next, create a new circuit and implement, in Logisim Evolution, an SR latch using NOR gates. This is the two-NOR circuit in which each output feeds back to the input of the other NOR.

It's the circuit with inputs "S" and "R" from the CS 5 notes...

### The Flip-Flop

Next, use your SR latch to implement a D latch or flip-flop (this was the end result of the CS 5 gold in-class exercise on Thursday). Call this subcircuit "Flip-flop."

### 12 nGbits of Memory!

...though it's not really one point per bit!

Finally, use your flip-flop circuit to create a 4x3 == 12-bit RAM in its entirety. It will have 4 addressable memory locations of three bits each.

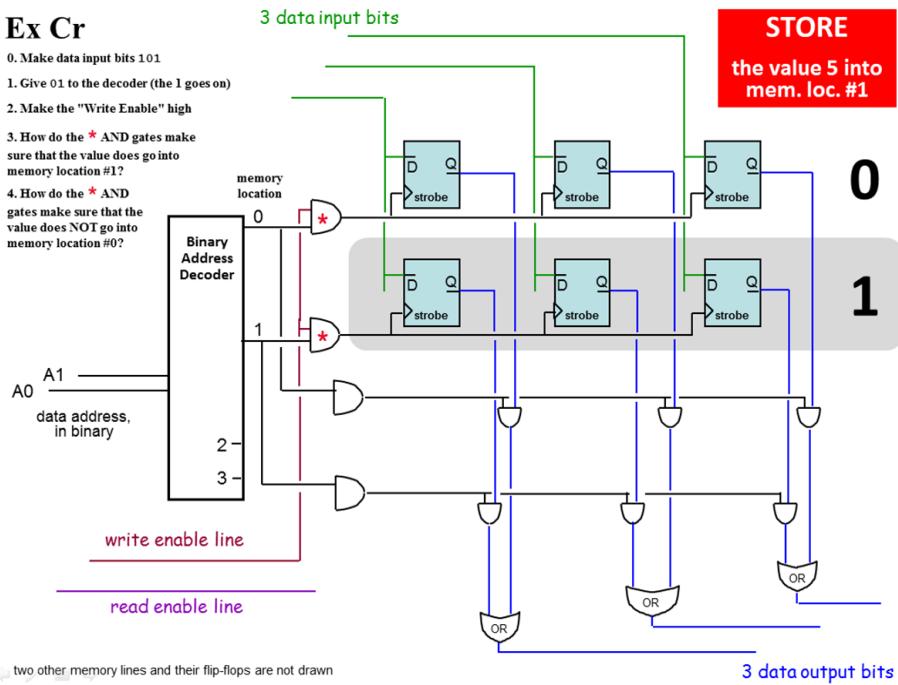
The RAM should have 2 bits of input to specify the 4 distinct memory locations (use your 2-bit decoder here!) It should have 3 bits of input for data, a 1-bit "read" indicator, and a 1-bit "write" indicator. Similarly, your RAM will have 3 bits of output data. Submit this as a subcircuit named "A\_12\_nGb\_memory" (12 nano-gigabits of memory) in your `evolution.circ` file. That subcircuit should already be present.

Make sure to label your inputs and outputs clearly!

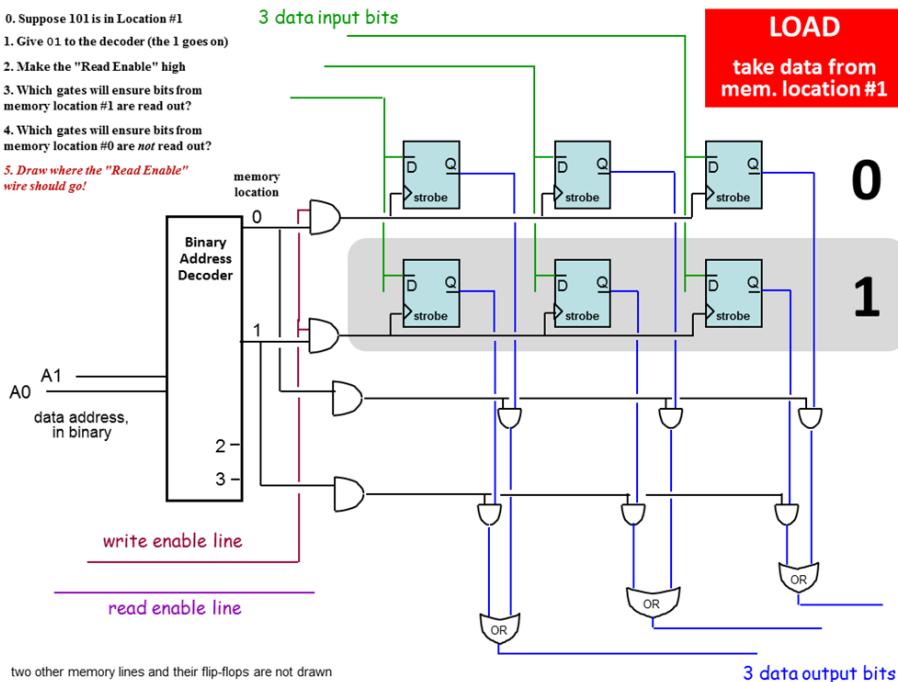
As a guide to how you might lay out the interactions (and to jog your own memory!), here are the images from Harvey Mudd's "memory" lecture. The first shows the principle of *writing*, or storing, to memory; the second shows how to *read*, or load, from memory. They include a couple of leading questions (and you can open them in a new tab for larger views...):

### Ex Cr

0. Make data input bits 1 0 1
1. Give 0 1 to the decoder (the 1 goes on)
2. Make the "Write Enable" high
3. How do the \* AND gates make sure that the value does go into memory location #1?
4. How do the \* AND gates make sure that the value does NOT go into memory location #0?



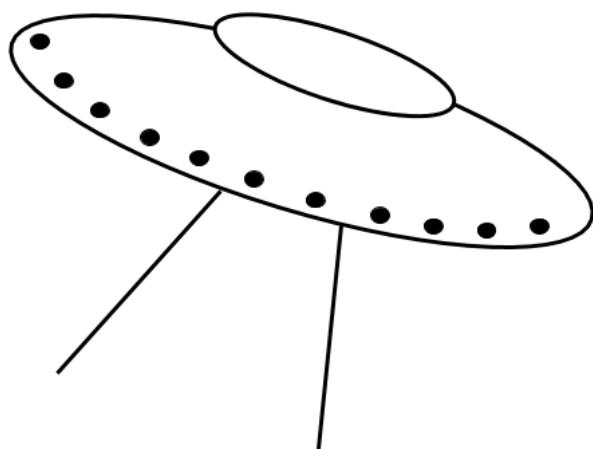
0. Suppose 101 is in Location #1
1. Give 0 1 to the decoder (the 1 goes on)
2. Make the "Read Enable" high
3. Which gates will ensure bits from memory location #1 are read out?
4. Which gates will ensure bits from memory location #0 are not read out?
5. Draw where the "Read Enable" wire should go!



Website design by Madeleine Masser-Frye and Allen Wu

## CSforAll - HmmmCountdown

### CS for All



CSforAll Web > Chapter4 > HmmmCountdown

This website accompanies the textbook *CS For All* by Alvarado, Dodds, Kuenning, and Libeskind-Hadas

### Hmmm...Assembly!

#### Starter Files!

Start by downloading the starter files, `Hmmmwork.zip` from this link. Included in this zip are starter files for all seven Hmmm problems and we highly recommend using them since the Hmmm program will be in the same directory.

For reference, Here is the Hmmm documentation: all of the instructions in the Hmmm language.

## Hmmm - Assembly Language!

This lab is all about programming in assembly language on a small computer model called Hmmm, the "Harvey Mudd Miniature Machine".

There are three parts:

- First we give you a bit of background. This short preamble will help you avoid a lot of potential pitfalls.
- Next, you're asked to write a Hmmm program that counts down to zero, starting from the *cube* of a positive input.
- The final part asks you to write a *pseudorandom-number generator* in Hmmm...

Happy Hmmming!

## Introduction to Assembly Language

Although the Hmmm model is not a physical computer, its design is fundamentally very similar to that of modern real computers.

Why not use a "real" computer? It's not worth learning the details of a real processor's instruction set: all of them are weighed down with lots of stuff that's not essential. (DNA shouldn't feel so bad!) Those many details are machine-specific: they don't apply to other computers.

The instructions in Hmmm are a subset common to *all processors*.

Hmmm has about 20 instructions. Modern computers typically have between twice and twenty times as many instructions. Engineering 85 (E85), a wonderful course taken by all engineering and many computer science students, will have you program in a real assembly language similar to Hmmm but with a larger (and more complicated) set of instructions.

## Why program in assembly language?

XKCD's reason

When a computer is first built, all it can do is execute machine-language instructions written in binary. Programming in this binary machine language is a pain! Therefore, the first thing that the designers of a computer (such as the Hmmm) typically do is write an "assembler."

With the assembler, programmers can now write programs with instructions such as add, jump, and the others in Hmmm. The assembler converts these instructions into their corresponding binary equivalents, aka the "machine language" that the computer's circuits execute.

Once an assembler exists, the next step is to use assembly language to write more powerful languages such as Python.

Don't worry—we won't ask you to implement Python in Hmmm! But writing a few short assembly-language programs will give you a sense of how one would start building such tools in the processor's native language.

There's another reason for understanding assembly language. Whenever you run a program in Python (or Java or C or...), that program is ultimately compiled (translated) into assembly or machine language so that the computer can run it. Learning to program in assembly language will allow you to understand what's **really** going on inside the machine when your program is run.

## What's the Difference Between Registers and Memory?

Hmmm has 16 registers named `r0` through `r15`, and 256 memory locations ("RAM"). A real processor might have have (about) the same number of registers as Hmmm—maybe a couple of times bigger—but would have "lots and lots" of RAM memory—typically on the order of millions or billions of memory locations that it could access.

Registers are where the computer actually *does* its computation. Registers are the limited amount of data the computer can keep in its "head."

RAM memory is where all of the other data—and *the program itself*—are located. Think of RAM as a "notebook" where the computer keeps lots of information that it can't otherwise remember.

Keep in mind that the actual program is located in memory and that the computer fetches one instruction at a time from that RAM memory into its "brain" (registers) so that it can actually execute that instruction. This process of fetching instructions from memory to execute on the processor is known as the *Von Neumann* architecture. All modern computers use the Von Neumann architecture.

## A Closer Look at Hmmm Instructions

Each instruction typically tells the computer to do something with its registers (add numbers, etc). When that instruction is done, the computer fetches the next instruction from memory. By convention, the program resides in memory beginning at memory location 0.

As in most computers, most Hmmm instructions "operate" on *registers*. If you look over your class notes, you'll see there are instructions such as `add` that take three registers as arguments, e.g.,

```
add r3 r1 r2
```

The values in the right two registers, `r1` and `r2`, are added together and the result is stored in the left-side register `r3`.

This instruction is Python's `r3 = r1 + r2`. Similarly, the `sub` (subtract), `mul` (multiply), and many other instructions operate entirely on registers.

There are exceptions! The most important are `jumpn`, `addn`, and `setn`. Note that each of these instructions ends in the letter n. This means that each takes a *raw number* as an argument. For instance, the number supplied to the `jumpn` instruction indicates the line to which the program will jump next. `Jumpn`'s relatives `jltzn`, `jgtzn`, `jnezn`, and `jeqzn` all accept a register *and* a raw number. For example, `jltzn r2 42` will jump to line 42 *only if the value in register r2 is less than zero*.

Feeding raw numbers into the instructions `addn` and `loadn` works in a very similar manner. Simply put, the command `addn r5 42` adds the number 42 to the pre-existing contents of register `r5` and then stores the result back in register `r5`.

Note that the instruction `addn rX -1` is an easy way to subtract 1 from a register!

The instruction `setn r3 42` simply puts the number 42 into register `r3`—this is also useful....

Let's get started with Hmmm!

## Downloading and Running Hmmm

To get started running Hmmm, you will need to download the zipped folder under the "starter files" heading above.

Unzip that file and you'll get a `Hmmmwork` folder with several files in it.

### Try it!

1. Try Hmmm out by cd'ing into the `Hmmmwork` folder that you downloaded and unzipped.
2. Then, run the usual command at the command-line:   `ipython`
3. Then, type `run hmmm pr1a.hmmm` to run Hmmm itself. Hmmm will assemble and run `pr1a.hmmm`.

**Not working?** You may be on Windows and need to *unzip* the folder before opening the files!

Or, if you've moved the `pr1a.hmmm` file away from the other files in that folder, it won't work—be sure to move the whole folder!

Here is what `pr1a.hmmm` looks like:

```
pr1a is an example program that
1) asks the user for an input
2) counts up from that input
3) keeps going and going...
```

```
00 read r1
get # from user to r1
01 write r1
print the value of r1
02 addn r1 1
add 1 to r1
03 jumpn 01
jump to line 01
04 halt
never halts! [use ctrl-c]

Lab task #1: Change Problem1 to "the cubic countdown"
```

After you type `run hmmm pr1a.hmmm`, you should see something like this:

```
-----| ASSEMBLY SUCCESSFUL |-----
```

```
0 : 0000 0001 0000 0001 00 read r1
get # from user to r1
1 : 0000 0001 0000 0010 01 write r1
print the value of r1
2 : 0101 0001 0000 0001 02 addn r1 1
add 1 to r1
3 : 1011 0000 0000 0001 03 jumpn 01
jump to line 01
4 : 0000 0000 0000 0000 04 halt
never halts! [use ctrl-c]
```

You will then see:

Enter number:

Things have worked and the program is running. It is asking you to enter a number (this is the effect of the `read r1` instruction at the start of the program).

Type in 32000 and hit return. You should see the simulator counting upward from 32000 or whatever number you typed in. When it gets to the maximum value, 32727, it will stop (because it overflows!).

(If you want to stop it early, hit control-c.)

### Trying another example, `example1`

Try the `example1.hmmm` example. (It's exemplary!) Here's `example1.hmmm` for you to look at:

```
00 read r1
```

```

get # from user to r1
01 read r2
ditto, for r2
02 mul r3 r1 r2
r3 = r1 * r2
03 write r3
print what's in r3
04 halt
stop.

```

Read the `example1` program instruction-by-instruction so that you understand what's happening in it; it shows how to read multiple inputs from the user and use them in a subsequent computation.

Try it out!

1. run `hmmm example1.hmmm`
2. The program will run and ask you for a number. Type 7, then type 6 for the next number.

The program will then print The Answer!

### The first problem: `pr1a.hmmm`, *Cubic Countdown*

For the first problem, `pr1a.hmmm`, your task is to change the code in the file so that it does the following:

1. First, it should ask the user for an input. Hmmm only allows integers between -32768 and 32767. For this problem, you may assume that the input will be *positive* and less than 30.
2. Next, your Hmmm program should compute the *cube* of that input. It should print the result. You'll need *multiple* multiplications to compute the cube! This will require multiple instructions. In general, it takes a *lot* more Hmmm instructions to do something than, say, Python lines of code!
3. Next, your `pr1a.hmmm` should count *downward* from that resulting value (the cube of the input) one integer at a time. It should print each one until it gets to 0.
4. When this countdown is at zero, the program should stop. The last value printed should be either 0 or 1. Either is completely OK as the final value.

#### Hint

A good way to do this problem is to write ONE step in the above list at a time and TEST your program to be sure that step works each time. For example, you might

- First, write a program that reads an input, computes and prints its cube, and then exits
- Then, *change* that program into one that reads an input, computes and prints its cube, and then prints *one less than that cube*, then exits.
- Finally, you might consider how to add a loop that will continue this process all the way to zero before exiting.

You don't *have* to use this development process, but the one-step-at-a-time approach is particularly crucial for assembly language, because it's difficult (at least for us humans!) to keep track of where everything is.

For reference, here are all of the Hmmm instructions (open in a new tab for a larger view...)

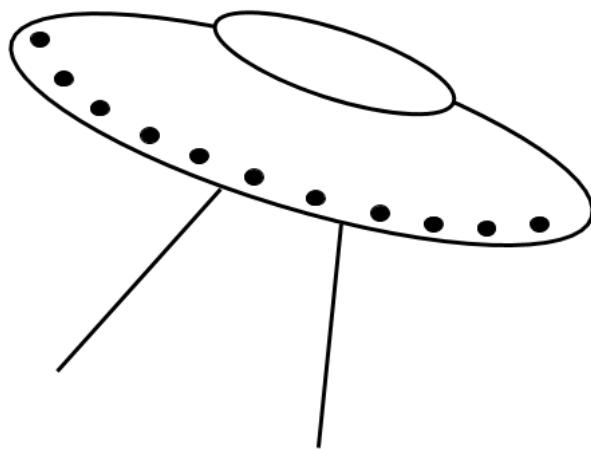
### Commenting is a must!

For this same reason—that keeping track of what is going on each step of the way is quite difficult for human readers—all of your Hmmm code should have a comment on *every line* except for null-operations, nops, which

are convenient to use as spacers so that you have room to grow your program without renumbering lines in the future.

# CSforAll - Randhmmm

## CS for All



CSforAll Web > Chapter4 > Randhmmm

This website accompanies the textbook *CS For All* by Alvarado, Dodds, Kuenning, and Libeskind-Hadas

### RandoHmmm Number Generation

For this problem, open the `pr1b.hmmm` file and the `pr1c.py` file... . Both are in `Hmmmwork`.

This starter code should already be in your `pr1b.hmmm` file:

```
00 read r1
input a
01 read r2
input c
02 read r3
input m
03 read r4
input X_0
04 read r5
```

```
input N
```

The next sections first explain these inputs, and then explain how to generate random numbers with them.

## The Math Behind Random Number Generation

In RNGs (random number generators), the numbers generated are not "*truly*" random because they are generated by a mathematical formula. Even so, they have statistical properties that make them behave in similar ways that truly random numbers would behave.

As a result, random-number generators are more precisely called *pseudo-random-number* generators. Here, we won't belabor the difference.

Our random-number generator creates a sequence of random numbers,  $x_0, x_1, x_2, \dots, x_n, x_{n+1}, \dots$ , as defined by this relationship:

$$x_{n+1} = (a x_n + c) \% m$$

where we get to choose

- $a$  a multiplier
- $c$  an additive amount ("offset")
- $m$  a divisor, but only the "mod" or remainder is used!
- $x_0$  a "seed value" or starter value, with  $0 \leq x_0 < m$ , i.e., between 0 and  $m$  (excluding  $m$ )

In typical random-number generators, these values are chosen for the user. Often this is done by using the number of milliseconds on the computer's clock.

Here, we will choose each of these values ourselves. We will also choose  $N$  the user is asked to enter the seed value,  $x_0$ .

## How Our Random-Number Generator Works...

You will implement a random-number generator named LCG, short for linear congruential generator.

The way an LCG random-number generator work is

1. Start with the seed value,  $x_0$       Thus,  $n$  is 0
2. Use the formula  $x_{n+1} = (a x_n + c) \% m$  to generate  $x_{n+1}$
3. Let  $x_{n+1}$  serve as the *new* seed value
4. Go back to step 2 and continue generating new values in this way, as long as you'd like...

Here is an example using small numbers:

- $a = 5$
- $c = 1$
- $m = 7$
- $x_0 = 0$

| i | Formula          | $X_i$ |
|---|------------------|-------|
| 0 | 0                | 0     |
| 1 | $(5*0 + 1) \% 7$ | 1     |
| 2 | $(5*1 + 1) \% 7$ | 6     |
| 3 | $(5*6 + 1) \% 7$ | 3     |
| 4 | $(5*3 + 1) \% 7$ | 2     |
| 5 | $(5*2 + 1) \% 7$ | 4     |
| 6 | $(5*4 + 1) \% 7$ | 0     |
| 7 | $(5*0 + 1) \% 7$ | 1     |

From here on out, everything repeats...

Notice that the random numbers generated this way are always between 0 and  $m-1$  because we are "modding" our numbers by  $m$ . The `mod` command is already in `Hmmm` (you don't need to write it!).

In the above example, the LCG repeats after generating  $m$  random values. If an LCG repeats after generating  $m$  numbers, it is said to have **full period**. This is a desirable property, since it means that it generates the largest possible amount of values before repeating.

Below, you'll make sure your generator has this property.

## Writing the RandoHMMM Number Generator

Your task is to implement this LCG algorithm in `Hmmm`.

First, the one-line summary: your program will generate and print  $N$  pseudorandom numbers, using our updating rule:

$$X_{n+1} = (a X_n + c) \% m$$

### Details

Your program should work as follows: The user will input **five** values in the following order. These inputs should already be handled by `pr1b.hmmm`:

- First, the user enters the number  $a$ , the multiplier in the LCG algorithm.
- Second, the user enters the number  $c$ , the increment in the LCG algorithm.
- Third, the user enters the number  $m$ , the modulus divisor in the LCG algorithm.
- Fourth, the user enters the seed,  $X_0$ , in the LCG algorithm.
- Fifth, the user enters a number  $N$ , indicating the number of pseudorandom numbers that should be printed.

Then, your `Hmmm` `Random` number generator should

- First, check to see if there are any more numbers to generate (base case!). Which register holds the value to check?
  - If there are no more to generate, jump to the end of the code
  - You may not know where the end will be yet—perhaps go too far (15 is too far) and come back to this line when you do know...
- Next, sequence the instructions you need to create the **next** value of  $X$ , e.g.,  $X_1$ 
  - Remember, from above, that  $X_1 = (a X_0 + c) \% m$
  - Or, in general,  $X_{n+1} = (a X_n + c) \% m$
- Note that it's totally OK to re-use the  $X_0$  register one or more times on the way to computing  $X_1$ , and so on...
- Once you've computed  $X_1$ , print it out
  - Don't print  $X_0$ : that's considered the *seed* of the random number generator
  - However, all the rest of the  $X_n$  values (with  $n \geq 1$ ) *are* considered part of the random-number stream, and those should be printed.
- Finally, be sure to update things that need to be updated, such as  $N$ , and then `jmpn` back, as appropriate.
  - You do need a `halt` in every `Hmmm` program at the very end of all of the branches.
- Note that—at this point—you'll know how large your program will be—and you'll be able to use that in any `jmpn` lines that may have been waiting...

## Summary and Hints...

Thus, your `Hmmm` program in `pr1b.hmmm` should print  $N$  pseudorandom numbers, beginning with  $X_1$  (again,  $X_0$  is not considered one of the pseudorandom numbers and is not printed.)

- Note that `mod` is built-in to `Hmmm`! *Don't rewrite mod yourself!*
- You may find you need to *copy* one register into another...
- if so, the `copy r4 r8` command copies the contents of register `r8` *into* register `r4`.
  - **Caution!**: the `copy` command is right-to-left.
  - **Aside** This is why Python's assignment operator is right-to-left!

For reference, here are all of the Hmmm instructions (open in a new tab for a larger view!)

## Checking Your Generator

To check that your random-number generator is working, try running it with the following inputs:

- First, enter the number  $a = 10$ , the multiplier in the LCG algorithm.
- Second, enter the number  $c = 7$ , the increment in the LCG algorithm.
- Third, enter the number  $m = 11$ , the modulus in the LCG algorithm.
- Fourth, enter the seed,  $x_0 = 3$ , the starting value in the LCG algorithm.
- Fifth, enter the number  $N = 10$ , indicating that 10 pseudorandom numbers should be printed.

More briefly:

```
10
7
11
3
10
```

The output should then be alternating 4s and 3s:

```
4
3
4
3
4
3
4
3
4
3
```

Clearly, these are **not** good values for our random-number generator: they are not very "random"! The next section will ask you to determine better -- then ideal -- values for  $a$  and  $c$ .

## Part 2: Picking Good Input Values for Random-Number Generation

Your boss at **SPRANG** Corporation (Spam-Processed RAndom Number Generation) has asked you to construct a random-number generator with  $m$  equal to 100.

You still need to find values of  $a$  and  $c$  that can be used with this value of  $m$  and produce *reasonable* random numbers. So, in this part you will choose "good" values for the parameters  $a$  and  $c$  in our random-number-generator algorithm.

Again, we will use  $m = 100$

### In Practice

Try two or three different values of  $a$  and  $c$  (with  $m = 100$ ,  $x_0 = 42$ , and  $N = 100$ ).

Perhaps  $a = 10$  and  $c = 5$  to start?

How different are all 100 outputs? Just judge this by looking at the output...

For completeness, here is the full set of values we're using at this point:

- First, enter the number  $a = 10$ , the multiplier in the LCG algorithm.
- Second, enter the number  $c = 5$ , the increment in the LCG algorithm.
- Third, enter the number  $m = 100$ , the modulus in the LCG algorithm.
- Fourth, enter the seed,  $x_0 = 42$ , the starting value in the LCG algorithm.
- Fifth, enter the number  $N = 100$ , indicating that 100 pseudorandom numbers should be printed.

More briefly:

```
10
5
100
42
100
```

### In Theory

It turns out that the LCG algorithm has its best-possible performance—that is, it generates a full set of  $m$  different values before repeating—if the following three conditions are met:

- Condition 1:  $c$  is relatively prime to  $m$  (this means  $c$  and  $m$  have no common divisors other than 1).
- Condition 2:  $(a-1)$  is divisible by all *prime factors* of  $m$  (not *all* factors of  $m$ , all *prime factors* of  $m$ ).
- Condition 3:  $(a-1)$  must be a multiple of 4 if  $m$  is a multiple of 4.

Note that

- The prime factors of  $m = 100$  are only 2 and 5.
- $100$  is a multiple of 4.
- Conditions 2 and 3, above, are not about  $a$  but about  $a-1$ .

With these facts in mind, see if you can find the *smallest* values of  $c$  and  $a$  that successfully generate **all** 100 values for  $m = 100$ . Keep  $x_0 = 42$  and  $N = 100$ .

It's *not* crucial that you find these on your own—if you're not sure, please ask!

However you find the numbers, **do test them out** with `pr1b.hmmm`—*Does it seem as though all 100 possible values are produced?*

You'll confirm this next.

### Part 3: Writing `unique`

But—it's difficult to be *sure* that there are no repeats in a list of 100 numbers. For this reason, your boss at SPRANG has asked you to *check* that all of the values from your LCG program are unique. This check can and should use Python.

To that end, open the file named `pr1c.py`, and in it edit the starter Python function named `unique(L)`.

Your function `unique(L)` will take a list as its only input and should return `True` if that list has only unique elements (no elements repeated) and `False` otherwise. A few hints:

- Your function needs to use only recursion, list indexing, and slicing.
- A good base case will check for the simplest possible list  $L$ . *What is the simplest possible list of values,  $L$ ?* It's not 0, but the 0-type *list*.
- You'll need two more cases, as well.
- You will find Python's `in` operator useful for checking if  $L[0]$  is in the rest of the list!

### Part 4: Testing `unique`

To test `unique`, simply run the file and then try a few inputs, e.g.,

```
>>> unique([7,8,9,1001,42])
```

```
True
```

```
>>> unique([2, 42, 3, 42])
```

```
False
```

## Part 5: Generating and Testing Your 100 Values

Now, we'll use your `unique` function to check whether your LCG program with the parameter values that you computed for `a` and `c` really give you full period when you run with `m` set to 100:

- First, run your Hmmm LCG program with those parameter values.
- Use any seed you want, from 0 to 99. We use 42.
- Also, make the fifth input `N` equal to 100, so that you will get 100 numbers of output.
- Finally, you need to check if those 100 numbers are unique.

*Fortunately*, you have your `unique` function.

*Unfortunately*, `unique` takes a list as its argument, **but** the output that we have on our screen is not a list (no commas and no brackets at the beginning and end).

No worries! First, notice the following bits of Python code in the `pr1c.py` file:

```
You'll paste your 100 numbers in this triple-quoted string:
```

```
NUMBERS = """
```

```
3
```

```
42
```

```
47
```

```
46
```

```
91
```

```
5
```

```
"""
```

```
def unique(L):
```

```
 """
```

```
 # you wrote this above ...
```

```
def test(S):
```

```
 """
```

```
 # see pr1c.py for this function ...
```

There are two things to note in the code above: first, there is a triple-quoted string named `NUMBERS` that contains a few integers.

Second, there is a small function named `test` that accepts a string (`S`), which certainly can be a triple-quoted string, and then uses `unique` to determine whether all of the integers in that triple-quoted string are unique.

The commented-out `print` statements are there in case you'd like to get a bit more intuition about how `test` works.

## Using the `test` Function

Copy and paste the output of your Hmmm program (100 numbers) into the triple-quoted string named `NUMBERS` *replacing the current contents of that string*.

Re-run the file. Then, run

```
test(NUMBERS)
```

at the Python shell's prompt.

If all goes well, Python will output `True`.

This means your 100-number random number generator did, in fact, generate 100 unique values, the best-possible result.

### **Finishing Up...**

Once you've checked your program to verify that it is correctly giving you 100 different values, lab is complete!

### **Update Your Resume!**

Congratulations! You can now officially put Hmmm on your resume, alongside Logisim (and Picobot!)

# CSforAll - HmmmPower

## CS for All

CSforAll Web > Chapter4 > HmmmPower

This website accompanies the textbook *CS For All* by Alvarado, Dodds, Kuenning, and Libeskind-Hadas

### Hmmm Power!

For this problem, you'll create a Hmmm program file in the folder you downloaded for the into Hmmm problem.

Keep this `pr2.hmmm` file in the *same Hmmmwork folder*, since you'll need the other files in it!

Your program should first ask for *two* nonnegative numbers from the user. Then, the program should compute the result obtained when you raise the first input to the power of the second input. Finally, it should print that result and then `halt`.

You can use this factorial example as a guide to how the program should work:

```
00 read r1 # Get number from user into r1
01 setn r2 1 # Put our result into r2
02 jeqzn r1 08 # Jump to line 7 if r1 == 0
03 mul r2 r2 r1 # Make r2 = r2 * r1
04 addn r1 -1 # Make r1 = r1 - 1
05 jumpn 02 # Jump back to line 2
06 nop
07 nop
08 write r2 # Write out the result, r2
09 halt
```

This is not a solution to this problem—remember that you'll need to edit this code to implement `power` instead of `factorial`!

Hint

- Use the existing `read` statement to get the base from the user.

- Add another `read` statement to get the power, say, into `r2`.
- *Keep 1 as your initial result value! (aka "Base Case")* Perhaps use `r3`.
- Then `test` to see if you're finished:
  - You're probably finished when the power == 0
- If you're not finished, you need to multiply once, reduce the remaining powers, and loop!

For this problem, you may assume that both inputs `n` will always be at least 0. Calculating 0 to the 0 power (or any other number to the 0 power) should result in 1. Here are a couple runs' worth of sample input and output:

```
Enter number: 2
```

```
Enter number: 5
```

```
32
```

```
Enter number: 42
```

```
Enter number: 1
```

```
42
```

```
Enter number: 42
```

```
Enter number: 0
```

```
1
```

```
Enter number: 0
```

```
Enter number: 0
```

```
1
```

Remember—you should have a comment of one line or more for every line of code that you write in order to explain what that line is doing. Also, test your program carefully, including the "edge cases" when one or both inputs are zero.

Website design by Madeleine Masser-Frye and Allen Wu

# CSforAll - FibonacciFun

## CS for All

CSforAll Web > Chapter4 > FibonacciFun

This website accompanies the textbook *CS For All* by Alvarado, Dodds, Kuenning, and Libeskind-Hadas

### Fibonacci Fun

The Fibonacci sequence is one of the most famous sequences of numbers in mathematics. The first Fibonacci number is 1, the second Fibonacci number is 1, and in general, the next Fibonacci number in the sequence is the sum of the previous two. The first few numbers in the sequence are 1, 1, 2, 3, 5, 8, 13, 21.

Your job is to write a Hmmm assembly language program in pr3.hmmm that takes a single input from the user, call it `n`, and prints the first `n` Fibonacci numbers. It's easiest to use the `pr3.hmmm` file from the folder you downloaded in the first hmmm problem.

You will probably want to copy the contents of one register `rX` into another `rY` during the course of this problem. Take a look at the Hmmm documentation—the command for copying `r1` into `r2`, i.e., `r2 = r1` in Python, is

```
copy r2 r1
```

Note that this instruction moves data from right to left, which mimics Python's assignment statements, e.g., `y = x` assigns `x`'s value into `y`.

For this problem, you may assume that the input `n` will always be at least 2. Here is one sample input and output:

```
Enter number: 10
1
1
2
3
```

5  
8  
13  
21  
34  
55

Remember to have a comment for every line of code that you write. Also, test your program carefully, starting at `n == 2`.

Website design by Madeleine Masser-Frye and Allen Wu

# CSforAll - ImageCompression

## CS for All

CSforAll Web > Chapter4 > ImageCompression

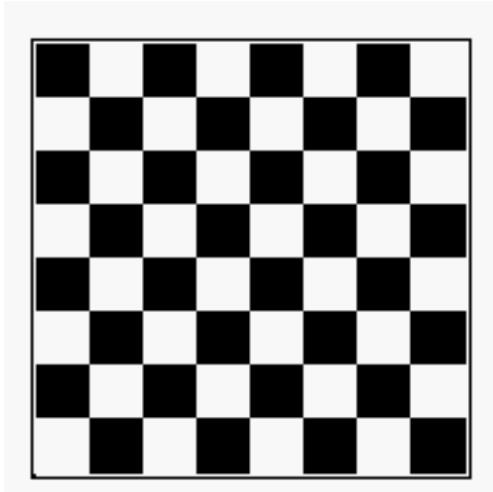
This website accompanies the textbook *CS For All* by Alvarado, Dodds, Kuenning, and Libeskind-Hadas

### Run-Length Image Compression

Up to now we've explored how numbers are symbols are represented in binary, but in this problem we'll explore the representation of images using 0's and 1's.

For this part you'll write two functions, `compress(I)` and `uncompress(C)`, along with one or more helper functions.

Let's begin by considering just 8-by-8 black-and-white images such as the one below:



Each cell in the image is called a "pixel". A white pixel is represented by the digit 0 and a black pixel is represented by the digit 1. The first digit represents the pixel at the top left corner of the image. The next digit represents the pixel in the top row and the second column. The eighth bit (digit) represents the pixel at the right end of the top row. The next bit represents the leftmost pixel in the second row and so forth. Therefore, the image above is represented by the following binary string of length 64:

```
'101010100101010110101010010101011010100101010110101001010101'
```

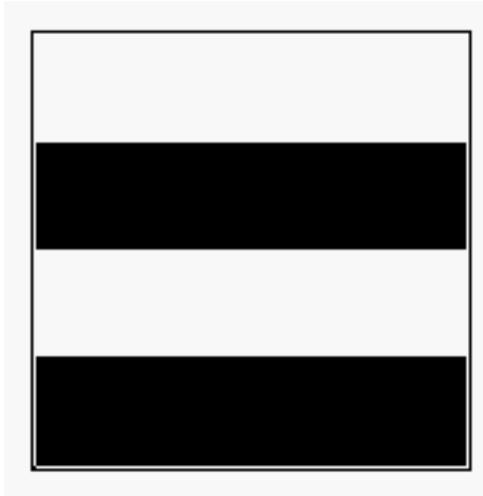
Another way to represent that same string in python is

```
'1010101001010101'*4
```

### The Background Story

So now what? Here's the gratuitous background story: You've been hired by MASA ("Mudd Air and Space Administration"). MASA has a deep-space satellite that takes 8-by-8 black-and-white images and sends them back to earth as binary strings of 64 bits as described above. To save precious energy required for transmitting data, MASA would like to "compress" the images sent into a format that uses as few bits as possible. One way to do so is to use the **run-length encoding algorithm**.

Imagine that we have an image that looks like this, for example:



Using our standard sequence of 64 bits, this image is represented by a binary string beginning with 16 consecutive 0's (for two rows of white pixels) followed by 16 consecutive 1's (for two rows of black pixels) followed by 16 consecutive 0's followed by 16 consecutive 1's.

Run-length encoding (which, by the way, is used as part of the JPEG image compression algorithm) says: Let's represent that image with the code "16 white, 16 black, 16 white, 16 black". That's a much shorter description than listing out the sequence of 64 pixels "white, white, white, white, ...".

### Run-Length Encoding

In general, our run-length coding represents an image by a sequence (called a "run-length sequence") of 8-bit bytes:

- The first bit of each byte represents the **bit** that will appear next in the image, either 0 or 1.
- The final seven bits contain the number *in binary* of those bits that appear consecutively at the current location in the image.

Notice that this run-length encoding will result in a relatively small number of bits to represent the 4-stripe image above. However, it will do very badly (in terms of the number of bits that it uses) in representing the checkerboard image that we looked at first. In general, run-length encoding does a good job "compressing" images that have large blocks of solid color. Fortunately, this is true of many real-world images, such as the images that MASA gets, which are mostly white with a few black spots representing celestial bodies.

**Function:**    `compress(S)`

Whew! So here's your task.

Write a function called `compress(S)`, whose argument is a binary string `S` of length less than or equal to 64 and returns another binary string as its result. The resulting binary string should be a run-length encoding of the original, as described above.

You may need a helper function or two—you may name them whatever you like. Also, you may want to copy a function or two from the Binary and/or Changing Bases problems!

Function: `uncompress(C)`

Next, write a function called `uncompress(C)` that "inverts" or "undoes" the compressing in your `compress` function.

That is, `uncompress(compress(S))` should give back `S`. Here's a cute test illustrating that point:

```
assert uncompress(compress(64*'0')) == 64*'0'
```

Again, helper functions are OK, as is using this week's previous problems or lab code you've written.

Here are a couple of examples of `compress` and `uncompress` in action:

```
In [1]: compress(64*'0')
Out[1]: '01000000'
```

```
In [2]: uncompress('10000101') # 5 1's
Out[2]: '11111'
```

```
In [3]: compress('11111')
Out[3]: '10000101'
```

```
In [4]: Stripes = '0'*16 + '1'*16 + '0'*16 + '1'*16
```

```
In [5]: compress(Stripes)
Out[5]: '0001000010010000001000010010000'
```

Website design by Madeleine Masser-Frye and Allen Wu

# CSforAll - RecursiveFibonacci

## CS for All

CSforAll Web > Chapter4 > RecursiveFibonacci

This website accompanies the textbook *CS For All* by Alvarado, Dodds, Kuenning, and Libeskind-Hadas

### Hmmm: Recursive Fibonacci

You've implemented a loop-based Hmmm program that writes out the Fibonacci sequence beginning 1, 1, 2, 3, 5, 8, 13, 21, ...

For this problem, you'll rewrite that Fibonacci generator, *recursively*.

It's easiest to write your code in the `pr5.hmmm` file included in the `Hmmmwork` zip since the `hmmm` program is already in the same directory.

Instead of sequentially, the Fibonacci sequence can expressed *recursively* by the mathematical *recurrence relation*:

```
Fib(1) = 1
Fib(2) = 1
Fib(n) = Fib(n-1) + Fib(n-2)
```

This extra-credit question asks you to create a recursive Hmmm program that produces **only one term** in the Fibonacci sequence...but that does so by implementing a function for computing the *n*th Fibonacci number.

### The Recursive Python Code to Emulate

Here is a recursive Python program for computing the *n*th term in the Fibonacci sequence:

```
def fib(n):
```

```

if n == 1 or n == 2:
 return 1
else:
 return fib(n-1) + fib(n-2)

```

Take a moment to make sure that you see why we need to have two base cases (`if n = 1 or n = 2`) in this case. Ask yourself what would happen if we only had the base case `if n == 1`.

Note that this program could have also been written without using recursion, but the objective here is to flex our recursion muscles. However, here is a slightly expanded version of the above recursive program that simply breaks up the steps into smaller ones. **It is this expanded version** that we ask you to rewrite in Hmmm—again converting the program as faithfully as possible:

```

def main():
 r1 = int(input()) # Line 1, get input from user
 r13 = fib(r1) # Line 2, call fib with input r1
 #print(r1) # (only for debugging - not to submit) Line 3, print r1
 print(r13) # Line 4, print the r1-th Fibonacci number
 return # Line 5, halt

def fib(r1):
 if r1 - 2 <= 0: # Line 6, if r1 - 2 <= 0 then this is fib(1) or fib(2), and the ans...
 return 1 # Line 7, so return 1
 else: # Line 8, otherwise...
 r13 = fib(r1-1) # Line 9, compute fib(r1-1)
 r3 = r13 # Line 10, store that result in another place, since r13 is special
 r13 = fib(r1-2) # Line 11, compute fib(r1-2)
 r13 = r13 + r3 # Line 12, now add the result from fib(r1-1) and fib(r1-2) and...
 return r13 # Line 13, ...that's the value that will be returned in r13

```

Be sure to put a comment in each line of your Hmmm program explaining which line in the above Python code it is implementing!

In a number of cases, several lines of Hmmm will be required to implement a single line of Python.

**Important:** Keep in mind what the "precious belongings" are that need to be pushed onto the stack for safekeeping. The `main` function has just one precious commodity when it calls `fib`—it's `r1` (notice that `main` uses `r1` by printing it after `fib` returns, so it is imperative that it be saved before the function call). The recursive call in line 9 has two precious belongings. (What are they?) The recursive call in line 11 has **three** precious belongings at that point.

### Walk-Through for this (Challenging!) Problem...

This is certainly a challenge—though, if completed, you will be able to say you've written recursive functions—in *assembly*—for what that is worth...

Here is a section-by-section overview of how one possible solution could work:

The idea is to translate into assembly (Hmmm) the *expanded* version of the generator above. This involves function calls (and recursion). A good analogy is the recursive factorial we analyzed in class, with its translation summarized in the slide that has the full program with lines and labels from its version of the "expanded" Python program.

The most important part is to have a high-level overview of the program first—then, the pieces can be assembled more confidently. Here are the pieces you'll need. The numbers of lines are likely overestimates (you can always use nops!)

- **Lines 0-10: implement main** (input, set start of stack (`setn r15 60`—be sure to use a value larger than your last line of code!), call to the fib function on line 11, printing afterwards)
- **Line 11: let's call this the start of the Fibonacci function** (you will `call` only to here, i.e., `calln r14 11`)
- **Lines 11-16: the BASE CASE of the Fibonacci function** (inspired by factorial's base case—this one is a bit different in details, but not in structure)
- **Line 17: starts the recursive case for the Fibonacci function** (you should jump here when the base case doesn't run)
- **Line 17-20: PUSH stuff on the stack** (to get ready for the recursive call to `fib(r1-1)`, you need to push `r1` and `r14` to the stack)
  - Note that this is a direct copy of the two-line PUSH block of Hmmm instructions from the factorial example...
- **Lines 21-25: setup and call of fib(r1-1)** (remember the call will be `calln r14 11`) (though you may not need 5 lines)
- **Lines 26-29: POP stuff back** (now that `r13` is the  $(r1-1)$ 'th Fibonacci number, you need to restore `r14` and `r1` from the stack to continue from "where things were" ...)
  - Note that this is a direct copy of the two-line POP block from factorial...
- **Line 30: copy r3 r13** This is a one-instruction translation of line 10 from Fibonacci's expanded Python, above.
- **Lines 31-36: PUSH stuff on the stack** (yes—again! For the upcoming call of `fib(r1-2)` Wow!)
  - Note that you need to push `r3`, since it's now a valuable piece of information needed later (along with the always-valuable `r1` and `r14`)

- Thus, you will need *three* lines!
- **Lines 37-41:** setup and call of fib(r1-2) (again, calln r14 11—and you may not need five lines)
- **Lines 42-47:** POP stuff back from the second call (again, *three* lines *will* be needed)
- **Line 48: Implement one instruction** corresponding to line 12 from the expanded Python, above
- **Line 49: Implement one instruction** corresponding to line 13 from the expanded Python, above

Phew! It's a lot—and a good indication of why this translation is done by machine, not by humans (anymore...). The reason we tackle it for one week is to really understand what all programming languages do (they translates from "natural" syntax to assembly language...).

Good luck with this (*very*) challenging problem!!

Website design by Madeleine Masser-Frye and Allen Wu

# CSforAll - Cryptography

## CS for All

CSforAll Web > Chapter4 > Cryptography

This website accompanies the textbook *CS For All* by Alvarado, Dodds, Kuenning, and Libeskind-Hadas

### Cryptography Using One-Time Pads!

In this problem, you'll implement the one-time encryption/decryption algorithm described in class. Here's the idea: Given two strings **S1** and **S2** of the same length, where **S1** is the string that we wish to encode and **S2** is a string representing the one-time pad, we first convert these strings into binary strings. Each character of **S1** or **S2** has a corresponding 8-bit representation and the binary representation of the string is the string comprising the concatenation of those 8-bit strings. So, both strings will be represented by binary sequences whose length is a multiple of 8.

Next, we compute the XOR of those two binary strings. Finally, we pack the XOR back into characters and return that string! This is all done by a function called `xorStrings(S1, S2)`. Here are a few examples of this function in action:

```
>>> encrypted = xorStrings("spam", "zng!") # "spam" is our secret, "zng!" is the one-time pad
>>> encrypted
'\t\x1e\x06L' # There are still four characters here, some are funny-looking control characters
>>> len(encrypted)
4
>>> xorStrings(encrypted, "zng!") # Now, we'll decrypt using our one-time pad
'spam'
```

You will need to write several small helper functions to get `xorStrings(S1, S2)` to do its job. Keep those functions very small and very simple. Other than the functions that you wrote for Changing Bases (you'll certainly use some of those functions as helper functions!) and other than `def` lines and docstrings, the total line count for our implementation (including `xorStrings` and all the new

helper functions) is 13 lines (and it could easily be shorter). Don't write more than 15 lines of code total. If you do, you're probably not using higher-order functions and/or list comprehensions to your full advantage.

Website design by Madeleine Masser-Frye and Allen Wu

# CSforAll - Ackerman

## CS for All

CSforAll Web > Chapter4 > Ackerman

This website accompanies the textbook *CS For All* by Alvarado, Dodds, Kuenning, and Libeskind-Hadas

### Ackermann's Function

Ackermann's function is named after its inventor/discoverer, Wilhelm Ackermann, an early 20th century logician.

It was one of the first functions proved to be *computable* but not *primitive recursive*. *Computable* means it can be evaluated by a computer, which you will demonstrate in this problem. We will not attempt to define *primitive recursive* here, but Wikipedia dares!

Ackermann's function is defined as

$$\begin{aligned} A(m, n) = & \quad n+1 && \text{if } m=0 \\ & A(m-1, 1) && \text{if } m>0 \text{ and } n=0 \\ & A(m-1, A(m, n-1)) && \text{if } m>0 \text{ and } n>0 \end{aligned}$$

And, here is a subset of the Ackermann function's values (the subset your implementation should be able to compute, in fact!)

Values of  $A(m, n)$

| $m \setminus n$ | 0 | 1  | 2  | 3  | 4   | 5   | 6   |
|-----------------|---|----|----|----|-----|-----|-----|
| 0               | 1 | 2  | 3  | 4  | 5   | 6   | 7   |
| 1               | 2 | 3  | 4  | 5  | 6   | 7   | 8   |
| 2               | 3 | 5  | 7  | 9  | 11  | 13  | 15  |
| 3               | 5 | 13 | 29 | 61 | 125 | 253 | 509 |

The following python program will compute A(m,n):

```
def A(m,n):
 if m == 0:
 return n + 1
 elif n == 0:
 return A(m - 1, 1)
 else:
 return A(m - 1, A(m, n - 1))
```

Your first task is to write a hmmm program that computes Ackermann's function. If we may suggest, using the pr6.hmmm file from the Hmmm zip is ideal.

Now to improve it! A recursive function call that does not alter its return value is said to be *tail-recursive*.

As an example of tail recursion, consider the following function, which determines whether the input, n, is a power of 2 (other than 1):

```
def powerOfTwo(n):
 if n % 2 == 1 : # check for n odd
 return False
 elif n == 2: # check for n=2
 return True
 else:
 return powerOfTwo(n / 2) # n is divisible by 2
```

Notice that the recursive call does all of its computation in the arguments -- and none with the value returned. The only thing the final `else` case does with the return value is `return` it itself!

This is the essence of tail-recursion.

At the assembly-language level, tail-recursive function calls can be optimized into pure loops.

Continuing our example, the following is a hmmm program for `powerOfTwo` that has *not* been optimized:

```
hmmm program to compute powers of two
00 setn r15 100 # initialize stack pointer
01 read r1 # read n into r1
```

```

02 calln r14 5 # call recursive powerOfTwo function
03 write r13 # write result (1=True, 0=False)
04 jumpn 1 # repeat
05 setn r2 2 # start of recursive powerOfTwo
06 mod r3 r1 r2 # base case n odd: set r3=n%2
07 jeqzn r3 10 # if n%2=0, jump
08 setn r13 0 # n is odd so load false into r13
09 jumpr r14 # return
10 sub r3 r1 r2 # base case n=2: set r3=n-2
11 jgtzn r3 15 # if n-2>0, jump
12 setn r13 1 # n=2 so load true into r13
13 jumpr r14 # return
14 nop
15 storer r1 r15 # push r1=n onto the stack
16 addn r15 1 # increment stack pointer
17 storer r2 r15 # push r2=2 onto the stack
18 addn r15 1 # increment stack pointer
19 storer r14 r15 # push return address onto the stack
20 addn r15 1 # increment stack pointer
21 div r1 r1 r2 # divide n by 2
22 calln r14 5 # compute powerOfTwo(n/2)
23 addn r15 -1 # decr stack pointer
24 loadr r14 r15 # pop return address
25 addn r15 -1 # decr stack pointer
26 loadr r2 r15 # pop r2=2
27 addnr r15 -1 # decr stack pointer
28 load r1 r15 # pop r1=n
29 jumpr r14 # return

```

Yet, because the call is tail-recursive, lines 15-29 can be greatly simplified:

```

15 div r1 r1 r2 # divide by 2
16 jumpn 5 # jump to compute PowerOfTwo on n/2

```

Here, we have replaced the `calln` command by a simple `jumpn`, and have eliminated all of the stack operations. Note that if we had needed to operate on the value returned by the recursive call, we could not have performed this simplification.

Modern compilers routinely perform this kind of tail-recursion optimization.

For the final part of this problem, modify your `hmmm` implementation of Ackermann's function -- by copying your `pr6.hmmm` file to `pr6tr.hmmm` -- and then altering the `Hmmm` assembly-language code to take advantage of tail recursion. Note: only 2 of the 3 calls can be optimized.

Happy Acker'ing!

Website design by Madeleine Masser-Frye and Allen Wu

# CSforAll - TowersofHanoi

## CS for All

CSforAll Web > Chapter4 > TowersofHanoi

This website accompanies the textbook *CS For All* by Alvarado, Dodds, Kuenning, and Libeskind-Hadas

### Towers of Hanoi

For this problem, you will write a Hmmm program named that solves the towers-of-Hanoi problem. You can write the code in the pr7.hmmm file included in Hmmmwork.

"The Towers of Hanoi" is a traditional puzzle involving moving different-sized disks among three pegs. The key constraint is that one is only allowed to place a disk onto an empty peg or another disk of larger radius.

An applet and a full explanation of the long history of this puzzle are available at this link.

Here is a summary of the algorithm for solving the puzzle:

```
def main():
 read Disks from user # This isn't real python syntax!
 # Ask the user for the number of
 # disks
 read FromPeg from user # Next, we ask the user for the
 # starting peg (1, 2, or 3)
 read ToPeg from user # Now we ask the user for the
 # destination peg (1, 2, or 3)
 hanoi(Disks, FromPeg, ToPeg) # Call hanoi to print out
 # the moves in the solution

def hanoi(Disks, FromPeg, ToPeg):
 if Disks == 1:
```

```

 print FromPeg
 print ToPeg
 return
 else:
 OtherPeg = the peg other than FromPeg and ToPeg
 # How do you find OtherPeg
 # without much work? Is there
 # a formula?
 hanoi(Disks - 1, FromPeg, OtherPeg)
 hanoi(1, FromPeg, ToPeg)
 hanoi(Disks - 1, OtherPeg, ToPeg)

```

Your job is to "compile" (or translate) this code into HMMM assembly language as faithfully as possible. By "faithful" we mean that you will have a section of HMMM code that corresponds to "main" and a section that corresponds to "hanoi". The "main" part will read in three inputs from the user and then call "hanoi" to solve it. Your "hanoi" code will have a base case at the top to handle the situation where there is just one disk, and will have a general case afterwards with three recursive calls. Although the program may look long (about 80-85 lines of HMMM code is typical here), much of it is repeated several times and can be cut-and-pasted.

Here is some sample input and output:

```

Enter number: 1
Enter number: 1
Enter number: 3
1
3

```

Notice that in this example we asked to move 1 disk from peg 1 to peg 3. The program printed out the instructions "1" and then "3" indicating that we should move a disk from peg 1 to peg 3. That is, the instructions appear as pairs of numbers indicating the peg to move from and the peg to move to. Here's a bigger example of moving 3 disks from peg 1 to peg 2:

```

Enter number: 3
Enter number: 1
Enter number: 2
1
2
1
3
2
3
1
2
3

```

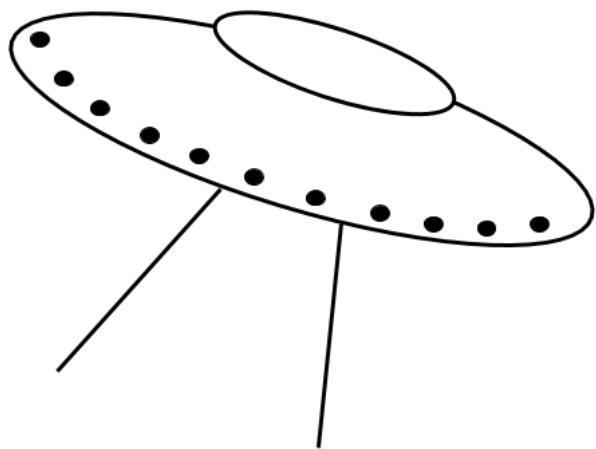
1  
3  
2  
1  
2

Website design by Madeleine Masser-Frye and Allen Wu

3

# CSforAll - RecursionExamples

## CS for All



CSforAll Web > Chapter2>FunctionFrenzy > RecursionExamples

This website accompanies the textbook *CS For All* by Alvarado, Dodds, Kuenning, and Libeskind-Hadas

### Examples: Recursion

power(b, p)

```
def power(b, p):
 """power calculates b**p via recursion
 Argument: b, a number
 Argument: p, an integer
 """
 if p == 0:
 return 1
 elif p < 0:
 # this is optional
```

```
 return 1.0/power(b, -p)
else:
 return b*power(b, p - 1)
```

add(m, n)

```
def add(m, n):
 """add calculates m + n via recursion and adding 1
 Argument: m, a number
 Argument: n, an integer
 """
 if n == 0:
 return m
 else:
 return add(m, n - 1) + 1
```

leng(s)

```
def leng(s):
 """leng returns the length of s
 Yes, it's already built in as len(s), but...
 Argument: s, which can be a string or list
 """
 if s == "" or s == []:
 # if empty string or empty list
 return 0
 else:
 return 1 + leng(s[1:])
```

vwl(s)

```
def vwl(s):
 """vwl returns the number of vowels in s
 Argument: s, which will be a string
 """
 if s == "":
 return 0
 # no vowels in the empty string
 elif s[0] in 'aeiou':
 return 1 + vwl(s[1:])
 else:
 return 0 + vwl(s[1:])
 # The 0 + isn't necessary but looks nice
```

mymax(L)

```
def mymax(L):
 """mymax returns the largest element in L
```

```

(this is also built-in, as max)
Argument: L, a _nonempty_ list
"""
if len(L) == 1:
 return L[0]
elif L[0] < L[1]:
 return mymax(L[1:])
drop the first
else:
 return mymax(L[0:1] + L[2:])
drop the second

```

`zeroest(L)`

```

def zeroest(L):
 """zeroest returns the element closest to 0 in L
 Argument: L, a _nonempty_ list
 """
 if len(L) == 1:
 return L[0]

 z = zeroest(L[1:])
 # what's the zeroest in the rest of L?

 if abs(L[0]) < abs(z):
 return L[0]
 # L[0] was zeroer!
 else:
 return z
 # z was zeroer!

```

# HMM Advanced Topics

[top]

## Advanced Topics for HMM (Harvey Mudd Miniature Machine)

Computer Science Department  
Harvey Mudd College

Last updated 23rd February 2018

### Table of Contents

- Introduction
- Loops
- Register conventions
- Function Calling Sequences
- Function Entry Sequences
- Function Exit Sequences
- Implementing a Stack
- Example
- Contact Information
- [Back to the main page](#)

### Introduction

This document aims at suggesting methods for implementing common programming constructs in HMM. It also includes an example which demonstrates these concepts in action.

### Loops

Loops are often useful, but implementing them can be tricky in assembly languages. In particular, it is critical to make sure that the loop jumps back to the proper point and has a means of exiting. The best strategy when implementing a loop is to reserve several registers beforehand as loop variables, and then enter the loop. Be careful not to initialize all of the loop variables in the loop, because this will most likely result in an infinite loop. The format of a basic loop is shown below:

```
1 # Initialize the start condition of the loop. May not be necessary in a

while-type loop
2 # Initialize the end condition of the loop. This will usually be a value to

compare against, but may not be necessary if two values within the loop
```

```

are compared against each other.
3 # The first line of the loop. Put the loop code here, and it will be

executed multiple times. This line is the loop repeat target.
4 # After the loop code, some sort of comparison must be made to decide

whether to jump to the loop escape target or the loop repeat target.

in many cases, there is no loop escape target, and the code either

jumps back or simply continues. In other cases, the code escapes on

some condition, in which case this line jumps conditionally to the

loop escape target.

5 # This line jumps unconditionally to the target not referenced in the

previous line. In many cases, this line is unnecessary.

```

Here is an example loop that is the equivalent of:

```
for i in range(3) :
 pass
```

Example code:

```

0 loadn r1 0
r1 will be the register which holds i
1 loadn r2 3
r2 is the end condition (for a subtract)

2 nop
This line would be the start of the loop body
3 addn r1 1
increment i
4 sub r3 r1 r2
compare i with end condition
5 jnez r3 2
if the end condition is not met, loop

6
This is the continuing code

```

## Register Conventions

By convention, several registers are reserved for special purposes. Register r15 is used as the *stack pointer*. Unlike most modern machines, the Hmmm stack grows upwards (the **pushr** and **popr** instructions enforce this convention). Register r14 is conventionally used for function return addresses, and r13 is reserved for return values from functions. Registers r1 through r4 can be used for scratch purposes, but when calling functions they contain the function arguments. All other registers can be used at will.

Also by convention, Hmmm is a *caller-save* machine. That means that a called function has the right to clobber any register except r14 and r15. When you call a function, you are responsible for saving any "precious possessions" (usually on the stack) and later restoring them.

## Function Calling Sequences

Although Hmmm does not enforce any specific method for calling functions, the following method is recommended:

- Use a sequence of **pushr** commands to save "precious possessions" on the stack. "Precious possessions" are any registers that contain values that you want to save. This often includes scratch registers and (if you are inside a non-main function) r14, the return address. It does *not* include r15, the stack pointer, or r13, where the function will return its result.
- Place the arguments to the function into r1, r2, ... as needed. Sometimes this will involve calculations.

- Call the function using `call r14, #`. Although the call instruction will accept any register, you should always use r14.
- Restore your "precious possessions" from the stack by using the `popr` instruction. Remember to do things in reverse order!
- The function result is now in r13. If you saved and restored your "precious possessions" correctly, everything else will be just as it was before you called the function.

## Function Entry Sequences

If you follow the function calling sequence just above, your function won't need to do anything special when it is entered. Your arguments will be in registers r1, r2, ... and your return address will be in r14. If you don't call any functions yourself, you don't need to do anything special; you are free to destroy any register except r14 and r15.

If you *do* call another function, you should save your own "precious possessions" (most emphatically including r14) on the stack before calling it. Often, the "precious possessions" will include your own arguments (registers r1 and up). They may also include other registers that contain intermediate results.

## Function Exit Sequences

When your function is finished, put the return value in r13. Then you can return to your caller with `jumpr r14`. Note that you have to use `jumpr`, not `jmpn`, to indicate that the return address is in a register (and thus not known a priori).

## Implementing a Stack

The easiest way to implement a stack is to use r15 as a stack pointer throughout the program and use `pushr` and `popr` to push and pop data. (Older Hmmm code might use `stor`, `load`, and `addn`, but that approach is much more painful.) Your stack operations should look something like this:

- To push register r14 onto the stack, use:  
`0 pushr r14, r15`
- To pop a value from the stack into register r1 use:  
`1 popr r1, r15`

Note that `pushr` adds 1 to r15 *after* you push, but `popr` subtracts 1 *before* it pops. That means that the stack pointer (r15) always points to an available memory location.

Remember that multiple pops must be executed in reverse order. The following code preserves the values of registers r1, r2, and r3 on a stack that starts at address 20:

```

0 loadn r15, 20
1 pushr r1, r15
2 pushr r2, r15
3 pushr r3, r15

use r1-r3 for other things here
4 popr r3, r15
5 popr r2, r15
6 popr r1, r15

```

Note that before the stack pointer can be used, it must be initialized with `loadn` to point to an empty area of memory. Although you could use a value as low as 7 (the first available empty space), it's often better to leave yourself a bit of wiggle room for later modifications, so here we chose 20.

## Example

This is an example Hmmm program that uses various advanced techniques. The function uses a stack for function calls, but is not recursive. The stack starts at address 40. The program also stores a list in memory, starting at address 50.

The program accepts a sequence of numbers, terminated with a 0. It then outputs each of the numbers in the same order that they were given, modulo the first number. Thus, inputs of 5, 11, 27, 3, and 8, would generate outputs of 1, 2, 3, and 3, which are 11 % 5, 27 % 5, 3 % 5, and 8 % 5. Here is the code for the program:

```

mods
created by Peter Mawhorter on 16.6.2006
Modified by Geoff Kuenning for new Hmmm architecture, 23.2.2018
#
The user enters a sequence of numbers, and the program prints each of those
numbers modulo the first number.

This Hmmm program illustrates several techniques, including:
1. Argument passing in registers
2. Function return values in registers
3. Saving precious registers on a stack
4. Storing lists of numbers in memory
5. Functions with no return value

Roughly equivalent python code:

def getNums():
"""Returns a list of user-provided numbers until the user enters a 0."""
num = int(input("Enter a number: "))
if num == 0:
return []
else:
return [num] + getNums()

def printMods(base, L):
"""Prints the list L modded by the base."""
if L == []:
return []
print(L[0] % base)
calcMods(base, L[1:])

def mods():
"""Takes a number and a list of numbers, and prints the list mod the
first number. List entry ends on an input of 0."""
base = int(input("Enter a number: "))
numbers = getNums([])
numbers = calcMods(base, numbers)
print numbers

Register usage:
#
r1 First function argument
r2 Second function argument
r5 Address of L (for printing)
r13 Function return value
r14 Function return address
r15 Stack pointer

Function "mods"
0 loadn r15, 40 # Initialize the stack
1 read r1
Read base

2 pushr r1, r15 # Save base on stack
3 loadn r1, 50
Tell getNums where to read numbers
4 calln r14, 11 # Call getNums

..returns address of first number in R13
5 popr r1, r15 # Recover base from stack

```

```

6 copy r2, r13 # Address of L is second argument to calcMods
7 calln r14, 17 # Print mods
8 halt
End of mods function
9 nop
Space for expansion
10 nop
Space for expansion

getNums function
r1: where to store the numbers read
Returns r1 in r13
11 copy r13, r1 # Save start of list for return purposes
Loop:
12 read r2
Read a number into r2
13 storer r2, r1
Save number in list
14 addn r1, 1
Step to next position in list
15 jnezn r2, 12
If number entered was nonzero, loop for more
16 jumpr r14
Otherwise, return to our caller

calcMods function
r1: modulus to use
r2: address of list of numbers to modify
No return value
17 loadr r3, r2
Get a number
18 jeqzn r3, 23
If it's zero, we're done
19 mod r3, r3, r1 # Calculate remainder
20 write r3
Print the result
21 addn r2, 1
Move to next number in list
22 jumpn 17
..and loop for more
23 jumpr r14
End of calcMods, return to caller

```

Here is the assembler output when given this program:

```

ASSEMBLY SUCCESSFUL
```

```

0 : 0010 1111 0010 1000 0 loadn r15, 40 # Initialize the s
1 : 0000 0001 0000 0001 1 read r1
Read base
2 : 0100 0001 1111 0011 2 pushr r1, r15 # Save base on sta
3 : 0010 0001 0011 0010 3 loadn r1, 50
Tell getNums whe
4 : 1011 1110 0000 1011 4 calln r14, 11 # Call getNums
5 : 0100 0001 1111 0010 5 popr r1, r15 # Recover base fro
6 : 0110 0010 1101 0000 6 copy r2, r13 # Address of L is
7 : 1011 1110 0001 0001 7 calln r14, 17 # Print mods
8 : 0000 0000 0000 0000 8 halt
End of mods func
9 : 0110 0000 0000 0000 9 nop

```

```

Space for expans
10: 0110 0000 0000 0000 10 nop
Space for expans
11: 0110 1101 0001 0000 11 copy r13, r1 # Save start of li
12: 0000 0010 0000 0001 12 read r2
Read a number in
13: 0100 0010 0001 0001 13 storer r2, r1
Save number in l
14: 0101 0001 0000 0001 14 addn r1, 1
Step to next pos
15: 1101 0010 0000 1100 15 jnezn r2, 12
If number entere
16: 0000 1110 0000 0011 16 jumpr r14
Otherwise, return
17: 0100 0011 0010 0000 17 loadr r3, r2
Get a number
18: 1100 0011 0001 0111 18 jeqzn r3, 23
If it's zero, we
19: 1010 0011 0011 0001 19 mod r3, r3, r1 # Calculate rem
20: 0000 0011 0000 0010 20 write r3
Print the result
21: 0101 0010 0000 0001 21 addn r2, 1
Move to next num
22: 1011 0000 0001 0001 22 jumpn 17
..and loop for m
23: 0000 1110 0000 0011 23 jumpr r14
End of calcMods,

```

After the successful assembly, the program will automatically run. Here is an example of running it:

```

Enter number (q to quit): 12
Enter number (q to quit): 15
Enter number (q to quit): 27
Enter number (q to quit): 36
Enter number (q to quit): 74
Enter number (q to quit): 9
Enter number (q to quit): 0
3
3
0
2
9
(2)%

```

## Contact Information

If this document is inaccurate or you believe that you have found a bug in either the assembler or the simulator, or even if you think that something should be added to any of these, please contact the CS5 staff at the usual cs5help address.

# Documentation for HMM (Harvey Mudd Miniature Machine)

Last update: 2024

## Quick reference: Table of Hmmm Instructions

| Instruction                          | Description                                                          | Aliases              |
|--------------------------------------|----------------------------------------------------------------------|----------------------|
| <b>System instructions</b>           |                                                                      |                      |
| <b>halt</b>                          | Stop!                                                                | None                 |
| <b>read rX</b>                       | Place user input in register rX                                      | None                 |
| <b>write rX</b>                      | Print contents of register rX                                        | None                 |
| <b>nop</b>                           | Do nothing                                                           | None                 |
| <b>Setting register data</b>         |                                                                      |                      |
| <b>setn rX N</b>                     | Set register rX equal to the integer N (-128 to +127)                | None                 |
| <b>addn rX N</b>                     | Add integer N (-128 to 127) to register rX                           | None                 |
| <b>copy rX rY</b>                    | Set rX = rY                                                          | <b>mov</b>           |
| <b>Arithmetic</b>                    |                                                                      |                      |
| <b>add rX rY rZ</b>                  | Set rX = rY + rZ                                                     | None                 |
| <b>sub rX rY rZ</b>                  | Set rX = rY - rZ                                                     | None                 |
| <b>neg rX rY</b>                     | Set rX = -rY                                                         | None                 |
| <b>mul rX rY rZ</b>                  | Set rX = rY * rZ                                                     | None                 |
| <b>div rX rY rZ</b>                  | Set rX = rY // rZ (integer division; rounds down; no remainder)      | None                 |
| <b>mod rX rY rZ</b>                  | Set rX = rY % rZ (returns the remainder of integer division)         | None                 |
| <b>Jumps!</b>                        |                                                                      |                      |
| <b>jmpn N</b>                        | Set program counter to address N                                     | None                 |
| <b>jumpr rX</b>                      | Set program counter to address in rX                                 | <b>jump</b>          |
| <b>jeqzn rX N</b>                    | If rX == 0, then jump to line N                                      | <b>jeqz</b>          |
| <b>jnezn rX N</b>                    | If rX != 0, then jump to line N                                      | <b>jnez</b>          |
| <b>jgtzn rX N</b>                    | If rX > 0, then jump to line N                                       | <b>jgtz</b>          |
| <b>jltzn rX N</b>                    | If rX < 0, then jump to line N                                       | <b>jltz</b>          |
| <b>calln rX N</b>                    | Copy addr. of next instr. into rX and then jump to mem. addr. N      | <b>call</b>          |
| <b>Interacting with memory (RAM)</b> |                                                                      |                      |
| <b>pushr rX rY</b>                   | Store contents of register rX onto stack pointed to by reg. rY       | None                 |
| <b>popr rX rY</b>                    | Load contents of register rX from stack pointed to by reg. rY        | None                 |
| <b>loadn rX N</b>                    | Load register rX with the contents of memory address N               | None                 |
| <b>storen rX N</b>                   | Store contents of register rX into memory address N                  | None                 |
| <b>loadr rX rY</b>                   | Load register rX with data from the address location held in reg. rY | <b>loadi, load</b>   |
| <b>storer rX rY</b>                  | Store contents of register rX into memory address held in reg. rY    | <b>storei, store</b> |

## Table of Contents

- [Introduction](#)
- [Using HMM](#)
- [Code Format](#)
- [Machine Layout](#)
- [The HMM Instruction Set](#)
- [Input and Output](#)
- [Halting the Program](#)
- [Diagnostic Features](#)
- [Examples](#)
- [Contact Information](#)
- [Advanced topics](#)

## Introduction

Hmmmm the (Harvey Mudd Miniature Machine) is a 16-bit, 26-instruction simulated assembly language with  $2^8 = 256$  16-bit words of memory. Hmmmm is written in Python, and it is intended as an introduction to assembly coding in general. Programs written in Hmmmm consist of numbered lines with one instruction per line, and comments.

Hmmmm is implemented as a single program written in Python. By default, `hmmmm` will assemble and run a file written in the Hmmmm assembly language. There are options that to assemble a program without executing it, to run a previously assembled program, and to invoke a built-in debugger.

## Installing Hmmmm

Hmmmm is available online in the form of source code from the HMC CS5 website at <http://www.cs.hmc.edu/twiki/bin/view/CS5/Resources>. If you plan on installing it on your own machine, you will need a working version of Python 3 but nothing more. Download the `hmmmm` file and put it in a directory where you will also write your Hmmmm programs.

## Using Hmmmm

Please refer to your lecture notes and the current Hmmmm assignment for an introduction to using Hmmmm.

## The Basics

To assemble (compile) and run a Hmmmm program, simply type "python `hmmmm`" at the command prompt. Hmmmm will ask for an input file, assemble it, and if assembly succeeds it will also run it. (On Macs, you may be able to save some effort by typing ".`/hmmmm filename.hmmmm`" where `filename.hmmmm` is the name of your Hmmmm program.

## Getting Fancy

Hmmmm accepts the -h, --help, -d, and -o options. -h and --help print explanations of all of the options. -d invokes the Hmmmm debugger. If -o is given, followed by a file name, the program is assembled and the result is written to the named file; later Hmmmm can be run on that file to execute it. For example:

```
python hmmmm program.hmmmm -o program.hb
python hmmmm -d program.hb
```

In debug mode, type "h" or "help" at the debug mode prompt for information on debugging commands, or see the [diagnostic features](#) section of this document.

## File Types

The assembler and the simulator are both file-type independent. Generally, however, files with a 'hmmm' (hmmm assembly) extension are Hmmm code, while files with an 'hb' (hmmm binary) extension are assembled Hmmm binary. (Because Hmmm is only a simulator, the binary files are actually text files and are readable and editable with standard text editors).

## Code Format

Each line of Hmmm code consists of a line number, an instruction, and one or more arguments. The line numbers must start at 0 and be consecutive integers, and they must be placed at the beginning of the line with no preceding characters. After the line number, use at least one character of whitespace to separate the instruction. The line number corresponds directly to the memory address of the line. Free (writable) memory begins at the address immediately after the last line of the program.

The instruction consists of a single word; all of the instructions are composed of lowercase alphabetic characters. After the instruction there must be at least one space and then 0 or more arguments, separated by any combination of whitespace and commas. The number of arguments depends on the instruction.

Each argument must be either a register or a number. Registers are denoted by 'r' followed by the number of the register, as in 'r3'. Numbers must fit in 8 bits. (Their decimal value must either be in the range -128 to 127 inclusive, or 0 to 255 inclusive, depending on the particular instruction. This is because numbers are represented using a method called "twos complement".) Each argument must be separated from any preceding arguments by any combination of whitespace and ',' characters, and the line may optionally be ended with some combination of whitespace and comments.

See the [examples](#) section for examples of proper and improper syntax.

Comments in Hmmm are identical to comments in Python: a '#' character begins a comment that continues to the end of the line. Comments are allowed both on empty lines and after the arguments on instruction lines. Comments on otherwise empty lines should not have line numbers, and will not be counted towards the line number of any following lines. Completely blank lines are also permitted

## Machine Organization

Hmmm simulates a computer that has sixteen 16-bit registers and 256 16-bit words of memory. The program is loaded into memory starting at location 0, so a program 12 lines long has 244 free words of memory starting at location 12 (locations 0 through 11 are occupied by the program's instructions). The program counter starts at location 0 and is incremented by 1 each cycle. It can also be redirected using the various **jump** commands. Each cycle, the simulator reads the instruction at the memory location pointed to by the program counter and executes it. This continues until it executes a **halt** instruction.

15 of the 16 registers are interchangeable from a hardware standpoint (although refer to the conventions below). The only special register is r0. When used as a source operand, r0 always provides a zero; when used as a destination it discards the result.

## The Hmmm Instruction Set

There are 26 different instructions in Hmmm, each of which accepts between 0 and 3 arguments. Two of the instructions, setn and addn, accept a signed numerical argument between -128 and 127. The load, store, call, and jump instructions accept an unsigned numerical argument between 0 and 255. All other instruction arguments are registers. In the code below, register arguments will be represented by 'rX', 'rY', and 'rZ', while numerical arguments will be represented by '#'. In real code, any of the 16 registers could take the place of 'rX' 'rY' or 'rZ'. The available instructions are:

| Assembly                  | Binary                   | Description                                                                                                                                                    |
|---------------------------|--------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>halt</b>               | 0000 0000 0000<br>0000   | Halt program                                                                                                                                                   |
| <b>nop</b>                | 0110 0000 0000<br>0000   | Do nothing                                                                                                                                                     |
| <b>read rX</b>            | 0000 XXXX 0000<br>0001   | Stop for user input, which will then be stored in register rX (input is an integer from -32768 to +32767).<br>Prints "Enter number: " to prompt user for input |
| <b>write rX</b>           | 0000 XXXX 0000<br>0010   | Print the contents of register rX on standard output                                                                                                           |
| <b>setn rX, #</b>         | 0001 XXXX #####<br>##### | Load an 8-bit integer # (-128 to +127) into register rX                                                                                                        |
| <b>loadr rX, rY</b>       | 0100 XXXX YYYY<br>0000   | Load register rX from memory word addressed by rY: rX = memory[rY]                                                                                             |
| <b>storer rX, rY</b>      | 0100 XXXX YYYY<br>0001   | Store contents of register rX into memory word addressed by rY:<br>memory[rY] = rX                                                                             |
| <b>popr rX rY</b>         | 0100 XXXX YYYY<br>0010   | Load contents of register rX from stack pointed to by register rY: rY -= 1;<br>rX = memory[rY]                                                                 |
| <b>pushr rX rY</b>        | 0100 XXXX YYYY<br>0011   | Store contents of register rX onto stack pointed to by register rY:<br>memory[rY] = rX; rY += 1                                                                |
| <b>loadn rX, #</b>        | 0010 XXXX #####<br>##### | Load register rX with memory word at address #                                                                                                                 |
| <b>storen rX, #</b>       | 0011 XXXX #####<br>##### | Store contents of register rX into memory word at address #                                                                                                    |
| <b>addn rX, #</b>         | 0101 XXXX #####<br>##### | Add the 8-bit integer # (-128 to 127) to register rX                                                                                                           |
| <b>copy rX, rY</b>        | 0110 XXXX YYYY<br>0000   | Set rX = rY                                                                                                                                                    |
| <b>neg rX, rY</b>         | 0111 XXXX 0000<br>YYYY   | Set rX = -rY                                                                                                                                                   |
| <b>add rX, rY,<br/>rZ</b> | 0110 XXXX YYYY<br>ZZZZ   | Set rX = rY + rZ                                                                                                                                               |
| <b>sub rX, rY,<br/>rZ</b> | 0111 XXXX YYYY<br>ZZZZ   | Set rX = rY - rZ                                                                                                                                               |
| <b>mul rX, rY,<br/>rZ</b> | 1000 XXXX YYYY<br>ZZZZ   | Set rX = rY * rZ                                                                                                                                               |
| <b>div rX, rY, rZ</b>     | 1001 XXXX YYYY<br>ZZZZ   | Set rX = rY // rZ                                                                                                                                              |
| <b>mod rX, rY,<br/>rZ</b> | 1010 XXXX YYYY<br>ZZZZ   | Set rX = rY % rZ                                                                                                                                               |
| <b>jumpr rX</b>           | 0000 XXXX 0000<br>0011   | Set program counter to address in rX                                                                                                                           |
| <b>jmpn n</b>             | 1011 0000 #####<br>##### | Set program counter to address #                                                                                                                               |
| <b>jeqzn rX, #</b>        | 1100 XXXX #####<br>##### | If rX = 0 then set program counter to address #                                                                                                                |
| <b>jnezn rX, #</b>        | 1101 XXXX #####<br>##### | If rX ≠ 0 then set program counter to address #                                                                                                                |
| <b>jgtzn rX, #</b>        | 1110 XXXX #####<br>##### | If rX > 0 then set program counter to address #                                                                                                                |
| <b>jltzn rX, #</b>        | 1111 XXXX #####<br>##### | If rX < 0 then set program counter to address #                                                                                                                |
| <b>calln rX, #</b>        | 1011 XXXX #####<br>##### | Set rX to (next) program counter, then set program counter to address #                                                                                        |

## Input and Output

Hmmm supports incredibly simplified I/O in the form of the **read** and **write** instructions.

The **read** instruction prompts the user to enter a number and, after a number is typed, puts it into the given register. If the number is too big or too small, or if it is otherwise bad, **read** will complain and prompt the user again. As a special convenience, if the user types "q" instead of a number, the program will immediately halt. The **write** instruction simply writes the given register to the user's console.

## Halting The Program

The **halt** instruction immediately ends the program and stops Hmmm. In addition, at any point during execution, the user can type ctrl-C to stop the program, and entering 'q' at either the debug-mode prompt or the input prompt will stop the program. Typing ctrl-D at any prompt will also stop the program.

## Diagnostic Features

The simulator program features a debug mode that is useful for diagnosing errors in the program. It is invoked using either the -d or --debug command-line options.

The debug mode prints information after executing each instruction, showing what instruction was just executed and where that instruction was found in memory (what the program counter was). It also prints the debug prompt. Any unrecognized input at the debug prompt causes the simulator to step one instruction forward. Recognized commands include 'c' or 'continue', 'd' or 'dump', 'h' or 'help', 'p' or 'print', 'q' or 'quit', and 'r' or 'run'.

### Command Effect

|                 |                                                                                                                                                                                                                      |
|-----------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>continue</b> | Causes the debugger to run through the rest of the program without prompting for debugging commands, but continuing to print debugging information.                                                                  |
| <b>dump</b>     | Immediately prints the contents of memory, printing the code lines first (one per line, in binary) followed by the numeric contents of the rest of memory in 6 columns, and then asks for another debugging command. |
| <b>help</b>     | Prints a short summary of the debug commands and returns to the prompt.                                                                                                                                              |
| <b>print</b>    | Prints the contents of the registers in a single column and returns to the prompt.                                                                                                                                   |
| <b>quit</b>     | Causes the program to exit immediately.                                                                                                                                                                              |
| <b>run</b>      | Causes the program to continue running as if it had been invoked with debug mode off: no debugging information is printed and no further prompts are given.                                                          |

## Examples

- [Good Code](#)
- [Bad Code](#)
- [Improper Syntax](#)

### Good Code:

This is a well-written and -commented program that takes two numbers, echoes the first, and returns the first divided by the second, or 0 if the second is 0.

```
program title
author and date
descriptive comment
0 read r1 # read dividend from the user
1 write r1 # echo the input
2 read r2 # read divisor from the user
3 jeqzn r2, 7 # jump to 7 if trying to divide by 0
```

```

4 div r3, r1, r2 # divide user's parameters
5 write r3 # print the result
6 halt

7 setn r3, 0 # 0 is the result for division by 0
8 write r3 # print the result
9 halt

```

This is the output from hmmm assembler when the program is run (note that the assembler truncates comments to make the lines fit neatly on your screen):

```
-----| ASSEMBLY SUCCESSFUL |-----
```

|                        |                                             |
|------------------------|---------------------------------------------|
| 0: 0000 0001 0000 0001 | 0  read r1      # read dividend from the us |
| 1: 0000 0001 0000 0010 | 1  write r1     # echo the input            |
| 2: 0000 0010 0000 0001 | 2  read r2      # read divisor from the use |
| 3: 1100 0010 0000 0111 | 3  jeqzn r2, 7 # jump to 7 if trying to di  |
| 4: 1001 0011 0001 0010 | 4  div r3, r1, r2 # divide user's paramete  |
| 5: 0000 0011 0000 0010 | 5  write r3     # print the result          |
| 6: 0000 0000 0000 0000 | 6  halt                                     |
| 7: 0001 0011 0000 0000 | 7  setn r3, 0   # 0 is the result for divis |
| 8: 0000 0011 0000 0010 | 8  write r3     # print the result          |
| 9: 0000 0000 0000 0000 | 9  halt                                     |

Enter number (q to quit): 42

42

Enter number (q to quit): 6

7

### Bad Code:

This is the same program, with the same functionality. However, this is very low-quality code.

```

0 read r1
1 write r1
2 read r2 # read r2
3 jeqzn r2, 7
4 div r3 r1, r2
5 write r3
6 halt # end
7 setn r3 0
8 write r3
9 halt

```

This is the assembler output for the above code:

```
-----| ASSEMBLY SUCCESSFUL |-----
```

|                        |                      |
|------------------------|----------------------|
| 0: 0000 0001 0000 0001 | 0  read r1           |
| 1: 0000 0001 0000 0010 | 1  write r1          |
| 2: 0000 0010 0000 0001 | 2  read r2 # read r2 |
| 3: 1100 0010 0000 0111 | 3  jeqzn r2, 7       |
| 4: 1001 0011 0001 0010 | 4  div r3 r1, r2     |
| 5: 0000 0011 0000 0010 | 5  write r3          |
| 6: 0000 0000 0000 0000 | 6  halt      # end   |
| 7: 0001 0011 0000 0000 | 7  setn r3 0         |
| 8: 0000 0011 0000 0010 | 8  write r3          |
| 9: 0000 0000 0000 0000 | 9  halt              |

Enter number (q to quit): 42

42

```
Enter number (q to quit): 7
6
```

### Improper Syntax:

This is the same program a third time, this time modified to demonstrate improper syntax. This program will not assemble.

```
program title
program title
author and date
descriptive comment
0 read r1, # trailing characters are not allowed
1 write[r1] # no grouping symbols allowed
2 read r2, r3 # too many arguments here
3 jeqzn r2, r7 # second argument must be a number
4 div r3, r1r2 # arguments must be separated
5 write 3 # write argument must be a register
5 halt # line number is incorrect
7 setn r3, 128 # maximum number for setn and addn commands is 127
 # (min is -128)
9 writer3 # instruction must be separated from argument
10halt # instruction must be separated from line number
```

Here is the assembly output for this program, demonstrating the error messages for the errors listed above:

ARGUMENT ERROR:

WRONG NUMBER OF ARGUMENTS.

DETECTED 2 ARGUMENTS, EXPECTED 1 ARGUMENTS

read r1,

SYNTAX ERROR ON LINE 1:

1 write[r1] # no grouping symbols allowed

ARGUMENT ERROR:

WRONG NUMBER OF ARGUMENTS.

DETECTED 2 ARGUMENTS, EXPECTED 1 ARGUMENTS

read r2, r3

ARGUMENT ERROR:

'r7' IS NOT A VALID NUMBER.

ARGUMENT ERROR:

WRONG NUMBER OF ARGUMENTS.

DETECTED 2 ARGUMENTS, EXPECTED 3 ARGUMENTS

div r3, r1r2

REGISTER ERROR:

'3' IS NOT A VALID REGISTER.

BAD LINE NUMBER AT LINE 6:

LINE NUMBER: 5 EXPECTED 6

ARGUMENT ERROR:

'128' IS OUT OF RANGE FOR THE ARGUMENT.

OPERATION ERROR:

'writer' IS NOT A VALID OPERATION.

SYNTAX ERROR ON LINE 9:

10halt # instruction must be separated from line number

\*\*\*\*\* ASSEMBLY TERMINATED UNSUCCESSFULLY \*\*\*\*\*
ASSEMBLY RESULTS:

```
0: ***ARGUMENT ERROR HERE*** 0 read r1, # trailing characters are n
1: ***SYNTAX ERROR HERE*** 1 write[r1] # no grouping symbols allow
```

```

2: ***ARGUMENT ERROR HERE*** 2 read r2, r3 # too many arguments here
3: ***ARGUMENT ERROR HERE*** 3 jeqzn r2, r7 # second argument must be
4: ***ARGUMENT ERROR HERE*** 4 div r3, r1r2 # arguments must be separa
5: ***REGISTER ERROR HERE*** 5 write 3 # write argument must be a
6: ***BAD LINE NUMBER HERE*** 5 halt # line number is incorrect
7: ***ARGUMENT ERROR HERE*** 7 setn r3, 128 # maximum number for s
8: ***OPERATION ERROR HERE*** 9 writer3 # instruction must be separ
9: ***SYNTAX ERROR HERE*** 10halt # instruction must be separ

```

\*\*\*\*\* ASSEMBLY FAILED, SEE ABOVE FOR ERRORS \*\*\*\*\*

Although it may not be obvious from the example above, the assembler will stop trying to assemble each line of code as soon as it finds an error. Thus, if a line of code has multiple errors in it, only one error will be reported. Once that error is fixed, the next error on that line will be reported if it is still there.

## Contact Information

If this document is inaccurate or you believe that you have found a bug in hmmm, or if you think that something should be added to it, please contact the CS5 staff at the usual cs5help address.

# Chapter 5: Imperative Programming — cs5book 1 documentation

## Navigation

- [index](#)
- [next |](#)
- [previous |](#)
- [cs5book 1 documentation »](#)

## Chapter 5: Imperative Programming

*The imperative is to define what is right and do it.*

—Barbara Jordan

### 5.1 A Computer that Knows You (Better than You Know Yourself?)

These days it seems that just about every website out there is trying to recommend something. Websites recommend books, movies, music, and activities. Some even recommend friends!

How does Netflix know what movies we're likely to enjoy watching or Amazon know what we'll like to read?



*Netflix recommended the movie “Alien” to me.*

The answer to this question lies in a broad and important subfield of computer science: *data mining*. Data mining focuses on extracting useful information from very large quantities of unstructured data.

In this chapter we will examine a simplified version of a fundamental data mining algorithm called *collaborative filtering (CF)*. Collaborative filtering is widely used in many successful recommendation systems, as well as many other application areas where there's a lot of data to be studied (e.g., financial markets, geological data, biological data, web pages, etc.). Our goal in this chapter is to build a music recommender system that uses a basic version of the CF approach.

Of course, our recommender system and our CF approach will necessarily be simplified. To build an industrial-strength recommender system is quite complicated. The system that won the Netflix prize (see <http://www.netflixprize.com> for more details) consists of several different sophisticated algorithms. See the paper “The BellKor Solution to the Netflix Grand Prize” by Yehuda Koren at [http://www.netflixprize.com/assets/Grand-Prize2009\\_BPC\\_BellKor.pdf](http://www.netflixprize.com/assets/Grand-Prize2009_BPC_BellKor.pdf)

To motivate the idea behind collaborative filtering, let's go back to the dark ages before the World Wide Web became pervasive. In those ancient times – perhaps before you were born – people discovered movies, books, restaurants, and music by asking for recommendations from people whom they knew to have tastes similar to their own.

Collaborative filtering systems work on this same fundamental principle—they find people who generally like the same things you do, and then recommend the things that they like that you might not have discovered yet. Actually, this is only one type of collaborative filtering called *user-based CF*. If you're dying to learn more about different types of CF, good! Keep taking CS courses. Or for a more immediate gratification, read the Wikipedia page: [http://en.wikipedia.org/wiki/Collaborative\\_filtering](http://en.wikipedia.org/wiki/Collaborative_filtering)

Let's consider a simple example in which a system is trying to recommend music you might like. First, the system needs to know something about your tastes, so you have to give it some information about what you like. Let's say you tell the system you like Lady Gaga, Katy Perry, Justin Bieber and Maroon 5.



*I talked to your roommate and learned that those are your favorite groups, so don't try to deny it!*

The system knows about the following additional five “stored” users (users for which we’ve already stored some data):

- April likes Maroon 5, The Rolling Stones and The Beatles
- Ben likes Lady Gaga, Adele, Kelly Clarkson, The Dixie Chicks and Lady Antebellum
- Cory likes Kelly Clarkson, Lady Gaga, Katy Perry, Justin Bieber and Lady Antebellum
- Dave likes The Beatles, Maroon 5, Eli Young Band and Scotty McCreery
- Edgar likes Adele, Maroon 5, Katy Perry, and Bruno Mars.

Determining a recommendation for you involves two steps:

- Determine which user (or users) have tastes most similar to yours.
- Choose and rank artists that the similar user likes. These are your recommendations.

Of course, there are many algorithms for performing each of the above steps, and many interesting problems that can arise in performing them, but again we will keep things relatively simple here. Our system will use the following simple CF algorithm:

- For each user, count the number of matching artists between your profile and the profile of that user.
- Choose the user with the highest number of matches.
- Recommend the artists who appear on their list and not on yours.

So in the above example, the system determines that Cory has the greatest overlap with you: He likes three of the same artists you like (Lady Gaga, Katy Perry and Justin Bieber) whereas none of the others have more than two groups in common with you. So, the recommendation system chooses Cory as the person whose tastes are most similar to yours. It also sees that Cory likes Lady Antebellum and Kelly Clarkson and therefore recommends that you check out these additional artists.

This is an admittedly simplified example and you can probably come up with a number of good questions about this approach. For example:

- What if I already know that I don’t like Kelly Clarkson? How could the system use this information?
- What if there are multiple people who tie for the best match with my tastes? Conversely, what if no one matches my tastes?
- In a real system, there are millions of users. How can the system efficiently find the best match?

Most of these answers are beyond the scope of this chapter, but we encourage you to think about how *you* might answer these questions as you read through the rest of this chapter. If you’re intrigued, great! This is exactly the kind of stuff you’ll learn as you continue your CS studies. But for now, back to the foundations for our music recommendation system.

### 5.1.1 Our Goal: A Music Recommender System

In the rest of this chapter we will build a music recommender system that implements the basic collaborative filtering algorithm described above. The transcript below shows an example of an interaction with the system.

Welcome to the music recommender system!

What is your name? Christine

Welcome Christine. Since you are a new user, I will need to gather some information about your music preferences.

Please enter an artist or band that you like: Maroon 5

Please enter another artist or band that you like,  
or just press enter to see your recommendations: Lady Gaga

Please enter another artist or band that you like,  
or just press enter to see your recommendations: Katy Perry

Please enter another artist or band that you like,  
or just press enter to see your recommendations: Justin Bieber

Please enter another artist or band that you like,  
or just press enter to see your recommendations:

Christine, based on the users I currently know about,  
I believe you might like:  
Lady Antebellum  
Kelly Clarkson

I hope you enjoy them! I will save your preferred artists and will have new recommendations for you in the future.

The system keeps track of users and their preferences. As more users rate more songs, the system can generate additional recommendations, both for new and existing users. So, the next time Christine uses the system, she's likely to get some new recommendations.

Before we introduce the techniques that we'll use to implement the recommendation system, take a few moments to think about the steps involved in the program demonstrated above. What data do you need to gather and store? How do you need to process the data? What do you need to produce as output?

Although there are many ways to actually write the program, the underlying components are the following:

- The ability to gather input from the user;
- The ability to repeat a task (e.g., asking for the user's preferred artists, comparing against stored users' preferences) many times;
- The ability to store and manipulate data in different ways (e.g., storing the user's responses, manipulating the stored users' preferences); and
- The ability to save data between program runs, and to load saved data (e.g., loading the stored users' preferences from a file, saving the current user's preferences to a file).

Throughout the rest of this chapter we will learn ways to perform each of the above tasks, ultimately resulting in a fully-functional music recommender system.

## 5.2 Getting Input from the User

The first task that we have listed above is to get input from the user. Fortunately, this is a simple task thanks to Python's built-in `input` function. (This function is called `raw_input` in Python 2.) Here's an example of some code that uses the `input` function:

```
def greeting():
 name = input('What is your name? ')
 print ('Nice to meet you,', name)
```

And here's how that code looks when it is run:

```
>>> greeting()
What is your name? Christine
Nice to meet you, Christine
```

The function `raw_input` takes one argument: a string that will be printed when the `raw_input` function is run (e.g., the string 'What is your name?' in the example above). When Python sees the `raw_input` command, it prints that string and then waits for the user to type in her own string. The user types in a string and presses Return (or Enter). At that point, Python assigns the user's input string to the variable appearing on the left-hand side of the `=` sign. In the example above, the string that the user typed in will be assigned to the variable `name`.

Python's `raw_input` function will always return a string. If you want that string to be converted into something else, like an integer or a floating point number (a number with a decimal point – also known as a "float"), that can be done using Python's data conversion functions. In the example below, we take the user's input string and convert it into a floating point number.

In the shell, this should appear as:

```
>>> plus42()
Enter a number: 15
15 + 42 = 57.0
```



*Proving that watermelons don't float!*

Note that converting data from one type to another (e.g., from strings to floats) can be “risky.” In the example above, we converted the user’s input string “15” into the floating point number `\(15.0\)`. However, had the user entered the string “watermelon”, Python’s attempt to convert that to a floating point number would have failed and resulted in an error message.

### 5.3 Repeated Tasks—Loops

The next feature that our program needs is to perform some set of actions multiple times. For example, the recommender system needs to ask the user to enter a number of artists that they like. For the moment, let’s change the system slightly so that instead of allowing the user to enter as many artists as she likes, the system asks the user to enter exactly four artists. The algorithm that performs this data-collection looks something like:

- **Repeat the following four times:**
  - Display a prompt to the user;
  - Obtain the user’s input; and
  - Record the user’s answer.



*Actually, 42 times is just the right number of times for me!*

Imagine that we have a few lines of Python code that gather and store input from the user. Since we want to do this four times, we could just make four copies of that code, one for each time that we want to ask the user for an artist. For only four preferences, this wouldn’t be so bad. But imagine that we wanted to ask the user for 10 preferences? Or 42? The code would quickly become long and cumbersome. Furthermore, if we decided that we wanted to make a small change to the way that we ask the user for input, we’d have to make that change in 4, 10, or 42 places in our program. This would be very annoying for the programmer! Moreover, ultimately we might want to let the user herself specify how many artists she wants to enter, instead of fixing that value before the program runs. In that case, we simply wouldn’t know in advance how many copies of the code to place in our program.

We have already seen one way to perform repeated tasks in Chapter 3. In fact, we’ve also seen a second: list comprehensions. This chapter presents another way to perform these same repeated tasks: *iteration* or *loops*.

Just as recursion allows a programmer to express self-similarity in a natural way, loops naturally express sequential repetition. It turns out that recursion and loops are often equally good choices for doing something repeatedly, but in many cases it’s much easier and more natural to use one rather than the other. More on that shortly!

#### 5.3.1 Recursion vs. Iteration at the Low Level



*If you skipped Chapter 4, no worries— you might jump to Section 5.3.2*

To illustrate the difference between iteration and recursion, let’s travel backwards in time and revisit the recursive factorial function:

```

def factorial(n):
 if n == 0:
 return 1
 else:
 answer = factorial(n - 1)
 return n * answer

```

In Chapter 4, we implemented this function in HMMM and saw that, at the machine level, the recursion uses a stack to keep track of the function calls and the values of the variables in each function call. In that chapter, we also looked at another very different implementation of the factorial function that was considerably simpler and did not use a stack. Here it is:

```

#
Calculate N factorial
Input: N
Output: N!
#
Register usage:
#
r1 N
r2 Running product
#
00 read r1 # read n from the user
01 loadn r13 1 # load 1 into the return register
 # (base case return value)
02 jeqz r1 06 # if we are at the base case, jump to the
 # end
03 mul r13 r13 r1 # else multiply the answer by n
04 addn r1 -1 # and decrement n
05 jump 02 # go back to line 2 and test for base case
 # again
06 write r13 # we're done so print the answer
07 halt # and halt

```

In contrast to the recursive version, this implementation uses a variable (`r13` in this example) to gradually “accumulate” the answer. Initially, `r13` is set to 1. In lines 3–5, we multiply `r13` by the current value of `n` (stored in register `r1`), decrement `n` by 1, and jumps back to line 2 to test if we should do this again. We repeat this loop until the value of `n` reaches 0. At that point, `r13` contains the product  $(n \times (n-1) \times \dots \times 1)$  and we’re done.

In some sense, this iterative approach is simpler than recursion because it just loops “round and round”, updating the value of a variable until it’s done. There’s no need to keep track (on the stack) of what “things were like” before the recursive call and reinstate those “things” (from the stack) when the recursion returns. Let’s now see how this works in Python.

### 5.3.2 Definite Iteration: for loops

In many cases, we want to repeat a certain computation a specific number of times. Python’s `for` loop expresses this idea. Let’s take a look at how we can use a `for` loop to implement the iteration in the recommendation program to collect a fixed number of artists that the user likes.

To begin, assume the user will enter three artists.

```

artists = []

for i in [0, 1, 2]:
 next_artist = input('Enter an artist that you like:')
 artists.append(next_artist)

print ('Thank you! We'll work on your recommendations now.')

```

In Python 2, remember to use `raw_input` instead of `input`.

This for loop will repeat its “body”, lines 4–5, exactly 3 times. Let’s look closely at how this works. Line 3 above is the *header* of the loop, with five required parts:

1. The keyword `for`;
2. The name of a variable that will control the loop. In our case that variable is named `i`. It’s safest to use a new variable name, created just for this `for` loop, or one whose old value is no longer needed;
3. The keyword `in`;
4. A sequence such as a list or a string. In our case it is the list `[0, 1, 2]`; and
5. A colon. This is the end of the header and the start of the `for` loop body.

As we noted, lines 4–5 are called the *body* of the loop. The instructions in the loop body must be indented consistently within the loop header, just as statements within functions are. Note that line 7 above is *not* part of the loop body because it is not indented.

So what exactly is going on in this `for` loop? Our variable `i` will initially take the first value in the list, which is 0. It will then perform the lines of code that are indented below the `for` loop. In our case, those are lines 4 and 5. Line 4 uses Python’s `raw_input` function (or `input` in Python 3) which prints the string

Enter an artist that you like:

and then pauses to let the user type in a response. Once the user has typed a response and pressed the ENTER (or RETURN) key, that response is placed in the variable named `next_artist`. So, `next_artist` will be a string that is the name of an artist. Line 5 uses Python’s `append` function to add that string to the list `artists`.

Python recognizes that this is the end of the body of the `for` loop because line 7 is not indented – that is, it’s at the same level of indentation as line 3. So, Python now goes back to line 3 and this time sets `i` to be 1. Now, lines 4 and 5 are performed again, resulting in the user being asked for another string and that string being appended to the end of the `artists` list.

Again, Python goes back up to line 3 and this time `i` takes the value 2. Once again, Python executes lines 4 and 5 and we get another artist and add it to the `artists` list.



*Four fors!*

Finally, `i` has no more values to take since we told it to take values from the list `[0, 1, 2]`. So now the loop is done and Python continues computing at line 7.

How could we ask the user to respond to four examples instead of three? Easy! We could modify the loop header to be:

```
for i in [0, 1, 2, 3]:
```

What about 25 iterations? Well, we could replace the list `[0, 1, 2, 3]` with the list `[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24]` but who wants to type all that? (We certainly didn’t find it very much fun!) Instead, we can use the built-in `range` function (that we saw in Chapter 3) to automatically generate that list:



*Remember that `range(25)` does indeed generate that 25-element list from 0 to 24.*

```
for i in range(25):
```

Does the control variable have to be named `i`? No – any valid variable name can be used here. For clarity and to avoid confusion, you should generally avoid using a name that you have already used earlier in your function, although we often reuse variable names in loops.

Finally, a word on style: In general, variable names like `i` are not very descriptive. However, a variable used in a loop is a very temporary variable – it serves the purpose of allowing us to loop and then once the loop is over we’re done with it. Therefore, it’s quite common to use short variable names in `for` loops, and the names `i`, `j`, and `k` are particularly popular.

### 5.3.3 How Is the Control Variable Used?

In the example above, the control variable took on a new value at every iteration through the loop, but we never explicitly *used* the value of variable inside the body of the loop. Therefore, it didn't matter what the list of elements in our for loop header was—these elements were ignored. We could have accomplished the same thing with any three-element list header:

```
for i in ['I', 'love', 'Spam']:
```

or

```
for i in [42, 42, 42]:
```

In both cases, we would have gone through the loop three times.

Sometimes the value of the control variable is important to the computation inside the for loop. For example, consider this iterative version of the factorial function. Here we've named the control variable factor to suggest its role:

```
def factorial(n):
 answerSoFar = 1
 for factor in range(1, n+1):
 answerSoFar = answerSoFar * factor
 return answerSoFar
```

When we call `factorial(4)`, the loop becomes:

```
for factor in range(1, 5):
 answerSoFar = answerSoFar * factor
```

What happens here?

- After the first time through the loop `answerSoFar` holds the result of its previous value (1) times the value of `factor` (1). When the loop completes its first iteration `answerSoFar` will be 1 (i.e.,  $1*1$ ).
- In the second iteration through the loop `answerSoFar` will again be assigned to hold the product of the previous value of `answerSoFar` times `factor`. Since `factor`'s value is now 2, `answerSoFar` will equal  $1*2$ , or 2, when this second loop iteration ends.
- After the third time through the loop, `answerSoFar` will be  $2*3$  or 6, and the fourth time through `answerSoFar` will become  $\backslash(4*6\backslash)$ , or 24.

The loop repeats exactly 4 times because `range(1, 5)` has four elements: [1, 2, 3, 4].

Let's now "unroll" this four-iteration loop to see what it does at each iteration. Calling `factorial(4)` becomes

```
factorial function begins
answerSoFar = 1

loop iteration 1
factor = 1
answerSoFar = answerSoFar * factor # answerSoFar becomes 1
iteration 2
factor = 2
answerSoFar = answerSoFar * factor # answerSoFar becomes 2
iteration 3
factor = 3
answerSoFar = answerSoFar * factor # answerSoFar becomes 6
iteration 4
factor = 4
answerSoFar = answerSoFar * factor # answerSoFar becomes 24

loop ends, 24 is returned
```

### 5.3.4 Accumulating Answers

Consider what happened to `answerSoFar` throughout the iterations above. It started with a value and it changed that value each time through the loop, so that when the loop completed the returned `answerSoFar` was the final answer. This technique of *accumulation* is very common, and the variable that "accumulates" the desired result is called an *accumulator*.

Let's look at another example. The `listDoubler` function below returns a new list in which each element is double the value of its corresponding element in the input list, `aList`. Which variable is the accumulator here?

(ch05\_1d)

In this case, the accumulator is the list `doubledList` rather than a number. It's growing in length one element at a time instead of one factor at a time as in `factorial`.

Let's consider one more example where the loop control variable's value is important to the functionality of a loop. Returning to our recommender program, let's write a loop that calculates the number of matches between two lists. This function will be useful when we're comparing the current user's preferences to the stored preferences of other users to determine which stored user most closely matches the current user's preferences. For this problem, we have two lists of artist/band names, for example:

```
>>> userPrefs
['Maroon 5', 'Lady Gaga', 'Justin Bieber']
>>> storedUserPrefs
['Maroon 5', 'Kelly Clarkson', 'Lady Gaga', 'Bruno Mars']
```

Our first version of this function will implement the following algorithm:

- Initialize a counter to 0
- For each artist in the user's preferences:
- If that artist is in the stored user's preferences too, add one to count

This simple algorithm is implemented with the following Python code:

```
def numMatches(userPrefs, storedUserPrefs):
 """ return the number of elements that match between
 the lists userPrefs and storedUserPrefs """
 count = 0
 for item in userPrefs:
 if item in storedUserPrefs:
 count += 1
 return count
```

In this function we've used a shorthand notation to increment the value of `count`.



*Similar shorthands include `-=`, `*=`, and `/=`*

The line

```
count += 1
```

is just shorthand for

```
count = count + 1
```

Notice that the code above is actually not specific to our music recommender function, but in fact can be used to compare any two lists and return the number of matching elements between them. For this reason, it's stylistically better to use generic list names for the parameters to this function – that is, we could replace the variable name `userPrefs` with a more general name such as `listA` and `storedUserPrefs` with `listB`.

```
def numMatches(listA, listB):
 """ return the number of elements that match between
 listA and listB """
 count = 0
 for item in listA:
 if item in listB:
 count += 1
 return count
```

Finally, let's complete the process of choosing the stored user with the highest number of matches to the current user. Let's assume that each stored user is represented by a list of that user's preferences and then we put all of those lists into one master list. Using the representation, the stored users' preferences might look like this:

```
[['Maroon 5', 'The Rolling Stones', 'The Beatles'],
 ['Lady Gaga', 'Adele', 'Kelly Clarkson', 'The Dixie Chicks', 'Lady Antebellum'],
 ['Kelly Clarkson', 'Lady Gaga', 'Katy Perry', 'Justin Bieber', 'Lady Antebellum'],
 ['The Beatles', 'Maroon 5', 'Eli Young Band', 'Scotty McCreery'],
 ['Adele', 'Maroon 5', 'Katy Perry', 'Bruno Mars']]
```

Note that we have not included the names of the stored users, since we only care about their preferences for now. However, we can think about the stored users as having indices. In the list above, index 0 corresponds to the user who likes ['Maroon 5', 'The Rolling Stones', 'The Beatles']. There are four other users with indices 1, 2, 3, and 4. Let's will assume that this list does not contain the preferences of the current user.

So, imagine that we want to calculate the index of the stored user with the best match to our current user. For example, if our current user likes ['Lady Gaga', 'Katy Perry', 'Justin Bieber', 'Maroon 5'] then the stored user at index 2 in the stored user list above has the most matches in common – three matches, whereas all other stored users have fewer matches.

The algorithm for finding the user with the best match to the current user is the following:

- Initialize the maximum number of matches seen so far to  $\backslash(0\backslash)$
- For each stored user:
- Count the number of matches between that stored user's preferences and the current user's preferences
- If that number of matches is greater than the maximum number of matches so far:
- Update the maximum number of matches seen so far
- Keep track of index of that user
- Return the index of the user with the maximum number of matches

How could we express this in Python? A first attempt might begin like this:

```
def findBestUser(userPrefs, allUsersPrefs):
 """ Given a list of user artist preferences and a
 list of lists representing all stored users'
 preferences, return the index of the stored
 user with the most matches to the current user. """
 max_matches = 0 # no matches found yet!
 best_index = 0
 for pref_list in allUsersPrefs:
 curr_matches = numMatches(userPrefs, pref_list)
 if curr_matches > max_matches:
 # somehow get the index of pref_list??
```

Note that we maintain a variable called `max_matches` that stores the maximum number of matches found so far. Initially it is set to 0, because we haven't done any comparisons yet and have no matches. We also maintain a variable called `best_index` that stores the index of the list in the `allUsersPrefs` that has the highest number of matches. We've initialized this to 0 as well. Is this a good idea? It suggests that the list with index 0 has the best number of matches so far. In fact, we haven't even looked at the list with index 0 at this point, so this may seem a bit weird. On the other hand, it doesn't hurt to initialize this variable to 0 since we're about to begin counting matches in the loop and the best solution will be at least 0.

Notice that we reach a dead end in the last line of our function above. We have the element from the `allPrefs` list, but we have no way of getting the `index` that corresponds to it, and it's that index that we're committed to finding. Can you think of a way to solve this problem—before peeking at the solution below?

Here's our next attempt—this time, more successful:

```
def findBestUser(userPrefs, allUsersPrefs):
 """ Given a list of user artist preferences and a
 list of lists represented all stored users'
 preferences, return the index of the stored
 user with the most matches to the current user. """
 max_matches = 0
```

```

best_index = 0
for i in range(len(allUsersPrefs)):
 curr_matches = countMatches(userPrefs,
 allPrefs[i])
 if curr_matches > max_matches:
 best_index = i
 max_matches = curr_matches
return best_index

```

What's the difference? Notice that now our `for` loop is iterating not over the lists in `allUsersPrefs` but rather over the indices of those lists. We do this using the `range` function to generate the list of all indices in `allUsersPrefs`. With this index variable, we can access the element in the list and we can store that index in our `best_index` variable.

### 5.3.5 Indefinite Iteration: while Loops

In all of the examples above, we knew exactly how many times we wanted to loop. However, in many cases, we *can't* know how many iterations we need — the number of iterations of the loop depends on some external factor out of our control.

Let's continue with our recommendation program. In the last section we collected a fixed number of preferred artists from the user. But in our original program we allowed the user to enter as many artists as they wished, whether that number was one or one thousand.

Recall that `for` loops always run a definite number of times, so this situation calls for a different kind of loop: the `while`. A `while` loop runs as long as its boolean condition is true.



*In other words, a while loop runs for a while.*

Let's take a close look at the structure of a `while` loop that implements the new desired behavior for our recommender program

```

newPref = input("Please enter the name of an \
artist or band that you like: ")

while newPref != "":
 prefs.append(newPref)
 newPref = input("Please enter an artist or band \
that you like, or just press enter to see recommendations: ")

print('Thanks for your input!')

```

In Python 2, remember to use `raw_input` rather than `input`.

The `while` loop is similar to the `for` loop in that it consists of a loop header (line 4) and a loop body (lines 5-7). The loop header consists of the following three elements, in order:

- The keyword `while`
- A boolean expression. In our example this expression is `newPref != "`
- A colon

As with `for` loops, the loop body must be indented under the loop header. Thus, line 9 in the above example is *not* inside the loop body. Note that both of the `input` statements are split over more than one line of text, but Python considers each really just one line because of the textbackslash symbols at the end of the lines.

A `while` loop will execute as long as the Boolean expression in the header evaluates to `True`. In this case, the Boolean expression is `newPref != "`. Assuming that the user entered a non-empty string in line 1, the expression `newPref != "` will evaluate to `True` the first time it is evaluated. Thus, we enter the body of the `for` loop and execute lines 5–7. If the user entered another non-empty string, then this Boolean will again evaluate to `True`. This will repeat until, eventually, the user enters no string – she just presses RETURN or ENTER. At this point, `newPref`

is an empty string. Thus, when we return to evaluate the Boolean expression `newPref != "` in the `while` statement, it is now `False` and the loop terminates, causing execution to continue at line 9.

### 5.3.6 for Loops vs. while Loops



*But aliens don't have thumbs!*

Often it's clear whether a computational problem calls for definite (`for`) or indefinite (`while`) iteration. One simple rule-of-thumb is that `for` loops are ideally suited to cases when you know exactly how many iterations of the loop will be conducted whereas `while` loops are ideal for cases when it's less clear in advance how many times the loop must repeat.

It is always possible to use a `while` loop to emulate the behavior of a `for` loop. For example, we could express the factorial function with a `while` loop as follows:

```
def factorial(n):
 answer = 1
 while n > 0:
 answer = answer * n
 n = n-1
 return answer
```



*...that saves typing!*

Here we've used `answer` instead of `answerSoFar` with the understanding that `answer` will not actually hold the value of the desired answer until the end of the loop—this is a common style for naming accumulator variables.

Be careful! Sometimes when people try to use a `while` loop to perform a specific number of iterations their code ends up looking like this:

```
def factorial(n):
 answer = 1
 while n > 0:
 answer = answer * n
 return answer
```

What happens when you run `factorial(5)` using the function above? The loop will run, but it will never stop!



This `while` loop depends on the fact that `n` will eventually reach 0, but in the body of the loop we never change the value of `n` and we never get `n!` Python will happily continue multiplying `answer * n` forever—or at least until you get tired of waiting and hit Ctrl-C to stop the program.

Go through the steps and see if you can spot the error in the following version:

(ch05\_fac)

This code will also run forever. But why? After all, we definitely do decrease `n`. This bug is more subtle. Remember that the `while` loop runs only the code in the body of the loop before repeating. Because the statement that subtracts 1 from `n` is not indented, it is not part of the `while` loop's body. Again, the loop variable does not change within the loop, and it runs forever.



The Apple Corporation's address is One Infinite Loop, Cupertino, CA

A loop that never ends is called an *infinite loop* and it is a common programming bug. When using a `while` loop, remember to update your loop-control variable inside the loop. It's an advantage of `for` loops that this updating is done automatically! In fact, the tendency to accidentally create infinite loops is so common that it leads us to an important takeaway message about the choice between `for` and `while` loops.

**Takeaway message:** *If you know in advance how many times you want to run a loop, use a `for` loop; if you don't, use a `while` loop.*

### 5.3.7 Creating Infinite Loops On Purpose

Sometimes infinite loops can come in handy. They're not actually infinite, but the idea is that we will stop them when we are "done," and we don't have to decide what "done" means until later in the loop itself.



*I'm pro-crastination!*

For example, consider a different version of the recommender loop above that gathers data from the user.

```
numCorrect = 0
```

```
while True: # run forever -- or at least as long as needed...
 newPref = input("Please enter an artist or band that you like, \
 or just press enter to see recommendations: ")
 if newPref:
 prefs.append(newPref)
 else:
 break

print('Thanks for your input!')
```

The body of the `else` statement contains one instruction: the `break` instruction. `break` will immediately halt the execution of the loop that it appears in, causing the code to jump to the next line immediately after the loop body. `break` can be used in any kind of loop, and its effect is always the same—if the code reaches a `break` statement, Python *immediately* exits the containing loop and proceeds with the next line after the loop. If you have one loop inside another loop (a perfectly OK thing to do, as you'll see below), the `break` statement exits only the innermost loop.



*Yes! I need a break!*

**You might ask, “Do we really need a break?”**

After all, the loop above can be written with a more informative condition, as we saw above. Which approach is better? It's a matter of style. Some prefer the “delayed decision” approach, writing loops that appear to run too long, only to break out of them from the inside; others prefer to put all of the conditions directly in the loop header. The advantage of the latter approach is that the condition helps clarify the context for the loop. The advantage of the former (in some cases anyway) is that it avoids the awkward double-input statement—there's no need to ask the user to enter their initial response in a separate input statement before the loop begins.

### 5.3.8 Iteration Is Efficient

The heart of imperative programming is the ability to change the value of variables—one or more accumulators—until a desired result is reached. These in-place changes can be more efficient, because they save the overhead of recursive function calls. For example, on our aging computer, the Python code

```
counter = 0
while counter < 10000:
 counter = counter + 1
```

ran in 2.6 milliseconds. The “equivalent” recursive program

```
def increment(value, times):
 if times <= 0:
 return value
 return increment(value + 1, times - 1)
```

```
counter = increment(0, 10000)
```

ran more than an order of magnitude slower, in 38.3 milliseconds.

Why the difference? Both versions evaluate 10,000 boolean tests; both execute the same 10,000 additions. The difference comes from the overhead of building and removing the stack frames used to implement the function calls made recursively.

Memory differences are even more dramatic: storing partial results on the stack can quickly exhaust even today’s huge memory stores.

## 5.4 References and Mutable vs. Immutable Data

At the beginning of this chapter we introduced the two key components of imperative programming: iteration and data mutation. We have already discussed iteration in some depth. Now we begin to explore the concept of data mutation.

### 5.4.1 Assignment by *Reference*

The assignment and re-assignment of values to one variable—the accumulator—characterizes imperative programming with loops. All of these assignments are efficient, but **only if the size of the copied data is small!** Floating-point values and typical integers are stored in a small space, often the size of one register (32 or 64 bits). They can be copied from place to place rapidly. For example, the assignment operation

```
suppose x refers to the value 42 right now
y = x
```

runs extremely fast, probably best measured in nanoseconds, as suggested by the timings in the previous section.

Lists, on the other hand, can grow very large. Consider this code:

```
suppose that list1 holds the value of
list(range(1000042)) right now
list2 = list1
```

This assignment makes `list2` refer to a 1,000,042-element list—a potentially expensive proposition, if it involved 1,000,042 individual integer assignments like the `y = x` example above. What’s more, there’s no guarantee the elements of `list1` aren’t lists themselvesdots .

How does Python make *both* integer and list assignments efficient? It does so through a simple, single rule for all of its data types: *Assignments copy a single reference*.

Reference? It turns out that when you assign a piece of data (e.g., an integer) to a variable, what you are actually doing is storing the memory location of that piece of data in the variable. We say that the variable holds a *reference* to the piece of data. When we think of a variable, we typically think only of the data to which the variable refers but you can also obtain the reference (i.e., the memory location of that data) with Python’s built-in `id()` function:

```
>>> x = 42
>>> x # Python will reply with x's value
42
>>> id(x) # asks for x's reference (the memory location of its contents)
```

```
505494448
```

```
this will be different on your machine!
```

When we talk about a variable's *value*, we mean the data to which the variable refers; when we talk about a variable's *reference*, we mean the memory location of that data. For example, in the code above, the value of *x* is 42, and its reference is 505494448, which is the location of the value 42 in memory. By the way, the idea of the memory location of a piece of data should be very familiar to you from Chapter 4. If you like, you can think of the variables that store the references as registers on the CPU. (This is not quite exactly correct, but a reasonable conceptual model.) Figure 5.1 illustrates this concept graphically.

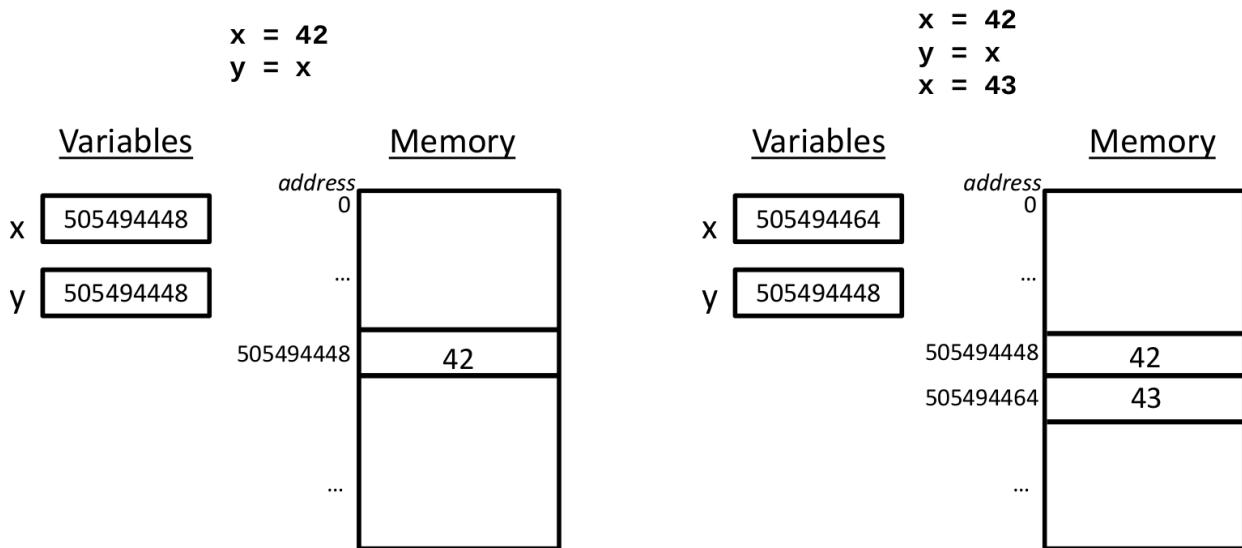


Figure 5.1: A depiction of how Python stores data. The boxes on the left are the variables, which you can think of as registers on the CPU, which store references to locations in memory. The actual data is stored in memory.

Python makes assignment efficient by *only copying the reference, and not the data*:

```
>>> x = 42 # this puts the value 42 in the next memory slot, 505494448 and then it gives x a copy
```

```
of that memory reference
>>> y = x # copies the reference in x into y, so that x and y both refer to the same integer 42 in memory
>>> id(x) # asks for x's reference (the memory location of its contents)
505494448
>>> id(y) # asks for y's reference (the memory location of its contents)
505494448
```

As you would expect, changes to *x* do not affect *y*:

```
>>> x = 43 # this puts 43 in the next memory slot, 505494464
>>> id(x)
505494464
x's reference has changed
>>> id(y)
505494448
but y's has not
```

The result of executing the above code is shown on the right in Figure 5.1.

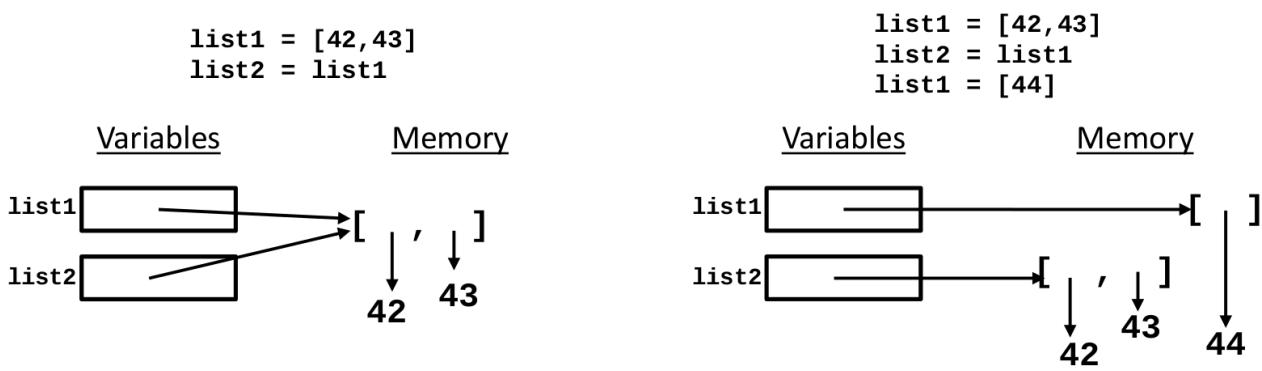


Figure 5.2: A box-and-arrow diagram abstraction of how Python stores list data. All of the elements in the list (and the list itself) are in memory, but we have abstracted away from the exact addresses of these elements.

Assignment happens the same way, regardless of the data types involved:



*It is possible to reprogram how assignment works for user-defined data types, but this is the default.*

```
>>> list1 = [42,43] # this will create the list [42,43] and give its location to list1
>>> list2 = list1 # give list2 that reference, as well

>>> list1 # the values of each are as expected
[42,43]
>>> list2
[42,43]

>>> id(list1) # asks for list1's reference (its memory location)
538664
>>> id(list2) # asks for list2's reference (its memory location)
538664
```

As the data we store gets more complicated, it becomes cumbersome to try to actually represent what is happening in memory faithfully, so computer scientists often use a type of diagram called a box-and-arrow diagram to illustrate how references refer to memory. A box-and-arrow diagram for the above code is shown on the left in Figure 5.2. In this figure, the list [42, 43] exists in memory at the location referred to by both `list1` and `list2`, but instead of writing out the actual memory location, we indicate that `list1` and `list2` refer to the same data in memory using arrows. In fact, notice something interesting about Figure 5.2: the elements in the list are also references to data in other locations in memory! This is a fundamental difference between integers and lists: a list refers to the memory location of a *collection* of potentially many elements, each of which has its own reference to the underlying data!

As with integers, if an assignment is made involving a list, *one reference is copied*, as shown in the code below, and on the right in Figure 5.2:

```
>>> list1 = [44] # will create the list [44] and make list1 refer to it
>>> id(list1) # list1's reference has changed
541600
>>> id(list2) # but list2's has not
538664
```

The assignment `list2 = list1` caused both `list1` and `list2` to refer to the same list.

This one-reference rule can have surprising repercussions. Consider this example, in which `x`, `list1[0]`, and `list2[0]` start out by holding the same reference to the value 42:

```
>>> x = 42 # to get started
>>> list1 = [x] # similar to before
>>> list2 = list1 # give list2 that reference, as well
>>> id(x) # all refer to the same data
505494448
```

```
>>> id(list1[0])
505494448
>>> id(list2[0])
505494448
```

What happens when we change `list1[0]`?

```
>>> list1[0] = 43 # this will change the reference held by the "zeroth" element of list1
>>> list1[0]
43
not surprising at all
>>> list2[0]
43
aha! list1 and list2 refer to the same list
>>> x
42
x is a distinct reference
```

Let's see

```
>>> id(list1[0]) # indeed, the reference of that zeroth element has changed
505494464
>>> id(list2[0]) # list2's zeroth element has changed too!
505494464
>>> id(x) # x is still, happily, the same as before
505494448
```

This example is illustrated in Figure 5.3 and codelens examples ‘ch05\_ref1’ and ‘ch05\_ref2’.

(ch05\_ref1)  
(ch05\_ref2)

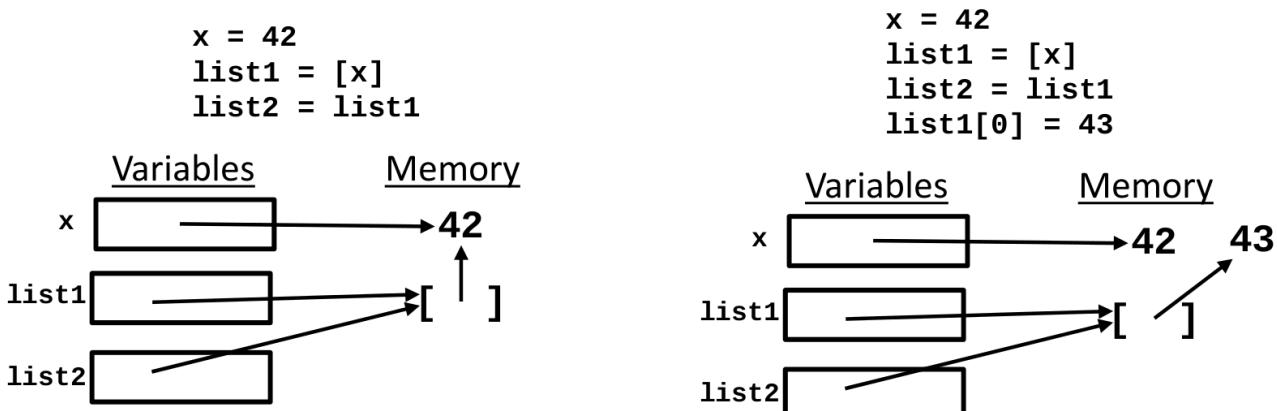


Figure 5.3: A graphical illustration of what happens when you modify the elements in a list that is referenced by two variables.

#### 5.4.2 Mutable Data Types Can Be Changed Using Other Names!

Note that `list2[0]` has been changed in the above example, even though no assignment statements involving `list2[0]` were run! Because `list2` is really just another name for the exact same data as `list1`, we say that `list2` is an *alias* of `list1`. Note that aliases should be used with caution as they can be both powerful and dangerous, as we explore more below.

Lists are an example of a *mutable* data type. They are mutable because their component parts (in this case, the elements of the list) can be modified. If you have two different variables that both refer to the same mutable piece of data – as we do with `list1` and `list2` above – changes that are made to the data’s components using one of those variables will also be seen when you use the other variable, since both variables refer to the same thing.



Two images here—mutation can be good ... or bad.

Data types that do not allow changes to their components are termed *immutable*. Integers, floating-point values, and booleans are examples of immutable types. This isn't surprising, because they do not have any accessible component parts that could mutate anyway.



I love to talk. I guess that makes me immutable!



Python lets you compute what bits make up these data types, but you can't change—or even read—the individual bits themselves.

Types with component parts can also be immutable: for example, strings are an immutable type. If you try to change a piece of a string, Python will complain:

```
>>> s = 'immutable'
>>> s[0] = ''
Traceback (most recent call last):
 File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

Keep in mind that, mutable or immutable, you can always use an assignment to make a variable refer to a different piece of data.

```
>>> s = 'immutable'
>>> s
'immutable'
>>> s = 'mutable'
>>> s
'mutable'
```

### Takeaway Message

There are a few key ideas to keep in mind:

- All Python variables have a value (the piece of data to which they refer) and a reference (the memory location of that piece of data).
- Python assignment copies only the reference.
- Mutable data types like lists allow assignment to component parts. Immutable data types like strings do not allow assignment to component parts.
- If you have two different variables that both refer to the same mutable piece of data, changes that are made to the data's components using one of the variables will also be seen when you use the other variable.
- It is always possible to use an assignment statement to make a variable refer to a different piece of data.

## 5.5 Mutable Data + Iteration: Sorting out Artists

Now that we have introduced the two fundamental concepts in imperative programming, we return to our recommendation example in order to motivate and illustrate the power of data mutation and iteration together. The example we will use is sorting a list of elements.

### 5.5.1 Why Sort? Running Time Matters

Before we dive into the details of sorting, let's establish (at least one reason) why it is useful to have data that is in sorted order. In short, sorted data makes data processing much faster.

But how do we measure what is “fast” vs. what is “slow?” The key to analyzing how long a program takes to run is to count the number of operations that it will perform for a given size input. Computer Scientists rarely care how fast or slow programs are for very small input: they are almost always fast if the input is small. However, when processing large inputs (for example, millions of users who have each rated hundreds or even thousands of artists), speed becomes critical.



*In fact, if we were trying to build a system to handle millions of users, we would need to make many more optimizations and use fundamentally different algorithms, but that is beyond the scope of this chapter. Here we show you how to speed things up to handle slightly larger inputs.*

So, let's look at how long it takes to calculate the number of matches between two lists using the function from section 5.3.4, reproduced here:

```
def numMatches(list1, list2):
 """ return the number of elements that match between
 list1 and list2 """
 count = 0
 for item in list1:
 if item in list2:
 count += 1
 return count
```

Let's say for the moment that each list has four elements in it. Take a moment to think about how many comparisons you think the program will make... then read on below.

First, you take the first element of the first list, and ask if it is in the second list. The `in` command is a bit deceptive because it hides a significant number of comparisons. How does Python tell if an item is in a list? It has to compare that item to every item in the list! So in this case, the first item in the first list is compared to every item (i.e., four of them) in the second list to determine whether it is in that list.

“Wait!” you say. If the item is actually in the list, it doesn't actually have to check all four, and can stop checking when it finds the item in question. This is exactly correct, but in fact it doesn't matter in our analysis. For the purpose of an analysis like this, computer scientists are quite pessimistic. They rarely care what happens when things work out well—what they care about is what might possibly happen in the *worst case*. In this case, the worst case is when the item is not in the list and Python has to compare it to every item in the list to determine it's not there. Since what we care about is this worst case behavior, we will perform our analysis as if we were dealing with the worst case.

So, back to the analysis: For the first item in the list, Python made four comparisons to the items in the second list – commands that were hidden away in the `in` command. Now our program moves on to the second item in the first list, where it again makes four comparisons with the second list. Similarly, it makes four comparisons for each of the third and the fourth elements in the first list. For a total of  $(4 + 4 + 4 + 4 = 4 * 4 = 16)$ . Again, this probably doesn't sound like such a bad number. After all, your computer can make 16 comparisons in *way* less than a second. But what if our lists were longer? What if the user had rated 100 artists and the comparison user had rated 1000 (high, but not crazy)? Then the system would have to do 1000 comparisons (to the items in the second list) for each of the 100 items in the first list for a total of  $(100 * 1000 = 10^5)$  comparisons. Still not huge, but hopefully you can see where this is going. In general, the matching algorithm we've written above takes  $(N*M)$  comparisons, where  $(N)$  is the size of the first list and  $(M)$  is the size of the second list. For simplicity, we might just assume that the two lists will always be the same length,  $(N)$ , in which case it makes  $(N^2)$  comparisons.

The good news is that we can do significantly better, but we have to make an assumption about the lists themselves. What if our lists were sorted alphabetically? How could that make our matching algorithm faster? The answer is that we can keep the lists “synchronized,” so to speak, and march through both lists at the same time, rather than pulling a single element out of the first list and comparing it to all of the elements in the

second. For example, if you are looking at the first element of the first list and it is “Black Eyed Peas” and the first element of the second list is “Counting Crows” you know that the Black Eyed Peas do not appear in the second list, because C is already past B. So you can simply discard the Black Eyed Peas and move on to the next artist in the first list.



*Discarding the “Black Eyed Peas” would save my hearing!*

Here’s what the new algorithm looks like. Remember that it assumes the lists will be in sorted order (we’ll talk about how to sort lists later).

- Initialize a counter to 0
- Set the current item in each list to be the first item in each list
- Repeat the following until you reach the end of one of the two lists:
- Compare the current items in each list.
- If they are equal, increment the counter and advance the current item of both lists to the next items in the lists.
- Else, if the current item in the first list is alphabetically before the current item in the second list, advance the current item in the first list.
- Else, advance the current item in the second list.
- The counter holds the number of matches.

Before you look at the code below, ask yourself: “What kind of loop should I use here? A for loop or while loop?” When you think you’ve got an answer you’re happy with, read on.

Here’s the corresponding Python code:

```
(ch05_matches)
```

Using `==`, `>`, `<`, etc. on strings is perfectly valid and these operators will compare strings alphabetically. Recall from section 4.2.3 that text is represented numerically. In this encoding all of the capital letters are mapped to consecutive numbers, with ‘A’ getting the lowest number and ‘Z’ getting the highest. The lower-case letters are also mapped to consecutive numbers, which are all higher than the number used for ‘Z’. It is these encodings that are used when strings are compared. This means that strings that start with capital letters will always come “alphabetically” before those that start with lower case letters. This issue is important to consider in our program (we haven’t yet), and we’ll discuss it below in Section 5.5.2.



*Python 2 and earlier versions use ASCII encoding, while Python 3 uses a slightly different encoding called Unicode, but everything here applies to both encodings.*

Now the question remains, is this approach really faster than the previous approach to comparing the elements in two lists? The answer is: definitely yes. Let’s again look at the number of comparisons that would need to be made to compare two lists with 4 elements each. Imagine that none of the elements match and that the lists are exactly interleaved alphabetically. That is, first the element in list 1 is smaller, then the element in list 2 is smaller, and so on, as in the lists `["Amy Winehouse", "Coldplay", "Madonna", "Red Hot Chili Peppers"]` and `["Black Eyed Peas", "Dave Matthews Band", "Maroon 5", "Stevie Nicks"]`.

With these two lists, the code above will never trigger on the first `if` condition—it will always increment either `i` or `j`, but not both. Furthermore, it will run out of elements in one list just before it runs out of elements in the second. Essentially it will look at all the elements in both lists.

At first it might seem like we haven’t made any improvements. After all, aren’t we still looking at all the elements of both lists? Aha, but now there’s an important difference! Whereas before we were looking at all the elements in the second list *for each element in the first list*, here we are only looking at all the elements in the second list *once*. In other words, each time through the loop, either `i` or `j` is incremented, and they are never decremented. So either `i` or `j` will hit the end of the list after all of the elements of that list have been looked at and one fewer than the elements of the other list have been looked at. So in this example, this means we will make exactly 7

comparisons. In general, if the lists are both length  $\sqrt{N}$ , the number of comparisons this algorithm will make is  $\sqrt{N+N-1}$  or  $\sqrt{2N - 1}$ . So even for the case where one list has 100 elements and the second has 1000, this is only about 1100 comparisons, a significant improvement over the  $\sqrt{10^5}$  of the previous approach!

In technical terms, computer scientists call this second algorithm a *linear time algorithm* because the equation which describes the number of comparisons is linear. The first algorithm is called a *quadratic time algorithm* because its equation is quadratic.

### 5.5.2 A Simple Sorting Algorithm: Selection Sort

Now that we've motivated at least one reason to sort data (there are many others—you can probably think of several of them), let's look at an algorithm for actually doing the sorting.

Let's consider the list of artists that the user likes that we collect by prompting the user at the start of the program. Recall that the code that gathers this list is the following:

```
newPref = input("Please enter the name of an artist \
or band that you like: ")

while newPref != "":
 prefs.append(newPref)
 newPref = input("Please enter an artist or band \
that you like, or just press enter to see recommendations: ")

print('Thanks for your input!')
```

Remember that in Python 2 you'll need to use `raw_input` rather than `input`.

The artists the user has entered are stored in the list `prefs`, but that list is entirely unsorted. We would like to sort this list alphabetically, and clean up the formatting of the text while we're at it.



*I don't think this will work for LMFAO!*

First, to make our lives easier, and to facilitate the matching process between user profiles, we will make a pass through the list to standardize the format of the artist names. In particular, we want to be sure that there is no leading (or trailing) whitespace, and that the artist or band names are in Title Case (i.e., the first letter of each word is capitalized, and the rest are lower case). Even though this format may fail for some bands, we'll work with it because it gives us a standard representation that allows the strings to be sorted and compared without us having to worry about case issues getting in the way.



*Recall that all uppercase strings are considered “less than” lower case strings, and it would be confusing to the user to alphabetically sort ZZ Top before adele.*

Because strings are immutable objects, we can't actually change them, but rather we have to generate new strings with the formatting we desire. Here is the code that accomplishes this goal (note that we also build a new list, which is not strictly necessary):

```
standardPrefs = []
for artist in prefs:
 standardPrefs.append(artist.strip().title())
```

The `strip` function returns a new string identical to `artist`, but without any leading or trailing whitespace. Then the `title` function returns that same string in Title Case.

Now that our data is standardized, we can sort it from smallest (alphabetically first) to largest. Sorting is an important topic of study in computer science in its own right.



*I sort of knew that it was important!*

Here we will discuss one sorting algorithm, but there's much more to be said on the subject.

To develop our algorithm, we start with small cases: what are the computational pieces that enable the rearrangement of a list?

A two-element list is the smallest (non-trivial) case to consider: in the worst case, the two elements would need to swap places with each other.

In fact, this idea of a `swap` is all we need! Imagine a large list that we'd like sorted in ascending order.

- First, we could find the smallest element. Then, we could swap that smallest element with the first element of the list.
- Next, we would search for the second-smallest element, which is the smallest element of the rest of the list. Then, we could swap it into place as the second element of the overall list.
- We continue this swapping so that the third-smallest element takes the third place in the list, do the same for the fourth, ... and so on ... until we run out of elements.

This algorithm is called *selection sort* because it proceeds by repeatedly selecting the minimum remaining element and moving it to the next position in the list.

What functions will we need to write selection sort? It seems we need only two:

- `index_of_min(aList, starting_index)`, which will return the index of the smallest element in `aList`, starting from `index starting_index`.
- `swap(aList, i, j)`, which will swap the values of `aList[i]` and `aList[j]`

Here is the Python code for this algorithm:

(ch05\_selSort)

Notice that the code above does not return anything. But when we run it, we see that it works. After the call to `selectionSort`, our list is sorted!

```
>>> standardPrefs
['Maroon 5', 'Adele', 'Lady Gaga']
>>> selectionSort(standardPrefs)
>>> standardPrefs
['Adele', 'Lady Gaga', 'Maroon 5']
```

### 5.5.3 Why `selectionSort` Works

Why does this code work? What is more, why does it work *even though `swap` does not have a return statement?* There are two key factors.

First, lists are mutable. Thus, two (or more) variables can refer to the same list, and changes that are made to list elements using one variable will also be seen when you use the other variables. But where are the two variables referring to the same list? It seems that `standardPrefs` is the only variable referring to the original three-element list in the example above.

This is the second factor: when inputs are passed into functions, the function parameters are assigned to each input just as if an assignment statement had been explicitly written, e.g.,

```
listToSort = standardPrefs
```

occurs at the beginning of the call to `selectionSort(standardPrefs)`.

As a result, as long as `listToSort` and `standardPrefs` refer to the same list, any changes that are made to `listToSort`'s elements will affect `standardPrefs`'s elements—because `listToSort`'s elements and `standardPrefs`'s elements are one and the same thing.

**Takeaway message:** *Passing inputs into a function is equivalent to assigning those inputs to the parameters of that function. Thus, the subtleties of mutable and immutable data types apply, just as they do in ordinary assignment.*

The left side of Figure 5.4 illustrates what is happening at the beginning of the first call to `swap`. `swap`'s variables are displayed in red, while `selectionSort`'s variables are displayed in black. Notice that `swap`'s `i` and `j` refer to the index locations in the list `aList` (which is just another name for the list `listToSort`).

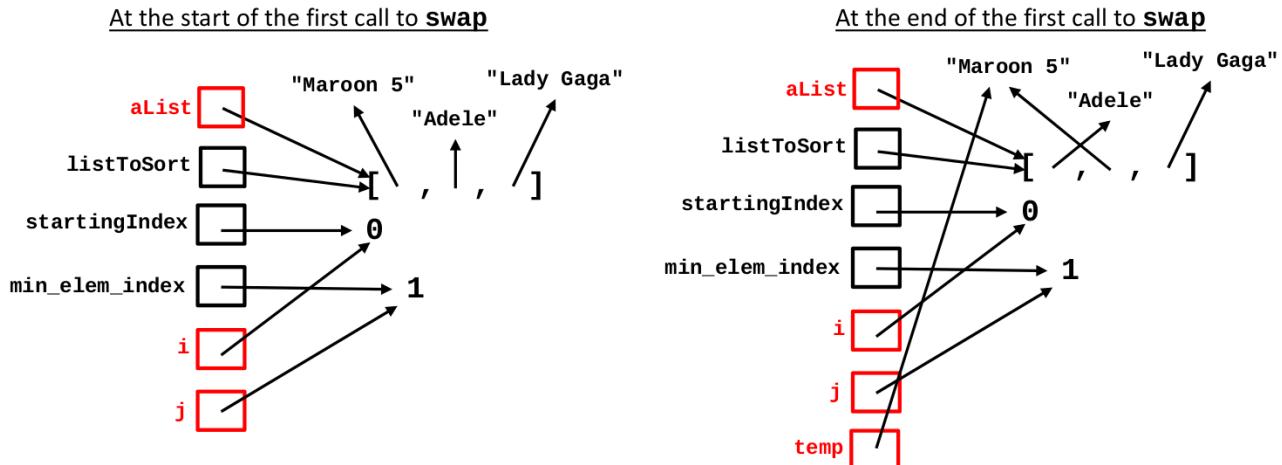


Figure 5.4: A graphical depiction of the variables in `selectionSort` and `swap` at the beginning and the end of the first call to `swap`

The right side of Figure 5.4 shows how the variables look at the very end of the first call to `swap`. When `swap` finishes, its variables disappear. But notice that even when `swap`'s `aList`, `i`, `j`, and `temp` are gone, the value of the list has been changed in memory. Because `swap`'s `aList` and `selectionSort`'s `listToSort` and the original `standardPrefs` are all references to the same collection of mutable elements, the effect of assignment statements on those elements will be shared among all of these names. After all, they all name the same list!

#### 5.5.4 A Swap of a Different Sort

Consider a very minor modification to the `swap` and `selectionSort` functions that has a very major impact on the results:

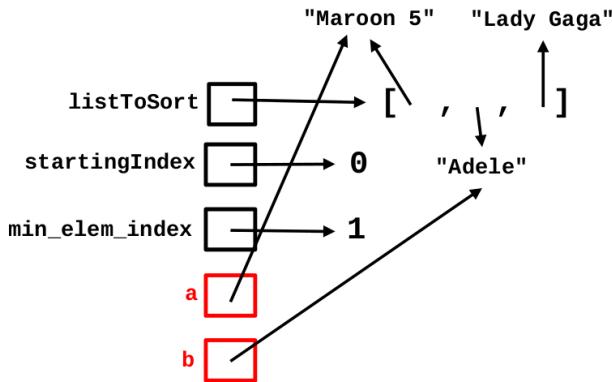
```
def selectionSort(listToSort):
 """sorts listToSort iteratively and in-place"""
 for starting_index in range(len(listToSort)):
 min_elem_index = index_of_min(listToSort, starting_index)
 # now swap the elements!
 swap(listToSort[starting_index], listToSort[min_elem_index])

def swap(a, b):
 """swaps the values of a and b"""
 temp = a
 a = b
 b = temp
```

The code looks almost the same, but now it does not work!

```
>>> standardPrefs # Assume this list is already populated with standardized user preferences
['Maroon 5', 'Adele', 'Lady Gaga']
>>> selectionSort(standardPrefs)
>>> standardPrefs
['Maroon 5', 'Adele', 'Lady Gaga']
```

At the start of the first call to `swap`



At the end of the first call to `swap`

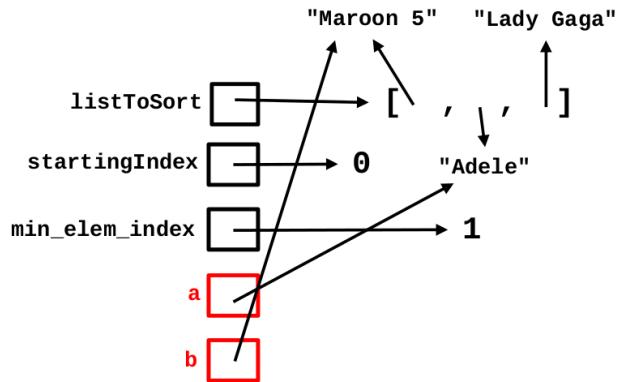


Figure 5.5: A graphical depiction of the variables in `selectionSort` and the new (and faulty) `swap` at the beginning and the end of the first call to `swap`

The variables `a` and `b` in `swap` indeed do get swapped, as the following before-and-after figure illustrates. But nothing happens to the elements of `alist` or, to say the same thing, nothing happens to the elements of `standardPrefs`.

Figure 5.5 illustrates what is happening before and after the call to `swap`.

What happened? This time, `swap` has only two parameters, `a` and `b`, whose references are copies of the references of `listToSort[start_index]` and `listToSort[min_elem_index]`, respectively. Keep in mind Python's mechanism for assignment: *Assignments make a copy of a reference*.

Thus, when `swap` runs this time, its assignment statements are working on *copies* of references. All of the swapping happens just as specified, so that the values referred to by `a` and `b` are, indeed, reversed. But the references held within `selectionSort`'s list `listToSort` have not changed. Thus, the value of `listToSort` does not change. Since the value of `standardPrefs` *is* the value of `listToSort`, nothing happens.

As we mentioned at the start of this section, Selection Sort is just one of many sorting algorithms, some of which are faster than others (Selection Sort is not particularly fast, though it's certainly not the slowest approach either). In practice, especially for now, you'll probably just call Python's built-in `sort` function for lists, which is efficient, and more importantly, correctly implemented.

```
>>> standardPrefs # Assume this list is already populated with standardized user preferences
['Maroon 5', 'Adele', 'Lady Gaga']
>>> standardPrefs.sort()
>>> standardPrefs
['Adele', 'Lady Gaga', 'Maroon 5']
```

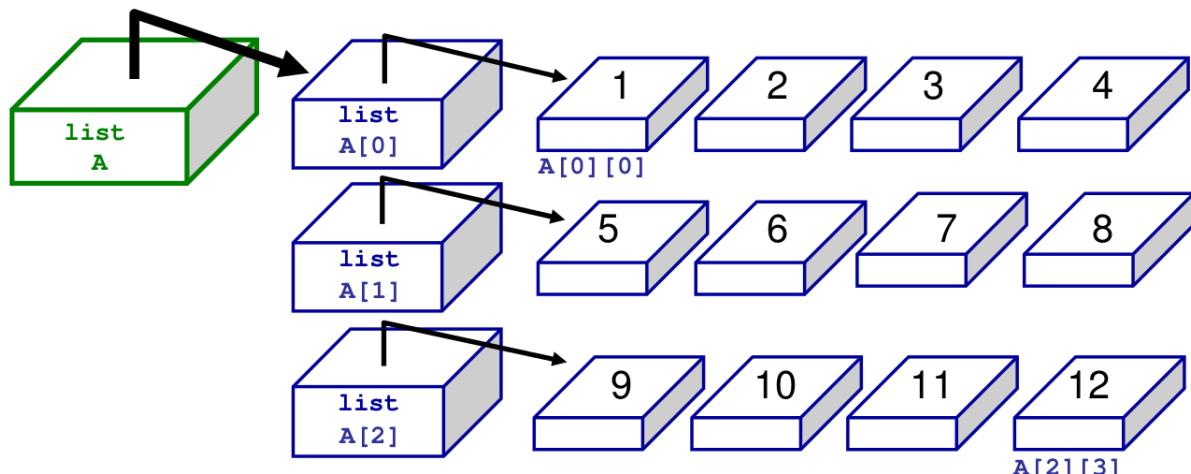


Figure 5.6: A graphical depiction of a 2D array. Following our model from above, the green box is stored in the CPU, and the rest of the data (including the references to other lists) is stored in memory.

### 5.5.5 2D Arrays and Nested Loops

It turns out that you can store more than just numbers in lists: arbitrary data can be stored. We've already seen many examples in Chapter 2 where lists were used to store strings, numbers, and combinations of those data types. Lists of lists are not only possible, but powerful and common.

In an imperative context, lists are often called *arrays*, and in this section we examine another common array structure: arrays that store other arrays, often called *2D arrays*.



*Our attorneys have advised us to note that technically, there are very subtle differences between “lists” and “arrays” – but the distinctions are too fine to worry about here.*

The concept behind a 2D array is simple: it is just a list whose elements themselves are lists. For example, the code

```
>>> a2DList = [[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]]
```

creates a list named `a2DList` where each of `a2DList`'s elements is itself a list. Figure 5.6 illustrates this 2D array (list) graphically. In the figure the array name has been shortened to just `A` for conciseness.

You can access the individual elements in the 2D array by using “nested” indices as shown below:

```
>>> a2DList[0] # Get the first element of a2DList
[1, 2, 3, 4]
>>> a2DList[0][0] # Get the first element of a2DList[0]
1
>>> a2DList[1][1] # Get the second element of a2DList[1]
6
```

We can also ask questions about `a2DList`'s height (the number of rows) and its width (the number of columns):

```
>>> len(a2DList) # The number of elements (rows) in a2DList
3
>>> len(a2DList[0]) # The number of elements in a list in a2DList (i.e., the number of columns)
4
```

Typically, arrays have equal-length rows, though “jagged” arrays are certainly possible.

This 2D structure turns out to be a very powerful tool for representing data. In fact, we've already seen an example of this in our recommender program: the stored users' preferences were represented by a list-of-lists, i.e., a 2D array. In this case, the array certainly was “jagged” as different stored users had rated different numbers of artists.

Above we wrote some code to standardize the representation of one particular user's preferences. But we might be unsure about the database—are those lists also standardized (and sorted)? If not, no problem. It's easy to write code that quickly standardizes all the elements in all of the lists:

(ch05\_stand)

In the above code, the outer loop controls the iteration over the individual users. That is, for each iteration of the outer loop, the variable `storedUser` is assigned a list containing the preferences of one of the users. The inner loop then iterates over the elements in that user's list of preferences. In other words, `artist` is simply a string.

We can also write the above code as follows:

```
def standardizeAll(storedPrefs):
 """ Returns a new list of lists of stored user preferences,
 with each artist string in Title Case,
 with leading and trailing whitespace removed.
 """

 standardStoredPrefs = []
 for i in range(len(storedPrefs)):
 standardStoredUser = []
 for j in range(len(storedPrefs[i])):
```

```

standardStoredUser.append(storedPrefs[i][j].strip().title())
standardStoredPrefs.append(standardStoredUser)
return standardStoredPrefs

```

This code does the same thing, but we use the indexed version of the `for` loop instead of letting the `for` loop iterate directly over the elements in the list. Either construct is fine, but the latter makes the next example more clear.

Because lists are mutable, we don't actually have to create a whole new 2D array just to change the formatting of the strings in the array. Remember that strings themselves are not mutable, so we can't directly change the strings stored in the list. However, we can change *which strings* are stored in the original list, as follows:

```

def standardizeAll(storedPrefs):
 """ Mutates storedPrefs so that each string is in
 Title Case, with leading and trailing whitespace removed.
 Returns nothing.
 """
 for i in range(len(storedPrefs)):
 for j in range(len(storedPrefs[i])):
 standardArtist = storedPrefs[i][j].strip().title()
 storedPrefs[i][j] = standardArtist

```

Notice that this code is slightly simpler than the code above, and avoids the overhead of creating a whole new list of lists.

### 5.5.6 Dictionaries

So far we've looked at one type of mutable data: arrays. Of course, there are many others. In fact, in the next chapter you'll learn how to create your own mutable data types. But more on that later. For now we will examine a built-in datatype called a *dictionary* that allows us to create mappings between pieces of data.



*Have you ever looked up dictionary in a dictionary?*

To motivate the need for a dictionary, let's once again return to the recommender program. Until now, in our examples we have not been associating stored user's names with their preferences. While we don't need to store the user's name with their examples in order to calculate the best matching user (assuming we know that the list of stored users does not contain the current user!), there are many advantages of doing so. For one, we can make sure that the user doesn't get matched against themselves! Additionally, it might be nice in an extended version of the system to suggest possible "music friends" to the user, whose tastes they may want to track.

So how will we associate user names with their preferences? One way would be to simply use lists. For example, we could have a convention where the first element in each user's stored preferences is actually the name of the user herself. This approach works, but it is not very elegant and is likely to lead to mistakes. For example, what if another programmer working on this system forgets that this is the representation and starts treating user names as artist names? Perhaps not a tragedy, but incorrect behavior nonetheless. In short, elegant design is important!

What we need is a single place to store the mapping from user names to user preferences that we can pass around to all of the functions that need to know about it. This is where the dictionary comes in handy! A dictionary allows you to create mappings between pieces of (immutable) data (the *keys*) and other pieces of (any kind of) data (the *values*). Here's an example of how they work:

```

>>> myDict = {}
creates a new empty dictionary

create an association between 'April' her
list of preferred artists.
'April' is the key and the list is the value
>>> myDict['April'] = ['Maroon 5', 'The Rolling Stones',
 'The Beatles']

```

```

Ditto for Ben and his list
>>> myDict['Ben'] = ['Lady Gaga', 'Adele',
 'Kelly Clarkson', 'The Dixie Chicks']
>>> myDict
display the mappings currently in the dictionary
{'April': ['Maroon 5', 'The Rolling Stones', 'The Beatles'],
'Ben': ['Lady Gaga', 'Adele', 'Kelly Clarkson', 'The Dixie Chicks']}
>>> myDict['April']
get the item mapped to 'April'
['Maroon 5', 'The Rolling Stones', 'The Beatles']
>>> myDict['f']
get the item mapped to 'f'. Will cause an error because

mappings are one-way--'f' is not

a valid key.
Traceback (most recent call last):
File "<pyshell\#14>", line 1, in <module>
myDict['f']
KeyError: 'f'
>>> myDict.has_key('f')
Check whether a key is in the

dictionary
False
>>> myDict.keys()
Get the keys in the dictionary
dict_keys(['April', 'Ben'])
Notice that this is a special kind of

object that you can iterate

over, but that's not a list
>>> myDict[1] = 'one'
keys can be any immutable type
>>> myDict[1.5] = [3, 5, 7] # values can also be mutable, and

we can mix types in the same

dictionary
>>> myDict[[1, 2]] = 'one'
Keys cannot be mutable

Traceback (most recent call last):
File "<pyshell\#36>", line 1, in <module>
myDict[[1, 2]] = 'one'
TypeError: list objects are unhashable

a shorthand way to create a dictionary
>>> userPrefs = {'April': ['Maroon 5', 'The Rolling Stones', 'The Beatles'],
'Ben': ['Lady Gaga', 'Adele', 'Kelly Clarkson', 'The Dixie Chicks']}
>>> userPrefs
{'April': ['Maroon 5', 'The Rolling Stones', 'The Beatles'],
'Ben': ['Lady Gaga', 'Adele', 'Kelly Clarkson', 'The Dixie Chicks']}

```

The order in which the key-value pairs appear in Python's output of `userPrefs` represents its internal representation of the data. They are not always in the same order.

Now let's look at how we can modify our recommender code to use a dictionary instead of a list of lists:

```

def getBestUser(currUser, prefs, userMap):
 """ Gets recommendations for currUser based on the users in userMap (a dictionary)

```

```

and the current user's preferences in prefs (a list) """
users = userMap.keys()
bestuser = None
bestscore = -1
for user in users:
 score = numMatches(prefs, userMap[user])
 if score > bestscore and currUser != user:
 bestscore = score
 bestuser = user
return bestuser

```

Notice that dictionaries allow us to make sure we're not matching a user to their own stored preferences! Pretty simple... and flexible too!

## 5.6 Reading and Writing Files

We've almost got all the pieces we need to build our recommender program. The last major piece we are missing is the ability to read and write files, which we will need to load and store the preferences for the users the system already knows about.

Fortunately, working with files is very simple in Python, and we illustrate file input and output (i.e., file I/O) through example. Imagine that our stored user preferences exist in a file named musicrec-store.txt. There is one line in the file for each user and the format of each line is the following:

```
username:artist1,artist2,...,artistN
```

We can write the following code to read in and process all the lines from this file:

```

def loadUsers(fileName):
 """ Reads in a file of stored users' preferences stored
 in the file 'fileName'.
 Returns a dictionary containing a mapping of user
 names to a list of preferred artists
 """
 file = open(fileName, 'r')
 userDict = {}
 for line in file:
 # Read and parse a single line
 [userName, bands] = line.strip().split(":")
 bandList = bands.split(",")
 bandList.sort()
 userDict[userName] = bandList
 file.close()
 return userDict

```

When we call this function we would pass it the filename musicrec-store.txt.

There are a few key pieces to this function. First, we have to open the file for reading. This task is accomplished with the line `file = open(fileName, 'r')`, which gives us a link to the contents of the file through `file`, but we can only read from this file (we cannot write to it). If we wanted to write to the file, we would specify '`w`' as the second argument to the `open` function; if we wanted to both read and write, we would pass '`w+`' as the second argument.

Once we have our handle to the file's contents (`file`) we can read lines from it using the `for` loop construct above. As the `for` loop iterates over `file`, what it is actually doing is pulling out one line at a time until there are no more lines, at which point the `for` loop ends.

Finally, when we have read all the data out of the file and stored it in the dictionary in our program, we can close the file using `file.close()`.

The function below shows the part of the code that writes the stored user preferences (including the current user) back to the file:

```

def saveUserPreferences(userName, prefs, userMap, fileName):
 """ Writes all of the user preferences to the file.
 Returns nothing. """
 userMap[userName] = prefs

```

```

file = open(fileName, "w")
for user in userMap:
 toSave = str(user) + ":" + ",".join(userMap[user]) + \
 "\n"
 file.write(toSave)
file.close()

```

## 5.7 Putting It All Together: Program Design

Now that we've got all the tools, it's time to construct the whole recommender program. However, as a disclaimer, large scale program design can at times feel more like an art than a science. Maybe more science than art! There are certainly guiding principles and theories, but you rarely "just get it right" on your first try, no matter how careful you are. So expect that in implementing any program of reasonable size, you'll need to make a couple of revisions. Think of it like writing a paper—if you like writing papers.



*CS is at least as creative and precise as careful writing; don't be afraid of creating—and redoing—several drafts!*

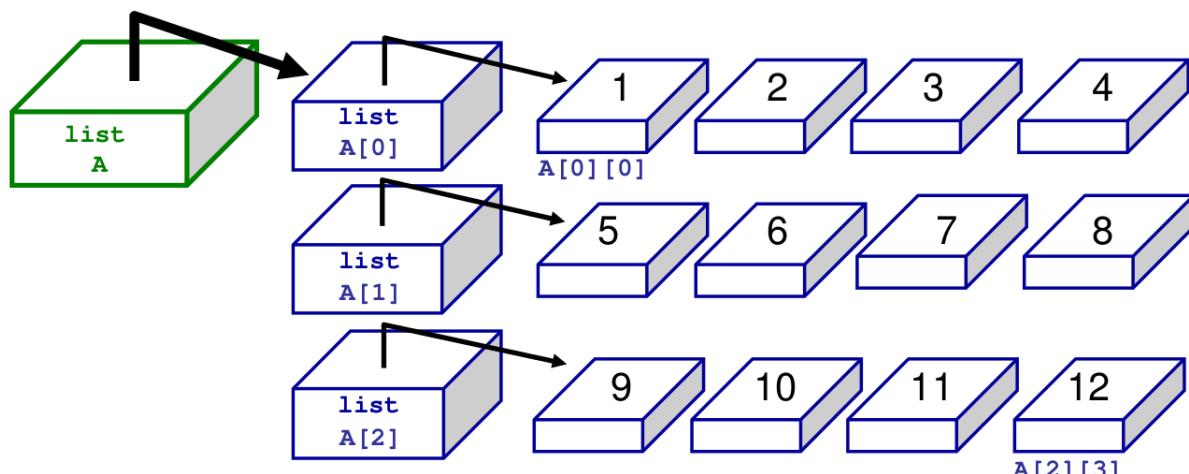
The first step to program design is trying to figure out what data your program is responsible for and how that data comes into the program (input), gets manipulated (computation) and is output from the program (output). In fact, this task is so important that we had you do this at the beginning of the chapter.

After identifying the data your program must keep track of, the next step is to decide what data structures you will use to keep track of this data. Do you need ints, lists, or dictionaries, or some other structure entirely? Take a moment to choose variable names and data types for each of the piece of data you identified at the beginning of the chapter (or that you identify now).

Finally, now that you have identified our data and the computation that our program needs to perform, you are ready to start writing functions. We can start with any function we like—there's no right order as long as we are able to test each function as we write it.

Our complete program is given below in listing 5.1. Let's return to the idea of collaborative filtering and examine how this program implements the basic CF algorithm we described in section 5.1. As we described there, the basic idea behind collaborative filtering is to try to make predictions about a user by looking at data from a number of other users. There are two basic steps to a CF algorithm:

- Find the user (or users) most similar to the current user,
- Make predictions based on what they like.



In the code in listing 5.1, these two steps are performed in helper functions called in `getRecommendations`. First, `findBestUser` finds and returns the user whose tastes are closest to the current user. Then `drop` returns a list of the artists who the best user likes who are not already in the current user's list.

Notice that even though the code is relatively short, we have chosen to separate the two stages of the algorithm into two separate functions for clarity. Generally it is a good idea for each semantic piece of your algorithm to have its own function. Notice also that `findBestUser` also relies on a helper function (`numMatches`), which again helps make the specific functionality of each piece of the code more clear.

One more item that's new in the code below is the last line:

```
if __name__ == "__main__": main()
```

You will notice that the function `main` implements the main control of the recommendation program. We could run our recommendation program by loading this code into the Python interpreter and then typing:

```
>>> main()
```

However, to keep the user from having to type this extra command, we include the above line, which tells the program to automatically run the function `main` as soon as the code is loaded into the interpreter.

Finally, you'll notice the variable `PREF_FILE` at the top of the program. This variable is known as a *global variable* because it is accessible to all of the functions in the program. By convention, global variable names are often ALL CAPS to distinguish them from local variables and parameters. Generally global variables should be avoided, but there are a few special cases such as this one where we want to avoid sprinkling a hard-coded value (in this case, the name of the file) throughout the program. Using a global variable for the filename makes it easy to change in the future.

```
A very simple music recommender system.
```

```
PREF_FILE = "musicrec-store.txt"

def loadUsers(fileName):
 """ Reads in a file of stored users' preferences
 stored in the file 'fileName'.
 Returns a dictionary containing a mapping
 of user names to a list preferred artists
 """
 file = open(fileName, 'r')
 userDict = {}
 for line in file:
 # Read and parse a single line
 [userName, bands] = line.strip().split(":")
 bandList = bands.split(",")
 bandList.sort()
 userDict(userName) = bandList
 file.close()
 return userDict

def getPreferences(userName, userMap):
 """ Returns a list of the user's preferred artists.

 If the system already knows about the user,
 it gets the preferences out of the userMap
 dictionary and then asks the user if she has
 additional preferences. If the user is new,
 it simply asks the user for her preferences.
 """
 newPref = ""
 if userName in userMap:
 prefs = userMap(userName)
 print("I see that you have used the system before.")
 print("Your music preferences include:")
 for artist in prefs:
 print(artist)
 print("Please enter another artist or band that you")
 print("like, or just press enter")
 newPref = input("to see your recommendations: ")
 else:
```

```

prefs = []
print("I see that you are a new user.")
print("Please enter the name of an artist or band")
newPref = input("that you like: ")

while newPref != "":
 prefs.append(newPref.strip().title())
 print("Please enter another artist or band that you")
 print("like, or just press enter")
 newPref = input("to see your recommendations: ")

Always keep the lists in sorted order for ease of
comparison
prefs.sort()
return prefs

def getRecommendations(currUser, prefs, userMap):
 """ Gets recommendations for a user (currUser) based
 on the users in userMap (a dictionary)
 and the user's preferences in pref (a list).
 Returns a list of recommended artists. """
 bestUser = findBestUser(currUser, prefs, userMap)
 recommendations = drop(prefs, userMap[bestUser])
 return recommendations

def findBestUser(currUser, prefs, userMap):
 """ Find the user whose tastes are closest to the current
 user. Return the best user's name (a string) """
 users = userMap.keys()
 bestUser = None
 bestScore = -1
 for user in users:
 score = numMatches(prefs, userMap[user])
 if score > bestScore and currUser != user:
 bestScore = score
 bestUser = user
 return bestUser

def drop(list1, list2):
 """ Return a new list that contains only the elements in
 list2 that were NOT in list1. """
 list3 = []
 i = 0
 j = 0
 while i < len(list1) and j < len(list2):
 if list1[i] == list2[j]:
 i += 1
 j += 1
 elif list1[i] < list2[j]:
 i += 1
 else:
 list3.append(list2[j])
 j += 1
 return list3

def numMatches(list1, list2):
 """ return the number of elements that match between
 two sorted lists """
 matches = 0

```

```

i = 0
j = 0
while i < len(list1) and j < len(list2):
 if list1[i] == list2[j]:
 matches += 1
 i += 1
 j += 1
 elif list1[i] < list2[j]:
 i += 1
 else:
 j += 1
return matches

def saveUserPreferences(userName, prefs, userMap, fileName):
 """ Writes all of the user preferences to the file.
 Returns nothing. """
 userMap[userName] = prefs
 file = open(fileName, "w")
 for user in userMap:
 toSave = str(user) + ":" + ",".join(userMap[user]) + \
 "\n"
 file.write(toSave)
 file.close()

def main():
 """ The main recommendation function """
 userMap = loadUsers(PREF_FILE)
 print("Welcome to the music recommender system!")

 userName = input("Please enter your name: ")
 print ("Welcome, ", userName)

 prefs = getPreferences(userName, userMap)
 recs = getRecommendations(userName, prefs, userMap)

 # Print the user's recommendations
 if len(recs) == 0:
 print("I'm sorry but I have no recommendations")
 print("for you right now.")
 else:
 print(userName, "based on the users I currently")
 print("know about, I believe you might like:")
 for artist in recs:
 print(artist)

 print("I hope you enjoy them! I will save your")
 print("preferred artists and have new")
 print("recommendations for you in the future")

 saveUserPreferences(userName, prefs, userMap, PREF_FILE)

if __name__ == "__main__": main()

```

## 5.8 Conclusion

We've covered a lot of ground in this chapter, ultimately resulting in a recommender program similar in spirit to ones that you undoubtedly use yourself.



*Are there any recommender programs that recommend recommender programs?*

We've also seen that there are often multiple different ways of doing the same thing. For example, recursion, for loops, and while loops all allow us to repeat a computational task. How do you determine which one to use? Some problems are inherently recursive. For example, the edit distance problem from Chapter 2 was very naturally suited for recursion because it can be solved using solutions to smaller versions of the same problem – the so-called “recursive substructure” problem. Other problems, like computing the factorial of a number, can be solved naturally – albeit differently – using recursion or loops. And then there are cases where loops seem like the best way to do business. The fact that there are choices like these to be made is part of what makes designing programs fun and challenging.

## Table Of Contents

- Chapter 5: Imperative Programming
  - 5.1 A Computer that Knows You (Better than You Know Yourself?)
    - \* 5.1.1 Our Goal: A Music Recommender System
  - 5.2 Getting Input from the User
  - 5.3 Repeated Tasks–Loops
    - \* 5.3.1 Recursion vs. Iteration at the Low Level
    - \* 5.3.2 Definite Iteration: for loops
    - \* 5.3.3 How Is the Control Variable Used?
    - \* 5.3.4 *Accumulating* Answers
    - \* 5.3.5 Indefinite Iteration: while Loops
    - \* 5.3.6 for Loops vs. while Loops
    - \* 5.3.7 Creating Infinite Loops On Purpose
    - \* 5.3.8 Iteration Is Efficient
  - 5.4 References and Mutable vs. Immutable Data
    - \* 5.4.1 Assignment by *Reference*
    - \* 5.4.2 Mutable Data Types Can Be Changed Using Other Names!
  - 5.5 Mutable Data + Iteration: Sorting out Artists
    - \* 5.5.1 Why Sort? Running Time Matters
    - \* 5.5.2 A Simple Sorting Algorithm: Selection Sort
    - \* 5.5.3 Why `selectionSort` Works
    - \* 5.5.4 A Swap of a Different Sort
    - \* 5.5.5 2D Arrays and Nested Loops
    - \* 5.5.6 Dictionaries
  - 5.6 Reading and Writing Files
  - 5.7 Putting It All Together: Program Design
  - 5.8 Conclusion

**Previous topic** Chapter 4: Computer Organization

**Next topic** Chapter 6: Fun and Games with OOPs: Object-Oriented Programs

## Quick search

Enter search terms or a module, class or function name.

## Navigation

- index
- next |
- previous |
- cs5book 1 documentation »

© Copyright 2013, hmc. Created using Sphinx 1.2b1.

# CSforAll - PythonBat

## CS for All

CSforAll Web > Chapter5 > PythonBat

This website accompanies the textbook *CS For All* by Alvarado, Dodds, Kuenning, and Libeskind-Hadas

### PythonBat Loops!

This problem invites practice with Python's loops: `for` and `while`.

There are 12 looping problems at the two PythonBat pages:

- Six "medium list problems" for using loops at <http://codingbat.com/python/List-2>
- Six "medium python string problems" for using loops at <http://codingbat.com/python/String-2>

For example, if you open the page of "medium python string problems", the first problem is named `double_char`. To quote its page,

*Given a string, return a string where for every char in the original, there are two chars.*

```
double_char('The') → 'TThhee'
double_char('AAbb') → 'AAAAbbbb'
double_char('Hi-There') → 'HHii--TThheerree'
```

The nice thing about the PythonBat pages is that they will check your code on the fly.

Here is a complete and correct answer for `double_char`:

```
def double_char(str):
 result = ''
 for c in str:
```

```
 result += c*2
 return result
```

Feel free to type—or paste—this in and then check that it does pass the tests.

Next, be sure you feel comfortable with the *strategy* for that `double_char` solution:

- Start with an "accumulator" variable at a suitable starting value
  - Here, that is the `result` variable, started at the value '' (the empty string)
- Compose a loop that accumulates the desired result
- Then return that result

**No docstrings needed!** *ONLY* for this PythonBat loop-practice problem, feel free not to create docstrings. These are pure practice!

Now, on to the details...

### Solve any 5 PythonBat Problems

For this problem, solve any five of the PythonBat problems on the two pages linked above.

- Be sure to name the functions the same as specified by PythonBat
- Copy your working and tested functions into your file

### Extra Credit: Solve the Others, Too!

Good luck with these problems!

Website design by Madeleine Masser-Frye and Allen Wu

# CSforAll - LoopingBack

## CS for All

CSforAll Web > Chapter5 > LoopingBack

This website accompanies the textbook *CS For All* by Alvarado, Dodds, Kuenning, and Libeskind-Hadas

### Looping Back...to Python

This problem asks you to read and alter three Python programs that use *loops*.

Loops are a powerful control structure for *iteration*, i.e., repeating an action a desired number of times.

Assembly language implements loops using the `jump` commands: this problem invites you to try Python's method for expressing them: `for` and `while`.

#### Looping for a while...

Repetition is what computers do best—and what people enjoy *not* having to do!

To get into the design mindset of using *loops*, paste this function (and comments) into a new file. This is a factorial function that uses a Python `for` loop:

```
def fac(n):
 """Loop-based factorial function
 Argument: a nonnegative integer, n
 Return value: the factorial of n
 """
 result = 1 # starting value - like a base case
 for x in range(1,n+1): # loop from 1 to n, inclusive
 result = result * x # update the result by mult. by x
```

```

 return result # notice this is AFTER the loop!

#
Tests for looping factorial
#
assert fac(0) == 1
assert fac(5) == 120

```

### Looping Function to Write #1: power(b,p)

To start, read over and run the above starter file. You should see the tests pass with the expected results.

Then, either copy and alter that function, or create a new function, based on that model, named `power(b, p)`, which should

- Accept any numeric value (base) `b`
- Accept any nonnegative integer (power) `p` (`p` can be 0)
- Return the value of `b**p`.
- The function must use a `for` loop! (writing `return b**p` is not in the spirit of this practice...)
- For this problem, `power(0, 0)` should return `1.0` (we realize this might push some mathematicians' tolerance *past its limit*)

Here are a few tests to paste and use: `# # tests for looping power #`

```

print("power(2, 5): should be 32 ==", power(2, 5))
print("power(5, 2): should be 25 ==", power(5, 2))
print("power(42, 0): should be 1 ==", power(42, 0))
print("power(0, 42): should be 0 ==", power(0, 42))
print("power(0, 0): should be 1 ==", power(0, 0))

```

Hint

The value of `n` in `factorial` is analogous to the value of `p` (the power) in the `power` function. (The base is simply used as the multiplied value...)

### Looping Function to Write #2: summedOdds(L)

Here is a loop-based `summed` function we looked at—or will look at—in class. Include this in your file (and try it out):

```

def summed(L):
 """Loop-based function to return a numeric list.
 ("sum" is built-in, so we're using a different name.)

```

```

 Argument: L, a list of integers.
 Result: the sum of the list L.
"""
result = 0
for e in L:
 result = result + e # or result += e
return result

tests!
assert summed([4, 5, 6]) == 15
assert summed(range(3, 10)) == 42

```

Either by using this as a template or by copy-paste-and-alter, create a loop-based function named `summedOdds(L)`, which should

- Accept any list of integers L.
- You may assume it includes only integers.
- It should return the sum of all of the *odd* elements in L
- ...using a loop!

Be sure to test!

```

tests!
assert summedOdds([4, 5, 6]) == 5
assert summedOdds(range(3, 10)) == 24

```

Hint

Use an `if` statement inside the loop and only add to `result` under the correct conditions

**Disclaimer:** The similarity of this function to any person's name, real or fictional, is entirely coincidental...

### Looping Function to Write #3: `untilARepeat( high )`

This function will let you explore what is sometimes called the *birthday paradox*:

- Even in a very small group of people (23 or more), there is greater than a 50-50 chance that a pair of those people share a birthday.
- This is explained well here, including a nice discussion of why it seems paradoxical.

First, paste and try out the number-guessing `while`-loop example from class:

```
import random

def countGuesses(hidden):
 """Uses a while loop to guess "hidden", from 0 to 99.
 Argument: hidden, a "hidden" integer from 0 to 99.
 Result: the number of guesses needed to guess hidden.
 """
 guess = random.choice(range(0, 100)) # 0 to 99, inclusive
 numguesses = 1 # we just made one guess, above
 while guess != hidden:
 guess = random.choice(range(0, 100)) # guess again!
 numguesses += 1 # add one to our number of guesses
 return numguesses
```

Then, using `countGuesses` as a starting point (feel free to copy and alter it), write a variant of this function named `untilARepeat(high)`, which

- Keeps a list `L` of all of the integers guessed. Start with `L = []`!
- Keeps looping as long as all of the elements in `L` are unique (no repeats).
  - Use a `while` loop!
  - Use the `unique(L)` function that's provided below—it returns a Boolean.
- Within the `while` loop:
  - Make a guess in the `range(0, high)`.
  - Count the number of guesses (add one each time to some kind of counting variable).
  - Add the guess on to the end of the list `L` (see below for a hint on this...).
- When the `while` loop finishes, the function should return the number of guesses needed until it gets a repeat.

#### Hint

- Remember that you can add 1 to a variable with `count += 1`
  - Similarly, you can add a new element `42` to the end of a list `L` with the line  
`L = L + [42]`
- The element needs to be in brackets, because only lists can be added to lists.

### Hint

Feel free to use this `uniq(L)` function provided here. You may have written this in the previous lab!

```
def unique(L):
 """Returns whether all elements in L are unique.
 Argument: L, a list of any elements.
 Return value: True, if all elements in L are unique,
 or False, if there is any repeated element
 """
 if len(L) == 0:
 return True
 elif L[0] in L[1:]:
 return False
 else:
 return unique(L[1:]) # recursion is OK in this function!
```

Once you've written your `untilARepeat` function, you should confirm that `untilARepeat(365)` produces surprisingly small numbers! For example,

```
In [1]: untilARepeat(365)
Out[1]: 25
```

```
In [2]: untilARepeat(365)
Out[2]: 8
```

```
In [3]: untilARepeat(365)
Out[3]: 23
```

```
In [4]: untilARepeat(365)
Out[4]: 33
```

...and then try running 10,000 times with a list comprehension:

```
L = [untilARepeat(365) for i in range(10000)]
```

You might find the average, maximum, and minimum of that list comprehension `L`:

```
In [1]: sum(L)/10000
```

```
In [2]: max(L)
```

```
In [3]: min(L)
```

```
In [4]: 42 in L
```

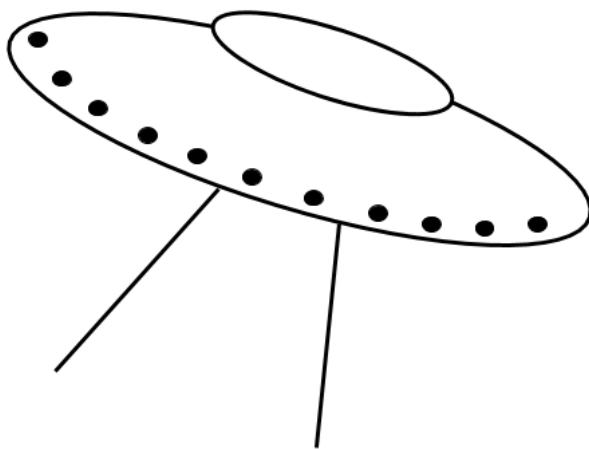
You don't need to hand in these results, but try them to see if their values match your intuition.

Sometimes it's tricky to know whether it's one's program—or intuition—that needs refinement!

Website design by Madeleine Masser-Frye and Allen Wu

# CSforAll - PifromPie

## CS for All



CSforAll Web > Chapter5 > PifromPie

This website accompanies the textbook *CS For All* by Alvarado, Dodds, Kuenning, and Libeskind-Hadas

### Pi from Pie

It is perhaps surprising that it is possible to estimate the mathematical constant pi without resorting to any techniques or operations more sophisticated than counting, adding, and multiplication. This problem asks you to write two functions that estimate pi (3.14159...) by *dart-throwing*.

### Computing Pi from Pie: Background

Imagine a circle inscribed within a square that spans the area where  $-1 \leq x \leq 1$  and  $-1 \leq y \leq 1$ . The area of the inscribed circle, whose radius is 1.0 would be pi.

If you were to throw darts at random locations in the square, only some of them would hit the circle inscribed within it. The ratio

area of the circle / area of the square

can be estimated by the ratio

number of darts that hit the circle / total number of darts thrown

As the number of darts increases, the second ratio, above, gets closer and closer to the first ratio. Since three of the four quantities involved are known, they can be used to approximate the area of the circle—this in turn can be used to approximate pi.

### Designing Your Dart-Throwing...

To throw a dart, you will want to generate random x and y coordinates between -1.0 and 1.0. Be sure to include the line

```
import random
```

near the top of your file. When you do this, you will now be able to use the function

```
random.uniform(-1.0, 1.0)
```

That line will return a floating-point value that is in the range from -1.0 to 1.0 . For example, you will be able to write

```
x = random.uniform(-1.0, 1.0)
```

### Helper Function to Write: `throwDart()`

With this background in mind, many have found it helpful to write a helper function that

- Throws one "dart" at the square by generating getting a random x and a random y coordinate between -1 and 1
- Determines whether that dart is within the circle of radius 1 centered at the origin—you can use the `math.sqrt` function to check this, though you may note that it's not strictly necessary!
- returns `True` if the dart hits the circle and `False` if the dart misses the circle
- remember that the dart will *always* hit the square, by the way the throw is designed...

This helper function could be used for both of this problem's main functions: `forPi` and `whilePi`.

However you design your Monte Carlo simulation, you should be sure—as always—to include an explanatory docstring for each of your functions!

### Main Function to Write #1: `forPi(n)`

Your `forPi(n)` function will accept a positive integer `n`.

It should "throw" `n` darts at the square.

Each time a dart is thrown, the function should print:

- The number of darts thrown so far.
- The number of darts thrown so far that have *hit* the circle.
- The resulting estimate of pi.

**Return value**—be sure to do this!

The `forPi` function should return the *final resulting estimate of pi* after `n` throws.

Here is an example run to show how `forPi` should work:

- Your printing will vary because of the randomness...
- However, it should converge to the real value of pi as the number of darts, `n` gets larger

```
In [1]: forPi(10)
1 hits out of 1 throws so that pi is 4.0
2 hits out of 2 throws so that pi is 4.0
3 hits out of 3 throws so that pi is 4.0
4 hits out of 4 throws so that pi is 4.0
4 hits out of 5 throws so that pi is 3.2
5 hits out of 6 throws so that pi is 3.33333333333
6 hits out of 7 throws so that pi is 3.42857142857
6 hits out of 8 throws so that pi is 3.0
7 hits out of 9 throws so that pi is 3.11111111111
8 hits out of 10 throws so that pi is 3.2
```

```
Out[1]: 3.2
```

### Main Function to Write #2: whilePi(error)

Your `whilePi(error)` function will accept a positive floating-point value, `error`.

It should then proceed to throw darts at the dartboard (the square) until the *absolute difference* between the function's estimate of pi and the real value of pi is less than `error`.

Your `whilePi` function requires the actual, known value of pi in order to determine whether or not its estimate is within the error range! Although this would not be available for estimating a truly unknown constant, for this function you include the line

```
import math
```

in your code and then use the value of `math.pi` as the actual value of pi .

Similar to your `forPi` function, for each dart throw your `whilePi` function should print:

- The number of darts thrown so far
- The number of darts thrown so far that have *hit* the circle
- The resulting estimate of pi

after each dart throw it makes.

#### Return value—be sure to do this!

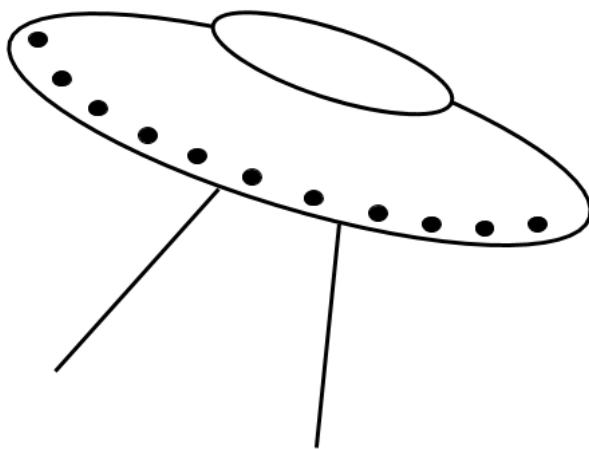
The `whilePi` function should return the *number of darts thrown* in order to reach the requested accuracy.

Here is an example run to show how `whilePi` works:

```
In [7]: whilePi(0.1)
1 hits out of 1 throws so that pi is 4.0
2 hits out of 2 throws so that pi is 4.0
3 hits out of 3 throws so that pi is 4.0
4 hits out of 4 throws so that pi is 4.0
5 hits out of 5 throws so that pi is 4.0
5 hits out of 6 throws so that pi is 3.33333333333
6 hits out of 7 throws so that pi is 3.42857142857
7 hits out of 8 throws so that pi is 3.5
7 hits out of 9 throws so that pi is 3.11111111111
```

```
Out[7]: 9
```

## CS for All



CSforAll Web > Chapter5 > TTSecurities

This website accompanies the textbook *CS For All* by Alvarado, Dodds, Kuenning, and Libeskind-Hadas

### TT Securities, Incorporated

#### Using Your Own Loops...

**Note for this problem:** Do *not* use the built-in functions `sum`, `min`, or `max` here—instead, as you work through the different menu options, write helper functions that handle the data appropriately. If you *really* want to use those functions, feel free to write versions of your own (also, we worked through examples in class, in fact—or will do so on Thursday)

#### Starter Code

This code may be a good place to start with this problem. We've named the primary function `main()` in order to set things up similar to how you'll write the larger version described below:

```
Example user-interaction looping program
```

```

def menu():
 """A function that simply prints the menu"""
 print()
 print("(0) Continue!")
 print("(1) Enter a new list")
 print("(2) Predict the next element")
 print("(9) Break! (quit)")
 print()

def predict(L):
 """Predict ignores its argument and returns
 what the next element should have been.
 """
 return 42

def find_min(L):
 """find min uses a loop to return the minimum of L.
 Argument L: a nonempty list of numbers.
 Return value: the smallest value in L.
 """
 result = L[0]
 for x in L:
 if x < result:
 result = x
 return result

def find_min_loc(L):
 """find min loc uses a loop to return the minimum of L
 and the location (index or day) of that minimum.
 Argument L: a nonempty list of numbers.
 Results: the smallest value in L, its location (index)
 """
 minval = L[0]
 minloc = 0

 for i in list(range(len(L))):
 if L[i] < minval:
 # a smaller one was found!
 minval = L[i]
 minloc = i

 return minval, minloc

def main():
 """The main user-interaction loop"""
 secret_value = 4.2

 L = [30, 10, 20]
 # an initial list

 while True:
 # the user-interaction loop
 print("\n\nThe list is", L)
 menu()
 choice = input("Choose an option: ")

 #
 # "Clean and check" the user's input

```

```

#
try:
 choice = int(choice)
make into an int!
except:
 print("I didn't understand your input! Continuing...")
 continue

run the appropriate menu option

#
if choice == 9:
We want to quit
 break
Leaves the while loop altogether

elif choice == 0:
We want to continue...
 continue
Goes back to the top of the while loop

elif choice == 1:
We want to enter a new list
 newL = input("Enter a new list: ")
enter _something_

#
"Clean and check" the user's input

#
try:
 newL = eval(newL)
eval runs Python's interpreter! Note: Danger!
 if type(newL) != type([]):
 print("That didn't seem like a list. Not changing L.")
 else:
 L = newL
Here, things were OK, so let's set our list, L
except:
 print("I didn't understand your input. Not changing L.")

elif choice == 2:
Predict and add the next element
 n = predict(L) # Get the next element from the predict function
 print("The next element is", n)
 print("Adding it to your list...")
 L = L + [n]
...and add it to the list

elif choice == 3:
Unannounced menu option!
 pass
this is the "nop" (do-nothing) statement in Python

elif choice == 4:
Unannounced menu option (slightly more interesting...)
 m = find_min(L)

```

```

print("The minimum value in L is", m)

elif choice == 5:
 # Another unannounced menu option (even more interesting...)
 minval, minloc = find_min_loc(L)
 print("The minimum value in L is", minval, "at day #", minloc)

else:
 print(choice, " ? That's not on the menu!")

print()
print("See you yesterday!")

```

## The TTS Problem ...

For this problem, you will implement a (text-based) menu of options for analyzing a list of stock prices (or any list of floating-point values). This provides an opportunity to use loops in a variety of ways, as well as experience handling user input.

The top-level function to write for this problem is called `main()`. Note that it takes no arguments. Instead, it should offer the user a menu with these choices:

- (0) Input a new list
- (1) Print the current list
- (2) Find the average price
- (3) Find the standard deviation
- (4) Find the minimum and its day
- (5) Find the maximum and its day
- (6) Your TT investment plan
- (9) Quit

Enter your choice:

Feel free to change the wording or text, but please keep the functionality of these choices intact. If you'd like to add additional menu options of your own design, please use different values for them (see extra credit, below).

Once the menu is presented, the program should wait for the user's input. (You may assume that the user will type an integer as input.) The function should then

- Print a warning message if the integer is not a valid menu option
- Quit if the user inputs 9
- Allow the user to input a new list of stock prices, if the user selects choice 0
- Print a table of days and prices, with labels, if the user selects choice 1
- Compute the appropriate statistics about the list for choices 2-6

For any option except 9, the function should reprompt the user with the menu after each choice.

Many of the pieces are straightforward, but here are a couple of pointers on two of them:

### formatting neatly...

This is optional, but if you'd like to print neatly-formatted columns, the following approach will help. Try pasting the examples at the Python prompt, just to get the hang of it). The formatting-string indicates *how* things are formatted, and the inputs to `.format` pass *what* values will get formatted that way.

Here, the printed values are in blue:

```
In [3]: print("{0: >4} : ${1: >6}".format("Day", "Price"))
Day : $ Price
```

```
In [4]: print("{0: >4} : ${1: >6}".format(3, 42))
3 : $ 42
```

```
In [5]: print("{0: >4} : $ {1: >6}".format(11, 27042))
11 : $ 27042
```

Briefly, the *formatting string*, namely "`{0: >4} : $ {1: >6}`" is saying three things:

- Print the format-argument #0 right-justified in a space with a width of 3"
- Then, print a space, a colon, a space, a dollar sign, and a space (all exactly as shown)
- Then, print the format-argument #1 right-justified in a space with a width of 4

Notice that the `.format(input0, input1)` after the formatting string provides the values needed!

Experiment with this to get it to look the way you'd like. And—printing without worrying about the format is totally fine, too!

- **The time-travel strategy:** For menu option 6, you will want to find the best day on which to buy and sell the stock in question in order to maximize the profit earned. However, the *sell day must be greater than or equal to the buy day*. You may want to adapt the example function `diff` from class to find this maximum profit.
- **Calculating the standard deviation:** Please use this formula to calculate the standard deviation of the stock. Note that  $L_{av}$  is the average (mean) value of the elements of the list  $L$ . Also, it's OK to assume that  $L$  will always be non-empty.

$$\sqrt{\frac{\sum_i (L[i] - L_{av})^2}{\text{len}(L)}}$$

If you're not familiar with the  $\Sigma$  notation, it means "add things up". Ask for help if you're not sure!

## Helper Functions

You **need to** write a helper function for each of the menu options 2-6. (The others already work or don't need an additional function.)

Also, especially for option #6, note that you can always return more than one value from a function. For example,

```
def f(x):
 return x, (x+4)
```

This function can be called as follows:

```
a, b = f(38)
```

In this case, `a` will receive the value 38 and `b` will receive 42.

Cool!

A bit further down this page, we have included an example run that illustrates one possible interface for your `main()` function.

### **Extra Credit: Creative Menu Options (up to +5 pts)**

If you'd like, you may add other menu options (under different numeric labels) that process the list in some other way of your design—it can be serious or not. Either way, your options will be graded on what they do and how smoothly they interact with the user.

Remember that the CS 5 graders get lonely—conversational programs are welcome!

### **Example Run of the Basic TTS Program**

Here is an example run—you do not need to follow this format exactly (though you may), but it's meant to show an example of each menu possibility:

```
In [1]: main()
(0) Input a new list
(1) Print the current list
(2) Find the average price
(3) Find the standard deviation
(4) Find the min and its day
(5) Find the max and its day
(6) Your TT investment plan
(9) Quit
```

```
Enter your choice: 0
Enter a new list of prices: [20, 10, 30]
(0) Input a new list
(1) Print the current list
(2) Find the average price
(3) Find the standard deviation
(4) Find the min and its day
(5) Find the max and its day
(6) Your TT investment plan
(9) Quit
```

```
Enter your choice: 1
```

| Day | Price |
|-----|-------|
| --- | ----  |
| 0   | 20.00 |
| 1   | 10.00 |
| 2   | 30.00 |

```
(0) Input a new list
(1) Print the current list
(2) Find the average price
(3) Find the standard deviation
(4) Find the min and its day
(5) Find the max and its day
(6) Your TT investment plan
(9) Quit
```

```
Enter your choice: 2
```

```
The average price is 20.0
```

```
(0) Input a new list
(1) Print the current list
(2) Find the average price
```

- (3) Find the standard deviation
- (4) Find the min and its day
- (5) Find the max and its day
- (6) Your TT investment plan
- (9) Quit

Enter your choice: 3

The st. deviation is 8.16496580928

- (0) Input a new list
- (1) Print the current list
- (2) Find the average price
- (3) Find the standard deviation
- (4) Find the min and its day
- (5) Find the max and its day
- (6) Your TT investment plan
- (9) Quit

Enter your choice: 4

The min is 10 on day 1

- (0) Input a new list
- (1) Print the current list
- (2) Find the average price
- (3) Find the standard deviation
- (4) Find the min and its day
- (5) Find the max and its day
- (6) Your TT investment plan
- (9) Quit

Enter your choice: 5

The max is 30 on day 2

- (0) Input a new list
- (1) Print the current list
- (2) Find the average price
- (3) Find the standard deviation
- (4) Find the min and its day
- (5) Find the max and its day
- (6) Your TT investment plan
- (9) Quit

Enter your choice: 6

Your TTS investment strategy is to

Buy on day 1 at price 10  
Sell on day 2 at price 30

For a total profit of 20

- (0) Input a new list
- (1) Print the current list
- (2) Find the average price
- (3) Find the standard deviation

- (4) Find the min and its day
- (5) Find the max and its day
- (6) Your TT investment plan
- (9) Quit

Enter your choice: 7

The choice 7 is not an option.

Try again

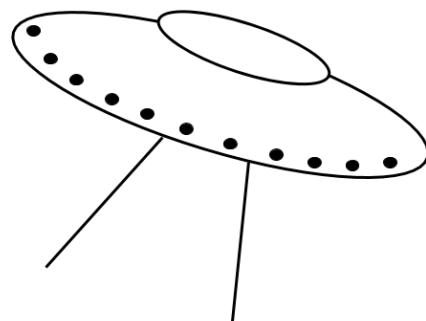
- (0) Input a new list
- (1) Print the current list
- (2) Find the average price
- (3) Find the standard deviation
- (4) Find the min and its day
- (5) Find the max and its day
- (6) Your TT investment plan
- (9) Quit

Enter your choice: 9

See you yesterday!

## CSforAll - 2DGameboard

### CS for All



CSforAll Web > Chapter5 > 2DGameboard

This website accompanies the textbook *CS For All* by Alvarado, Dodds, Kuennen, and Libeskind-Hadas

#### Tic Tac Toe + $N$ in a Row...

This problem asks you to write eight small functions—all very closely related—that operate on Python 2D data, that is, on lists-of-lists.

The application we have in mind is a *gameboard* in which your program will determine:

- Whether there are three-in-a-row of a single character (four functions), and
- Whether there are N-in-a-row of a single character (four analogous functions)

## Getting Started...

To begin, copy these two functions into a new file:

```
here is a function for printing 2D arrays
(lists-of-lists) of data

def print2d(A):
 """print2d prints a 2D array, A
 as rows and columns
 Argument: A, a 2D list of lists
 Result: None (no return value)
 """
 NR = len(A)
 NC = len(A[0])

 for r in range(NR): # NR == numrows
 for c in range(NC): # NC == numcols
 print(A[r][c], end=' ')
 print()

 return None # this is implied anyway,
 # when no return statement is present

some tests for print2d
A = [['X', ' ', 'O'], ['O', 'X', 'O']]
print("2-row, 3-col A is")
print2d(A)

A = [['X', 'O'], [' ', 'X'], ['O', 'O'], ['O', 'X']]
print("4-row, 2-col A is")
print2d(A)

create a 2D array from a 1D string
def createA(NR, NC, s):
 """Returns a 2D array with
 NR rows (numrows) and
 NC cols (numcols)
 using the data from s: across the
```

```

 first row, then the second, etc.
 We'll only test it with enough data!
"""
A = []
for r in range(NR):
 newrow = []
 for c in range(NC):
 newrow += [s[0]] # add that char
 s = s[1:] # get rid of that first char
 A += [newrow]
return A

a couple of tests for createA:
A = [['X', ' ', 'O'], ['O', 'X', 'O']]
newA = createA(2, 3, 'X OOXO')
assert newA == A
print("Is newA == A? Should be True:", newA == A)

A = [['X', 'O'], [' ', 'X'], ['O', 'O'], ['O', 'X']]
newA = createA(4, 2, 'XO XOOOX')
assert newA == A

```

Try out the file by running it at the ipython command line.

**Representation!** Notice that all of the 2D data in this problem will be lists-of-lists-of-single-characters:

- The overall structure, typically called `A`, is a list of rows
- Each row is a list of data elements
- Each data element is a single-character string
- In fact, we will stick with only three strings:
  - 'X', a capital X,
  - 'O', a capital O,
  - and ' ', the space character (this is *not* the empty string!).

### First Four Functions to Write: *three-in-a-row*

The first four functions to write check whether a three-in-a-row occurs

- in a specific direction (noted in the function name),
- for a specific character `ch`,
- at a specific starting location row and column: `r_start` and `c_start`, and
- within a given 2D data array, `A`.

Each one should return `False`

- if there is ***NO ROOM*** for a three-in-a-row starting at **r\_start** and **c\_start** (check for this first!), or
- if **r\_start** or **c\_start** is out of bounds of **A**, or
- (even if there is room in bounds), if there is NOT a three-in-a-row pattern within **A** entirely matching the element **ch** in the specified direction starting at the **r\_start** or **c\_start** location.

On the other hand, each function should return **True**

- only if there ***IS*** a three-in-a-row pattern within **A** entirely matching the element **ch** in the specified direction starting at the **r\_start** or **c\_start** location.

Hint

Consider the following pattern, designed for the "east" direction.

Here is one in-bounds check and then an example **for** loop:

```
for the 3-in-a-row-east function:

NR = len(A) # number of rows is len(A)
NC = len(A[0]) # number of cols is len(A[0])

if r_start >= NR:
 return False # out of bounds in rows

other out-of-bounds checks...
if c_start > NC - 3:
 return False # out of bounds in cols

are all of the data elements correct?
for i in range(3): # loop index i as needed
 if A[r_start][c_start+i] != ch: # check for mismatches
 return False # mismatch found--return False

return True # loop found no mismatches--return True
```

Note that for other directions:

- You will need other checks (to be sure things don't go out of bounds).
- Also, you'll need to adjust the loop for different directions
- The above example only checks for east-heading 3-in-a-rows.

## Functions to Write...

Here are the signatures of the four functions to write:

1. def inarow\_3east(ch, r\_start, c\_start, A):
  - This should start from `r_start` and `c_start` and check for three-in-a-row ***eastward*** of element `ch`, returning True or False, as appropriate
2. def inarow\_3south(ch, r\_start, c\_start, A):
  - This should start from `r_start` and `c_start` and check for three-in-a-row ***southward*** of element `ch`, returning True or False, as appropriate
3. def inarow\_3southeast(ch, r\_start, c\_start, A):
  - This should start from `r_start` and `c_start` and check for three-in-a-row ***southeastward*** of element `ch`, returning True or False, as appropriate
4. def inarow\_3northeast(ch, r\_start, c\_start, A):
  - This should start from `r_start` and `c_start` and check for three-in-a-row ***northeastward*** of element `ch`, returning True or False, as appropriate

These functions could be combined to handle any three-in-a-row across a game-board, e.g., for Tic-Tac-Toe. You'll create more general versions next...

Here are four tests for each one—*please do paste these into your file and make sure they work!*

```
tests of inarow_3east
A = createA(3, 4, 'XXOXXX000000')
#print2d(A)
assert inarow_3east('X', 0, 0, A) == False
assert inarow_3east('0', 2, 1, A) == True
assert inarow_3east('X', 2, 1, A) == False
assert inarow_3east('0', 2, 2, A) == False

tests of inarow_3south
A = createA(4, 4, 'XXOXXX0XX00 000X')
#print2d(A)
assert inarow_3south('X', 0, 0, A) == True
assert inarow_3south('0', 2, 2, A) == False
assert inarow_3south('X', 1, 3, A) == False
assert inarow_3south('0', 42, 42, A) == False

tests of inarow_3southeast
A = createA(4, 4, 'X00XXX0XX X0000X')
#print2d(A)
```

```

assert inarow_3southeast('X', 1, 1, A) == True
assert inarow_3southeast('X', 1, 0, A) == False
assert inarow_3southeast('O', 0, 1, A) == True
assert inarow_3southeast('X', 2, 2, A) == False

tests of inarow_3northeast
A = createA(4, 4, 'XOXXXXOXOX0000X')
#print2d(A)
assert inarow_3northeast('X', 2, 0, A) == True
assert inarow_3northeast('O', 3, 0, A) == True
assert inarow_3northeast('O', 3, 1, A) == False
assert inarow_3northeast('X', 3, 3, A) == False

```

### From 3 to N: *N-in-a-row*

Tic-tac-toe is a solved game! Let's handle *arbitrary* gameboards...

To that end, you'll generalize your three-in-a-row functions to N-in-a-row ones.

Each will have one more argument at the end, an integer *N*, which represents the number of identical elements (equal to *ch*) that need to be found to return *True*

- If it's out of bounds—or even in-bounds, but counting *N* would take it out of bounds—your functions should return *False*.
- Of course, your functions should also return *False* even if it's entirely in-bounds, but there aren't *N*-in-a-row!

Here are the signatures of the four N-in-a-row functions to write:

1. `def inarow_Neast(ch, r_start, c_start, A, N):`
  - This should start from *r\_start* and *c\_start* and check for N-in-a-row *eastward* of element *ch*, returning *True* or *False*, as appropriate.
2. `def inarow_Nsouth(ch, r_start, c_start, A, N):`
  - This should start from *r\_start* and *c\_start* and check for N-in-a-row *southward* of element *ch*, returning *True* or *False*, as appropriate.
3. `def inarow_Nsoutheast(ch, r_start, c_start, A, N):`
  - This should start from *r\_start* and *c\_start* and check for N-in-a-row *southeastward* of element *ch*, returning *True* or *False*, as appropriate.
4. `def inarow_Nnortheast(ch, r_start, c_start, A, N):`
  - This should start from *r\_start* and *c\_start* and check for N-in-a-row *northeastward* of element *ch*, returning *True* or *False*, as appropriate.

Again, here are four tests for each one—*please do paste these into your file and make sure they work!*

```
tests of inarow_Neast
A = createA(5, 5, 'XXOXXX000000XXXX XXX00000')
print2d(A)
assert inarow_Neast('0', 1, 1, A, 4) == True
assert inarow_Neast('0', 1, 3, A, 2) == True
assert inarow_Neast('X', 3, 2, A, 4) == False
assert inarow_Neast('0', 4, 0, A, 5) == True

tests of inarow_Nsouth
A = createA(5, 5, 'XXOXXX000000XXXX0XXX000X0')
print2d(A)
assert inarow_Nsouth('X', 0, 0, A, 5) == False
assert inarow_Nsouth('0', 1, 1, A, 4) == True
assert inarow_Nsouth('0', 0, 1, A, 6) == False
assert inarow_Nsouth('X', 4, 3, A, 1) == True

tests of inarow_Nsoutheast
A = createA(5, 5, 'XOO XXXOX000XXXX0XXX000XX')
print2d(A)
assert inarow_Nsoutheast('X', 1, 1, A, 4) == True
assert inarow_Nsoutheast('0', 0, 1, A, 3) == False
assert inarow_Nsoutheast('0', 0, 1, A, 2) == True
assert inarow_Nsoutheast('X', 3, 0, A, 2) == False

tests of inarow_Nnortheast
A = createA(5, 5, 'XOO XXXOX000X0XXX0XXX000XX')
print2d(A)
assert inarow_Nnortheast('X', 4, 0, A, 5) == True
assert inarow_Nnortheast('0', 4, 1, A, 4) == True
assert inarow_Nnortheast('0', 2, 0, A, 2) == False
assert inarow_Nnortheast('X', 0, 3, A, 1) == False
```

You'll be using these `inarow` functions over the next two weeks as you implement a version of Connect Four!

Website design by Madeleine Masser-Frye and Allen Wu

# CSforAll - ASCIIArt

## CS for All

CSforAll Web > Chapter5 > ASCIIArt

This website accompanies the textbook *CS For All* by Alvarado, Dodds, Kuenning, and Libeskind-Hadas

### ASCII Art

In this assignment you will revisit a classic art form: ASCII art!

**Important limitation!** For these problems, you should **not** use Python's string-multiplication or string-addition operators. Because our goal is to use loop constructs, use loops to achieve the repetition that those operators might otherwise provide. There is one exception, however—you **may** use string-multiplication with the space character ' '. That is, you can create any number of consecutive spaces with constructs like

```
' '*5
```

### ASCII Art Problems

The goal of this problem is to further solidify your reasoning skills with loops and nested loops. For many of the problems (especially the striped diamond), you will have to think carefully about the value of your loop-control variable as your loop or loops execute. "Debugging by random permutation"—that is, arbitrarily changing the values of your loop conditions or variables—will lead to much frustration. The path to success on this assignment is to reason carefully about your loops!

There are five problems here; the first two are worth 1 point each and the other three are worth 2 points each.

The `printCrazyStripedDiamond` is worth a possible +5 points beyond the others. It is, however, *crazy...*

## **printRect**

Write a function named `printRect` that takes three arguments, `width`, `height`, and `symbol`, and prints a `width`-by-`height` rectangle of `symbols` on the screen.

```
In [1]: printRect(4, 6, '%')
% % %
% % %
% % %
% % %
% % %
% % %
```

### Hint

If you look back at the slides from the first day of nested loops, you'll see that we did precisely this problem! The only differences are that

- The width is a variable, instead of a constant
- The height is a variable, instead of a constant
- The character printed is a variable, instead of a constant

## **printTriangle**

Create a function `printTriangle` that takes three arguments: `width`, `symbol`, and `rightSideUp`, and prints a triangle of symbols on the screen. `width` is a number that determines the width of the base of the triangle and `rightSideUp` is a Boolean that determines whether the triangle is printed right-side-up (`True`) or upside-down (`False`).

```
In [1]: printTriangle(3, '@', True)
@
@
@

In [2]: printTriangle(3, '@', False)
@ @ @
```

```
© ©
©
```

### printBumps

Now, use your `printTriangle` function to write a function called `printBumps(num, symbol1, symbol2)` that will print the specified number of two-symbol "bumps", where each bump is larger than the last, as in the following example:

```
In [1]: printBumps(4, '%', '#')
%

%
% %

%
% %
% % %

%
% %
% % %
% % % %

#
```

### printDiamond

For these "diamond" functions, you **may** use string multiplication, but only for strings of blank spaces, such as `' '*n` or the like. Each visible character should be printed separately, just as in the functions earlier in this problem. Also, you don't *have* to use the string `*` operator for strings of spaces, either.

Write a function called `printDiamond(width, symbol)` that prints a diamond of `symbol` whose maximum width is determined by `width`.

```
In [1]: printDiamond(3, '&')
&
& &
& & &
& &
&
```

### printStripedDiamond

Next, write a function called `printStripedDiamond(width, sym1, sym2)` that prints a "striped diamond" of `sym1` and `sym2`.

For example:

```
In [1]: printStripedDiamond(7, '.', '%')
.
. %
. % .
. % . %
. % . % .
. % . % . %
% . % . %
. % . %
% . %
.
.
```

### printCrazyStripedDiamond

Finally, (and this is worth +5 points *beyond* the +8 for the previous figures) write a function called `printCrazyStripedDiamond(width, sym1, sym2, sym1Width, sym2Width)` that prints a "striped diamond" of `sym1` and `sym2` where the stripes can have varied widths. `sym1Width` determines the width of the stripe made of symbol 1 and `sym2Width` determines the width of the stripe made of symbol 2.

For example:

```
In [1]: printCrazyStripedDiamond(7, '.', '%', 2, 1)
.
```

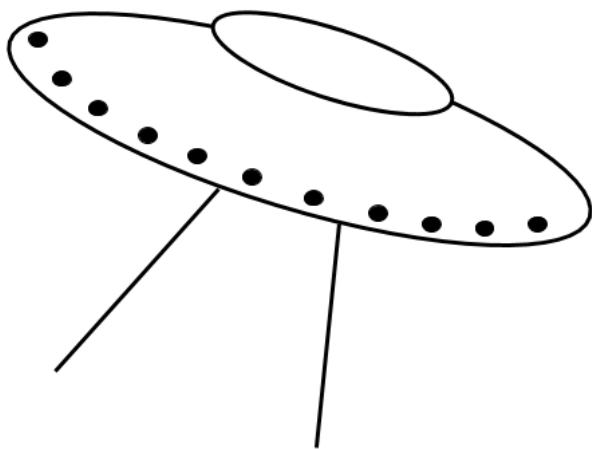
. . . %  
. . % .  
. . % . .  
. . % . . %  
. . % . . % .  
. % . . % .  
% . . % .  
. . % .  
. % .  
% .

.

Website design by Madeleine Masser-Frye and Allen Wu

# CSforAll - GameofLife

## CS for All



CSforAll Web > Chapter5 > GameofLife

This website accompanies the textbook *CS For All* by Alvarado, Dodds, Kuenning, and Libeskind-Hadas

### The Game of Life

#### John Conway's *Game of Life*

The Game of Life is a *cellular automaton* invented by John Conway, a mathematician from Cambridge. The Game of Life is not so much a "game" in the traditional sense, but rather a process that transitions over time according to a few simple rules. The process is set up as a grid of cells, each of which is "alive" or "dead" at a given point in time. At each time step, the cells live or die according to the following rules:

1. A cell that has fewer than two live neighbors dies (because of isolation)
2. A cell that has more than 3 live neighbors dies (because of overcrowding)
3. A cell that is dead and has exactly 3 live neighbors comes to life
4. All other cells maintain their state

Although these rules seem simple, they give rise to complex and interesting patterns. For more information and a number of interesting patterns see [http://en.wikipedia.org/wiki/Conway%27s\\_Game\\_of\\_Life](http://en.wikipedia.org/wiki/Conway%27s_Game_of_Life)

## Game\_of\_Life.

In this problem, you will implement a Python program to run the Game of Life.

### **Thinking about life...**

As always, it is important to break the problem down into pieces and develop the program in stages so that others can understand the code and so that you can ensure that each piece is correct before building on top of it. We will break this problem down into the following steps:

- Creating a 2D array of cells
- Displaying the board (in various colors) and updating it with new data
- Allowing the user to change the state of the cells
- Implementing the update rules for the "Game of Life"
- (Optionally) Running and stopping the simulation

Before you start, you need to develop a scheme for keeping track of your data. Basically, the data you need to maintain are the states of all of the cells in the board. To do this, you should keep track of this data in a 2D array of integer values, where 0 represents an empty (off) cell and 1 represents a live (on) cell.

### **Files to start with...**

Start by downloading the .zip file from the following link:

[gameoflife.zip](#)

It will be easiest to place this zip file on the desktop and extract it there. There is some support code for graphically displaying your life generations, but that will be the final piece of the problem. First, you'll implement the basic functionality for creating 2D arrays of data, changing them, and having them evolve according to the rules of Life.

### **Step 1: Creating a 2D "board" of cells: `createOneRow` and `createBoard`**

In the `gameoflife.py` file, you will see this example function:

```
def createOneRow(width):
 """Returns one row of zeros of width "width".
 You might use this in your createBoard(width, height) function."""
 row = []
 for col in range(width):
 row += [0]
 return row
```

This function offers a starting point for creating *one-dimensional* lists—but the same idea applies for building arbitrarily deep nested list structures.

**`createBoard(width, height)`** Building on this example, write a function named `createBoard(width, height)` that creates and returns a new 2D list of `height` rows and `width` columns in which all of the data elements are 0 (no graphics quite yet, just a Python list!).

#### ***Avoid re-implementing the `createOneRow` function!***

Rather, use `createOneRow` inside your `createBoard` in the same way that 0 is used to accumulate individual elements in `createOneRow`. Here is a template—copy and paste this and then fill in the parts you'll need to complete it:

```
def createBoard(width, height):
 """Returns a 2D array with "height" rows and "width" columns."""
 A = []
 for row in range(height):
 A += [SOMETHING]
 # use the above function so that SOMETHING is one row!!
 return A
```

That's all you'll need! Again, the idea is to follow the example of `createOneRow`—but instead of adding a 0 each time, the function would add a whole row of 0s, namely the result from `createOneRow`!

Test out your `createBoard` function!

```
In [1]: A = createBoard(5, 3)
In [2]: A
Out[2]: [[0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0]]
```

### Printing your 2D board of cells: `printBoard`

You no doubt noticed that when Python prints a 2D list, it blithely ignores its 2D structure and flattens it out into one line (perhaps wrapping, if needed). To print your board in 2D using ASCII (we will use graphics once it's working), copy this function into your file.

*Note: there's one line missing!* Perhaps you already see it...

```
def printBoard(A):
 """This function prints the 2D list-of-lists A."""
 for row in A:
 # row is the whole row
 for col in row:
 # col is the individual element
 print(col, end="")
 # print that element
```

Next, try out `printBoard`. As is, it's *not quite right*:

```
In [1]: A = createBoard(5, 3)
In [2]: printBoard(A)
0000000000000000
```

**Your task Add one line** so that `printBoard` shows our *2D* array as two-dimensional!

The line is `print()`—an empty print statement. *Where should it go?!*

Be sure to test your improved `printBoard`:

```
In [1]: A = createBoard(5, 3)
In [2]: printBoard(A)
00000
00000
00000
```

### Adding patterns to 2D arrays: `diagonalize`

To get used to looping over 2D arrays of data, copy this function named `diagonalize(A)` into your file:

```
def diagonalize(width, height):
 """Creates an empty board and then modifies it
 so that it has a diagonal strip of "on" cells.
 But do that only in the *interior* of the 2D array.
 """

```

```

A = createBoard(width, height)

for row in range(1, height - 1):
 for col in range(1, width - 1):
 if row == col:
 A[row][col] = 1
 else:
 A[row][col] = 0

return A

```

This function, `diagonalize`, accepts a desired width and height. It then creates an array `A` and sets `A`'s data so that its elements are all 0 *except for the **interior** diagonal* where `row == col`.

Try displaying the result with these commands:

```
In [1]: A = diagonalize(7, 6)
```

```

In [2]: A
Out[2]:
[[0, 0, 0, 0, 0, 0, 0],
 [0, 1, 0, 0, 0, 0, 0],
 [0, 0, 1, 0, 0, 0, 0],
 [0, 0, 0, 1, 0, 0, 0],
 [0, 0, 0, 0, 1, 0, 0],
 [0, 0, 0, 0, 0, 1, 0],
 [0, 0, 0, 0, 0, 0, 0]]

```

```

In [3]: printBoard(A)
0000000
0100000
0010000
0001000
0000100
0000000

```

Take a moment to note the direction the diagonal is running—that indicates which way the *rows* of the board are being displayed: top-to-bottom, in this case.

Note that this code uses `range(1, height - 1)` and `range(1, width - 1)` in order to avoid changing the edges.

- We will continue this pattern throughout the problem, since the Game of Life does not handle edge cells (they don't have 8 neighbors).

Also, this example shows that the `height` and `width` do not have to be the same—though it's certainly OK if they are.

### More patterns: `innerCells(w, h)`

Based on the example of `diagonalize`, write a variation named `innerCells(w, h)` that returns a 2D array that has **all** live cells—with a value of 1—*except* for a one-cell-wide border of empty cells (with the value of 0) around the edge of the 2D array.

For example, you might try

```
In [1]: A = innerCells(5, 5)
```

```

In [2]: printBoard(A)
remember: up-arrow for this!
00000
01110
01110
01110
00000

```

This is only a small variation of `diagonalize`!

### More patterns: `randomCells(w, h)`

Next, create a function named `randomCells(w, h)`, which returns an array of randomly-assigned 1's and 0's *except* that the outer edge of the array is still completely empty (all 0's) as in the case of `innerCells`.

Here is one of our runs:

```
In [1]: A = randomCells(10, 10)
```

```
In [2]: printBoard(A)
up-arrow!
0000000000
0100000110
0001111100
0101011110
00001111000
0010101010
0010111010
0011010110
0110001000
0000000000
```

You might recall that `random.choice([0, 1])` will return either a 0 or a 1. You will need to import `random` to use it!

### More building blocks: `copy(A)`

Each of the updating functions so far creates a new set of cells without regard to an old "generation" that it might depend on. Conway's Game of Life, on the other hand, follows a set of cells by changing one generation into the next.

To see why `copy(A)` is a crucial helper function for this process, try the following commands:

```
In [1]: A = innerCells(5, 5)
create a 5x5 innerCells board
```

```
In [2]: printBoard(A)
up-arrow
00000
01110
01110
01110
00000
```

```
In [3]: newA = A
creates a false ("shallow") copy
```

```
In [4]: printBoard(newA)
up-arrow and edit... LOOKS the same...
00000
01110
01110
01110
00000
```

```
In [5]: A[2][2] = 5
set old A's center to 5
```

```
In [6]: printBoard(A)
there's a 5 in the middle...
00000
01110
01510
01110
00000
```

```
In [7]: printBoard(newA)
note that newA changed, too--Aargh!
00000
01110
01510
01110
00000
```

Here we have made a shallow "copy" of A, naming that shallow copy newA.

However, newA is simply a copy of the *reference* to the original data in A!

As a result, when A's data changes, so does newA's data, even though we never touched newA's data!

The above example shows *shallow* copying: the copying of a *reference* to data, rather than making a full copy of all of the data.

Making a full copy of all of the data is called *deep* copying.

**Writing copy(A), a "deep" copier...** Starting with this code:

```
def copy(A):
 """Returns a DEEP copy of the 2D array A."""
 height = len(A)
 width = len(A[0])
 newA = createBoard(width, height)

 for row in range(1, height - 1):
 for col in range(1, width - 1):

 # what single line should be here to copy each

 # element of A into the corresponding element of newA?

 return newA
```

finish its implementation to create a function named copy(A), which will make a *deep* copy of the 2D array A.

As usual, don't worry about the outside edges: they'll always stay 0. The loops only touch the non-edge cells.

Thus, copy will accept a 2D array A as its input. And, copy will return a brand-new 2D array of data that has the same pattern as the original array.

You can make sure your copy function is working properly with this example:

```
In [1]: A = innerCells(5, 5)
create a 5x5 innerCells board
```

```
In [2]: printBoard(A)
up-arrow
00000
01110
```

```
01110
01110
00000
```

```
In [3]: newA = copy(A)
```

```
In [4]: printBoard(newA)
00000
01110
01110
01110
00000
```

```
In [5]: A[2][2] = 5
```

```
In [6]: printBoard(A)
changes it!
00000
01110
01510
01110
00000
```

```
In [7]: printBoard(newA)
UNchanged!
00000
01110
01110
01110
00000
```

This time, `newA` has *not* changed even though `A` did: it's a true, "deep" copy.

### Aside: Python's built-in `deepcopy`

Your `copy` function returns a deep copy of its input. This is useful enough to be part of Python. To use the library call, you can run

```
from copy import deepcopy
```

```
newA = deepcopy(A)
```

Now you've seen how it's implemented: element-by-element! Feel free to use either one.

### More building blocks: `innerReverse(A)`

Copying is a straightforward way that a new "generation" of array elements might depend on a previous one.

Next you'll write a function that *changes* one generation of cells into a new generation.

To that end, write a function `innerReverse(A)` that takes an old 2D array `A` (an old "generation") and then creates a new generation `newA` of the same shape and size, using `createBoard` or `copy`.

However, with `innerReverse` the new generation should be the "opposite" of `A`'s cells everywhere except on the outer edge.

In the same spirit as `innerCells`, you should make sure that the new generation's outer edge of cells is always all 0.

However, for inner cells—those not on the edge—where `A[row][col]` is a 1, the new array's value will be a 0—and vice versa.

Hint

Just paste your `copy(A)` function, and then add an `if/else` appropriately!

Try out your `innerReverse` function by displaying an example. This one uses `randomCells`:

In [1]: `A = randomCells(8, 8)`

In [2]: `printBoard(A)`

```
00000000
01011010
00110010
00000010
01111110
00101010
01111010
00000000
```

In [3]: `A2 = innerReverse(A)`

In [4]: `printBoard(A2)`

```
00000000
00100100
01001100
01111100
00000000
01010100
00000100
00000000
```

*Aside:* You might point out that it would be possible to simply change the old argument `A`, rather than create and return new data—this is true for simply reversing the pattern of array elements, but it is *not* true when implementing the rules of Conway's Game of Life—there, changing cells without copying would change the number of neighbors of other cells!

## John Conway's Game of Life

For this step, you'll create two functions:

- `countNeighbors(row, col, A)` and
- `next_life_generation(A)`

### Helper function: `countNeighbors(row, col, A)`

It will help to write a helper function, `countNeighbors(row, col, A)`, which should return *the number of live neighbors* for a cell in the board `A` at a particular `row` and `col`.

There are two basic approaches to `countNeighbors`:

1. Write two *small* for-loops to examine the nine cells centered **at `A[row][col]`**.
  - You'll naturally add the center. This is OK—just be sure to *subtract* it before returning!
2. **OR**, write eight `if` statements to check all eight possible neighbors...
  - Note that you want ALL eight to run. So, use `if` for all of them—don't use `elif`: it's too "exclusive" 😊

**Checking `countNeighbors`** Define or paste this 5x5 array `A` and then check a few cells' neighbor counts:

In [1]: `run gameoflife`

In [2]: `A = [[0, 0, 0, 0, 0], [0, 0, 1, 0, 0],`

```
[0, 0, 1, 0, 0],
[0, 0, 1, 0, 0],
[0, 0, 0, 0, 0]]
```

In [3]: printBoard(A)

```
00000
00100
00100
00100
00000
```

In [4]: countNeighbors(2, 1, A)

Out[4]: 3

# Correct! There are 3 live neighbors here

In [5]: countNeighbors(2, 2, A)

Out[5]: 2

# Be sure not to count the cell itself!

In [6]: countNeighbors(0, 1, A)

Out[6]: 1

### Main function: next\_life\_generation(A)

Finally, you'll write `next_life_generation`, which implements the rules of Life.

*This will be most similar to `innerReverse`, so you might use that as a template.*

Here is a starting signature for `next_life_generation`:

```
def next_life_generation(A):
 """Makes a copy of A and then advances one
 generation of Conway's Game of Life within
 the *inner cells* of that copy.
 The outer edge always stays at 0.
 """
```

This `next_life_generation` function should accept a 2D array `A`, representing the "old" generation of cells, and it should return a *new generation* of cells, each either 0 or 1, based on John Conway's rules for the *Game of Life*:

1. Be sure to create a `newA`, the new generation, which is of the same size as `A`, the old generation.
2. All edge cells stay zero (0), as usual (but see the extra challenges, below).
3. A cell that has fewer than two live neighbors dies (because of loneliness).
4. A cell that has more than 3 live neighbors dies (because of overcrowding).
5. A cell that is dead and has exactly 3 live neighbors comes to life.
6. All other cells maintain their existing state (are given the value of the corresponding old cell).

For concreteness, let's call the new generation of cells you're returning `newA` in order to contrast it with `A`.

**As suggested in `innerReverse`, always keep all of the outer-edge cells empty!** This is simply a matter of limiting your loops to an appropriate range. However, it greatly simplifies the four update rules, above, because it means that you will only update the interior cells, each of which *has a full set of eight neighbors without going out of bounds*.

**Warnings/hints:** There are a few things to keep in mind:

- Only count neighbors within the old generation `A`. Change only the new generation, `newA`.
- Be sure to set *every* value of `newA` (the new data), whether or not it differs from `A`.
- A cell is **NOT** a neighbor of itself.
- A 2x2 square of cells is statically stable (if isolated)—you might try it on a small grid for testing purposes
- A 3x1 line of cells oscillates with period 2 (if isolated)—also a good pattern to test.

Here is a set of tests to try based on the last suggestion in the list above:

```
In [1]: A = [[0, 0, 0, 0, 0], [0, 0, 1, 0, 0], [0, 0, 1, 0, 0], [0, 0, 1, 0, 0], [0, 0, 0, 0, 0]]
```

```
vertical bar
```

```
In [2]: printBoard(A)
```

```
00000
00100
00100
00100
00000
```

```
In [3]: A2 = next_life_generation(A)
```

```
In [4]: printBoard(A2)
```

```
00000
00000
01110
00000
00000
```

```
In [5]: A3 = next_life_generation(A2)
```

```
In [6]: printBoard(A3)
```

```
00000
00100
00100
00100
00000
```

and so on.

Once your Game of Life is working, look for some of the other common patterns, e.g., other statically stable forms ("rocks"), as well as oscillators ("plants") and others that will move across the screen, known as gliders ("animals" or "birds").

## Graphical Life!

Once your `next_life_generation` code is working you can (and should!) watch it in action with the provided simulator. To boot it up, use the following command:

```
run gameoflife_graphics.py
```

This file is counting on these things:

- You have `gameoflife.py` in the same directory.
- You have written `next_life_generation`, `randomCells`, and `createBoard` correctly in that file (named `gameoflife.py`).

## Using the simulator

1. To start it, run `start()`. You will be asked whether you want to start with a blank or random board.
2. Then, a window will pop up with your board. Click in the grey area to turn squares on or off. The green border is interpreted as 0s and cannot be changed.
3. To start the simulation, click anywhere outside the board (but within the window) and it will run until you click outside the board again.
4. DO NOT CLOSE THE WINDOW WHILE IT RUNS! If you do this, the window will still try to pop up because it's still running and your best hope is to kill python.
5. Once it has stopped running you can continue to modify the board and/or run it again.
6. To exit, simply x out of the window. To run again, use `start()`.

*Features!* There are a few functions you can use to adjust the simulator to your liking.

- `setColor()`: Change the color scheme using a built in one from matplotlib or entering rgb tuples
  - For built in themes, the far left hue becomes the blank cells, the middle hue is active cells, and the far right hue is the border

- `stepsPerClick()`: Use this function if you'd like the program to run a finite number of steps each time you click outside the board. It will ask how many per click, and you can run the function again to turn that mode off.
- `timePerStep(float)`: The argument here is the number of seconds you'd like the board to display each step before going to the next generation. Going much lower than 0.5 sec is limited by processing speed (depending on board size).
- `sideLength(int)`: The argument is the number of cells wide/tall you'd like your square grid to be.

### Variations on the Game of Life [Optional]

That said, there's always more!

If you'd like, you might try implementing a variation on the Game of Life:

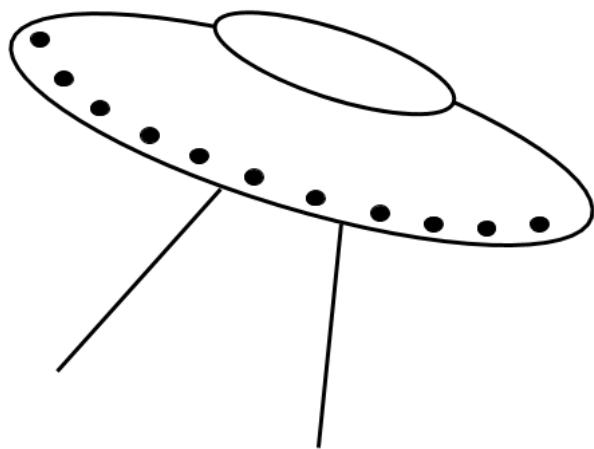
**Variation One: the doughnut** For this variation, remove the margin of empty cells around the board. In this case, the board should have the topology of a doughnut—so that the left and right edges of the board become neighbors, as do the top and bottom edges. This makes neighbor-counting and avoiding out-of-bounds errors trickier. Yet with a single function that "looks up" the actual location of any given `row` and `col` coordinates, it's not too bad...

**Variation Two: alien life** Life is considered a 23/3 cellular automaton, because cells survive with 2 or 3 neighbors (the two digits before the slash) and they are born with 3 neighbors (the digit after the slash). Many (perhaps all) of the other survival/birth possibilities have been studied, as well. Some even have names: for example, 1358/357 is called "Amoeba" and 1/1 is called "Gnarl." For this variation on life, choose a different set of survival/birth rules, perhaps from this reference of them and implement a key that switches between John Conway's Game of Life and your chosen set of rules (with another key to go back...).

**Variation Three: More cells!** You can also create rules that allow for cells of more than two types. You may set colors for up to 8 types of cells, as well, numbered 0 through 7.

## CSforAll - Mandelbrot

### CS for All



CSforAll Web > Chapter5 > Mandelbrot

This website accompanies the textbook *CS For All* by Alvarado, Dodds, Kuenning, and Libeskind-Hadas

#### Creating the Mandelbrot Set

##### The Mandelbrot Set

In this problem you will build a program to visualize and explore the points in and near the Mandelbot Set. In doing so, you will have the chance to:

- Use loops and nested loops to solve *complex* problems (quite literally!)
- Develop a program using *incremental design*, i.e., by starting with a simple task and gradually adding levels of complexity
- Connect with mathematics and other disciplines that use fractal modeling

## Introduction to for Loops!

Because you will create images in this lab, there is some additional starter code—grab and unzip **mandelbrot.zip**.

To build some intuition about loops, first write two short functions in your file:

- Write a function named `mult(c, n)` that returns the product `c` times `n`, but without multiplication. Instead, it should start a value (named `result`) at 0 and repeatedly *add* the value of `c` into that `result`. It should use a `for` loop to make sure that it adds `c` the correct number of times. After the loop finishes, it should return the result, both conceptually and literally.

The value of `n` will be a positive integer. To get you started, here is a snippet of the function that initializes the value of `result` to 0 and builds the loop itself:

```
def mult(c, n):
 """Mult uses only a loop and addition
 to multiply c by the positive integer n
 """
 result = 0
 for i in range(n):

 # update the value of result here in the loop

 assert mult(3, 5) == 15
```

Here are a couple of cases to try:

```
In [1]: mult(6, 7)
Out[1]: 42
```

```
In [2]: mult(1.5, 28)
Out[2]: 42.0
```

- The next function will build the basic Mandelbrot update step, which is  $z = z^{**2} + c$  for some constant `c`.

To that end, write a function named `update(c, n)` that starts a new value, `z`, at zero, and then repeatedly updates the value of `z` using the assignment statement  $z = z^{**2} + c$  for a total of `n` times. In the end, the function should return the final value of `z`. The value of `n` will be a positive integer. Here are the `def` line and docstring to get you started:

```
def update(c, n):
 """Update starts with z = 0 and runs z = z**2 + c
 for a total of n times. It returns the final z.
 """

```

Here are a couple of cases to try:

```
In [1]: update(1, 3)
Out[1]: 5
```

```
In [2]: update(-1, 3)
Out[2]: -1
```

```
In [3]: update(1, 10)
Out[3]: a really big number!
```

```
In [4]: update(-1, 10)
Out[4]: 0
```

You'll use these ideas (through a variant of the `update` function) in building the Mandelbrot set, next:

## Introduction to the Mandelbrot Set

The *Mandelbrot set* is a set of points in the complex plane that share an interesting property that is best explained through the following process:

- Choose a complex number  $c$ .
- With this  $c$  in mind, start with  $z_0 = 0$ .
- Then repeatedly iterate as follows:
- $z_{n+1} = z_n^2 + c$

The Mandelbrot set is the collection of all complex numbers  $c$  such that this process does *not* diverge to infinity as  $n$  gets large. In other words, for some  $c$ , if  $z_n$  diverges to infinity (becomes unreasonably large) then  $c$  is *not* in the set; otherwise it is.

There are other, equivalent definitions of the Mandelbrot set. For example, the Mandelbrot set consists of those points in the complex plane for which the associated *Julia set* is connected. Admittedly, this requires defining Julia sets, which we won't do here...

The Mandelbrot set is a *fractal*, meaning that its boundary is so complex that it can not be well-approximated by one-dimensional line segments, regardless of how closely one zooms in on it. There are many available references.

## The `inMSet` Function

The next task is to write a function named `inMSet(c, n)` that accepts a complex number  $c$  and an integer  $n$ .

This function will return a Boolean:

- True if the complex number  $c$  is in the Mandelbrot set and
- False otherwise.

First, we will introduce Python's built-in support for complex numbers.

## Python and Complex Numbers

In Python a complex number is represented in terms of its real part  $x$  and its imaginary part  $y$ . The mathematical notation would be  $x+yi$ , but in Python the imaginary unit is typed as `1.0j` or `1j`, so that

```
c = x + y*1j
```

would assign the variable  $c$  to the complex number with real part  $x$  and imaginary part  $y$ .

Unfortunately, `x + yj` does not work, because Python thinks you're using a variable named `yj`.

Also, the value `1 + j` is not a complex number: Python assumes you mean a variable named `j` unless there is an `int` or a `float` directly in front of it. Use `1 + 1j` instead.

**Side note:** Mathematicians use  $i$ ; engineers use  $j$  because in engineering,  $i$  refers to electrical current. Python follows the engineering convention, which can be a source of *infinite* confusion!

**Try it out** Just to get familiar with complex numbers, at the Python prompt try

```
In [1]: c = 3 + 4j
```

```
In [2]: c
```

```
Out[2]: (3+4j)
```

```
In [3]: abs(c)
```

```
Out[3]: 5.0
```

```
In [4]: c**2
```

Out[4]: (-7+24j)

Python is happy to use the power operator (`**`) and other operators with complex numbers. However, note that you cannot compare complex numbers directly—they are 2d points, so there's no "greater than"! Thus, you cannot write `c > 2` for a complex `c` (it will `TypeError`).

You CAN compare the magnitude, however: `abs(c) > 2`. Note that the built-in `abs` function returns the magnitude of a complex number.

### Thinking About the `inMSet` Function:

To determine whether or not a number  $c$  is in the Mandelbrot set, you will

- Start with  $z_0 = 0 + 0j$  and then
- repeatedly iterate  $z_{n+1} = z_n^2 + c$

to see if this sequence of  $z_0, z_1, z_2$ , etc. stays bounded.

To put it another way, we need to know whether or not the magnitude of these  $z_k$  go off toward infinity.

Truly determining whether or not this sequence goes off to infinity isn't feasible. To make a reasonable guess, we will have to decide on two things:

- The number of times we are willing to wait for the  $z_{n+1} = z_n^2 + c$  process to run
- A value that will represent "infinity"

We will run the update process  $n$  times. The  $n$  is the second argument to the function `inMSet(c, n)`. This is a value you will experiment with, but 25 is a good starting point.

The value for infinity can be surprisingly low! It has been shown that if the absolute value of a complex number  $z$  ever gets larger than 2 during the update process, then the sequence will *definitely* diverge to infinity.

There is no equivalent rule that tells us that the sequence definitely *does not* diverge, but it is *very likely* it will stay bounded if `abs(z)` does not exceed 2 after a reasonable number of iterations, and  $n$  is that "reasonable" number, starting at 25.

### Writing `inMSet`

You should **copy** your `update` function and change its name to `inMSet`.

**It's definitely better** to copy and change that old function—*do not call `update` directly*.

To get you started, here is the first line and a docstring for `inMSet`:

```
def inMSet(c, n):
 """inMSet accepts
 c for the update step of z = z**2+c
 n, the maximum number of times to run that step
 Then, it returns
 False as soon as abs(z) gets larger than 2
 True if abs(z) never gets larger than 2 (for n iterations)
 """
```

The `inMSet` function should return `False` if the sequence  $z_{n+1} = z_n^2 + c$  ever yields a  $z$  value whose magnitude is greater than 2. It returns `True` otherwise.

Note that you will **not** need different variables for  $z_0, z_1, z_2$ , and so on. Rather, you'll use a single variable `z`. You'll update the value of `z` within a loop, just as in `update`.

**Make sure** that you are using `return False` somewhere *inside* your loop. You will want to `return True` **after** the loop has finished all of its iterations!

Check your `inMSet` function by copying and pasting these examples:

```
In [1]: c = 0 + 0j
this one is in the set
```

```
In [2]: inMSet(c, 25)
Out[2]: True
```

```
In [3]: c = 3 + 4j
this one is NOT in the set
WARNING: this one might freeze or crash Python you're not returning False
UNLESS your function returns False *as soon as* the magnitude is larger than 2
that is, you need to return False _inside_ the loop (inside your if or your else, whichever's appropriate)!
```

```
In [4]: inMSet(c, 25)
False
```

```
In [5]: c = 0.3 + -0.5j
this one is also in the set
```

```
In [6]: inMSet(c, 25)
Out[6]: True
```

```
In [7]: c = -0.7 + 0.3j
this one is NOT in the set
```

```
In [8]: inMSet(c, 25)
Out[8]: False
```

```
In [9]: c = 0.42 + 0.2j
```

```
In [10]: inMSet(c, 25)
this one _seems_ to be in the set
Out[10]: True
```

```
In [11]: inMSet(c, 50)
but at 50 tries, it turns out that it's not!
Out[11]: False
```

### Getting too many `True`s?

If so, you might be checking for `abs(z) > 2` *after* the for loop finishes. Be sure to check *inside* the loop!

There is a subtle reason you need to check inside the loop:

Many values get so large so fast that they overflow the capacity of Python's floating-point numbers. When they do, *they cease to obey greater-than / less-than relationships*, and so the test will fail. The solution is to check whether the magnitude of `z` ever gets bigger than 2 **inside** the for loop, in which case you should immediately return `False`.

The `return True`, however, needs to stay outside the loop!

As the last example illustrates, when numbers are close to the boundary of the Mandelbrot set, many additional iterations may be needed to determine whether they escape. This is why it is so computationally intensive to build high-resolution images of the Mandelbrot set.

## Creating Images with Python

Try out this code to get started:

```
from cs5jpg3 import *
You might already have this line at the top...

def weWantThisPixel(col, row):
 """This function returns True if we want to show
 the pixel at col, row and False otherwise.
 """
 if col % 10 == 0 and row % 10 == 0:
 return True
 else:
 return False

def test():
 """This function demonstrates how
 to create and save a PNG image.
 """
 width = 300
 height = 200
 image = PNGImage(width, height)

 # create a loop in order to draw some pixels

 for col in range(width):
 for row in range(height):
 if weWantThisPixel(col, row):
 image.plotPoint(col, row)

 # we looped through every image pixel; we now write the file

 image.saveFile()
```

Save this code, and then run it by typing `test()`, with parentheses, at the Python shell.

If everything goes well, `test()` will run through the nested loops and print a message that the file `test.jpg` has been created. That file should appear in the same directory as your starter file.

Both Windows and Mac computers have nice built-in facilities for looking at jpg-type images; `jpg` is short for *portable network graphics*. For many people, double clicking on the icon of the `test.jpg` image will display it. (For me, on a Windows machine, opening `test.jpg` in a browser was even better.)

Either way, for the above function, it should be all white except for a regular, sparse point field, plotted wherever the row number and column number were both multiples of 10:

You can zoom in and out of images with the menu options or shortcuts, as well.

### An Image Thought-Experiment to Consider...

Before changing the above code, **write a short comment** under the `test` function in your file describing how the image would change if you changed the line

```
if col % 10 == 0 and row % 10 == 0:
 to the line
if col % 10 == 0 or row % 10 == 0:
```

Then, make that change from `and` to `or` and try it. On both on Macs and PCs, the image does not have to be re-opened: if you leave the previous image window open, its image will update automatically.

Just for practice, you might try creating other patterns in your image by changing the `test` and `weWantThisPixel` functions appropriately.

## Some notes on how the test function works...

There are three lines of the `test` function that warrant a closer look:

- `image = PNGImage(width, height)` This line of code creates a variable of type `PNGImage` with the specified height and width. The `image` variable holds *the whole image!* This is similar to the way a single variable—often called `L`—can hold an arbitrarily large list of items. When information is gathered together into a list or an image or another structure, it is called a *software object* or just an *object*.

We will build objects of our own design in a couple of weeks; this lab is an opportunity to use them without worrying about how to create them from scratch.

- `image.plotPoint(col, row)` An important property of software *objects* is that they carry around and call functions of their own! They do this using the dot `.` operator. Here, the `image` object is calling its own `plotPoint` function to place a pixel at the given column and row. Functions called in this way are sometimes called *methods*.
- `image.saveFile()` This line creates the new `test.jpg` file that holds the jpg image. It demonstrates another *method* (i.e., function) of the software object named `image`.

## From Pixel Coordinates to Complex Coordinates

### The problem

Ultimately, we need to plot the Mandelbrot set within the complex plane. However, when we plot points in the image, we must manipulate *pixels* in their own coordinate system.

As the `testImage()` example shows, pixel coordinates start at  $(0, 0)$  (in the lower left) and grow to  $(\text{width}-1, \text{height}-1)$  in the upper right. In the example above, `width` was 300 and `height` was 200, giving us a small-ish image that will render quickly.

The Mandelbrot Set, however, lives in the box

$-2.0 \leq x$  (or real coordinate)  $\leq +1.0$

and

$-1.0 \leq y$  (or imaginary coordinate)  $\leq +1.0$

which is a  $3.0 \times 2.0$  rectangle.

So, we need to convert from each pixel's `col` integer value to a floating-point value, `x`. We also need to convert from each pixel's `row` integer value to the appropriate floating-point value, `y`.

### The solution

One function, named `scale`, will convert coordinates in general.

So, you'll next write this `scale` function:

```
scale(pix, pixMax, floatMin, floatMax)
```

that can be run as follows:

```
In [1]: scale(150, 200, -1.0, 1.0)
```

Here, the arguments mean the following:

- The first argument is the current pixel value: we are at column 150 or row 150
- The second argument is the maximum possible pixel value: pixels run from 0 to 200 in this case

- The third argument is the minimum floating-point value. *This is what the function will return when the first argument is 0.*
- The fourth argument is the maximum floating-point value. *This is what the function will return when the first argument is pixMax.*

Finally, the **return value** should be the floating-point value that corresponds to the integer pixel value of the first argument.

The return value will always be somewhere between floatMin and floatMax (inclusive).

**This function will NOT use a loop.** In fact, it's really just arithmetic. You will need to ask yourself

- How to use the quantity `pix / pixMax`
- How to use the quantity `floatMax - floatMin`

Hint

You wrote a very similar function (`interp`) in the Function Fun problem.

### A Start to the scale Function

To compute this conversion back and forth from pixel coordinates to complex coordinates, write a function that starts as follows:

```
def scale(pix, pixMax, floatMin, floatMax):
 """scale accepts
 pix, the CURRENT pixel column (or row)
 pixMax, the total # of pixel columns
 floatMin, the min floating-point value
 floatMax, the max floating-point value
 scale returns the floating-point value that
 corresponds to pix
 """

```

The docstring describes the arguments:

- `pix`, an integer representing a pixel column
- `pixMax`, the total number of pixel columns available
- `floatMin`, the floating-point lower endpoint of the image's axis
- `floatMax`, the floating-point upper endpoint of the image's axis

Note that there is no `pixMin` because the pixel count always starts at 0.

Again, the idea is that `scale` will return the floating-point value between `floatMin` and `floatMax` that corresponds to the position of the pixel `pix`, which is somewhere between 0 and `pixMax`. This diagram illustrates the geometry of these values:

Once you have written your `scale` function, here are some test cases to try to be sure it is working:

```
In [1]: scale(100, 200, -2.0, 1.0)
halfway from -2 to 1 should be -0.5
Out[1]: -0.5
```

```
In [2]: scale(100, 200, -1.5, 1.5)
halfway from -1.5 to 1.5 should be 0.0
Out[2]: 0.0
```

```
In [3]: scale(100, 300, -2.0, 1.0)
1/3 of the way from -2 to 1 should be -1.0
Out[3]: -1.0
```

```
In [4]: scale(25, 300, -2.0, 1.0)
1/12 of the way from -2 to 1 should be -1.75
Out[4]: -1.75
```

```
In [5]: scale(299, 300, -2.0, 1.0)
your exact value may differ slightly...
Out[5]: 0.99
```

**Note** Although we initially described `scale` as computing x-coordinate (real-axis) floating-point values, your `scale` function works equally well for both the x- and the y-dimensions. You don't need a separate function for the vertical axis!

### Visualizing the Mandelbrot Set in Black and White: `mset`

This part asks you to put the pieces from the above sections together into a function named `mset()` that computes the set of points in the Mandelbrot set on the complex plane and creates a bitmap of them, of size `width` by `height`.

#### x and y ranges

To focus on the interesting part of the complex plane, we will limit the ranges of `x` and `y` to

```
-2.0 <= x or real coordinate <= +1.0
and
-1.0 <= y or imaginary coordinate <= +1.0
```

which is a 3.0 x 2.0 rectangle.

**How to get started?** Start by copying the code from the test function and renaming it as `mset`:

```
def mset():
 """Creates a 300x200 image of the Mandelbrot set
 """
 width = 300
 height = 200
 image = PNGImage(width, height)
```

```
create a loop in order to draw some pixels
```

```
for col in range(width):
 for row in range(height):

Use scale twice:
once to create the real part of c (x)

x = scale(..., ..., ..., ...)

once to create the imaginary part of c (y)

y = scale(..., ..., ..., ...)
```

```
THEN, create c, choose n, and test:
c = x + y*1j
n = 25
if inMSet(c, n):
 image.plotPoint(col, row)
```

```
we looped through every image pixel; we now write the file
image.writeFile()
```

To build the Mandelbrot set, you will need to change a number of behaviors in this function—start where the comment suggests that you should `Use scale twice`:

- For each pixel `col`, you need to compute the *real (x) coordinate* of that pixel in the complex plane. Use the variable `x` to hold this x-coordinate, and use the `scale` function to find it!
- For each pixel `row`, you need to compute the *imaginary (y) coordinate* of that pixel in the complex plane. Use the variable `y` to hold this y-coordinate, and again use the `scale` function to find it! Even though this will be the imaginary *part* of a complex number, it is simply a normal floating-point value.
- Using the real and imaginary parts computed in the prior two steps, create a variable named `c` that holds a **`complex`** value with those real (`x`) and imaginary (`y`) parts, respectively. Recall that you'll need to multiply `y*1j`, not `y*j`!
- Finally, your test for which pixel `col` and `row` values to plot will involve `inMSet`, the first function you wrote. You'll want to specify a value for the argument named `n` to that `inMSet` function. I'd start with a value of 25 for `n`.

Once you've composed your function, try

```
In [1]: mset()
```

and check to be sure that the image you get is a black-and-white version of the Mandelbrot set, e.g., something like this:

### Adding Features to your `mset` Function

You don't have to use black and white!

The `image.plotPoint` method accepts an optional third argument that represents the *color* of the point you'd like to plot. Here is an example:

```
image.plotPoint(col, row, (0, 0, 255))
```

The third argument here is a list that uses *parentheses* instead of square-brackets. Parenthesized lists are called *tuples* in Python. Tuples are faster to access than lists, but their elements cannot be assigned to, so they're often used for constants such as colors.

The three elements of the color tuple above are *red*, then *green*, then *blue*: each of those three color components must be an integer from 0 to 255. Thus, the tuple above, `(0,0,255)` is pure blue.

To change the *background* of the set, add the lines

```
else:
 image.plotPoint(col, row, (0, 0, 0))
```

within the loops that run over each `col` and each `row`. This will explicitly plot, in black, all of the points *not* in the Mandelbrot Set

**Try it out!** perhaps first by changing your Mandelbrot set to orange `(255, 175, 0)` on top of a black background `(0, 0, 0)`.

Then, change the colors to something more to your liking!

Feel free to enlarge your image, too—it will take longer to render, but you'll be able to resolve more detail in the result!

### No Magic Numbers!

*Magic Numbers* are simply literal numeric values that you have typed into your code. They're called *magic* numbers because if someone tries to read your code, the values and purpose of those numbers seem to have been pulled out of a hat. For example, your `mset` function might call `inMSet(c, 25)` (at least, mine did). A newcomer to your code (and this problem) would have no idea what that `25` represented or why you chose that value.

To keep your code as flexible and expandable as possible, it's a good idea to avoid using these "magic numbers" for important quantities in various places in your functions. Instead, it's better to collect all of those magic numbers at the very top of your functions (after the docstring) and to give them useful names that suggest their

purpose. It's common, though not required, to use all caps for these values—for example, you might have the line:

```
NUMITER = 25
of updates
```

where NUMITER is the number of iterations to be used by the `inMSet` function. The function call would then look like

```
inMSet(c, NUMITER)
```

In addition to being clearer, this makes it *much* easier to add or change functionality in your code—all of the important quantities are defined **one time** and in **one place**.

For this part of the lab, **move all of your magic numbers to the top of the function and give them descriptive names**. For example, these five lines are a good starting point:

```
NUMITER = 25
of updates, from above
XMIN = -2.0
the smallest real coordinate value
XMAX = 1.0
the largest real coordinate value
YMIN = -1.0
the smallest imag coordinate value
YMAX = 1.0
the largest imag coordinate value
```

These variables can then be changed in one place in order to alter the "window" to the Mandelbrot Set being plotted.

In the next part, you'll change these values to zoom into the Mandelbrot set.

## Zooming In!

For this part, simply run your `mset` function with some different values of XMAX, XMIN, YMAX, and YMIN to create images of different parts of the Mandelbrot Set.

For example, try the values

```
NUMITER = 25
XMIN = -1.3
XMAX = -1.0
YMIN = .1
YMAX = .3
```

Then, try this range again, this time using `NUMITER = 50`—you'll see that fewer points are included because of the additional time to escape!

Another range worth checking out is the "infinite snowmen":

```
NUMITER = 25
XMIN = -1.2
XMAX = -.6
YMIN = -.5
YMAX = -.1
```

**Optionally**, feel free to look around to find another set of values that shows an interesting piece of the set—and note what they are in a comment in your file. As a guide, you might consider the suggestions at this page on the "Seahorse Valley".

Note that the *aspect ratio* of the image is 3:2 (horizontal:vertical), and if you keep this aspect ratio in your ranges, the set will be scaled naturally.

It will work with different ratios, but to maintain the natural scaling, you would have to change the height and width of the image accordingly. Or you could *compute* the height and width, but that's not required for this lab.

If you'd like to visualize the different *escape velocities* or change the resolution or perhaps *mandelbrotify* another image, these next sections describe how to add features to your Mandelbrot-set program. They're optional, but fun!

### ***Completely Optional: Visualizing Escape Velocities***

These extensions are optional (but fun!)—this first one lets you see the relative speeds with which the points in and/or around the Mandelbrot set escape to infinity.

Images of fractals often use color to represent *how fast* points are diverging toward infinity when they are not contained in the fractal itself. For this problem, create a new version of `mset` called `msetColor`. Simply copy-and-paste your old code, because its basic behavior will be the same. However, you should alter the `msetColor` function so that it plots points that escape to infinity *more quickly* in different colors.

Thus, have your `msetColor` function plot the Mandelbrot set as before. In addition, however, use at least three different colors to show how quickly points outside the set are diverging—this is their "escape velocity." Making this change will require a change to your `inMSet` helper function, as well. We suggest that you copy, paste, and rename `inMSet` so that you can change the new version without affecting the old one.

There are several ways to measure the "escape velocity" of a particular point. One is to look at the resulting magnitude after the iterative updates. Another is to count the number of iterations required before it "escapes" to a magnitude that is greater than 2. An advantage of the latter approach is that there are fewer different escape velocities to deal with.

Choose one of those approaches—or design another of your own—to implement `msetColor`.

### **Mandelbrotifying Another Image**

The `jpg` library can read images, too. (As a warning, it's not overly fast in converting them to Python lists...)

The result is that you can use another image's pixels to determine the look of the points in (or out) of your Mandelbrot set, e.g.,

Here is an example function to show how to read in an image (the `alien.jpg` image from the `jpgs` folder):

```
def example():
 """Shows how to access the pixels of an image.
 inputPixels is a list of rows, each of which is a list of columns,
 each of which is a list [r,g,b]
 """
 inputPixels = getRGB("./jpgs/alien.jpg")
 inputPixels = inputPixels[::-1] # the rows are reversed

 height = len(inputPixels)
 width = len(inputPixels[0])
 image = PNGImage(width, height)

 for col in range(width):
 for row in range(height):
 if col%10 < 5 and row%10 < 5: # only plot some of the pixels
 image.plotPoint(col, row, inputPixels[row][col])
```

```
image.saveFile()
```

The above code produces the following image:

Try *Mandelbrotifying* this image—or another jpg you'd like to use...

# CSforAll - SpellingatMillisoft

## CS for All

CSforAll Web > Chapter5 > SpellingatMillisoft

This website accompanies the textbook *CS For All* by Alvarado, Dodds, Kuenning, and Libeskind-Hadas

### Spelling At Millisoft

Here is the gratuitous "true story": You've landed a summer internship at Millisoft. One day the CEO, Gill Bates, comes into your office sipping on a Jolt Cola. "I've decided that Millisoft is going to have a new spell checking product called "spam" and it's yours to develop and implement! As an incentive, you'll get a lifetime supply (two six-packs) of Jolt when it's done."

Spell checking of this type is both useful to those of us who have trouble spelling (or trying) and also useful in biological databases where the user types in a sequence (e.g. a DNA or amino acid sequence) and the system reports back the near matches.

How do we measure the similarity of two strings? The "edit distance" between two strings S1 and S2 is the **minimum** number of "edits" that need to be made to the strings to get them to match. An "edit" is either a **replacement** of a symbol with another symbol or the **deletion** of a symbol. For example, the edit distance between "spam" and "xsam" is 2. We can first delete the "x" in "xsam" leading to "sam" and then delete the "p" in "spam" to also make "sam". That's two edits, for an edit distance of 2. That's the best possible in this case. Of course, another possible sequence of edits would have been to delete the "s" and "p" from "spam" to make "am" and delete the "x" and "s" from "xsam" to make also "am". That's 4 edits, which is not as good as 2.

Here's a function that calculates the edit distance:

```
def ED(first, second):
 '''Returns the edit distance between the strings first and second.'''
 if first == '':
 return len(second)
 else:
 if first[0] == second[0]:
 return ED(first[1:], second[1:])
 else:
 return 1 + min(ED(first[1:], second),
 ED(first, second[1:]))
```

```

elif second == '':
 return len(first)
elif first[0] == second[0]:
 return ED(first[1:], second[1:])
else:
 substitution = 1 + ED(first[1:], second[1:])
 deletion = 1 + ED(first[1:], second)
 insertion = 1 + ED(first, second[1:])
 return min(substitution, deletion, insertion)

```

Now, here's ED in action:

```
In [1]: ED("spam", "xsam")
Out[1]: 2
```

```
In [2]: ED("foo", "")
Out[2]: 3
```

```
In [3]: ED("foo", "bar")
Out[3]: 3
```

```
In [4]: ED("hello", "below")
Out[4]: 3
```

```
In [5]: ED("yes", "yelp")
Out[5]: 2
```

### It's Cute, But It's Slow!

Try your ED function on a few pairs of long words. For example, here's my attempt to observe the *extraordinary* slowness of this program! Exercise your *originality* in finding other pairs of very long words to try out!

```
In [1]: ED("extraordinary", "originality")
```

Wait for a bit—you will get an answer (it's 10).

Since the recursive program is very slow, Gill has asked you to reimplement it using memoization. Write a new function called `fastED(S1, S2, memo)` that computes the edit distance using a memo (Python dictionary) to memoize previously computed results.

After writing `fastED(S1, S2)`, test it to make sure that it's giving the "write" answer. Here are a few more test cases:

```
In [1]: fastED("antidisestablishment", "antiquities", {})
Out[1]: 13
```

```
In [2]: fastED("xylophone", "yellow", {})
```

```

Out[2]: 7

In [3]: fastED("follow", "yellow", {})
Out[3]: 2

In [4]: fastEd("lower", "hover", {})
Out[4]: 2

topNmatches(word, nummatches, ListOfWords)

```

Next, you'll write a function whose signature is

```
def topNmatches(word, nummatches, ListOfWords)
```

which takes in

- `word`, a string which is the word to match (using `fastED`)
- `nummatches`, an integer which is 0 or greater
- `ListOfWords`, a list of strings against which to match `word`

and, from there, `topNmatches` should output the *alphabetically-sorted* list of a total of `nummatches` words from `ListOfWords` that have the lowest edit-distance scores with the input `word`

- you should be sure to return only N words. If there is a tie at the last place among those N, there will be an ambiguity as to which words to return
- we will not test it in those "tie" cases, however, so you're welcome to handle that ambiguity as you see fit

Here are a couple of examples:

```

In [1]: topNmatches("spam", 3, ["spam", "seam", "wow", "cs5blackrocks", "span", "synecdoch"]
Out[1]: ['seam', 'spam', 'span']

In [2]: topNmatches("spam", 1, ["spam", "seam", "wow", "cs5blackrocks", "span", "synecdoch"]
Out[2]: ['spam']

we would not test the above example with nummatches == 2, since the output would be ambiguous

```

And here are a few details that may be of help:

- You will need to find the score (edit distance) for each word in the master list of words. One way to do this is to construct another list that is just like your master list, except that each entry in that new list will be a tuple of the form `(score, word)`. For example, the tuple `(42, "spam")` would mean that the word "spam" has edit distance 42 from the word that the user entered. You can use `map` to build this list of tuples!

- You'll need to sort the words by score. While mergesort is very fast, the amount of recursion that it requires will likely exceed the recursion limit permitted by your computer. Therefore, use the very fast sorting algorithm built into Python, which is named `sort`. This function is used as follows: If `L` is the name of your list (it can have any name you like) then the syntax `L.sort()` will modify `L` by sorting it in increasing order. This will sort `L` but will not return anything. So, use the line `L.sort()` rather than `sortedList = L.sort()`. It's a strange syntax, but it works (and we'll talk about the reason for this syntax in a few weeks.) You may wish to sort a list of items, each of which is a list or a tuple. For example, if you have a list `L` that looks like this: `[[42, "hello"], [15, "spam"], [7, "chocolate"]]` and you do `L.sort()`, it will sort `L` using the first element in each list as the sorting key. So, `L.sort()` in this case will change `L` to the list `[[7, "chocolate"], [15, "spam"], [42, "hello"]]`.
- You'll use `sort` again in order to get your ultimate list of words into alphabetical order!

You'll put this together into an interactive program, known as `SPAM` in the next part... .

## SPAM!

Finally, your last task is to write a function called `spam()` that loads in a large master list of words and then repeatedly does the following:

- The memo is initialized to be an empty dictionary.
- The user is shown the prompt `spell check>` and prompted to type in a word.
- If the word is in the master list, the program reports `Correct`. You can test if a string is in a list by using the `in` keyword as in `if "spam" in ["everyone", "loves", "spam"]` returns `True`.
- If the word is not in the master list, the program should compute the edit distance between the word and *every* word in the master list. Then the 10 most similar words should be reported, in order of smallest to largest edit distance. The program should also report how long it took to find this list.

### A Note on Performance

Your spell checker will call the memoized `fastED` function repeatedly. Each time the spell checker calls `fastED` it could send it a "fresh" empty dictionary. This would be reasonably fast. But, you can get *even better* performance by NOT re-initializing the memo dictionary each time you call `fastED`. In other words, when the spell checker starts up, it will begin with an empty dictionary. But, each time `fastED` is called, it will simply be called with the current `memo` dictionary rather than a brand new empty one. Do you see why this is likely to make your spell checker faster?

### Back to Our Regularly Scheduled Program (So to Speak)...

Here is an example of what your program will look like when running. The actual times may vary from computer to computer, so don't worry if you see different running times on your computer. Moreover, if there are ties in the edit distance scores, you may break those ties arbitrarily when sorting.

```
In [1]: spam()
spell check> hello
Correct
spell check> spam
Suggested alternatives:
scam
seam
sham
slam
spa
span
spar
spasm
spat
swam
Computation time: 2.06932687759 seconds
```

Here are the ingredients that you will need:

- Download `3esl.txt` into the same directory (folder) where your program resides. This is our master list of words: It is simply a file with 21877 words in alphabetical order. Save this file on your machine. (On the Macs, push control and mouse-click on this link and then choose to save this link in a file.) Make sure that this file is in the same directory as your program.
- The following three lines will open the file `3esl.txt`, read it, and split it into a list called `words`. ***Be sure these lines are INSIDE your `spam` function*** and not at the top-level of the file (they will cause all of your tests to fail!) So, the top of your function will look like this (with a better docstring, for sure):

```
def spam():
```

```
""" docstring """
f = open("3esl.txt")
contents = f.read()
words = contents.split("\n")
```

- You'll need to prompt the user for input. The Python function `input(S)` displays the string `S` and then waits for the user to enter a string and hit the return key. The string that was typed in by the user is now the value returned by the `input` function. For example

```
userInput = input("spell check> ")
```

displays the string `spell check>`, waits for the user to type in a string, and then returns that string so that `userInput` now stores that string. (You'll find that things look nicer if your prompt string ends with a blank.)

- To compute the amount of time that transpires between two points in your program, we recommend the following:
  - First, have the line `import time` at the top of your program to import the `time` package;
  - Next, any time you like, call `time.time()` to capture the number of seconds (a floating point number) that have elapsed since some point in the past (perhaps when your program started, or perhaps some other well known time—it doesn't really matter!). By capturing `time.time()` at two different places and subtracting the first value from the second, you can determine the elapsed time in that part of the program. Here is an example:

```
import time

def time_example():
 yadda, yadda, yadda
 startTime = time.time()
 blah, blah, blah
 endTime = time.time()
 print("The elapsed time executing blah, blah, blah was", endTime - startTime)
```

Website design by Madeleine Masser-Frye and Allen Wu

# CSforAll - MarkovText1

## CS for All

CSforAll Web > Chapter5 > MarkovText1

This website accompanies the textbook *CS For All* by Alvarado, Dodds, Kuenning, and Libeskind-Hadas

### Markov Text Generation

#### Getting Started

Use this as the starting point for your file:

```
coding: utf-8
#
the top line, above, is important --
it ensures that Python will be able to use this file,
even if you paste in text with Unicode characters
(beyond ASCII)
for an more expansive example of such a file, see
http://www.cl.cam.ac.uk/~mgk25/ucs/examples/UTF-8-demo.txt
#
#
Name:
#
#
function #1
#
def createDictionary(filename):
 pass

#
function #2
#
```

```

def generateText(d, N):
 pass

 #
 # Your 500-or-so-word CS essay (paste into these triple-quoted strings below):
 #
 """"
 """
 #
 #

```

Your goal in this section is to write a program that is capable of generating "meaningful" text all by itself! You will accomplish this goal by writing a Markov text generation algorithm.

### Using Python's Dictionaries...

Python has a built-in data type (class) known as a *dictionary*. Try the examples below at the Python shell just to familiarize yourself with how dictionaries work. Briefly, they're containers like lists, but better!

```

In [1]: d = {} # creates an empty dictionary, d
Notice that these are CURLY braces!

In [2]: d[1993] = 'rooster' # en.wikipedia.org/wiki/Chinese_zodiac

In [3]: d
Out[3]: {1993: 'rooster'} # 1993 is a KEY, 'rooster' is a VALUE

In [4]: d[1994] = 'dog'

In [5]: d
Out[5]: {1993: 'rooster', 1994: 'dog'}

In [6]: d[1994]
Out[6]: 'dog'

```

```

In [7]: d[1995]
Traceback (most recent call last):
 File "<stdin>", line 1, in <module>
KeyError: 1995

In [8]: 1994 in d # in checks for KEYS only
Out[8]: True

In [9]: 1995 in d
Out[9]: False

In [10]: d.keys()
Out[10]: dict_keys([1993, 1994])

In [11]: d.values()
Out[11]: dict_values(['rooster', 'dog'])

In [12]: d.items()
Out[12]: dict_items([(1993, 'rooster'), (1994, 'dog')])

```

This problem uses a Python dictionaries to model—and then generate—text.

### **Example Code (File-Reading and Using Dictionaries...)**

Python examples of how dictionaries can be used to analyze text in a *vocabulary-counter* (as opposed to a word-counter) are here in this example code—a good reference for some of what you'll need...

### **Markov Text Generation**

Here's the basic idea: English is a language with a lot of structure. Words have a tendency (indeed, an obligation) to appear only in certain sequences. Grammatical rules specify legal combinations of different parts of speech. E.g., the phrase "The cat climbs the stairs" obeys a legal word sequence. "Stairs the the climbs cat", does not. Additionally, semantics (the meaning of a word or sentence), further limits possible word combinations. "The stairs climb the cat" is a perfectly legal sentence, but it doesn't make much sense and you are very unlikely to encounter this word ordering in practice.

Even without knowing the formal rules of English, or the meaning of English words, we can get an idea of what word combinations are legal simply by

looking at well-formed English text and noting the combinations of words that tend to occur in practice. Then, based on our observations, we could generate *new* sentences by randomly selecting words according to commonly occurring sequences of these words. For example, consider the following text:

"I love roses and carnations. I hope I get roses for my birthday."

If we start by selecting the word "I", we notice that "I" may be followed by "love", "hope" and "get" with equal probability in this text. We randomly select one of these words to add to our sentence, e.g. "I get". We can repeat this process with the word "get", necessarily selecting the word "roses" as the next word since in our sample "get" is always (i.e., once) followed by "roses". Continuing this process could yield the phrase "I get roses and carnations". Note that this is a valid English sentence, but not one that we have seen before. Other novel sentences we might have generated include "I love roses for my birthday," and "I get roses for my birthday".

More formally, the process we use to generate these sentences is called a *first-order Markov process*. A first-order Markov process is a process in which the state at time  $t+1$  (i.e. the next word) depends only on the state at time  $t$  (i.e., the previous word). In a second-order Markov process, the next word would depend on the *two* previous words, and so on. Our example above was a first-order process because the choice of the next word depended only on the current word. Note that the value of the next word is independent of that word's position and depends only on its immediate history. That is, it doesn't matter if we are choosing the 2nd word or the 92nd. All that matters is what the 1st or the 91st word is, respectively.

### Your Text Analyzer and Text Generator

In the first part of this assignment you will implement a first-order Markov text generator. Writing this function will involve two functions: (1) one to process a file and create a dictionary of legal word transitions and (2) another to actually generate the new text.

Without special handling, your code will consider words different even if they differ only by capitalization or added punctuation. This is completely OK for this assignment: `spam` and `Spam` and `spam.` (with the trailing period included) may all be considered distinct.

Here are details on the two functions to write:

### `createDictionary(filename)`

`createDictionary(filename)` accepts a string, which is the name of a text file containing some sample text. It should return a dictionary whose keys are words encountered in the text file and whose entries are a list of words that may legally follow the key word. Note that you should determine a way to keep track of frequency information. That is, if the word "cheese" is followed by the word "pizza" twice as often as it is followed by the word "sandwich", your dictionary should reflect this trend. For example, you might keep multiple copies of a word in the list.

The dictionary returned by `createDictionary` will allow you to choose word  $t+1$  given a word at time  $t$ . But how do you choose the *first* word, when there is no preceding word to use to index into your dictionary?

To handle this case, your dictionary should include the string "\$", which will represent the *sentence-start symbol*. The first word in the file should "follow" this string. In addition, each word in the file that follows a sentence-ending word should follow this string. A sentence-ending word will be defined to be any raw, space-separated word whose last character is a period ., a question mark ?, or an exclamation point !

#### ***How do I determine whether a word ends in a punctuation mark?***

The easiest way is to check `w[-1]`. We will only worry about '.', '?', and '!'.  
• Remember that, if you use `or`, you need to have each test fully described, e.g.,

```
if w[-1] == '.' or w[-1] == '?' or ...
```

- You may recall the `in` alternative (from vowel-checking long ago...): `if w[-1] in '.!':`

#### **Strategy**

Note that two of the class slides contain *almost* the entirety of this function. We'll include images here and a bit more description below:

Model creation in Python

```

$: [I, I] d[final value]
I : [like, eat]
like : [spam]
eat : [poptarts]

prev $

word I like spam. I eat poptarts!

```

```

d = {}
prev = '$'

for word in LoW:
 if prev not in d: d[prev] = [word]
 else: d[prev] += [word]

 prev = _____

```

most/hast common?

```

def vocab_count(filename):
 """Vocabulary-counting program"""
 f = open(filename)
 text = f.read()
 f.close()

 LoW = text.split()
 print("There are", len(LoW), "words.") }

 d = {}

 for word in LoW:
 if word not in d:
 d[word] = 1
 else:
 d[word] += 1

 print("There are", len(d), "distinct words.\n")

 return d # Return d for later use by other code...

```

file handling

word counting

Tracking the number of occurrences of each word with a dictionary, d.

(Note that you can open those images in a new tab for a larger version... .)

In particular, you will want to (1) use the file-reading code that is at the top of the right-hand image, but *not* the word-counting code there...

Instead, you'll want the code relating nw and pw (which stand for "new word" and "previous word," respectively) from the left-hand image. Here is the result:

```

pw = '$'

for nw in LoW:
 if pw not in d:
 d[pw] = [nw]
 else:
 d[pw] += [nw]

 pw = nw

 # then check for whether that new pw ends in
 # punctuation -- if it _does_ then set pw = '$'

```

The one thing not included in these examples is how to handle words that end in a punctuation mark. That is left to you (but is hinted at in the hashtags-comments above)...

### Checking your code...

To check your code, paste the following text into a plain-text file (for example, into a new ".txt" file window in VScode):

A B A. A B C. B A C. C C C.

Save this file as t.txt in the same directory where your file lives. Then, see if

your dictionary `d` matches the sample below:

```
In [1]: d = createDictionary('t.txt')

In [2]: d
Out[2]:
{'$': ['A', 'A', 'B', 'C'],
 'A': ['B', 'B', 'C.'],
 'B': ['A.', 'C.', 'A'],
 'C': ['C', 'C.']}
```

The elements within each list need not be in the same order, but they should appear in the quantities shown above for each of the four keys, 'A', 'C', 'B', and '\$'.

Here are the contents of the `poptarts` file, named `a.txt`, from class.

- If you're using Mac's Text Edit, be sure to choose *Format ... Make Plain Text*—you need the file to be a `.txt` file, not an `.rtf`:

```
I like poptarts and 42 and spam.
Will I get spam and poptarts for
the holidays? I like spam poptarts!
```

You'll want to be sure that the output dictionary from this file is the same as the one in the class notes (note that the order of the keys can vary):

```
In [1]: d = createDictionary('a.txt')

In [2]: d
Out[2]:
{'and': ['42', 'spam.', 'poptarts'],
 '$': ['I', 'Will', 'I'],
 'for': ['the'],
 'get': ['spam'],
 'I': ['like', 'get', 'like'],
 'spam': ['and', 'poptarts!'],
 '42': ['and'],
 'Will': ['I'],
 'poptarts': ['and', 'for'],
 'the': ['holidays?'],
 'like': ['poptarts', 'spam']}
```

`generateText(d, n)`

`generateText(d, n)` will take in a dictionary of word transitions `d` (generated in your `createDictionary` function, above) and a positive integer, `n`. Then,

`generateText` should print a string of `n` words.

The first word should be randomly chosen from among those that can follow the sentence-starting string `"$"`. Remember that `random.choice` will choose one item randomly *from a list!* The second word will be randomly chosen among the list of words that could possibly follow the first, and so on. When a chosen word ends in a period `.`, a question mark `?`, or an exclamation point `!`, the `generateText` function should detect this event and start a new sentence by again choosing a random word from among those that follow `"$"`.

Don't include the `'$'` in the output text itself—it will be a marker internal to your function.

For this problem, you should *not* strip the punctuation from the raw words of the text file. Leave the punctuation as it appears in the text—and when you generate words, don't worry if your generated text does not end with legal punctuation, i.e., you might end without a period, which is ok. The text you generate won't be perfect, but you might be surprised at how good it is!

You may assume there won't be any isolated punctuation (or other pathological cases) in the input. It's best to handle any unexpected cases by simply starting a new sentence. Specifically, if your process ever encounters a word that has only appeared as the last word in your training set (i.e., it does not have any words that follow it), you should simply start generating text again with the sentence-start symbol `"$"`.

Here are two examples that use the dictionary `d`, from above. Yours will differ because of the randomness, but should be similar in spirit.

```
In [3]: generateText(d, 20)
Out[3]: B C. C C C. C C C C C C C C. C C C. A
```

```
In [4]: generateText(d, 20)
Out[4]: A B A. C C C. B A B C. A C. B A. C C C C C C.
```

### Your Results! The 500-Word CS Essay!

Or *C-eSSay*, perhaps... !

As the final part of this problem, find a plain-text file of interest, create a first-order Markov model from it, and generate some of your own text! Think of it as a "CS essay."

At the bottom of your file, include a triple-quoted string (comment) with:

- The name of the source text (and author, if applicable) that you used for your CS essay (you *don't* need to include the source text itself!).
- At least 500 words of your generated text.

- Please do point out at the top any highlights you'd like to single out from your generated text. 

The input text you want to use is up to you! Random scenes from Shakespeare might strike your fancy... Dr. Seuss or nursery rhymes can (sometimes) work well... Song lyrics, speeches, or your own papers—all are possibilities.

**Warning**—do not attempt to use CS essays as *actual* essays!

**Help! How do I get a plain-text file**, which I can then use for input?

The easiest way to get a plain text file is to

1. Copy the text from wherever it is using the menu option or control-c (command-c on a Mac)
2. Open a blank text file using Notepad (Windows) orTextEdit (Mac)
  - **Warning!** By default, TextEdit saves files as *rich text format* or *.rtf*. You can change this by choosing *Format...Make Plain Text*.
  - You should do that to ensure that you get a plain-text *.txt* file.
3. Save your plain-text *.txt* file, e.g., *spam.txt*, in the same folder as your file.
4. That should do it!

Good luck!

#### **Extra Credit: Word Generation**

The same process that you used to generate sentences can also be used at the letter level to generate English words. Adapt your Nth-order word generator, above, into an Nth-order analyzer and generator that work ***character-by-character***, instead of word-by-word. Try different values of N and add a comment on how large N needs to be before the text generated seems English-like (or whatever other language you might want to try!)

Website design by Madeleine Masser-Frye and Allen Wu

# CSforAll - MarkovText2

## CS for All

CSforAll Web > Chapter5 > MarkovText2

This website accompanies the textbook *CS For All* by Alvarado, Dodds, Kuenning, and Libeskind-Hadas

### Markov Text Generation!

Your goal in this problem is to write a program that "learns" English (or any text, for that matter) using the k-th order Markov Model algorithm that we described in class. Here are the functions that your program should have. You can have other helper functions as well, but we will test the following functions:

#### Adding \$ to Your Text With `dollarify(wordList, k)`

As discussed in class, it will be important to have special \$ delimiters embedded in the list of words. Specifically, at the beginning of each sentence, we should have exactly k of these delimiters. Here's an example. It also shows Python's string `split` feature which you'll need to split the words in a string into a list of words:

```
>>> test = "A B C. A B B! D A A?"
>>> wordList = test.split() # splits the string up into a list of words
>>> wordList
['A', 'B', 'C.', 'A', 'B', 'B!', 'D', 'A', 'A?']
>>> dollarify(wordList, 2)
['$', '$', 'A', 'B', 'C.', '$', '$', 'A', 'B', 'B!', '$', '$', 'D', 'A', 'A?']
```

#### Building the Markov Model With `markov_model(wordList, k)`

The `markov_model(wordList, k)` function should accept the list of words in the text and should return a k-th order Markov model based on that text. Specifically, that Markov model should be a Python dictionary. For many more details, see the Details section below. For now, we will show three examples of `markov_model(text, k)` in action. Remember that the order of elements in

Python dictionaries does not matter. (We provide a function below that can compare two dictionaries for equivalence.)

```
>>> text = "A B C. A B B C B. A B C C C D B B. B B C C D D B C."
>>> wordList = text.split()
>>> mm1 = markov_model(wordList, 1) # first-order Markov model
>>> mm1
{('C',): ['B.', 'C', 'C', 'D', 'C', 'D'],
 ('$',): ['A', 'A', 'A', 'B'], # things that are first or follow a period
 ('A',): ['B', 'B', 'B'],
 ('B',): ['C.', 'B', 'C', 'C', 'B.', 'B', 'C', 'C.'],
 ('D',): ['B', 'D', 'B']}

>>> mm2 = markov_model(wordList, 2) # second-order Markov model
>>> mm2
{('B', 'C'): ['B.', 'C', 'C'],
 ('$', '$'): ['A', 'A', 'A', 'B'],
 ('D', 'D'): ['B'],
 ('$', 'A'): ['B', 'B', 'B'],
 ('C', 'C'): ['C', 'D', 'D'],
 ('$', 'B'): ['B'],
 ('A', 'B'): ['C.', 'B', 'C'],
 ('D', 'B'): ['B.', 'C.'],
 ('C', 'D'): ['B', 'D'],
 ('B', 'B'): ['C', 'C']}

>>> mm3 = markov_model(wordList, 3) # third-order Markov model
>>> mm3
{('$', '$', 'B'): ['B'],
 ('A', 'B', 'B'): ['C'],
 ('C', 'C', 'D'): ['B', 'D'],
 ('B', 'B', 'C'): ['B.', 'C'],
 ('C', 'D', 'D'): ['B'],
 ('$', 'B', 'B'): ['C'],
 ('$', '$', '$'): ['A', 'A', 'A', 'B'],
 ('C', 'C', 'C'): ['D'],
 ('B', 'C', 'C'): ['C', 'D'],
 ('$', 'A', 'B'): ['C.', 'B', 'C'],
 ('C', 'D', 'B'): ['B.'],
 ('$', '$', 'A'): ['B', 'B', 'B'],
 ('A', 'B', 'C'): ['C'],
 ('D', 'D', 'B'): ['C.']}
```

### Generating Text Using `gen_from_model(mmodel, numwords)`

Next, you should have a `gen_from_model(mmodel, numwords)` function that generates words from a model. In this case, we will print the words, rather than return them.

The `gen_from_model(mmodel, numwords)` function should accept a Markov model named `mmodel`, generated by the `markov_model` function above, and an integer, `numwords`, which is the number of words for it to print from that model. Then `gen_from_model` should determine the order of the Markov model (`mmodel`) and generate `numwords` words from it, starting with the all-'\$' tuple (see below). This function does not need to return anything - it simply uses `print` to print the generated text.

Here is an example with the simple text used above. Your output may look different due to randomness. But, note that the generated "sentences" can have as many Cs in a row as the random-number generation allows, but it can never have more than two Bs or two Ds in a row. A sentence can never include an A after the first letter, and it's impossible for a B to follow a C, unless it ends the "sentence." These are some of the characteristics that we'll look for when running your code!

```
>>> text = "A B C. A B B C B. A B C C C D B B. B B C C D D B C."
>>> d = markov_model(text.split(), 2)
>>> gen_from_model(d, 100)
A B C B. B B C C D B B. A B B C B. B B C C C D B B. B B C C D D B B.
B B C C D B B. B B C B. A B B C B. A B C. A B C B. A B C C D D B B. A
B B C B. A B C C D B C. A B B C B. A B B C B. B B C C D B B. B B C B.
B B C B.
```

### Putting It All Together With `markov(fileName, k, length)`

Finally, you should have a function called `markov(fileName, k, length)` that takes a string containing a file name, the model parameter `k`, and a positive integer `length` indicating the length of the output that you'd like to have generated. This function opens a file, reads in the contents, generates the `k`-th order Markov Model, and then generates output of the given `length`.

Take a look at this `IODemo.py` file to see how to read and write files.

### Some Useful Details!

- You will need to be able to recognize punctuation (periods, exclamation points, question marks). Since you'll need to do this in multiple functions, it makes sense in this case to have a global variable `PUNCTUATION = [".", "!", "?"]`. Now, you can test if something is in the that list using Python's `in` keyword as in `if blah in PUNCTUATION:`.

- You will need to split a string into a list of words. If `myString` is a string, then `myString.split()` returns a list of the words in that string.
- When you're generating text, you'll need to print words. Imagine that `nextWord` is a single word that you wish to print. If you use `print(nextWord)`, the word will be printed but then a newline will be generated so that the next time you print, you'll print on the next line. If you use instead `print(nextWord, end=' ')` then a space ' ' will be printed after the word rather than a newline. Later, if you really want to print a newline, you can just use `print()` to do that.
- Remember `.append`? For example, if you have a list `L` and you wish to add `42` to the end, you can do it with `L.append(42)`. There's also an `.extend` that works like this: If you have a list `L` and a list `M` and you want *all* of the items in `M` to be appended to the end of `L`, you can use the syntax `L.extend(M)`. Note that both of these commands change `L`. Moreover, you would actually just have a single line of code like `L.append(42)` or `L.extend(M)` and NOT `L = L.append(42)` or `L = L.extend(M)`.
- Remember that dictionary keys need to be immutable, so we can use tuples but not lists as keys. But, the values can be anything, including lists. You can change a list to a tuple by using `tuple(L)` where `L` is a list.
- One thing about tuples is that ("spam") may look to you like a tuple of length of 1, but Python will think that you're just parenthesizing the string "spam". So, if you want to build a tuple of just one element, you add a comma after that element like this: ("spam",). That comma tells Python that this is a tuple and not a parenthesized item.
- As words are generated, we proceed as follows. We start with a sequence of  $k$  words which are simply this: ('\$', '\$', ..., '\$') (with  $k$  dollar signs). Then this  $k$ -word sequence is used to select a random next word using the Markov dictionary of valid next words. The next word should be selected at random but weighted by its probability of occurring as a next word (e.g. in the example above, spam would be twice as likely as chocolate to appear after I like). Here's an easy way to do this: First, have the line `import random` at the top of your file to import the random package. This package contains many useful features, but the only one we need here is `random.choice(L)` where `L` is some list. This will return a randomly selected element of that list. You can google "python random" for more information on this package, but you don't need anything else in this assignment.

### Try It Out!

Finally, create a Markov model with input text of your choosing and an order ( $k$ ) of your choice (at least 2), and then generate at least 100 words and copy-and-paste the output into a comment at the bottom of your python file. The

output need not end with punctuation - you can have it stop mid-sentence.

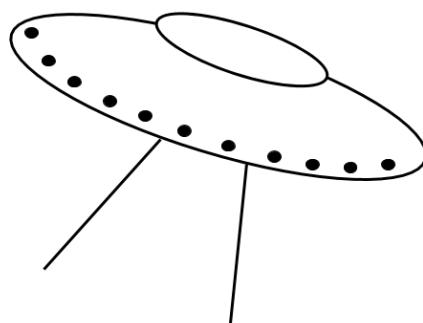
The text that you use is your choice. Shakespeare, Dr. Seuss, song lyrics etc.  
Here are a few that we've found.

- Shakespeare's sonnets
- Famous poems
- 35 Famous speeches

Website design by Madeleine Masser-Frye and Allen Wu

# CSforAll - Quadtrees

## CS for All



CSforAll Web > Chapter5 > Quadtrees

This website accompanies the textbook *CS For All* by Alvarado, Dodds, Kuennen, and Libeskind-Hadas

### Quadtrees!

First the gratuitous backstory! You've been hired as a summer intern at SASA (the Shmorbodian Air and Space Administration) and assigned to the image processing group. SASA's deep space probes capture black-and-white images and must send those images back to Earth. The amount of power required to send an image is proportional to the length of the encoding of the image, so

SASA wants to compress those images as much as possible. In addition, those images sometimes need to be processed (e.g., rotated, flipped, etc.). Your job is to develop software base on quadtrees to compress and manipulate binary (black-and-white) images. Your colleagues at SASA have decided that quadtrees are the way to go - they tend to compress images very well and they allow for fast manipulation of those images.

In this problem, you'll add code to the quadtree.py.zip starter file that we've provided (you'll have to unzip it).

In the `quadtree.py` file, you'll see a string global variable `test1`. This string has length 64 and represents an 8x8 binary image. This is one of two ways that we'll use to input a binary image. (The second way is to load an existing `.png` file; more on that later.)

Once you've loaded `quadtree.py` into Python, try this:

Notice that the function `stringToArray(bstring)` takes a string of 0's and 1's like the `test1` string above and returns a 2-dimensional array representation for that image. Try it! The `renderASCII(array)` function takes a 2D array as input and displays the image as 0's and 1's on the screen. The `renderImage(array)` takes a 2D representation of the image and opens a matplotlib window rendering of the image. You'll need to close the image window before Python is willing to continue.

**quadrants(array)**

It will be useful later to have a function that takes a 2D array and returns a list of four arrays, one for each of the northwest, northeast, southwest, and southeast quadrants, in that order. Here's an example of this function in action. About seven lines of code suffice. Using higher-order functions or list comprehensions will keep this short and sweet!

```

>>> NW, NE, SW, SE = quadrants(array1) # Nice syntax!
>>> NW
[[0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0]]
>>> renderASCII(NW)
0000
0000
0000
0000
>>> NE
[[0, 0, 1, 1], [1, 1, 0, 0], [0, 0, 1, 1], [0, 0, 1, 1]]
>>> SW
[[1, 1, 1, 1], [1, 1, 1, 1], [1, 1, 1, 1], [1, 1, 1, 1]]
>>> SE
[[1, 1, 1, 1], [1, 1, 1, 1], [1, 1, 1, 1], [1, 1, 1, 1]]

```

Recall the convenient syntax used here: `quadrants(array)` returns a list of four arrays, and the first line above gives names to each of the four elements in that list.

#### `makeQuadtree(array)`

A quadtree is yet another representation of a binary image! This representation is frequently much more compact than the 2D array and thus can be used to compress the image. Moreover, many image operations such as rotation, flipping, and inverting the 0's and 1's can be done much faster in the quadtree representation than in the original 2D array representation.

*We'll assume that our images are black and white (1 and 0), always squares, and the dimension is a power of 2 (e.g., 2, 4, 8, 16, 32) pixels on each side.*

Recall that the idea of a quadtree is this: Let's assume, for example, that the image has dimension 8 (as in the example above). Then, if the image is solid 0's, we represent the image as just a 0. If the image is solid 1's, we represent the image as just a 1. If the image is not solid, we partition it into four quadrants and recursively represent the image as list [NW, NE, SW, SE] where NW, NE, SW, and SE are the quadtrees for the northwest, northeast, southwest, and southeast quadrants, respectively. For example, for the `array1` 2D array above, we'd get this:

```

>>> qtree1 = makeQuadtree(array1)
>>> qtree1
[0, [[0, 0, 1, 1], [1, 1, 0, 0], 0, 1], 1, 1]

```

Your first task is to write the `makeQuadTree(array)` function that takes a 2D array of 0's and 1's (remember, it's a square with dimension that is a power of 2) and returns the corresponding quadtree representation.

To that end, you may wish to have a few helper functions. Here are some functions that we found useful:

- `solidZero(array)` returns `True` if the entire 2D array is all 0's and otherwise returns `False`. This requires only one line of code. (*Hint:* The `max` function takes a list as input and returns the maximum element in that list.)
- `solidOne(array)` is analogous for an array of all 1's. It too only requires a single line of code!

You're welcome to write another helper function or two. But, using the functions that you've already written, you shouldn't need to write more than 10 more lines of code total here.

Finally, write at least two more small additional test strings (analogous to `test1`) to check that your `makeQuadtree(array)` function is working correctly and include those tests in your file. Name your first two additional test strings `test2` and `test3`. They should be at least 4x4 pixels in size, but we recommend that at least one of them be 8x8 so that you can better test your code for correctness. You can have more tests if you like, but we'll just test those two.

#### `makeArray(quadtree, dim)`

Next, you'll need a way to go from the compact quadtree representation back to a 2D array in order to render the image. To that end, your next task is to write a function called `makeArray(quadtree, dim)` that takes a quadtree and the original dimension of the image, `dim`. (Recall that `dim` will always be a power of 2.) Using our running example, `qtree1` is the quadtree that we built from `array1` above. Now, we'll convert the quadtree back to its original array form:

```
>>> output = makeArray(qtree1, 8)
>>> output == array1
True
```

Notice that this shows that we recovered the original array from the quadtree. Of course, we had to know that the image was 8x8 to make this work! (One cool thing though is that you can make larger or smaller versions of your original image by entering a `dim` value that is larger or smaller than that of the original image!) Your `makeArray` function will likely be between 10 and 15 lines long, depending on how you write it.

It may be useful to you to be able to construct a solid block of 0's or 1's. Here's a simple way to do that (take a moment to see how this works, since you'll want to write your own code like this in future assignments):

```
def solidArray(value, pixels):
 """ Takes a value (0 or 1) and a number of pixels and returns a 2D array of pixelsxpixels
 bits all of which are set to the given value. """
 return [[value]*pixels for row in range(pixels)]
```

`invert(quadtree)`, `rotateRight(quadtree)`, `flipHorizontal(quadtree)`,  
and `flipDiagonal(quadtree)`

Finally, Millisoft wants to be able to efficiently manipulate images in their compact quadtree representation. To that end write the following three functions:

- `invert(quadtree)` takes a quadtree as input and returns a quadtree of the same size, but all of the 0's are replaced by 1's and all of the 1's are replaced by 0's.
- `rotateRight(quadtree)` takes a quadtree as input and returns a quadtree that represents the image rotated 90 degrees to the right (clockwise).
- `flipHorizontal(quadtree)` takes a quadtree as input and returns a quadtree that is the mirror image with respect to the horizontal line passing through the center of the image.
- `flipDiagonal(quadtree)` takes a quadtree as input and returns a quadtree that is the mirror image with respect to the diagonal line passing through the NE and SW corners of the image. (Note that this function may *seem* like it's the same as `rotateRight`, but, in general, these two functions will not always produce the same output! Rotating right and flipping diagonally are not the same.)

These functions should be very short. About 5 lines per function should suffice. *These functions should operate on quadtrees and should not convert the quadtree to a binary array at any point.* These functions will be fast because they are operating on the compact quadtree representation rather than the generally much larger 2D array representation.

Here are some examples using our running `array1` example:

```
>>> array1
[[0, 0, 0, 0, 0, 0, 1, 1], [0, 0, 0, 0, 1, 1, 0, 0], [0, 0, 0, 0, 0, 0, 1, 1], [0, 0, 0, 0,
>>> renderImage(array1) # <-- render the image to see what it looks like
>>> qtree1
[0, [[0, 0, 1, 1], [1, 1, 0, 0], 0, 1], 1, 1]
>>> invert1 = invert(qtree1)
>>> invert1
[1, [[1, 1, 0, 0], [0, 0, 1, 1], 1, 0], 0, 0]
>>> renderImage(makeArray(invert1, 8)) # <-- render the inverted image (first convert to 8x8)
>>> rotate1 = rotateRight(qtree1)
>>> renderImage(makeArray(rotate1, 8)) # <-- render the rotated image (first convert to 8x8)
```

Here are examples for `flipHorizontal` and `flipDiagonal`:

```
>>> flipH1 = flipHorizontal(qtree1)
>>> flipH1
[1, 1, 0, [0, 1, [1, 1, 0, 0], [0, 0, 1, 1]]]
>>> flipD1 = flipDiagonal(qtree1)
>>> flipD1
[1, [1, [0, 1, 0, 1], 0, [1, 0, 1, 0]], 1, 0]
```

## One More Test, and Making Your Own Images!

Finally, download the alien.png (download by right-clicking, or CTRL-clicking on a Mac) which is a 32x32 image) and spam.png which is a 256x256 image. You can extract the array representation of a black-and-white .png file using the function `pngToArray(filename)` that we provided in the `quadtree.py` file. That function takes a string as input (where the string is the name of the file) and returns a 2D array representation. Test all of your functions on these images. You should be able to build a quadtree representation of each image; invert, rotate, and flip the image, and render the image on the screen to see that these functions work.

If you'd like to generate your own .png images, you can use an online tool such as Piskel to create your own pattern. Piskel's default images are 32x32. After you're done drawing, click on the icon on the far right that looks like a mountain. That's the export button. Choose to export as a .png and then click the button to download as a .png. Before you use that image, you'll need to convert it to a format that our .png reader can read. On a mac, you can double-click the image to open it in Preview. Then, choose "Export" and select png. There will be a little check-box labelled "Alpha". De-select "Alpha" and save your image. Now, our `pngToArray(filename)` will work! On a PC, you may not need to do the analogous thing. If you have trouble with this, come visit Ran in office hours and we'll sort things out. Of course, generating your own .png files is totally optional - so don't worry if you choose not to do this.

Put that file in the directory with your code and then you can extract the array using the provided `pngToArray` function described above.

Website design by Madeleine Masser-Frye and Allen Wu

# Chapter 6: Fun and Games with OOPs: Object-Oriented Programs — cs5book 1 documentation

## Navigation

- [index](#)
- [next |](#)
- [previous |](#)
- [cs5book 1 documentation »](#)

## Chapter 6: Fun and Games with OOPs: Object-Oriented Programs

*I paint objects as I think them, not as I see them.*

—Pablo Picasso

### 6.1 Introduction

In this chapter we'll develop our own “killer app”: A 3D video game called “Robot versus Zombies.” By the end of this chapter you'll have the tools to develop all kinds of interactive 3D programs for games, scientific simulations, or whatever else you can imagine.

We're sneaky! The *true* objective of this chapter is to demonstrate a beautiful and fundamental concept called *object-oriented programming*. Object-oriented programming is not only the secret sauce in 3D graphics and video games, it's widely used in most modern large-scale software projects. In this chapter, you'll learn about some of the fundamental ideas in object-oriented programming. And, yes, we'll write that video game too!

### 6.2 Thinking Objectively

Before we get to the Robot versus Zombies video game, let's imagine the following scenario. You're a summer intern at Lunatix Games, a major video game developer. One of their popular games, Lunatix Lander, has the player try to land a spaceship on the surface of a planet. This requires the player to fire thrusters to align the spaceship with the landing site and slow it down to a reasonable landing speed. The game shows the player how much fuel remains and how much fuel is required for certain maneuvers.

As you test the game, you notice that it often reports that there is not enough fuel to perform a key maneuver when, in fact, you are certain there should be just the right amount of fuel. Your task is to figure out what's wrong and find a way to fix it.

Each of the rocket's two fuel tanks has a capacity of 1000 units; the fuel gauge for each tank reports a value between 0 and 1.0, indicating the fraction of the capacity remaining in that tank. Here is an example of one of your tests, where `fuelNeeded` represents the fraction of a 1000 unit tank required to perform the maneuver, and `tank1` and `tank2` indicate the fraction of the capacity of each of the two tanks. The last statement is checking to see if the total amount of fuel in the two tanks equals or exceeds the fuel needed for the maneuver.

```
>>> fuelNeeded = 42.0/1000
>>> tank1 = 36.0/1000
>>> tank2 = 6.0/1000
>>> tank1 + tank2 >= fuelNeeded
False
```



*Bummer!*

Notice that  $\frac{36}{1000} + \frac{6}{1000} = \frac{42}{1000}$ ) and the fuel needed is exactly  $(\frac{42}{1000})$ . Strangely though, the code reports that there is not enough fuel to perform the maneuver, dooming the ship to crash.

When you print the values of `fuelNeeded`, `tank1`, `tank2`, and `tank1 + tank2` you see the problem:



*To be precise, imprecision occurs because computers use only a fixed number of bits of data to represent data. Therefore, only a finite number of different quantities can be stored. In particular, the fractional part of a floating-point number, the mantissa, must be rounded to the nearest one of the finite number of values that the computer can store, resulting in the kinds of unexpected behavior that we see here.*

```
>>> fuelNeeded
0.04200000000000003
>>> tank1
0.03599999999999997
>>> tank2
0.00600000000000001
>>> tank1 + tank2
0.04199999999999996
```



*A rational thing to have!*

This example of *numerical imprecision* is the result of the inherent error that arises when computers try to convert fractions into floating-point numbers (numbers with a decimal-point representation). However, assuming that all of the quantities that you measure on your rocket are always rational numbers—that is fractions with integer numerators and denominators—this imprecision problem can be avoided! How? Integers don’t suffer from imprecision. So, for each rational number, we can store its integer numerator and denominator and then do all of our arithmetic with integers.



*Or even a fraction of them all.*

For example, the rational number  $(\frac{36}{1000})$  can be stored as the pair of integers  $(36)$  and  $(1000)$  rather than converting it into a floating-point number. To compute  $(\frac{36}{1000} + \frac{6}{1000})$  we can compute  $(36+6 = 42)$  as the numerator and  $(1000)$  as the denominator. Comparing this to the `fuelNeeded` value of  $(\frac{42}{1000})$  involves separately comparing the numerators and denominators, which involves comparing integers and is thus free from numerical imprecision.

So, it would be great if Python had a way of dealing with rational numbers as pairs of integers. That is, we would like to have a “rational numbers” type of data (or *data type*, as computer scientists like to call it) just as Python has an integer data type, a string data type, and a list data type (among others). Moreover, it would be great if we could do arithmetic and comparisons of these rational numbers just as easily as we can do with integers.

The designers of Python couldn’t possibly predict all of the different data types that one might want. Instead, Python (like many other languages) has a nice way to let you, the programmer, define your own new types and then use them nearly as easily as you use the built-in types such as integers, strings, and lists.

This facility to define new types of data is called *object-oriented programming* or OOP and is the topic of this chapter.

### 6.3 The Rational Solution

Let's get started by defining a rational number type. To do this, we build a Python "factory" for constructing rational numbers. This factory is called a *class* and it looks like this:

```
class Rational:
 def __init__(self, num, denom):
 self.numerator = num
 self.denominator = denom
```

Don't worry about the weird syntax; we'll come back to that in a moment when we take a closer look at the details. For now, the big idea is that once we've written this `Rational` class (and saved it in a file with the same name but with the suffix `.py` at the end, in this case `Rational.py`) we can "manufacture" – or more technically *instantiate* – new rational numbers to our heart's content. Here's an example of calling this "factory" to instantiate two rational numbers  $\frac{36}{1000}$  and  $\frac{6}{1000}$ :

```
r1 = Rational(36, 1000)
r2 = Rational(6, 1000)
```

What's going on here? When Python sees the instruction

```
r1 = Rational(36, 1000)
```



*Perhaps this is self -ish of Python.*

it does two things. First, it instantiates an empty object which we'll call `self`. Actually, `self` is a *reference* to this empty object as shown in Figure 6.1.

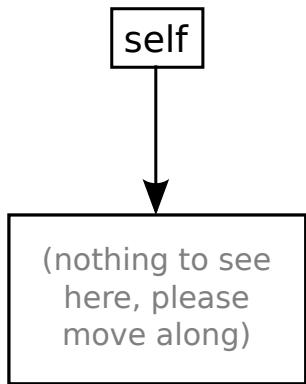


Figure 6.1: `self` refers to a new empty object. It's only empty for a moment!

Next, Python looks through the `Rational` class definition for a function named `_init_` (notice that there are two underscore characters before and after the word "init"). That's a funny name, but it's a convention in Python. Notice that in the definition of `_init_` above, that function seems to take *three* arguments, but the line `r1 = Rational(36, 1000)` has only supplied *two*. That's weird, but—as you may have guessed—the first argument is passed in automatically by Python and is a reference to the new empty `self` object that we've just had instantiated for us.

The `_init_` function takes the reference to our new empty object called `self`, and it's going to add some data to that object. The values  $\frac{36}{1000}$  are passed in to `_init_` as `num` and `denom`, respectively. Now, when the `_init_` function executes the line `self.numerator = num`, it says, "go into the object referenced by `self`, give it a variable called `numerator`, and give that variable the value that was passed in as `num`." Similarly, the line `self.denominator = denom` says "go into the object referenced by `self`, give it a variable called `denominator`, and give that variable the value that was passed in as `denom`." Note that the names `num`, `denom`, `numerator`, and `denominator` are not special—they are just the names that we chose.



Python uses the name “*attributes*” for the variables that belong to a class. Some other languages use names like “*data members*”, “*properties*”, “*fields*”, and “*instance variables*” for the same idea.

The variables `numerator` and `denominator` in the `__init__` function are called **attributes** of the `Rational` class. A class can have as many attributes as you wish to define for it. It’s pretty clear that a rational number class would have to have at least these two attributes!

The last thing that happens in the line

```
r1 = Rational(36, 1000)
```

is that the variable `r1` is now assigned to be a reference to the object that Python just created and we initialized. We can see the contents of the rational numbers as follows:

```
>>> r1.numerator
36
```

We used the “dot” in the function `__init__` as well, when we said `self.numerator = num`. The “dot” was doing the same thing there. It said, “go into the `self` object and look at the attribute named `n numerator`.” Figure 6.2 shows the situation now.

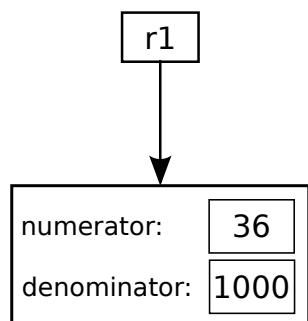


Figure 6.2: `r1` refers to the `Rational` object with its `numerator` and `denominator`.

We note that in our figures in this chapter, we’re representing memory in a somewhat different (and simpler) way than we used in Chapter 5. For example, Figure 6.2 shows the values of `numerator` and `denominator` as if they were stored inside the variables. In reality, as we saw in Chapter 5, the integer values would be somewhere else in memory, and the variables would store references to those values.

In our earlier example, we “called” the `Rational` “factory” twice to instantiate two different rational numbers in the example below:

```
r1 = Rational(36, 1000)
r2 = Rational(6, 1000)
```

The first call, `Rational(36, 1000)` instantiated a rational number, `self`, with `numerator \(36\)` and `denominator \(1000\)`. This was called `self`, but then we assigned `r1` to refer to this object. Similarly, the line `r2.numerator` is saying, “go to the object called `r2` and look at its attribute named `n numerator`.” It’s important to keep in mind that since `r1` and `r2` are referring to two different objects, each one has its “personal” `numerator` and `denominator`. This is shown in Figure 6.3.

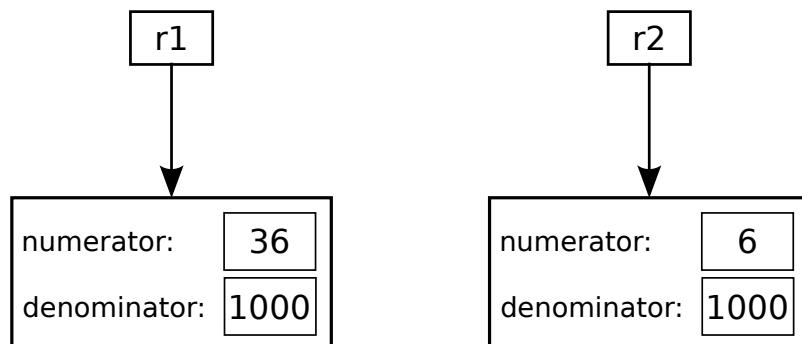


Figure 6.3: Two Rational numbers with references `r1` and `r2`.



*This would be a short chapter if that was the whole story!*

Let's take stock of what we've just seen. First, we defined a factory—technically known as a *class*—called `Rational`. This `Rational` class describes a template for manufacturing a new type of data. That factory can now be used to instantiate a multitude of items—technically known as *objects*—of that type. Each object will have its own variables—or “attributes”—in this case numerator and denominator, each with their own values.

That's cute, but is that it? Remember that our motivation for defining the `Rational` numbers was to have a way to manipulate (add, compare, etc.) rational numbers without having to convert them to the floating-point world where numerical imprecision can cause headaches (and rocket failures, and worse).

Python's built-in data types (such as integers, floating-point numbers, and strings) have the ability to be added, compared for equality, etc. We'd like our `Rational` numbers to have these abilities too! We'll begin by adding a function to the `Rational` class that will allow us to add one `Rational` to another and return the sum, which will be a `Rational` as well. A function defined inside a class has a special fancy name—it's called a *method* of that class. The `_init_` method is known as the *constructor* method.

Our `add` method in the `Rational` class will be used like this:

```
>>> r1.add(r2)
```

This should return a `Rational` number that is the result of adding `r1` and `r2`. So, we should be able to write:

```
>>> r3 = r1.add(r2)
```

Now, `r3` will refer to the new `Rational` number returned by the `add` method. The syntax here may struck you as funny at first, but humor us; we'll see in a moment why this syntax is sensible.

Let's write this `add` method! If  $r1 (= \frac{a}{b})$  and  $r2 (= \frac{c}{d})$  then  $r1+r2 (= \frac{ad+bc}{bd})$ . Note that the resulting fraction might be simplified by dividing out by terms that are common to the numerator and the denominator, but let's not worry about that for now. Here is the `Rational` class with its shiny new `add` method:

```
class Rational:
 def __init__(self, num, denom):
 self.numerator = num
 self.denominator = denom

 def add(self, other):
 newNumerator = self.numerator * other.denominator +
 self.denominator * other.numerator
 newDenominator = self.denominator*other.denominator
 return Rational(newNumerator, newDenominator)
```

What's going on here!? Notice that the `add` method takes in two arguments, `self` and `other`, while our examples above showed this method taking in a single argument. (Stop here and think about this. This is analogous to what we saw earlier with the `_init_` method.)

To sort this all out, let's consider the following sequence:

```
>>> r1 = Rational(1, 2)
>>> r2 = Rational(1, 3)
>>> r3 = r1.add(r2)
```

The instruction `r1.add(r2)` does something funky: It calls the `Rational` class's `add` method. It seems to pass in just `r2` to the `add` method but that's an optical illusion! In fact, it passes in two values: *first* it *automatically* passes a reference to `r1` and *then* it passes in `r2`. This is great, because our code for the `add` method is expecting two arguments: `self` and `other`. So, `r1` goes into the `self` “slot” and `r2` goes into the `other` slot. Now, the `add` method can operate on those two `Rational` numbers, add them, construct a new `Rational` number representing their sum, and then return that new object.



“Snazzy” is a technical term.

Here’s the key: Consider some arbitrary class `Blah`. If we have an object `myBlah` of type `Blah`, then `myBlah` can invoke a method `foo` with the notation `myBlah.foo(arg1, arg2, ..., argN)`. The method `foo` will receive *first* a reference to the object `myBlah` followed by all of the `N` arguments that are passed in explicitly. Python just knows that the first argument is always the automatically-passed-in reference to the object before the “dot”. The beauty of this seemingly weird system is that the method is invoked by an object and the method “knows” which object invoked it. Snazzy!

After performing the above sequence of instructions, we could type:

```
>>> r3.numerator
>>> r3.denominator
```

What would we see? We’d see the numerator and denominator of the Rational number `r3`. In this case, the numerator would be `\(5\)` and the denominator would be `\(6\)`.

Notice that instead of typing `r3 = r1.add(r2)` above, we could have instead have typed `r3 = r2.add(r1)`. What would have happened here? Now, `r2` would have called the `add` method, passing `r2` in for `self` and `r1` in for `other`. We would have gotten the same result as before because addition of rationals is commutative.

## 6.4 Overloading



“Spiffy” is yet another technical term.

So far, we have built a basic class for representing rational numbers. It’s neat and useful, but now we’re about to make it even spiffier.

You probably noticed that the syntax for adding two Rational numbers is a bit awkward. When we add two integers, like `\(42\)` and `\(47\)`, we certainly don’t type `42.add(47)`, we type `42+47` instead.

It turns out that we can use the operator “`+`” to add Rational numbers too! Here’s how: We simply change the name of our `add` method to `_add_`. Those are two underscore characters before and two underscore characters after the word `add`. Python has a feature that says “if a function is named `_add_` then when the user types `r1 + r2`, I will translate that into `r1._add_(r2)`.” How does Python know that the addition here is addition of Rational numbers rather than addition of integers (which is built-in)? It simply sees that `r1` is a Rational, so the “`+`” symbol must represent the `_add_` method in the Rational class. We could similarly define `_add_` methods for other classes and Python will figure out which one applies based on the type of data in front of the “`+`” symbol.



This is “good” overloading. “Bad” overloading involves taking more than 18 credits in a term.

This feature is called *overloading*.

We have overloaded the “`+`” symbol to give it a meaning that depends on the context in which it is used. Many, though not all, object-oriented programming languages support overloading. In Python, overloading addition is just the tip of the iceberg. Python allows us to overload all of the normal arithmetic operators and all of the comparison operators such as “`==`”, “`!=`”, “`<`”, among others.

Let’s think for a moment about comparing rational numbers for equality. Consider the following scenario, in which we have two different Rational objects and we compare them for equality:

```
>>> r1 = Rational(1, 2)
>>> r2 = Rational(1, 2)
>>> r1 == r2
False
```



“Blob” really is a technical term!

Why did Python say “`False`”? The reason is that even though `r1` and `r2` look the same to us, each one is a reference to a *different* object. The two objects have identical contents, but they are different nonetheless, just as two identical twins are two different people. Another way of seeing this is that `r1` and `r2` refer to different blobs of memory, and when Python sees them ask if `r1 == r2` it says “Nope! Those two references are not to the same memory location.” Since we haven’t told Python how to compare `Rationals` in any other way, it simply compares `r1` and `r2` to see if they are referring to the very same object.

So let’s “overload” the “`==`” symbol to correspond to a function that will do the comparison as we intend. We’d like for two rational numbers to be considered equal if their ratios are the same, even if their numerators and denominators are not the same. For example  $\left(\frac{1}{2} = \frac{42}{84}\right)$ . One way to test for equality is to use the “cross-multiplying” method that we learned in grade school: Multiply the numerator of one of the fractions by the denominator of the other and check if this is equal to the other numerator-denominator product. Let’s first write a method called `_eq_` to include in our `Rational` number class to test for equality.

```
def __eq__(self, other):
 return self.numerator * other.denominator ==
 self.denominator * other.numerator
```

Now, if we have two rational numbers such as `r1` and `r2`, we could invoke this method with `r1.__eq__(r2)` or with `r2.__eq__(r1)`. But because we’ve used the special name `_eq_` for this method, Python will know that when we write `r1 == r2` it should be translated into `r1.__eq__(r2)`. There are many other symbols that can be overloaded. (To see a complete list of the methods that Python is happy to have you overload, go to <http://docs.python.org/2/reference/datamodel.html#special-method-names>.)



This example serves to doubly underscore the beauty of overloading!

For example, we can overload the “`>=`” symbol by defining a method called `_ge_` (which stands for greater than or equal). Just like `_eq_`, this method takes two arguments: A reference to the *calling object* that is passed in automatically (`self`) and a reference to another object to which we are making the comparison. So, we could write our `_ge_` method as follows:

```
def __ge__(self, other):
 return self.numerator * other.denominator >=
 self.denominator * other.numerator
```

Notice that there is only a tiny difference between how we implemented our `_eq_` and `_ge_` methods. Take a moment to make sure you understand why `_ge_` works.



We hope that you aren’t feeling overloaded at this point. We’d feel bad if you “object”ed to what we’ve done here.

Finally, let’s revisit the original fuel problem with which we started the chapter. Recall that due to numerical imprecision with floating-point numbers, we had experienced mission failure:

```
>>> fuelNeeded = 42.0/1000
```

```
>>> tank1 = 36.0/1000
>>> tank2 = 6.0/1000
>>> tank1 + tank2 >= fuelNeeded
False
```

In contrast, we can now use our slick new Rational class to save the mission!

```
>>> fuelNeeded = Rational(42, 1000)
>>> tank1 = Rational(36, 1000)
>>> tank2 = Rational(6, 1000)
>>> tank1 + tank2 >= fuelNeeded
True
```

Mission accomplished!

## 6.5 Printing an Object

Our Rational class is quite useful now. But check this out:



*0x6b918!?* What the heck is that?!

```
>>> r1 = Rational(1, 2)
>>> r2 = Rational(1, 3)
>>> r3 = r1 + r2
>>> r3
<Rational.Rational instance at 0x6b918>
>>> print(r3)
<Rational.Rational instance at 0x6b918>
```

Notice the strange output when we asked for `r3` or when we tried to `print(r3)`. In both cases, Python is telling us, “`r3` is a Rational object and I’ve given it a special internal name called `0x blah, blah, blah.`”

What we’d really like, at least when we `print(r3)`, is for Python to display the number in some nice way so that we can see it! You may recall that Python has a way to “convert” integers and floating-point numbers into strings using the built-in function `str`. For example:

```
>>> str(1)
'1'
>>> str(3.142)
'3.142'
```

So, since the `print` function wants to print strings, we can print numbers this way:

```
>>> print(str(1))
1
>>> print("My favorite number is " + str(42))
My favorite number is 42
```

In fact, other Python types such as lists and dictionaries also have `str` functions:

```
>>> myList = [1, 2, 3]
>>> print("Here is a very nice list: " + str(myList))
Here is a very nice list: [1, 2, 3]
```

Python lets us define a `str` function for our own classes by overloading a special method called `_str_`. For example, for the `Rational` class, we might write the following `_str_` method:

```
def __str__(self):
 return str(self.numerator) + "/" + str(self.denominator)
```

What is this function returning? It’s a string that contains the numerator followed by a forward slash followed by the denominator. When we type `print(str(r3))`, Python will invoke this `_str_` method. That function first calls the `str` function on `self.numerator`. Is the call `str(self.numerator)` recursive? It’s not! Since `self.numerator` is an integer,

Python knows to call the `str` method for integers here to get the string representation of that integer. Then, it concatenates to that string another string containing the forward slash, `/`, indicating the fraction line. Finally, to that string it concatenates the string representation of the denominator. Now, this string is returned. So, in our running example from above where `r3` is the rational number  $\frac{5}{6}$ , we could use our `str` method as follows:

```
>>> r3
<Rational.Rational instance at 0x6b918>
>>> print("Here is r3: "+str(r3))
Here is r3: 5/6
```

Notice that in the first line when we ask for `r3`, Python just tells us that it is a reference to `Rational` object. In the third line, we ask to convert `r3` into a string for use in the `print` function. By the way, the `_str_` method has a closely related method named `_repr_` that you can read about on the Web.

## 6.6 A Few More Words on the Subject of Objects

Let's say that we wanted (for some reason) to change the numerator of `r1` from its current value to 42. We could simply type

```
r1.numerator = 42
```

In other words, the internals of a `Rational` object can be changed. Said another way, the `Rational` class is mutable. (Recall our discussion of mutability in Chapter 5.) *In Python, classes that we define ourselves are mutable (unless we add fancy special features to make them immutable).* To fully appreciate the significance of mutability of objects, consider the following pair of functions:

```
def foo():
 r = Rational(1, 3)
 bar(r)
 print r

def bar(number):
 number.numerator += 1
```

What happens when we invoke function `foo`? Notice that function `bar` is not returning anything. However, the variable `number` that it receives is presumably a `Rational` number, and `foo` increments the value of this variable's numerator. Since user-defined classes such as `Rational` are mutable, this means that the `Rational` object that was passed in will have its numerator changed!

How does this actually work? Notice that in the function `foo`, the variable `r` is a reference to the `Rational` number  $\frac{1}{3}$ . In other words, this `Rational` object is somewhere in the computer's memory and `r` is the address where this blob of memory resides. When `foo` calls `bar(r)` it is passing the reference (the memory location) `r` to `foo`. Now, the variable `number` is referring to that memory location. When Python sees `number.numerator += 1` it first goes to the memory address referred to by `number`, then uses the "dot" to look at the `nominator` part of that object, and increments that value by 1. When `bar` eventually returns control to the calling function, `foo`, the variable `r` in `foo` is still referring to that same memory location, but now the `nominator` in that memory location has the new value that `bar` set.



*Our legal team objected to us using the word "everything" here, but it's close enough to the truth that we'll go with it.*

This brings us to a surprising fact: *Everything in Python is an object!* For example, Python's list datatype is an object. "Wait a second!" we hear you exclaim. "The syntax for using lists doesn't look anything like the syntax that we used for using `Rationals`!" You have a good point, but let's take a closer look.

For the case of `Rationals`, we had to make a new object this way:

```
r = Rational(1, 3)
```

On the other hand, we can make a new list more simply:

```
myList = [42, 1, 3]
```

In fact, though, this list notation that you've grown to know and love is just a convenience that the designers of Python have provided for us. It's actually a shorthand for this:

```
myList = list()
myList.append(42)
myList.append(1)
myList.append(3)
```

Now, if we ask Python to show us myList it will show us that it is the list [42, 1, 3]. Notice that the line myList = list() is analogous to r = Rational(1, 3) except that we do not provide any initial values for the list. Then, the append method of the list class is used to append items onto the end of our list. Lists are mutable, so each of these appends changes the list!

Indeed, the list class has many other methods that you can learn about online. For example, the reverse method reverses a list. Here's an example, based on the myList list object that we created above:

```
>>> myList
[42, 1, 3]
>>> myList.reverse()
>>> myList
[3, 1, 42]
```

Notice that this reverse method is not returning a new list but rather mutating the list on which it is being invoked.

Before moving on, let's reflect for a moment on the notation that we've seen for combining two lists:

```
>>> [42, 1, 3] + [4, 5]
[42, 1, 3, 4, 5]
```

How do you think that the "+" symbol works here? You got it—it's an overloaded method in the list class! That is, it's the `_add_` method in that class!

Strings, dictionaries, and even integers and floats are all objects in Python! However, a few of these built-in types - such as strings, integers, and floats - were designed to be immutable. Recall from the previous chapter that this means that their internals cannot be changed. You can define your own objects to be immutable as well, but it requires some effort and it's rarely necessary, so we won't go there.

## 6.7 Getting Graphical with OOPs



*Warning! This section contains graphical language!*

We've seen that object-oriented programming is elegant and, hopefully, you now believe that it's useful. But what about 3D graphics and our video game? That's where we're headed next!

To get started, you'll need to get the "Jupyter VPython" 3D graphics system for Python 3. You can install it by running `pip install vpython` at the command prompt. You should also make sure you have the most recent version of Anaconda installed.

Once you have VPython installed, run `jupyter notebook` from the command prompt. A window should pop up in your browser. In that window, select the New button on the upper right, then select VPython under Notebooks.

In the new window that pops up, you'll need to import "vpython" as follows:

```
>>> from vpython import *
```

Next, in the same cell, type:

```
>>> b = box()
```

Then hit the Run button. A white box should pop up in a small window below the cell (the box you typed into). If you change your code and want to re-run your code, you will need to click the Restart the Kernel button

(near the Run button), then click the red Restart button in the box that pops up. Then wait for a blue Kernel Ready message to flash in the upper right hand corner, and then press Run.



*In the display window (where the box is displayed), click and drag with the right mouse button (hold down the command key on a Macintosh). Drag left or right, and you rotate around the scene. To rotate around a horizontal axis, drag up or down. Click and drag up or down with the middle mouse button to move closer to the scene or farther away (on a 2-button mouse, hold down the left and right buttons; on a 1-button mouse, hold down the Option key).*

What you'll see now on the screen is a white box. It might look more like a white square, so rotate it around to see that it's actually a 3D object.

As you may have surmised, `box` is a class defined in the `vpython` module. The command

```
>>> b = box()
```

invoked the constructor to create a new `box` object and we made `b` be the name, or more precisely “a reference”, to that box.

Just like our `Rational` number class had `numerator` and `denominator` attributes, the `box` class has a number of attributes as well. Among these are the box’s length, height, and width; its position, its color, and even its material properties. Try changing these attributes at the command line as follows:

```
>>> b.length = 0.5 # the box's length just changed
>>> b.width = 2.0 # the box's width just changed
>>> b.height = 1.5 # the box's height just changed
>>> b.color = vector(1.0, 0.0, 0.0) # the box turned red
>>> b.texture = textures.wood # it's wood-grained!
>>> b.opacity = 0.42 # it's translucent!
```

When we initially made our `b = box()` it had “default” values for all of these attributes. The `length`, `width`, and `height` attributes were all 1.0, the `color` attribute was white, and the `textures` attribute was a boring basic one. Notice that some of the attributes of a `box` are fairly obvious: `length`, `width`, and `height` are all numbers. However, the `color` attribute is weird. Similarly, the `texture` property was set in a strange way. Let’s take a closer look at just one of these attributes and you can read more about others later on the VPython documentation website. (You will want to look up ‘Glowscript documentation’. If you only search for ‘vpython documentation’, you will only find the documentation for Classic VPython, the predecessor of Jupyter VPython.)

Clear the cell you are in (except for the import statement). Let’s make a new box and ask Python for its `color` attribute by running the following code:

```
>>> c = box()
>>> c.color
(1.0, 1.0, 1.0)
```

VPython represents color using a tuple with three values, each between `\(0.0\)` and `\(1.0\)`. The three elements in this tuple indicate how much red (from `\(0.0\)`, which is none, to `\(1.0\)`, which is maximum), green, and blue, respectively, is in the color of the object.



*If your roommate sees you staring at your fingers, just explain that you are doing something very technical.*

So, `\((1.0, 1.0, 1.0)\)` means that we are at maximum of each color, which amounts to bright white. The tuple `\((1.0, 0.0, 0.0)\)` is bright red and the tuple `\((0.7, 0.0, 0.4)\)` is a mixture of quite a bit of red and somewhat less blue.

The `box` class has another attribute called `pos` that stores the position of the center of the box. The coordinate system used by VPython is what’s called a “right-handed” coordinate system: If you take your right hand and

stick out your thumb, index finger, and middle finger so that they are perpendicular to one another with your palm facing you, the positive  $\langle x \rangle$  axis is your thumb, the positive  $\langle y \rangle$  axis is your index finger, and the positive  $\langle z \rangle$  axis is your middle finger.



*When you used your mouse to rotate the scene, that actually rotated the entire coordinate system.*

Said another way, before you start rotating in the display window with your mouse, the horizontal axis is the  $\langle x \rangle$  axis, the vertical axis is the  $\langle y \rangle$  axis, and the  $\langle z \rangle$  axis points out of the screen.

Take a look at the position of the box by adding a call to `c.pos`. When you re-run the code, you'll see this:

```
>>> c.pos
<0.000000, 0.000000, 0.000000>
```



*Is vector class also called linear algebra?*

VPython has a class called `vector`, and `pos` is an object of this type, as we can tell by the vector notation. Nice! The `box` class is defined using the `vector` class. Using a `vector` class inside the `box` class is, well, very classy! “OK,” we hear you concede grudgingly, “but what’s the point of a `vector`? Why couldn’t we just use a tuple or a list instead?” Here’s the thing: The `vector` class has some methods defined in it for performing `vector` operations. For example, the `vector` class has an overloaded addition operator for adding two `vector`s:

```
>>> v = vector(1, 2, 3)
>>> w = vector(10, 20, 30)
>>> v + w
<11.000000, 22.000000, 33.000000>
```

This class has many other `vector` operations as well. For example, the `norm()` method returns a `vector` that points in the same direction but has magnitude (length) 1:

```
>>> u = vector(1, 1, 0)
>>> u.norm()
<0.707107, 0.707107, 0.000000>
```



*Take a look at the rich set of other `vector` operations on the VPython web site in order to dot your i’s and cross your t’s or, more precisely, to dot your scalars and cross your vectors!*

So, while we *could* have represented `vector`s using lists, we wouldn’t have a nice way of adding them, normalizing them, and doing all kinds of other things that `vector`s like to do.

But, our objective for now is to change our `box`’s `pos` `vector` in order to move it. We can do this, for example, as follows:

```
>>> c.pos = vector(0, 1, 2)
```

While we can always create a `box` and change its attributes afterwards, sometimes it’s convenient to just set the attributes of the `box` at the time that the `box` is first instantiated. The `box` class constructor allows us to set the values of attributes at the time of construction like this:

```
>>> d = box(length = 0.5, width = 2.0, height = 1.5, color = vector(1.0, 0.0, 0.0))
```

Whatever attributes we don’t specify will get their default values. For example, since we didn’t specify a `vector` value for `pos`, the `box`’s initial position will be the origin.

VPython has lots of other shape classes beyond boxes including spheres, cones, cylinders, among others. While these objects have their own particular attributes (for example a sphere has a radius), all VPython objects share some useful methods. One of these methods is called `rotate`. Not surprisingly, this method rotates its object. Let's take `rotate` out for a spin!

Try this with the box `b` that we defined above:

```
>>> d.rotate(angle=pi/4)
```

We are asking VPython to rotate the box `b` by  $\frac{\pi}{4}$  radians. The rotation is, by default, specified in radians about the  $(x)$  axis.

Now, let's put this all together to write a few short VPython programs just to flex our 3D graphics muscles. First, let's write a *very* short program that rotates a red box forever (clean out your cell before copying this code in):

```
from vpython import *

def spinBox():
 myBox = box(color = vector(1.0, 0.0, 0.0))
 while True:
 # Slow down the animation to 60 frames per second.
 # Change the value to see the effect!
 rate(60)
 myBox.rotate(angle=pi/100)
```

Second, take a look at the program below. Try to figure out what it's doing before you run it.

```
from vpython import *
import random

def spinboxes():
 boxList = []
 for boxNumber in range(0,10):
 x = random.randint(-5,5)
 y = random.randint(-5,5)
 z = random.randint(-5,5)
 red = random.random() # random number between 0 and 1
 green = random.random() # random number between 0 and 1
 blue = random.random() # random number between 0 and 1
 newBox = box()
 newBox.pos = vector(x, y, z)
 newBox.color = vector(red, green, blue)
 newBox.axis =
 random.choice([vector(1,0,0),vector(0,1,0),vector(0,0,1)]) # makes boxes rotate in random directions
 boxList.append(newBox)
 # the physics loop, which updates the world
 while True:
 rate(60)
 for myBox in boxList:
 myBox.rotate(angle=pi/100)
spinboxes()
```

This is very cool! We now have a list of objects and we can go through that list and rotate each of them.

## 6.8 Robot and Zombies, Finally!

It's time to make our video game! The premise of our game is that we will control a robot that moves on the surface of a disk (a large flat cylinder) populated by zombies. The player will control the direction of the robot with a GUI (Graphical User Interface) that YOU will make! Our GUI should have the following: Two buttons to speed up and slow down the robot, two buttons to turn the robot left and right, and one button to quit the program. (We will give you the code for the GUI so that we can focus on making our robots and alien.) Meanwhile, the zombies will each move and turn independently by random amounts.

The program will be in an infinite loop. At each step, the player's robot will take a small step forward. The

buttons will simultaneously control the robot's turn amounts. Similarly, each zombie will turn a random amount and then take a small step forward. Our game will have no particular objective, but you can add one later if you like. Perhaps, the objective is to run into as many zombies as possible – or perhaps avoid them.

To get started, we wish to define a player robot class that we'll use to manufacture (“instantiate”) the player's robot and another class that allows to instantiate zombies. It particularly makes sense to have a zombie *class* because a class allows us to instantiate many *objects* – and we indeed plan to have many zombies!

In fact, the player's robot and zombies have a lot in common. They are 3D entities that should be able to move forward and turn. Because of this commonality, we would be replicating a lot of effort if we were to define a robot class and a zombie class entirely separately. On the other hand, the two classes are not going to be identical because the player's robot looks different from zombies (we hope) and because the robot will be controlled by the player while the zombies move on their own.

So, here's the key idea: We'll define a class – let's call it `GenericBot` – that has all of the attributes that any entity in our game – a player robot or a zombie – should have. Then, we'll define a `PlayerBot` class and a `ZombieBot` class both of which “inherit” all of the attributes and methods of the `GenericBot` and *add* the special extras (e.g., how their bodies look) that differentiate them.

Our `GenericBot` class will have a constructor, an `_init_` method, that takes as input the initial position of the bot, its initial heading (the direction in which it is pointing), and its speed (the size of each step that it makes when we ask it to move forward). Here's the code; we'll dissect it below.

```
from vpython import *
import math
import random

class GenericBot:
 def __init__(self, position = vector(0, 0, 0),
 heading = vector(0, 0, 1), speed = 1):
 self.position = position
 self.heading = heading.norm()
 self.speed = speed
 self.parts = []

 def update(self):
 self.turn(0)
 self.forward()

 def turn(self, angle):
 # convert angle from degrees to radians (VPython
 # assumes all angles are in radians)
 theta = math.radians(angle)
 self.heading = rotate(self.heading, angle = theta, axis = vector(0, 1, 0))
 for part in self.parts:
 part.rotate(angle = theta, axis = vector(0, 1, 0),
 origin = self.position)

 def forward(self):
 self.position += self.heading * self.speed
 for part in self.parts:
 part.pos += self.heading * self.speed
```

Let's start with the `_init_` method – the so-called “constructor” method. It takes three input arguments: A `position` (a VPython `vector` object indicating the bot's initial position), a `heading` (a vector indicating the direction that the bot is initially pointing), and `speed` (a number indicating how far the bot moves at each update step). Notice that the line:

```
def __init__(self, position = vector(0, 0, 0),
 heading = vector(0, 0, 1), speed = 1):
```

provides *default values* for these inputs. This means that if the user doesn't provide values for these input arguments, the inputs will be set to these values. (You may recall that the `box` class had default arguments as well. We could either define a new `box` with `b = box()` in which case we got the default values or we could specify

our own values for these arguments.) If the user provides only some of the input arguments, Python will assume that they are the arguments from left-to-right. For example, if we type

```
>>> mybot = GenericBot(vector(1, 2, 3))
```

then Python assumes that `vector(1, 2, 3)` should go into the `position` variable and it uses the default values for `heading` and `speed`. If we type

```
>>> mybot = GenericBot(vector(1, 2, 3), vector(0, 0, 1))
```

then the first vector goes into the `position` argument and the second goes into the `heading` argument. If we want to provide values in violation of the left-to-right order, we can always tell Python which value we are referring to like this:

```
>>> mybot = GenericBot(heading=vector(0, 1, 0))
```



*It was de-fault of de-authors for this de-gression!*

Now, Python sets the `heading` to the given value and uses the default values for the other arguments.

OK, so much for defaults! The `_init_` method then sets its position (`self.position`), heading (`self.heading`), speed (`self.speed`), and parts (`self.parts`) attributes. The `self.heading` is normalized to make the length of the vector a unit vector (a vector of length 1) using the `vector` class's `norm()` method. The `self.parts` list will be a list of VPython 3D objects – boxes, spheres, and so forth – that make up the body of the bot. Since the player bot and the zombie bots will look different, we haven't placed any of these body parts into the list just yet. That's coming soon!

Notice that the `GenericBot` has three additional methods: `update`, `forward`, and `turn`. In fact, the `update` method simply calls the `turn` method to turn the bot 0 radians (we'll probably change that later!) and then calls the `forward` method to move one step at the given speed. The `turn` method changes the bot's `self.heading` so that it heads in the new direction induced by the turning angle and then rotates each of the parts in the bot's `self.parts` list by that same angle.

Something **very lovely** and subtle is happening in the `for` loop of the `turn` method! Notice that each `part` is expected to be a VPython object, like a `box` or `sphere` or something else. Each of those objects has a `rotate` method. Python is saying here “hey part, figure out what kind of object you are and then call your `rotate` method to rotate yourself.” So, if the first part is a `box`, then this will call the `box` `rotate` method. If the next part is a `sphere`, the `sphere`'s `rotate` method will be called here. This all works great, as long as each element in the `self.parts` list has a `rotate` method. Fortunately, all VPython shapes do have a `rotate` method.

We'll also just point out quickly that the line

```
part.rotate(angle = theta, axis = vector(0, 1, 0),
 origin = self.position)
```

is telling that part to rotate by an angle `theta` about a vector aligned with `vector(0, 1, 0)` (the  $\langle y \rangle$ -axis) but starting at the vector given by `self.position`. Under our assumption that the  $\langle y \rangle$ -axis is “up”, this effectively rotates the object about a line that runs through the center of its body from “its feet to its head”. That is, it rotates the body parts the way we would like it to rotate, as opposed to the default rotation which is about the  $\langle x \rangle$ -axis.

The `forward` method changes the bot's `self.position` vector by adding to it the `heading` vector scaled by the bot's `self.speed`. Note that `self.position` is just the bot's own self-concept of where it is located. We also need to physically move all of the parts of the bot's body, which is done by changing the `pos` position vector of each VPython objects in the `self.parts` list, again by the `self.heading` vector scaled by `self.speed`.



*Drumroll, please!*

Next comes the most amazing part of this whole business! We now define the `ZombieBot` class that *inherits* all of the methods and attributes of the `GenericBot` but adds the components that are specific to a zombie. Here's the code and we'll discuss it in a moment.

```
class ZombieBot(GenericBot):
 def __init__(self, position = vector(0, 0, 0),
 heading = vector(0, 0, 1)):
 GenericBot.__init__(self, position, heading)
 self.body = cylinder(pos = self.position,
 axis = vector(0, 4, 0),
 radius = 1,
 color = vector(0, 1, 0))
 self.arm1 = cylinder(pos = self.position + vector(0.6, 3, 0),
 axis = vector(0, 0, 2),
 radius = .3,
 color = vector(1, 1, 0))
 self.arm2 = cylinder(pos = self.position + vector(-0.6, 3, 0),
 axis = vector(0, 0, 2),
 radius = .3,
 color = vector(1, 1, 0))
 self.halo = ring(pos = self.position + vector(0, 5, 0),
 axis = vector(0, 1, 0),
 radius = 1,
 color = vector(1, 1, 0))
 self.head = sphere(pos = self.position + vector(0, 4.5, 0),
 radius = 0.5,
 color = vector(1, 1, 1))
 self.parts = [self.body, self.arm1, self.arm2,
 self.halo, self.head]

 def update(self):
 # call turn with a random angle between -5 and 5
 # degrees
 self.turn(random.uniform(-5, 5))
 self.forward()
```

The class definition begins with the line: `class ZombieBot(GenericBot)`. The `GenericBot` in parentheses is saying to Python “this class inherits from `GenericBot`.” Said another way, a `ZombieBot` “is a kind of” `GenericBot`. Specifically, this means that a `ZombieBot` has the `__init__`, `update`, `turn`, and `forward` methods of `GenericBot`. The class `GenericBot` is called the *superclass* of `ZombieBot`. Similarly, `ZombieBot` is called a *subclass* or *derived class* of `GenericBot`.

Note that `ZombieBot` has its own `__init__` constructor method. If we had not defined this `__init__` then each time we constructed a `ZombieBot` object, Python would automatically invoke the `__init__` from `GenericBot`, the superclass from which `ZombieBot` was derived. However, since we've defined an `__init__` method for `ZombieBot`, that method will get called when we instantiate a `ZombieBot` object. It's not that the `GenericBot`'s constructor isn't useful to us, but we want to do some other things too. Specifically, we want to populate the list of body parts, `self.parts`, with the VPython shapes that constitute a zombie.

We get two-for-the-price-of-one by first having the `ZombieBot`'s `__init__` method call the `GenericBot`'s `__init__` method to do what it can do for us. This is invoked via `GenericBot.__init__(self, position, heading)`. This is saying, “hey, I know that I'm a `ZombieBot`, but that means I'm a kind of `GenericBot` and, as such, I can call any `GenericBot` method for help. In particular, since the `GenericBot` already has an `__init__` method that does some useful things, I'll call it to set my `position` and `heading` attributes.”

After calling the `GenericBot` constructor, the `ZombieBot` constructor continues to do some things on its own. In particular, it defines some VPython objects and places them in the `parts` list of body parts. You might notice that all of those body parts are positioned relative to the `ZombieBot`'s `position` – which is just a vector that we defined that keeps track of where the bot is located in space.

Since `ZombieBot` inherited from `GenericBot`, it automatically has the `update`, `turn`, and `forward` methods defined in the `GenericBot` class. The `turn` and `forward` methods are fine, but the `update` method needs to be replaced to turn the `ZombieBot` at random. Thus, we provide a new `turn` method in the `ZombieBot` class.

Now, imagine that we do the following:

```
>>> zephyr = ZombieBot()
>>> zephyr.update()
```

The first line creates a new `ZombieBot` object. Since we didn't provide any inputs to the constructor, the default values are used and zephyr the zombie is at position (0, 0, 0) and heading in direction (0, 0, 1). The second line tells zephyr to update itself. Python checks to see if the `ZombieBot` class contains an `update` method. It does, so that method is invoked. That method then calls the `turn` method with a random angle between -5 and 5 degrees. Python checks to see if the `ZombieBot` class has its own `turn` method. Since it doesn't, Python goes to the superclass, `GenericBot`, and looks for a `turn` method there. There is one there and that's what's used! Next, the `update` method calls the `forward` method. Since there's no `forward` method defined in `ZombieBot`, Python again goes to the superclass and uses the `forward` method there.

The very nice thing here is that `ZombieBot` inherits many things from the superclass from which it is derived and only changes – or *overrides* – those methods that it needs to customize for zombies.

Now, we can do something similar for the player's robot:

```
class PlayerBot(GenericBot):
 def __init__(self, position = vector(0, 0, 0),
 heading = vector(0, 0, 1)):
 GenericBot.__init__(self, position, heading)
 self.body = cylinder(pos = self.position + vector(0, 0.5, 0),
 axis = vector(0, 6, 0),
 radius = 1,
 color = vector(1, 0, 0))
 self.head = box(pos = vector(0, 7, 0) + self.position,
 length = 2,
 width = 2,
 height = 2,
 color = vector(0, 1, 0))
 self.nose = cone(pos = vector(0, 7, 1) + self.position,
 radius = 0.5,
 axis = vector(0, 0, 1),
 color = vector(1, 1, 0))
 self.wheel1 = cylinder(pos = self.position + vector(1, 1, 0),
 axis = vector(0.5, 0, 0),
 radius = 1,
 color = vector(0, 0, 1))
 self.wheel2 = cylinder(pos = self.position + vector(-1, 1, 0),
 axis = vector(-0.5, 0, 0),
 radius = 1,
 color = vector(0, 0, 1))
 self.parts = [self.body, self.head, self.nose,
 self.wheel1, self.wheel2]

 def update(self):
 self.turn(0) # we'll leave the turn handling up to our buttons...
 self.forward()
```

The `PlayerBot` class also inherits from the `GenericBot` class. It again calls the `GenericBot`'s constructor for help initializing some attributes and then defines its own attributes. It uses the `turn` and `forward` methods from the superclass, and although the `update` method isn't any different, it reminds us that the inputs for the robot come from the button handlers in the GUI section.

Recall that a short awhile ago we noted that all Python shapes have a `rotate` method. That's because all of these shapes – boxes, spheres, cylinders, cones, and others – inherit from a `shape` superclass that defines a `rotate` method. Therefore, they all “know” how to rotate because they inherited that “knowledge” from their parent class. That parent class has many features – methods and attributes – that its children need, just like our `GenericBot` class has features that are used by its children – the derived classes `ZombieBot` and `PlayerBot`.

Finally, here's our game! In this file, we import the `vpython` graphics package; the `robot.py` file containing the `GenericBot`, `ZombieBot`, and `PlayerBot` classes, and the `random` and `math` packages. The main function instantiates a large flat VPython cylinder object, that we name `ground`, that is the surface on which the bots will move. Its radius is given by a global variable `GROUND_RADIUS`. We then instantiate a single `PlayerBot` named `player` and call a

helper function called `makeZombies` that instantiates many zombies (the number is given by the global variable `ZOMBIES`) and returns a list of `ZombieBot` objects at random positions on our cylindrical playing area. Finally, the main function enters an infinite loop. At each iteration, we check to see if the player's location is beyond the radius of the playing area, and if so turn the robot 180 degrees so that its next step will hopefully be inside the player area. Now, each zombie is updated by calling its `update` method. Here too, if the zombie has stepped out of the playing area, we rotate it at random (180 degrees plus or minus 30 degrees). We then also include code of our own to read user input (which we won't detail here). And so, voilá!

```
from vpython import *
from robot import *
import random
import math
import ipywidgets as widgets

variable declarations
global userbot
global running
running = True
GROUND_RADIUS = 50
ZOMBIES = 20

declare our buttons
fastButton = widgets.Button(description = 'F', width = '60px', height = '60px')
slowButton = widgets.Button(description = 'S', width = '60px', height = '60px')
leftButton = widgets.Button(description = 'L', width = '60px', height = '60px')
rightButton = widgets.Button(description = 'R', width = '60px', height = '60px')
fillerButton0 = widgets.Button(description = "", width = '60px', height = '60px')
resetButton = widgets.Button(description = 'Reset', width = '120px', height = '60px')
quitButton = widgets.Button(description = 'Quit', width = '120px', height = '60px')
fillerButton1 = widgets.Button(description = "", width = '120px', height = '60px')
scene.caption = "To use the directional pad, click on a marked direction. F = Faster, S = Slower, L = turn Left and R = turn Right."

These functions set up our buttons to read in inputs
def fastButton_handler(s):
 global userbot
 userbot.speed += 0.1
fastButton.on_click(fastButton_handler)

def slowButton_handler(s):
 global userbot
 userbot.speed -= 0.1
slowButton.on_click(slowButton_handler)

def leftButton_handler(s):
 global userbot
 userbot.turn(5)
leftButton.on_click(leftButton_handler)

def rightButton_handler(s):
 global userbot
 userbot.turn(-5)
rightButton.on_click(rightButton_handler)

def quitButton_handler(s):
 global running
 running = False
 print("Exiting the main loop. Ending this vPython session...")
quitButton.on_click(quitButton_handler)

now arrange and display our GUI
container0 = widgets.HBox(children = [fillerButton0, fastButton, fillerButton0, quitButton])
```

```

container1 = widgets.HBox(children = [leftButton, fillerButton0, rightButton, fillerButton1])
container2 = widgets.HBox(children = [fillerButton0, slowButton, fillerButton0, fillerButton1])
display(container0)
display(container1)
display(container2)

def main():
 global userbot
 global running
 ground = cylinder(pos = vector(0, -1, 0),
 axis = vector(0, 1, 0),
 radius = GROUND_RADIUS)
 userbot = PlayerBot()
 zombies = makeZombies()
 while running:
 rate(30)
 userbot.update()
 if mag(userbot.position) >= GROUND_RADIUS:
 userbot.turn(180)
 for z in zombies:
 z.update()
 if mag(z.position) >= GROUND_RADIUS:
 z.turn(random.uniform(150, 210))

def makeZombies():
 zombies = []
 for z in range(ZOMBIES):
 theta = random.uniform(0, 360)
 r = random.uniform(0, GROUND_RADIUS)
 x = r * cos(math.radians(theta))
 z = r * sin(math.radians(theta))
 zombies.append(ZombieBot(position = vector(x, 0, z)))
 return zombies
main()

```

## 6.9 Conclusion



*I suppose that “yucky” is yet another technical term.*

This is all really neat, but why is object-oriented programming such a big deal? As we saw in our Rational example, one benefit of object-oriented programming is that it allows us to define new types of data. You might argue, “Sure, but I could have represented a rational number as a list or tuple of two items and then I could have written functions for doing comparisons, addition, and so forth without using any of this class stuff.” You’re absolutely right, but you then have exposed a lot of yucky details to the user that she or he doesn’t want to know about. For example, the user would need to know that rational numbers are represented as a list or a tuple and would need to remember the conventions for using your comparison and addition functions. One of the beautiful things about object-oriented programming is that all of this “yuckiness” (more technically, “implementation details”) is *hidden* from the user, providing a *layer of abstraction* between the use and the implementation of rational numbers.

Layer of abstraction?! What does *that* mean? Imagine that every time you sat in the driver’s seat of a car you had to fully understand various components of the engine, transmission, steering system, and electronics just to operate the car. Fortunately, the designers of cars have presented us with a nice layer of abstraction: the steering wheel, pedals, and dashboard. We can now do interesting things with our car without having to think about the low-level details. As a driver, we don’t need to worry about whether the steering system uses a rack and pinion or something entirely different. This is precisely what classes provide for us. The inner workings of a class are securely “under the hood,” available if needed, but not the center of attention. The user of the class doesn’t

need to worry about implementation details; she or he just uses the convenient and intuitive provided methods. By the way, the “user” of your class is most often you! You too don’t want to be bothered with implementation details when you use the class—you’d rather be thinking about bigger and better things at that point in your programming.

Object-oriented design is the computer science version of *modular design*, an idea that engineers pioneered long ago and have used with great success. Classes are modules. They encapsulate logical functionality and allow us to reason about and use that functionality without having to keep track of every part of the program at all times. Moreover, once we have designed a good module/class we can reuse it in many different applications.

Finally, in our Robot and Zombies game, we saw the important idea of inheritance. Once we construct one class, we can write special versions that inherit all of the methods and attributes of the “parent” or superclass, but also add their own unique features. In large software systems, there can be a large and deep *hierarchy* of classes: One class has children classes that inherit from it which in turn have their own children classes, and so forth. This design methodology allows for great efficiencies in reusing rather than rewriting code.

**Takeaway message:** *Classes—the building blocks of object-oriented designs and programs—provide us with a way of providing abstraction that allows us to concentrate on using these building blocks without having to worry about the internal details of how they work. Moreover, once we have a good building block we can use it over and over in all different kinds of programs.*

## Table Of Contents

- Chapter 6: Fun and Games with OOPs: Object-Oriented Programs
  - 6.1 Introduction
  - 6.2 Thinking Objectively
  - 6.3 The Rational Solution
  - 6.4 Overloading
  - 6.5 Printing an Object
  - 6.6 A Few More Words on the Subject of Objects
  - 6.7 Getting Graphical with OOPs
  - 6.8 Robot and Zombies, Finally!
  - 6.9 Conclusion

**Previous topic** Chapter 5: Imperative Programming

**Next topic** Chapter 7: How Hard is the Problem?

## Quick search

Enter search terms or a module, class or function name.

## Navigation

- index
- next |
- previous |
- cs5book 1 documentation »

© Copyright 2013, hmc. Created using Sphinx 1.2b1.

# Chapter 7: How Hard is the Problem? — cs5book 1 documentation

## Navigation

- [index](#)
- [previous |](#)
- [cs5book 1 documentation »](#)

## Chapter 7: How Hard is the Problem?

*It is a mistake to think you can solve any major problems just with potatoes.*

—Douglas Adams

### 7.1 The Never-ending Program

Chances are you've written a program that didn't work as intended. Many of us have had the particularly frustrating experience of running our new program and observing that it seems to be running for a very long time without producing the output that we expected. "Hmmm, it's been running now for nearly a minute. Should I stop the program? Or, perhaps, since I've invested this much time already I should give the program another minute to see if it's going to finish," you say to yourself. Another minute passes and you ask yourself if you should let it run longer or bail out now. Ultimately, you might decide that the program is probably stuck in some sort of loop and it isn't going to halt, so you press Control-C and the program stops. You wonder though if perhaps the program was just a moment away from giving you the right answer. After all, the problem might just inherently require a lot of computing time.

Wouldn't it be nice to have some way to check if your program is eventually going to halt?



*The answer to that question is “yes!” Of course that would be nice!*

Then, if you knew that your program wasn't going to halt, you could work on debugging it rather than wasting your time watching it run and run and run.

In fact, perhaps we could even write a program, let's call it a “halt checker,” that would take *any* program as input and simply return a Boolean: `True` if the input program eventually halts and `False` otherwise. Recall from Chapter 3 that functions can take other functions as input. So there is nothing really strange about giving the hypothetical halt checker function another function as input. However, if you're not totally comfortable with that idea, another alternative is that we would give the halt checker a string as input and that string would contain the code for the Python function that we want to check. Then, the halt checker would somehow determine if the function in that string eventually halts.

For example, consider the following string which we've named `myProgram`:

```
myProgram = 'def foo(): \
 return foo()'
```

This string contains the Python code for a function called `foo`. Clearly, that function runs forever because the function calls itself recursively and there is no base case to make it stop. Therefore, if we were to run our hypothetical `haltChecker` function it should return `False`:

```
>>> haltChecker(myProgram)
False
```

How would we write such a halt checker? It would be easy to check for certain kinds of obvious problems, like the problem in the program `foo` in the example above. It seems though that it might be quite difficult to write a halt checker that could reliably evaluate *any* possible program that we would give it. After all, some programs are very complicated with all kinds of recursion, for loops, while loops, etc. Is it even possible to write such a halt checker program? In fact, in this chapter we'll show that this problem is impossible to solve on a computer. That is, it is not possible to write a halt checker program that would tell us whether *any* other program eventually halts or not.

How can we say it's impossible!? That seems like an irresponsible statement to make. After all, just because nobody has succeeded in writing such a program yet doesn't allow us to conclude that it's impossible. You've got a good point there – we agree! However, the task of writing a halt checker truly is *impossible* – it doesn't exist now and it will never exist. In this chapter we'll be able to prove that beyond a shadow of a doubt.

## 7.2 Three Kinds of Problems: Easy, Hard, and Impossible.

Computer scientists are interested in measuring the “hardness” of computational problems in order to understand how much time (or memory, or some other precious resource) is required for their solution. Generally speaking, time is the most precious resource so we want to know how much time it will take to solve a given problem. Roughly speaking, we can categorize problems into three groups: “easy”, “hard”, or “impossible.”



Recall that an “algorithm” is a computational recipe. It is more general than a “program” because it isn’t specific to Python or any other language. However, a program implements an algorithm.

An “easy” problem is one for which there exists a program – or algorithm – that is fast enough that we can solve the problem in a reasonable amount of time. All of the problems that we’ve considered so far in this book have been of that type. No program that we’ve written here has taken days or years of computer time to solve. That’s *not* to say that coming up with the program was always easy for us, the computer scientist. That’s not what we mean by “easy” in this context. Rather, we mean that there *exists* an algorithm that runs fast enough that the problem can be solved in a reasonable amount of time. You might ask, “what do you mean by *reasonable* ?” That’s a reasonable question and we’ll come back to that shortly.

In contrast, a “hard” problem is one for which we can find an algorithm to solve it, but every algorithm that we can find is so slow as to make the program practically useless. And the “impossible” ones, as we suggested in the introduction to this chapter, are indeed just that - absolutely, provably, impossible to solve no matter how much time we’re willing to let our computers crank away on them!

### 7.2.1 Easy Problems

Consider the problem of taking a list of  $\mathcal{O}(N)$  numbers and finding the smallest number in that list. One simple algorithm just “cruises” through the list, keeping track of the smallest number seen so far. We’ll end up looking at each number in the list once, and thus the number of steps required to find the smallest number is roughly  $\mathcal{O}(N)$ . A computer scientist would say that the running time of this algorithm “grows proportionally to  $\mathcal{O}(N)$ .”

In Chapter 5 we developed an algorithm, called *selection sort*, for sorting items in ascending order. If you don’t remember it, don’t worry. Here’s how it works in a nutshell: Imagine that we have  $\mathcal{O}(N)$  items in a list– we’ll assume that they are numbers for simplicity - and they are given to us in no particular order. Our goal is to sort them from smallest to largest. In Chapter 5, we needed that sorting algorithm as one step in our music recommendation system.

Here’s what the *selection sort* algorithm does. It first “cruises” through the list, looking for the smallest element. As we observed a moment ago, this takes roughly  $\mathcal{O}(N)$  steps. Now the algorithm knows the smallest element and it puts that element in the first position in the list by simply swapping the first element in the list with the smallest element in the list. (Of course, it might be that the first element in the list *is* the smallest element in the list, in which case that swap effectively does nothing - but in any case we are guaranteed that the first element in the list now is the correct smallest element in the list.)

In the next phase of the algorithm, we’re looking for the second smallest element in the list. In other words, we’re looking for the smallest element in the list excluding the element in the first position which is now known to be the first smallest element in the list. Thus, we can start “cruising” from the second element in the list and this second phase will take  $\mathcal{O}(N-1)$  steps. The next phase will take  $\mathcal{O}(N-2)$  steps, then  $\mathcal{O}(N-3)$ , and all the

way down to 1. So, the total number of steps taken by this algorithm is  $\lfloor N + (N-1) + (N-2) + \dots + 1 \rfloor$ . There are  $\lfloor N \rfloor$  terms in that sum, each of which is at most  $\lfloor N \rfloor$ . Therefore, the total sum is certainly less than  $\lfloor N^2 \rfloor$ . It turns out, and it's not too hard to show, the sum is actually  $\lfloor \frac{N(N+1)}{2} \rfloor$  which is approximately  $\lfloor \frac{N^2}{2} \rfloor$ .



You might hear a computer scientist say: “The running time is big-Oh of  $\lfloor N^2 \rfloor$ ” - written  $\lfloor O(N^2) \rfloor$ . That’s CS-talk for what we’re talking about here.

A computer scientist would say that the running time of this algorithm “grows proportionally to  $\lfloor N^2 \rfloor$ .” It’s not that the running time is necessarily exactly  $\lfloor N^2 \rfloor$ , but whether it’s  $\lfloor \frac{1}{2} N^2 \rfloor$  or  $\lfloor 42 N^2 \rfloor$ , the  $\lfloor N^2 \rfloor$  term dictates the shape of the curve that we would get if we plotted running time as a function of  $\lfloor N \rfloor$ .

As another example, the problem of multiplying two  $\lfloor N \times N \rfloor$  matrices using the method that you may have seen in an earlier math course takes time proportional to  $\lfloor N^3 \rfloor$ . All three of these algorithms - finding the minimum element, sorting, and multiplying matrices - have running times of the form  $\lfloor N^k \rfloor$  where  $\lfloor k \rfloor$  is some number: We saw that  $\lfloor k = 1 \rfloor$  for finding the minimum,  $\lfloor k = 2 \rfloor$  for selection sorting, and  $\lfloor k = 3 \rfloor$  for our matrix multiplication algorithm. Algorithms that run in time proportional to  $\lfloor N^k \rfloor$  are said to run in *polynomial time* because  $\lfloor N^k \rfloor$  is a polynomial of degree  $\lfloor k \rfloor$ . In practice, polynomial time is a reasonable amount of time. Although you might argue that an algorithm that runs in  $\lfloor N^{42} \rfloor$  steps is nothing to brag about, in fact using polynomial time as our definition of “reasonable” time is, um, er, reasonable – for reasons that will be apparent shortly.

### 7.2.2 Hard Problems

Imagine now that you are a salesperson who needs to travel to a set of cities to show your products to potential customers. The good news is that there is a direct flight between *every pair of cities* and, for each pair, you are given the cost of flying between those two cities. Your objective is to start in your home city, visit each city *exactly once*, and return back home at lowest total cost. For example, consider the set of cities and flights shown in Figure 7.1 and imagine that your start city is Aville.

../\_imagesstar.png

Figure 7.1: Cities and flight costs.

A tempting approach to solving this problem is to use an approach like this: Starting at our home city, Aville, fly on the cheapest flight. That’s the flight of cost 1 to Beesburg. From Beesburg, we could fly on the least expensive flight to a city that we have not yet visited, in this case Ceefield. From Ceefield we would then fly on the cheapest flight to a city that we have not yet visited. (Remember, the problem stipulates that you only fly to a city once, presumably because you’re busy and you don’t want to fly to any city more than once - even if it might be cheaper to do so.) So now, we fly from Ceefield to Deesdale and from there to Eetown. Uh oh! Now, the constraint that we don’t fly to a city twice means that we are forced to fly from Eetown to Aville at a cost of 42. The total cost of this “tour” of the cities is  $\lfloor 1 + 1 + 1 + 1 + 42 = 46 \rfloor$ . This approach is called a “greedy algorithm” because at each step it tries to do what looks best at the moment, without considering the long-term implications of that decision. This greedy algorithm didn’t do so well here. For example, a much better solution goes from Aville to Beesburg to Deesdale to Eetown to Ceefield to Aville with a total cost of  $\lfloor 1 + 2 + 1 + 2 + 3 = 9 \rfloor$ . In general, greedy algorithms are fast but often fail to find optimal or even particularly good solutions.

It turns out that finding the optimal tour for the Traveling Salesperson Problem is very difficult. Of course, we could simply enumerate every one of the possible different tours, evaluate the cost of each one, and then find the one of least cost. If we were to take this approach in a situation with  $\lfloor N \rfloor$  cities, how many different tours would we have to explore?

../\_imagestime.png

Table 7.1: Time to solve problems of various sizes using algorithms with different running times on a computer capable of performing one billion operations per second. Times are in seconds unless indicated otherwise.



We're using  $\backslash(N!)$  rather than  $\backslash((N-1)!)$  here for simplicity. They only differ by a factor of  $\backslash(N\backslash)$ , which is negligible in the grand scheme of things.

Notice that there are  $\backslash(N-1\backslash)$  first choices for the first city to visit. From there, there are  $\backslash(N-2\backslash)$  choices for the next city, then  $\backslash(N-3\backslash)$  for the third, etc. Therefore there are a total of  $\backslash((N-1 \times (N-2) \times (N-3)\dots \times 1\backslash)$ . That product is called "  $\backslash(N-1\backslash)$  factorial" and denoted  $\backslash((N-1)!)$ . The exclamation point is appropriate because that quantity grows very rapidly. While  $\backslash(5!)$  is a modest  $\backslash(120\backslash)$ ,  $\backslash(10!)$  is over 3 million and  $\backslash(15!)$  is over one trillion. Computers are fast, but examining one trillion different tours would take a long time on even the fastest computer.



*That's a lot of exclamation marks!*

Table 7.1 demonstrates just how bad a function like  $\backslash(N!)$  is in comparison to polynomial functions like  $\backslash(N\backslash)$ ,  $\backslash(N^2\backslash)$ ,  $\backslash(N^3\backslash)$ , and  $\backslash(N^5\backslash)$ . In this table, we are imagining a computer that is capable of performing one billion "operations" per second so that our algorithm for finding the minimum element in a list in time proportional to  $\backslash(N\backslash)$  can find the minimum in a list of one billion items in one second. We're being pretty generous here, and assuming that this computer can also enumerate one billion traveling salesperson tours in one second, but this will only make our case against the brute-force enumeration of traveling salesperson tours even stronger!

Ouch!  $\backslash(N!)$  is clearly very bad! One might be tempted to believe that in a few years, when computers get faster, this will no longer be a problem. Sadly, functions like  $\backslash(N!)$  grow so fast that a computer that is 10 or 100 or 1000 times faster provide us with very little relief. For example, while our  $\backslash(N!)$  algorithm takes  $\backslash(8400\backslash)$  trillion years for a problem of size  $\backslash(30\backslash)$  on our current one billion operations per second computer, it will take  $\backslash(8.4\backslash)$  trillion years on a computer that is 1000 times faster. That's hardly good news!

The Traveling Salesperson Problem is just one of many problems for which algorithms exist but are much too slow to be useful - regardless of how fast computers become in the future. In fact, the Traveling Salesperson Problem is in a group - or "class" - of problems called the *NP-hard* problems. Nobody knows how to solve any of these problems efficiently (that is, in polynomial time like  $\backslash(N\backslash)$  or  $\backslash(N^2\backslash)$  or  $\backslash(N^k\backslash)$  for any constant  $\backslash(k\backslash)$ ). Moreover, these problems all have the property that if any *one* of them can be solved efficiently (in polynomial time) then, amazingly, *all* of these problems can be solved efficiently.

Metaphorically, we can think of the NP-hard problems as a giant circle of very big dominoes - one domino for the Traveling Salesperson Person and one for every NP-hard problem. We don't know how to knock any of these dominoes over (metaphorically, how to solve any of them efficiently) *but* we know that if any one of them falls over, all of the rest of them will fall (be solvable by an efficient algorithm) as well.

Unfortunately, many important and interesting problems are NP-hard. For example, the problem of determining how a protein folds in three dimensions is NP-hard. That's truly unfortunate because if we could predict how proteins fold, we could use that information to design better drugs and combat a variety of diseases. As another example, imagine that we have a number of items of different size that we want to pack into shipping containers of some given size. How should we pack our items into shipping containers so as to minimize the number of containers used? This problem too is NP-hard. Many games and puzzles are also NP-hard. For example, the problem of determining whether large Sudoku puzzles are solvable is NP-hard as are problems related to solving Minesweeper games and many others.

The good news is that just because a problem is NP-hard does not mean that we can't find reasonably good - albeit probably not perfect - solutions. Computer scientists work on a variety of strategies for dealing with NP-hard problems. One strategy is something called *approximation algorithms*. An approximation algorithm is an algorithm that runs fast (in polynomial time) and finds solutions that are guaranteed to be within a certain percentage of the best possible solution. For example, for certain types of Traveling Salesperson Problems we can quickly find solutions that are guaranteed to be no more than 50% more expensive than the best possible tour. You might reasonably ask, "How can you guarantee that a solution is within 50% of optimal when you

have no way of efficiently finding the optimal solution?” This is indeed surprising and seemingly magical, but in fact such guarantees can be made. This topic is often studied in an “Algorithms” course.



*That's a shameless sales pitch for another course if there ever was one!*

There are many other approaches as well to dealing with NP-hard problems. One approach is called *heuristic* design. A heuristic is essentially a “rule of thumb” for approaching a problem. For example, for the Traveling Salesperson Problem, you might use the “greedy” heuristic that we mentioned earlier, namely, start by visiting the city that you can get to most cheaply. From there, visit the cheapest city that you get to that you haven’t visited yet. Continue in this fashion until you’ve visited each city once and then fly back home. That’s a simple rule and it often finds reasonably good solutions, although it sometimes does quite badly as we saw earlier. In contrast to approximation algorithms, heuristics don’t make any promises on how well they will do. There are many other approaches for dealing with NP-hard problems and this is an active area of research in computer science. For an example of an interesting biologically inspired technique for dealing with NP-hard problems, see below.

Note

### **Genetic Algorithms: A Biologically-Inspired Approach for Dealing with NP-hard Problems.**

One technique for dealing with NP-hard problems is borrowed from biology. The idea is to “evolve” reasonably good solutions to our problem by simulating evolution on a population of possible solutions to our problem. Returning to our Traveling Salesperson example, let’s call the cities in Figure 7.1 by their first letters:  $\backslash(A\backslash)$ ,  $\backslash(B\backslash)$ ,  $\backslash(C\backslash)$ ,  $\backslash(D\backslash)$  and  $\backslash(E\backslash)$ . We can represent a tour by a sequence of those letters in some order, beginning with  $\backslash(A\backslash)$  and with each letter appearing exactly once. For example, the tour Aville to Beesburg to Deesdale to Eetown to Ceefield and back to Aville would be represented as the sequence  $\backslash(ABDEC\backslash)$ . Notice that we don’t include the  $\backslash(A\backslash)$  at the end because it is implied that we will return to  $\backslash(A\backslash)$  at the end.

Now, let’s imagine a collection of some number of orderings such as  $\backslash(ABDEC\backslash)$ ,  $\backslash(ADBCE\backslash)$ ,  $\backslash(AECDB\backslash)$ , and  $\backslash(AEBDC\backslash)$ . Let’s think of each such ordering as an “organism” and the collection of these orderings as a “population”. Pursuing this biological metaphor further, we can evaluate the “fitness” of each organism/ordering by simply computing the cost of flying between the cities in that given order.

Now let’s push this idea one step further. We start with a population of organisms/orderings. We evaluate the fitness of each organism/ordering. Now, some fraction of the most fit organisms “mate”, resulting in new “child” orderings where each child has some attributes from each of its “parents.” We now construct a new population of such children for the next generation. Hopefully, the next generation will be more fit - that is, it will, on average, have less expensive tours. We repeat this process for some number of generations, keeping track of the most fit organism (least cost tour) that we have found and report this tour at the end.

“That’s a cute idea,” we hear you say, “but what’s all this about mating Traveling Salesperson orderings?” That’s a good question - we’re glad you asked! There are many possible ways we could define the process by which two parent orderings give rise to a child ordering. For the sake of example, we’ll describe a very simple (and not very sophisticated) method; better methods have been proposed and used in practice.

Imagine that we select two parent orderings from our current population to reproduce (we assume that any two orderings can mate):  $\backslash(ABDEC\backslash)$  and  $\backslash(ACDEB\backslash)$ . We choose some point at which to split the first parent’s sequence in two, for example as  $\backslash(ABD \mid EC\backslash)$ . The offspring ordering receives  $\backslash(ABD\backslash)$  from this parent. The remaining two cities to visit are  $\backslash(E\backslash)$  and  $\backslash(C\backslash)$ . In order to get some of the second parent’s “genome” in this offspring, we put  $\backslash(E\backslash)$  and  $\backslash(C\backslash)$  in the order in which they appear in the second parent. In our example, the second parent is  $\backslash(ACDEB\backslash)$  and  $\backslash(C\backslash)$  appears before  $\backslash(E\backslash)$ , so the offspring is  $\backslash(ABDCE\backslash)$ .

Let’s do one more example. We could have also chosen  $\backslash(ACDEB\backslash)$  as the parent to split, and split it at  $\backslash(AC \mid DEB\backslash)$ , for example. Now we take the  $\backslash(AC\backslash)$  from this parent. In the other parent,  $\backslash(ABDEC\backslash)$ , the remaining cities  $\backslash(DEB\backslash)$  appear in the order  $\backslash(BDE\backslash)$ , so the offspring would be  $\backslash(ACBDE\backslash)$ .

In summary, a genetic algorithm is a computational technique that is effectively a simulation of evolution with natural selection. The technique allows us to find good solutions to hard computational problems by imagining candidate solutions to be metaphorical organisms and collections of such organisms to be populations. The population will generally not include every possible “organism” because there are usually far too many! Instead,

the population comprises a relatively small sample of organisms and this population evolves over time until we (hopefully!) obtain very fit organisms (that is, very good solutions) to our problem.

## 7.3 Impossible Problems!

So far, we've talked about "easy" problems – those that can be solved in polynomial time - and "hard" problems - those that can be solved but seemingly require impractically huge amounts of time to solve. Now we turn to problems that are downright impossible to solve, no matter how much time we are willing to spend.

First, we'd like to establish that there even exist impossible problems. We'll do this, in essence, by showing that the number of computational problems is much larger than the number of different programs. Therefore, there must be some problems for which there exist no programs. This is a strange and beautiful idea. It's strange because it ultimately will allow us to establish that there are problems that can't be solved by programs but it doesn't actually tell us what any of those problems are! This is what a mathematician calls an *existence proof*: We show that something exists (in this case uncomputable problems) without actually identifying a concrete example! The idea is also beautiful because it uses the notion of different sizes of infinities. In a nutshell, it turns out that there are an infinite number of different programs that we could write but there is a *larger* infinite number of different computational problems. "Larger infinities!?" we hear you exclaim. "Have you professors totally lost your minds?" Perhaps, but that's not relevant here. Let's dive in to our adventure in infinities, large and small.

### 7.3.1 "Small" Infinities



*Sorry, you'll just have to imagine that. After all, if we actually gave them to you, you'd probably eat them.*

Imagine that we gave you three jelly beans. Of course, you are good at counting and you instantly recognize that you have three jelly beans, but pardon us for a moment as we look at this another way. You have three jelly beans because you can match up your jelly beans with the set of counting numbers  $\{\{1, 2, 3\}\}$ . A mathematician would say that there is a *bijection* – or "perfect matching" – between your set of three jelly beans and the set of counting numbers  $\{\{1, 2, 3\}\}$ .

More precisely, a bijection is a matching of elements from one set (e.g., our set of jelly beans) to another set (e.g., our set of numbers  $\{\{1, 2, 3\}\}$ ) such that every element in the *first set* is matched to a *distinct* element in the second set and every element in the *second set* is matched to something from the first set. In other words, a bijection is a "perfect matching" of elements between our two sets. We say that two sets have the same *cardinality* (or "size") if there is a bijection between them.

That seems pretty obvious and more than a bit pedantic. But here's where it gets interesting! Once we accept that two sets have the same cardinality if there exists a bijection between them, it's natural to see what happens when the two sets are infinite. Consider, for example, the set of counting numbers  $\{\{1, 2, 3, 4, \dots\}\}$  and the set of even counting numbers  $\{\{2, 4, 6, 8, \dots\}\}$ . Both sets are clearly infinite. Moreover, it seems that the set of counting numbers is about twice as large as the set of even counting numbers. Strangely though, the two sets have the same cardinality: We can establish a bijection - a perfect matching – between the two! Here it is:

`../_images/count.png`

Notice that the mapping here associates each counting number  $(x)$  with the even counting number  $(2x)$ . Let's see if it's truly a bijection. Is every counting number associated with a distinct even counting number? Yes, because each pair of counting numbers is matched to two different even numbers. And, is every even counting number matched in this way to some counting number? Yes, because any even counting  $(y)$  number is matched to the counting number  $(y/2)$ . So, strangely, these two sets have the same cardinality!

By similar arguments, the set of all integers (the counting numbers, the negatives of the counting numbers, and 0) also has a bijection with the counting numbers, so these two sets also have the same size. Amazingly, even the set of all rational numbers (numbers that are of the form  $\frac{p}{q}$  where  $(p)$  and  $(q)$  are integers) has a bijection with the counting numbers. That's truly strange because on the face of it, it seems that there are *way* more rational numbers than counting numbers. But such is the reality of infinity!

## “Larger” Infinities

Any set that has the same cardinality as the counting numbers is said to be *countably infinite*. So, as we noted above, the set of even counting numbers, the set of integers, and the set of all rational numbers are all countably infinite. If that’s the case, aren’t *all* infinite sets countably infinite?

Surprisingly, the answer is “no”!

The story begins in Germany in the last years of the nineteenth century. A brilliant mathematician name Georg Cantor gave a beautiful argument that shows that the set of real numbers – the set of all numbers, including the rational and irrational numbers - contains a “larger” infinity of elements than the set of counting numbers.

Cantor’s proof made many mathematicians in his day very uncomfortable and, in fact, many went to their graves feeling that there must be a problem somewhere in Cantor’s logic. Cantor’s proof is not only rock-solid but it is one of the most important results in all of mathematics. Moreover, as we’ll see shortly, it’s the very basis for showing that there exist uncomputable problems.

Cantor’s proof is based on the method of *proof by contradiction*. (See the sidebar on proofs by contradiction.) The approach is to assume that something is true (e.g., that the set of real numbers is countably infinite) and derive from that a contradiction (e.g.,  $\exists(1=2)$  or something equally obviously false). We are therefore forced to conclude that our initial assumption (e.g., that the set of real numbers is countably infinite) must also be false (because if we assume it’s true we are led to conclude something that is definitely false).

Note

### A Quick Primer on Proofs by Contradiction

In case you haven’t seen the technique of *proof by contradiction* before, let’s take a brief aside to demonstrate it with a famous mathematical result that goes back to the days of ancient Greece.

The Greeks wondered if all numbers are rational - that is, whether every number can be expressed as the ratio of two integers like  $\frac{1}{2}$ ,  $\frac{3}{7}$ , etc. According to legend, Hippasus of Metapontum discovered that  $\sqrt{2}$  is not rational - it cannot be written as a ratio of two integers. The ancient Greeks, also according to legend, treated this result as an official secret, and Hippasus was murdered when he divulged it.

Here’s the proof. Assume that  $\sqrt{2}$  is rational. Then it can be written as a ratio of two integers,  $\frac{p}{q}$ . Every fraction can be written in lowest terms (that is, canceling out common factors), so let’s assume that  $\frac{p}{q}$  is in lowest terms. In particular, that means that  $p$  and  $q$  cannot both be even, since if they were we would have cancelled out the multiples of 2 from each of  $p$  and  $q$  when putting the fraction in lowest terms.

OK, so now  $\sqrt{2} = \frac{p}{q}$ . Let’s square both sides to get  $2 = \frac{p^2}{q^2}$  and now let’s multiply both sides by  $q^2$  to get  $2q^2 = p^2$ . Since  $2q^2$  is even, this means that  $p^2$  is even. If  $p^2$  is even, clearly  $p$  must be even (since the square of an odd number is always odd!). Since  $p$  is even, let’s rewrite it as  $p = 2\ell$  where  $\ell$  is also an integer. So now,  $2q^2 = p^2 = (2\ell)^2 = 4\ell^2$ . Since  $2q^2 = 4\ell^2$ , we can divide both sides by 2 and we get  $q^2 = 2\ell^2$ . Aha! So  $q^2$  is even! But this means that  $q$  is even. That’s a contradiction because when we wrote  $\sqrt{2} = \frac{p}{q}$  we stipulated that  $\frac{p}{q}$  is in lowest terms, and thus  $p$  and  $q$  could not both be even.

We’ve just established that if  $\sqrt{2} = \frac{p}{q}$  in lowest terms then *both*  $p$  and  $q$  must be even and thus  $\frac{p}{q}$  is not in lowest terms. That’s like saying if it’s raining then it’s not raining. In that case, it can’t be raining! And in just the same way in our case,  $\sqrt{2}$  cannot be written as a fraction.

In a nutshell, a proof by contradiction shows that something is false by first assuming that it’s true and then deducing an absurd (namely false) outcome. This forces us to conclude that our assumption was false, thereby proving what we seek to prove!

Cantor not only showed that the real numbers are uncountably infinite, he showed that even the set of real numbers between 0 and 1 are uncountably infinite! Here’s how his proof goes: Assume that the set of real numbers between 0 and 1 is countably infinite. (Note that he’s setting up the proof by contradiction. He’s hoping to eventually conclude that something absurd follows from this assumption, therefore forcing us to conclude that the assumption is false!) Then, there must exist a bijection – a perfect matching – between the set of counting numbers and the set of real numbers between 0 and 1. We don’t know what that bijection looks like, but it would need to match every counting number with a distinct real number between 0 and 1.

A real number between 0 and 1 can be represented by a decimal point followed by an infinite number of digits. For example, the number  $.5$  can be represented as  $(0.5000\ldots)$ . The number  $\frac{1}{3}$  can be

represented as  $\langle .333\ldots \rangle$ . The fractional part of  $\langle (\pi) \rangle$  would be  $\langle (.141592654\ldots) \rangle$ . So a bijection between the counting numbers and the real numbers between 0 and 1 could be represented by a table that looks “something” like this. We say “something” because we don’t know what the actual association would be between counting numbers and real numbers would be, except for the fact that it is necessarily a bijection so we must associate a distinct real number with every counting number and every real number must appear somewhere on the right-hand column of the listing.

[../\\_images/count2.png](#)

Now, Cantor says: “Aha! Well now that you have given me the bijection, I’m going to look at that bijection and highlight the first digit of the real number matched with counting number 1, the second digit of the real number matched with counting number 2, the third digit of the real number matched with counting number 3, and so forth, all the way down the list. This is going down the diagonal of the listing of real numbers indicated in boldface in the table below.

[../\\_images/count3.png](#)

“Now,” says Cantor, “Watch this! I’m going to write down a new number as follows: My new number differs from the real number matched with 1 by changing its first digit (e.g., changing 5 to 6). Similarly, my new number differs from the real number matched with 2 by changing its second digit (e.g., changing 3 to 4). It differs from the real number matched with 3 by changing its third digit (e.g., changing from 1 to 2), etc. In general, for any counting number  $\langle (n) \rangle$ , look at the real number matched to  $\langle (n) \rangle$  in your bijection. My number will differ from that real number in the  $\langle (n^{\text{th}}) \rangle$  digit. For example, for the hypothetical bijection that begins as in the table above, the real number that I construct would begin with  $\langle (.642\ldots) \rangle$ .”

“What’s with that number?” you ask. “Well,” says Cantor, “You promised me that you had a bijection between the counting numbers and the real numbers between 0 and 1, and that real number that I just constructed is certainly between 0 and 1. So where is it in the table that represents your bijection?” He’s got a good point. Where is it? You might argue, “Be patient, it’s matched with some very large counting number way down that list.” But if so, you must be able to tell Cantor what counting number that is. Perhaps, Cantor’s new number is matched with the counting number  $\langle (10^{99}) \rangle$ . But in that case, by the way that Cantor constructed his new number, it would differ from that number in the billionth digit. That’s a problem because we asserted that we had a bijection between the counting numbers and the real numbers between 0 and 1, and therefore every real number in that range, including Cantor’s new number, must appear in that bijection.

So, the conclusion is that no matter what bijection we try to establish between the counting numbers and the real numbers between 0 and 1, Cantor can always use his *diagonalization method* (changing the digits along the diagonal) to show that our putative matching fails to be a bijection. In other words, no bijection exists between the counting numbers and the real numbers between 0 and 1 and we have shown that the real numbers are not countably infinite!

**A fine point:** Our attorneys have advised us to say one more thing here. Note that some real numbers have two different representations when we use an infinite decimal expansion. For example  $\langle (0.3) \rangle$  is the same as  $\langle (0.2999\ldots) \rangle$ . Indeed, the problem is with an infinite number of 9’s. Every real number whose decimal expansion eventually has an infinite sequence of 9’s can be represented by changing the digit before the first of those 9’s and then replacing all of the 9’s by zeroes. So, it might be that Cantor’s new number in his diagonalization proof is actually *not different* from all numbers in our bijection table, it just *looks different* because it has a different representation. The solution to this problem is to always agree to use the “nicer” representation of numbers when we have the choice - that is, never use the representation that has an infinite number of consecutive 9’s. After all, this is just an issue of representing numbers and we can always represent a number that contains an infinite sequence of 9’s with its alternate representation! Now, when Cantor diagonalizes, he must be careful not to construct a number with an infinite sequence of consecutive 9’s, because we decided to avoid that representation. One easy solution to this is to simply have Cantor avoid ever writing the digit 9 in the new number he constructs. In particular, if he was changing the digit 8 in one of the real numbers in the table, rather than letting him change it to 9 (which might lead to an infinite sequence of 9’s), have him change it to any digit other than 8 or 9. This will still give him the contradiction that he seeks, and avoids the legal tangle that our attorneys have been worrying about.



*I wonder how much they billed us for that?*

### 7.3.2 Uncomputable Functions

We've just seen that the set of real numbers between 0 and 1 is such a large infinity that its elements can't be matched up perfectly with the counting numbers. But what does this have to do with computer science?



Your “favorite” language is officially Python, but this would work just as well for any programming language.

Our goal is to show that there exist problems that are not solvable by any program. We'll do this using Cantor's diagonalization method. Here's the plan: First we'll show that the set of all programs is countably infinite; that is, there is a bijection between the counting numbers and the set of all programs in your favorite language.

Then, we'll show that the set of problems does not have a bijection with the counting numbers (it's a larger infinite set), and thus there is no matching between programs and problems.

Let's start with the set of programs. A program is, after all, nothing more than a string of symbols that we interpret as a sequence of instructions. Here's our plan: We'll list out all possible strings in alphabetical order from short ones to longer ones. Each time we list one out, we'll check if it's a valid Python program. If it is, we'll match it to the next available counting number.

There are 256 different symbols that are used in a standard computer symbol set. So, we'll first list the 256 strings of length 1. None of them are valid Python programs. Then we'll move on and generate  $\backslash(256 \backslash \text{times} 256\backslash)$  strings of length 2. None of these are valid Python programs. Eventually, we'll come across a valid Python program and we'll match that with the counting number 1. Then, we'll march onwards, looking for the next valid Python program. That program will be matched with 2, and so forth.

While this process is laborious, it does show that we can find a bijection between counting numbers and Python programs. Every counting number will be matched to a distinct Python program and every Python program will eventually be generated this way, so it will be matched to some counting number. So, there is a bijection between the set of counting numbers and the set of Python programs. The set of Python programs is, therefore, countably infinite.

Now let's count the number of problems. The problem with problems is that there are so many different kinds of them! So, to make our life easier, let's restrict our attention to a special kind of problem where we take a counting number as input and return a Boolean: `True` or `False`. For example, consider the problem of determining if a given counting number is odd. That is, we'd like a function called `odd` that takes any counting number as input and returns `True` if it's odd and `False` otherwise. That's an easy function to write in Python. Here it is:

```
def odd(X):
 return X % 2 == 1
```

A function that takes any counting number as input and returns a Boolean is called a *counting number predicate*. Counting number predicates are obviously a very, very special kind of problem.

It will be convenient to represent counting number predicates as infinite strings of “T” (for `True`) and “F” (for `False`) symbols. We list all the possible inputs to the predicate along the top and, for each such input, we put a “T” if the predicate should return `True` for this input and we put a “F” otherwise. For example, the odd predicate would be represented as shown in Figure 7.2

`../_images/input.png`

Table 7.2: Representing the odd counting number predicate as an infinite list of “T” and “F” symbols, one for each possible input to the predicate.

Table 7.3 shows a list of several counting number predicates. The first one is the predicate for determining if its input is odd. The next one is the predicate that determines if its input is even, the next one is for primes, etc. Incidentally, all of these predicates have relatively simple Python functions.

`../_images/input2.png`

Table 7.3: A few sample counting number predicates and the values they return for small non-negative integers.

Now, suppose you claim to have a bijection between the counting numbers and counting number predicates. That is, you have a listing that purportedly perfectly matches the counting numbers to the counting number predicates. Such a bijection might look something like Table 7.4. In this table each row shows a counting number

on the left and the predicate with which it is matched on the right. In this example, the counting number 1 is matched to the odd predicate, the counting number 2 is matched to the even predicate, etc. However, this is just an example! Our plan is to show that no matter how we try to attempt to match counting numbers to predicates, the attempt will necessarily fail.

../\_images/input3.png

Table 7.4: Applying Cantor diagonalization to the counter number predicates. The table shows an attempted bijection of counting numbers to predicates. Cantor diagonalization is used to construct a predicate that is not in this list.

Once again Cantor reappears to foil your plan. “Aha!” he says. “I can diagonalize your predicates and create one that’s not on your list.

And sure enough, he does exactly that, by defining a predicate that returns values different from everything you listed. How? For the predicate matched to the counting number 1, he makes sure that if that predicate said “T” to 1 then his predicate would say “F” to 1 (and if it said “F” to 1 his predicate would say “T” to 1). So his predicate definitely differs from the predicate matched to the counting number 1. Next, his predicate is designed to differ from the predicate matched to the counting number 2 by ensuring that it says the opposite of that predicate on input 2, and so forth down the diagonal! In this way, his new predicate is different from every predicate in the list. This diagonalization process is illustrated in Table 7.4.

So, Cantor has established that there is at least one predicate that’s not on your list and thus the counting number predicates must be uncountably infinite, just like the real numbers.

What have we shown here? We’ve shown that there are more counting number predicates than counting numbers. Moreover, we showed earlier that the number of programs is equal to the number of counting numbers. So, there are more counting number predicates than programs and thus there must be some counting number predicates for which there exist no programs.

Said another way, we’ve shown that there exist some “uncomputable” problems. That’s surprising and amazing, but the key words there are “there exist.” We haven’t actually demonstrated a particular problem that is uncomputable. That’s our next mission.

## 7.4 An Uncomputable Problem

In the last section we proved that there *exist* functions that can’t be computed by any program. The crux of the argument was that there are more problems than programs, and thus there must be some problems with no corresponding programs.

That’s surprising and amazing, but it would be nice to see an example of an actual problem that cannot be solved by a program. That’s precisely what we’ll do in this section!

### 7.4.1 The Halting Problem

At the beginning of this chapter we noted that it would be very useful to have halt checker program that would take another program (like one that we are working on for a homework assignment) as input and determine whether or not our input program will eventually halt. If the answer is “no”, this would tell us that our program probably has a bug!

To make things just a bit more realistic and interesting, we note that our homework programs usually take some input. For example, here’s a program that we wrote for a homework assignment. (Exactly what the objective of this homework problem was now escapes us!)

```
def homework1(X):
 if X == "spam": return homework1(X)
 else: return "That was not spam!"
```

Notice that this program runs forever if the input is ‘spam’ and otherwise it halts. (It returns a string and it’s done!)

So, imagine that we now want a function called `haltChecker` that takes *two* inputs: A string containing a program, `P`, and a string, `S`, containing the input that we would like to give that program. The `haltChecker` then returns `True` if program `P` would eventually halt if run on input string `S` and it returns `False` otherwise.



Evidently, you can buy just about anything on the Internet.

We'd *really* like to have this `haltChecker`. We searched on the Internet and found one for sale! Here's what the advertisement says: "Hey programmers! Have you written programs that seem to run forever when you know that they *should* halt? Don't waste your precious time letting your programs run forever anymore! Our new `haltChecker` takes *any* program `P` and *any* string `S` as input and returns `True` or `False` indicating whether or not program `P` will halt when run on input string `S`. It's only \$99.95 and comes in a handsome faux-leather gift box!"

"Don't waste your money!"—one of our friends advised us. "We could write the `haltChecker` ourselves by simply having it run the program `P` on string `S` and see what happens!" That's tempting, but do you see the problem? The `haltChecker` must *always* return `True` or `False`. If we let it simply run program `P` on input string `S`, and it happens that `P` runs forever on `S`, then the `haltChecker` also runs forever and doesn't work as promised. So, a `haltChecker` would probably need to somehow examine the contents of the program `P` and the string `S`, perform some kind of analysis, and determine whether `P` will halt on input `S`.

But it's true that we shouldn't waste our money on this product because we're about to show that the `haltChecker` cannot exist. Before doing that, however, our lawyers have advised us to point out that it *is* possible to write a `haltChecker` that works correctly for *some* programs and their input strings, because certain kinds of infinite loops are quite easy to detect. But that is not good enough! We want one that is absolutely positively 100% reliable. It must render a decision, `True` or `False`, for *any* program `P` and input `S` that we give it.

A hypothetical `haltChecker` would need to take two inputs: a program and a string. For simplicity, let's assume that the program is actually given as a string. That is, it's Python code in quotation marks. The `haltChecker` will interpret this string as a program and analyze it (somehow!). So, we might do something like this with our hypothetical `haltChecker`:

```
>>> P = 'def homework1(X):\n if X == "spam": return homework1(X)\n else: return "That was not spam!"'\n>>> S = 'spam'\n>>> haltChecker(P, S)\nFalse\n>>> haltChecker(P, 'chocolate')\nTrue
```

We will show that a `haltChecker` cannot exist using a proof by contradiction again. This is a general and powerful technique for showing that something can't exist.

Here's a metaphorical sketch of what we're about to do. Let's imagine that someone approaches you and tells you, "Hey, I've got this lovely magic necklace with a crystal oracle attached to it. If you ask it any question about the wearer of the necklace that can be answered with `True` or `False`, it will always answer the question correctly. I'll sell it to you for a mere \$99.95."

It sounds tempting, but such a necklace cannot exist. To see this, assume by way of contradiction that the necklace does exist. You can force this necklace to "lie" as follows: First you put the necklace on. Then, your friend asks the oracle necklace: "Will my friend accept this handful of jelly beans?" If the oracle necklace says `True` then your friend can ask you "Would you like this handful of jelly beans?" You now answer `False` and the oracle has been caught giving the wrong answer! Similarly, if the oracle necklace says `False` then, when your friend asks you if you would like the jelly beans, you say `True` and you have again caught the oracle in a lie. So, we are forced to conclude that an always accurate oracle necklace cannot exist.

Necklaces? Oracles? Jelly Beans? "You professors need a vacation," we hear you say. Thank you for that, it is indeed almost vacation time, but let's see first how this necklace metaphor bears on the `haltChecker`.

OK then. Here we go! Assume by way of contradiction that there exists a `haltChecker`. Then, we'll place that `haltChecker` function in a file that also contains the program `paradox` below:

```
def paradox(P):\n if haltChecker(P, P):\n while True:\n print 'Look! I am in an infinite loop!'
```

```
else: return
```

Take a close look at what's happening here. This `paradox` program takes a single string `P` as input. Next, it gives that string as *both* inputs to the `haltChecker`. Is that OK?! Sure! The `haltChecker` takes two strings as input, the first one is interpreted as a Python program and the second one is any old string. So, let's just use `P` as both the program string for the first input and also as the string for the second input.

Now, let's place the code for the `paradox` function in a string and name that string `P` like this:

```
>>> P = "def paradox(P):
 if haltChecker(P, P):
 while True:
 print 'Look! I am in an infinite loop!'
 else: return"
```

Finally, let's give that string `P` as the input to our new `paradox` function:

```
>>> paradox(P)
```

What we're doing here is analogous to you foiling the hypothetical oracle necklace; in this case the hypothetical oracle is the `haltChecker` and you are the invocation `Paradox(P)`.

Let's analyze this carefully to see why. First, notice that this `paradox` program either enters an infinite loop (the `while True` statement loops forever, printing the string "Look! I am in an infinite loop!" each time through the loop) or it returns, thereby halting.

Next, notice that when we run the `paradox` function on the input string `P`, we are actually running the program `paradox` on the input string containing the program `paradox`. It then calls `haltChecker(P, P)` which must return `True` or `False`. If `haltChecker(P, P)` returns `True`, it is saying "It is true that the program `P` that you gave me will eventually halt when run on input string `P`." In this case, it is actually saying, "It is true that the program `paradox` that you gave me will eventually halt when run on the input program `paradox`." At that point, the `paradox` program enters an infinite loop. In other words, if the `haltChecker` says that the program `paradox` will eventually halt when run on the input `paradox`, then the program `paradox` actually runs forever when given input `paradox`. That's a contradiction; or perhaps we should say, that's a paradox!

Conversely, imagine that `haltChecker(P, P)` returns `False`. That is, it is saying, "The program `P` that you gave me will not halt on input `P`." But in this case, `P` is `paradox`, so it's actually saying "The program `paradox` will not halt on input `paradox`." But in that case, we've designed the `paradox` program to halt. Here again, we have a contradiction.

We could easily write the `paradox` program if we only had a `haltChecker`. However, we just saw that this leads to a contradiction. Thus, we're forced to conclude that our one assumption – namely that a `haltChecker` exists – must be false. In the same way, we concluded that an oracle necklace cannot exist because if it did then we could devise a logical trap just as we have done here!

## 7.5 Conclusion

In this chapter we've taken a whirlwind tour of two major areas of computer science: complexity theory and computability theory. Complexity theory is the study of the "easy" and "hard" problems - problems for which algorithmic solutions exist but the running time (or memory or other resources) may vary from modest to outrageous. In fact, complexity theory allows us to classify problems into more than just two categories "easy" and "hard", but into many different categories that ultimately provide deep and surprising insights into what makes some problems solvable by efficient algorithms while others require vastly more time (or memory).



*Thanks for reading!*

Computability theory explores those problems that are "impossible" to solve, such as the halting problem and many others. In fact, computability theory provides us with some very powerful and general theorems that allow us to quickly identify what problems are uncomputable. Among these theorems is one that says that virtually any property that we would like to test about the behavior of a program is uncomputable. In the case of the halting problem, the behavior that we sought to test was halting. Another behavior we might be interested in testing is that of having a virus that writes values into a certain part of your computer's memory. Results from

computability theory tell us that testing an input program for that property is also uncomputable. So, if your boss ever tells you, “your next project is to write a completely reliable virus detector,” you might want to read a bit more about computability theory so that you can show your boss that it’s not just hard to do that, it’s downright impossible.

## Table Of Contents

- Chapter 7: How Hard is the Problem?
  - 7.1 The Never-ending Program
  - 7.2 Three Kinds of Problems: Easy, Hard, and Impossible.
    - \* 7.2.1 Easy Problems
    - \* 7.2.2 Hard Problems
  - 7.3 Impossible Problems!
    - \* 7.3.1 “Small” Infinities
    - \* “Larger” Infinities
    - \* 7.3.2 Uncomputable Functions
  - 7.4 An Uncomputable Problem
    - \* 7.4.1 The Halting Problem
  - 7.5 Conclusion

**Previous topic** Chapter 6: Fun and Games with OOPs: Object-Oriented Programs

## Quick search

Enter search terms or a module, class or function name.

## Navigation

- index
- previous |
- cs5book 1 documentation »

© Copyright 2013, hmc. Created using Sphinx 1.2b1.

# Index — cs5book 1 documentation

1 capture

10 Sep 2019

| Aug  | SEP  | Oct  |
|------|------|------|
|      | 10   |      |
| 2018 | 2019 | 2020 |

success

fail

About this capture

COLLECTED BY

Organization: Internet Archive

The Internet Archive discovers and captures web pages through many different web crawls. At any given time several distinct crawls are running, some for months, and some every day or longer. View the web archive through the Wayback Machine.

Collection: Live Web Proxy Crawls

Content crawled via the Wayback Machine Live Proxy mostly by the Save Page Now feature on web.archive.org.

Liveweb proxy is a component of Internet Archive's wayback machine project. The liveweb proxy captures the content of a web page in real time, archives it into a ARC or WARC file and returns the ARC/WARC record back to the wayback machine to process. The recorded ARC/WARC file becomes part of the wayback machine in due course of time.

TIMESTAMPS



The Wayback Machine - <https://web.archive.org/web/20190910144749/https://www.cs.hmc.edu/csforall-book/genindex.html>

**Navigation**

- [index](#)
- [cs5book 1 documentation »](#)

## Index

[Symbols](#) | [\\_](#) | [A](#) | [B](#) | [C](#) | [D](#) | [E](#) | [F](#) | [G](#) | [H](#) | [I](#) | [K](#) | [L](#) | [M](#) | [N](#) | [O](#) | [P](#) | [Q](#) | [R](#) | [S](#) | [T](#) | [U](#) | [V](#) | [W](#) | [Z](#)

**Symbols**

## 2D arrays

eq  
init

\_str

A

|                 |                    |
|-----------------|--------------------|
| abstraction     | anonymous function |
| algorithm       | arrays             |
| analog computer | ASCII              |
| AND gate        | assembly language  |

B

|           |        |
|-----------|--------|
| base 10   | Binary |
| base 2    | bits   |
| base case | byte   |

C

|                               |                       |
|-------------------------------|-----------------------|
| central processing unit (CPU) | composition           |
| class                         | computability theory  |
| collaborative filtering       | computationally hard  |
| comment                       | conditional jump      |
| compiled                      | conditional statement |
| compiler                      |                       |

D

|                |            |
|----------------|------------|
| data mining    | dictionary |
| decryption key | division   |
| default values | docstring  |
| derived class  |            |

E

---

edit distance    empty string

F

G

global variable

## H

---

header hierarchy

---

## I

---

|               |                      |
|---------------|----------------------|
| immutable     | instruction register |
| index         | interpreter          |
| infinite loop | iteration            |
| inherits      |                      |

---

## K

---

keys

---

## L

---

|                       |                     |
|-----------------------|---------------------|
| lambda calculus       | list comprehensions |
| left-hand rule        | list concatenation  |
| linear time algorithm | loops               |
| list, [1]             |                     |

---

## M

---

|                             |                                |
|-----------------------------|--------------------------------|
| method                      | multiple assignment            |
| minterm expansion principle | multiple levels of abstraction |
| modular design              | mutable                        |

---

## N

---

|                             |                       |
|-----------------------------|-----------------------|
| natural language processing | numerical imprecision |
| no-operation                | nybble                |
| NOT                         |                       |

---

## O

---

|                |             |
|----------------|-------------|
| objects        | OR          |
| opcode         | OR gate     |
| open bracket   | overloading |
| operation code | overrides   |
| operators      |             |

---

## P

---

|                   |                 |
|-------------------|-----------------|
| parameter passing | predicate       |
| Picobot           | program counter |
| polymorphism      | public key      |
| pop               | push            |
| precedence        |                 |

---

## Q

---

quadratic time algorithm

---

## R

---

|                                 |                    |
|---------------------------------|--------------------|
| Random Access Memory (RAM)      | result             |
| recursive definition            | return value       |
| recursive substructure property | right-hand rule    |
| registers                       | ripple-carry adder |

---

## S

---

|                       |                      |
|-----------------------|----------------------|
| S-R latch             | stack pointer        |
| scope                 | state                |
| secret key            | string               |
| selection sort        | string concatenation |
| sieve of Eratosthenes | subclass             |
| sign-magnitude        | substitution         |
| slicing               | superclass           |
| stack, [1]            | symmetric            |
| stack discipline      | syntax               |

---

## T

---

|                 |                  |
|-----------------|------------------|
| top-down design | two's complement |
| truth table     |                  |

---

## U

---

|                    |       |
|--------------------|-------|
| unconditional jump | UTF   |
| Unicode            | UTF-8 |
| user-based CF      |       |

---

## V

---

|             |                          |
|-------------|--------------------------|
| value       | von Neumann architecture |
| values, [1] | von Neumann John         |
| variable    |                          |

---

## W

---

while loop    worst case

---

## Z

---

zoogenesis

---

### Quick search

Enter search terms or a module, class or function name.

## Navigation

- [index](#)
- [cs5book 1 documentation »](#)

© Copyright 2013, hmc. Created using Sphinx 1.2b1.