From Vision to Execution
The Creative Process in Computer Science
(with examples in Python)

DAVID G. WONNACOTT

Haverford College
Haverford, Pa.
# Table of contents

Prologue ................................................................. 7

1 What is Computer Science? ........................................... 9
  1.1 Problems and Solutions ........................................... 9
  1.2 Solutions and Non-Solutions ..................................... 11
  1.3 Comparing Algorithms ........................................... 13
  1.4 Summary .......................................................... 15
  1.5 Further Resources ............................................... 15
  1.6 On the Reading of Technical Writing ........................... 15

2 Understanding the Problem to be Solved: Specifications and Test Suites .............................. 17
  2.1 Preconditions and Postconditions ............................... 17
  2.2 Test Suites ...................................................... 18
  2.3 Putting it All Together: Solo and Team Approaches .......... 19
  2.4 Including Problem Definitions in Programs ................... 20
  2.5 Describing the Form of the Question .......................... 22
  2.6 Summary ........................................................ 24
  2.7 Further Resources ............................................... 24
  2.8 Exercises ....................................................... 26

3 Expression and Execution of Algorithms ........................................ 27
  3.1 Facts and Definitions ........................................... 27
  3.2 Expressing Algorithms as Python Functions .................... 28
  3.3 Execution by Following Instructions Step-by-step .......... 32
  3.4 Execution by Substitution ....................................... 35
  3.5 Complete Programs and Program Organization ................. 40
  3.6 Summary ........................................................ 42
  3.7 Further Resources ............................................... 42
  3.8 Exercises ....................................................... 43

4 Techniques for Algorithm Design ........................................ 45
  4.1 Mathematical Derivation ......................................... 46
  4.2 Top-down Design (and Bottom-up Implementation) ............ 49
  4.3 Design by Cases ................................................ 54
  4.4 Basic Recursive Design ......................................... 57
  4.5 Advanced Recursive Design ..................................... 60
  4.6 Generate and Filter ............................................. 62
    4.6.1 Testing if a Group is a Subclique ......................... 63
    4.6.2 Generating all Possible Subcliques ....................... 67
    4.6.3 Filtering .................................................. 70
4.7 Reducing a New Problem to a Solved Problem ........................................... 72
4.8 Summary ........................................................................................................ 72
4.9 Further References ......................................................................................... 73
4.10 Exercises ........................................................................................................ 74

5 Correctness of Algorithms ............................................................................. 77
5.1 Defining Correctness ...................................................................................... 77
5.2 Formal Verification ......................................................................................... 78
  5.2.1 Verification of our Exponentiation Algorithms ....................................... 83
  5.2.2 Of Proofs and Proofs ............................................................................... 85
  5.2.3 Of Programs and Proofs ........................................................................... 86
5.3 Testing ............................................................................................................ 86
5.4 Other Uses of Verification and Testing Techniques ...................................... 88
5.5 Summary ........................................................................................................ 88
5.6 Other Resources ............................................................................................. 89
5.7 Exercises ........................................................................................................ 90

6 Computational Complexity ............................................................................ 91
6.1 Describing Complexity .................................................................................. 91
6.2 Improving Complexity .................................................................................. 95
6.3 Giving Upper and Lower Bounds on Complexity ....................................... 97
6.4 Functions With Higher Complexity ............................................................. 98
6.5 Complexity, Models of Execution, and Program Speed ............................. 99
  6.5.1 Redundant Function Calls and Speed ..................................................... 99
  6.5.2 Tail Recursion and Speed ...................................................................... 100
  6.5.3 Operation Selection and Speed ............................................................. 100
  6.5.4 Memory Systems and Speed ................................................................. 100
  6.5.5 Concurrency, Speed, and Heat .............................................................. 100
  6.5.6 How to Make Programs Fast ............................................................... 101
6.6 Summary ........................................................................................................ 101
6.7 Further References ....................................................................................... 101
6.8 Exercises ........................................................................................................ 102

7 Imperative Programming ................................................................................ 103
7.1 Order of Execution ....................................................................................... 103
7.2 What is a Variable? Sequences and Updates ............................................. 104
  7.2.1 Testing Functions with Sequences of Variable Updates ....................... 105
  7.2.2 Reasoning About One Example with a “Box and Arrow Diagram” ..... 105
  7.2.3 Recording Facts for Each Program Point in Assertions .................... 107
  7.2.4 Converting to “Static Single Assignment Form” .................................. 107
7.3 if without else or return ............................................................................... 110
  7.3.1 Testing Imperative Programs with Conditionals ................................ 111
  7.3.2 Drawing Box and Arrow Diagrams for Conditionals .......................... 111
  7.3.3 Detailed Assertions and Conditional Statements ............................... 111
  7.3.4 Static Single Assignment and Gated Single Assignment Forms .......... 114
7.4 Loops ............................................................................................................. 117
  7.4.1 while loops ........................................................................................... 117
  7.4.2 for loops and range expressions ......................................................... 117
  7.4.3 Testing of Functions With Loops ......................................................... 119
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.4.4 Loops, Box and Arrow Diagrams, and Row and Column Diagrams</td>
<td>119</td>
</tr>
<tr>
<td>7.5 Lists and Modification of Objects</td>
<td>119</td>
</tr>
<tr>
<td>7.6 Creating and Using Procedures</td>
<td>120</td>
</tr>
<tr>
<td>7.7 Correctness of Imperative Programs</td>
<td>121</td>
</tr>
<tr>
<td>7.7.1 Code Reviews</td>
<td>121</td>
</tr>
<tr>
<td>7.7.2 Verification</td>
<td>121</td>
</tr>
<tr>
<td>7.7.3 Software Testing and Coverage Metrics</td>
<td>121</td>
</tr>
<tr>
<td>7.8 Summary</td>
<td>121</td>
</tr>
<tr>
<td>7.9 Further References</td>
<td>121</td>
</tr>
<tr>
<td>7.10 Exercises</td>
<td>122</td>
</tr>
<tr>
<td>8 Reasoning About Loops</td>
<td>123</td>
</tr>
<tr>
<td>8.1 Algorithm Design with Loops</td>
<td>123</td>
</tr>
<tr>
<td>8.2 Correctness of Loop-Based Algorithms</td>
<td>123</td>
</tr>
<tr>
<td>8.2.1 Loop Pre- and Post-Conditions and Invariants</td>
<td>123</td>
</tr>
<tr>
<td>8.2.2 Choosing Appropriate Logical Statements for Loops</td>
<td>124</td>
</tr>
<tr>
<td>8.2.2.1 Logical statements that allow a successful proof</td>
<td>124</td>
</tr>
<tr>
<td>8.2.2.2 Imagine an “evil debugger”</td>
<td>124</td>
</tr>
<tr>
<td>8.2.2.3 Identify plausible or implausible sets of variable values</td>
<td>126</td>
</tr>
<tr>
<td>8.2.3 Correctness Proofs for Loops</td>
<td>126</td>
</tr>
<tr>
<td>8.3 Loops and Tail Recursion</td>
<td>127</td>
</tr>
<tr>
<td>8.4 Anagrams and Sorting Algorithms</td>
<td>127</td>
</tr>
<tr>
<td>8.5 Summary</td>
<td>127</td>
</tr>
<tr>
<td>8.6 Further References</td>
<td>127</td>
</tr>
<tr>
<td>8.7 Exercises</td>
<td>128</td>
</tr>
<tr>
<td>Appendix A A “Starting Set” of Python</td>
<td>129</td>
</tr>
<tr>
<td>A.1 Elements of a Python Program</td>
<td>129</td>
</tr>
<tr>
<td>A.2 Comments</td>
<td>129</td>
</tr>
<tr>
<td>A.3 Functions and Parameters</td>
<td>130</td>
</tr>
<tr>
<td>A.4 Statements and Expressions</td>
<td>131</td>
</tr>
<tr>
<td>A.4.1 return statements</td>
<td>131</td>
</tr>
<tr>
<td>A.4.2 Literal value expressions</td>
<td>131</td>
</tr>
<tr>
<td>A.4.3 Function call expressions</td>
<td>131</td>
</tr>
<tr>
<td>A.4.4 Arithmetic expressions</td>
<td>131</td>
</tr>
<tr>
<td>A.4.5 Comparison expressions and Boolean operators</td>
<td>132</td>
</tr>
<tr>
<td>A.4.6 Strings and string operations</td>
<td>133</td>
</tr>
<tr>
<td>A.4.7 Definitions of parameters and local variables</td>
<td>133</td>
</tr>
<tr>
<td>A.4.8 Conditional statements and expressions (if)</td>
<td>134</td>
</tr>
<tr>
<td>A.4.9 Variable use expressions</td>
<td>136</td>
</tr>
<tr>
<td>A.4.10 Preconditions, postconditions, and progress conditions</td>
<td>136</td>
</tr>
<tr>
<td>A.4.11 Input and output</td>
<td>137</td>
</tr>
<tr>
<td>A.4.12 Separating input and output from computation</td>
<td>137</td>
</tr>
<tr>
<td>A.4.13 Errors</td>
<td>138</td>
</tr>
<tr>
<td>A.4.14 Loops and multiple definitions</td>
<td>139</td>
</tr>
<tr>
<td>Appendix B Proof Techniques</td>
<td>140</td>
</tr>
<tr>
<td>B.1 What is a proof?</td>
<td>140</td>
</tr>
<tr>
<td>B.2 Axioms of Boolean Algebra</td>
<td>143</td>
</tr>
<tr>
<td>Section</td>
<td>Page</td>
</tr>
<tr>
<td>----------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>B.3 Axioms for Integers</td>
<td>144</td>
</tr>
<tr>
<td>Appendix C Logical Assertions Program File</td>
<td>145</td>
</tr>
<tr>
<td>Index</td>
<td>146</td>
</tr>
</tbody>
</table>
Prologue

These course notes cover most of the first semester computer science course at Haverford College. They are a work in progress, with occasional omissions or “need to add this” notes, but serve as a guide and supplement to most of material of our course. The goal of this course is to introduce students without prior experience with computing to fundamental intellectual tools of computer scientists. This includes algorithms and programs, and also approaches to thinking about them, such as call trees, specifications, verification, code reviews, and testing methodologies. We have found that this focus provides those students who wish to continue with computer science (either for its own sake, or in support of some other goal) with an excellent foundation for further work, and that it provides students who seek only a single semester (or year) of computer science with a deep appreciation of the thought processes involved in this field.

The URL http://cs.haverford.edu/people/davew/FVtE/FVtE-2015.pdf can be used to obtain a copy of this 2015-2016 edition of these notes. Permission is granted for individuals to use this PDF if directly downloaded from that site, but please do not distribute copies. If you would like to assign this as reading for a course, or edit it, please contact davew@cs.haverford.edu.
Chapter 1
What is Computer Science?

Computer science is the study of the representation and manipulation of information. Computer scientists create and study information processing systems in many ways; some require the use of a computer, but others require only a place to sketch a diagram or equation, or a friend with whom one can discuss ideas. It is easy to overlook communication and contemplation when one thinks of the creation of computer software, but these are at least as important and rewarding as tinkering with a computer, and thus must play a central role in the study of computer science.

In this first course, we focus almost entirely on the manipulation of information, leaving detailed questions of representation of information for a later course. The information we manipulate will be represented in ways that should be familiar, for example numbers can be represented with a sequence of numerals, for example using 42 for the number forty two. Numbers can be represented in other ways: the ancient roman system used letters, with XLII representing the number we write as 42; the ancient Mayan system arranged yet another set of symbols in yet another way. However, the properties of the numbers remain the same regardless of how we represent them, e.g. seventeen more than twenty five must always be forty two.

1.1 Problems and Solutions

Clear and precise communication is often well served by definitions and by examples. We begin our discussion of information manipulation with some of each, to bring out the subtle meanings of familiar terms like “problem” and “solution” in the context of computer science.

Definition 1.1. A problem instance is a precise question or request about specific information.

For example, “Is 37 prime?” and “Given a map of the United States, find a set of colors for states such that no two neighboring states (not including states that only touch at one point, such as Colorado and Arizona) have the same color.”, are problem instances. A problem instance may have one answer (as in the case of testing whether 37 is prime) many answers (as in the map coloring), or no answer (“Find an integer that, when multiplied by itself, gives 17.”).

Definition 1.2. A general problem (or simply a problem, if the context makes this clear) is a precise question that can be asked about any element of a well-defined class of inputs.

For example, “Given any positive integer, determine whether or not it has any integer factors other than 1 and itself.” and “Given any two-dimensional map made up of a collection of finite regions, find a set of colors such that no two neighboring regions (not counting regions that touch only at one or more corners) have the same color.”, are general problems. Note that it is important to state a problem or problem instance as precisely as possible, spelling out exactly what is meant in vague cases (such as states with touching corners).
While a problem instance can be answered with specific information (e.g., the answers “Yes, 37 is prime.” or a colored map in which New York is blue, Pennsylvania red, etc.), general problems typically cannot. We answer general problems (when possible) with a set of precise instructions that must lead to a correct result for any input in the allowed class. For example, we can answer the general problem of testing to see if a positive integer \( x \) is prime with the instructions “For each integer \( y \) between 2 and \( x - 1 \), find \( \frac{x}{y} \), and report “Not Prime” if one of these ratios is an integer, and “Prime” otherwise.”. For the map coloring problem, we can answer “Create a new color for every region on the map.” (assuming we can always distinguish as many different colors as there are regions). This kind of answer is known as an algorithm.

**Definition 1.3.** An **algorithm** is a specific set of instructions that can be followed to produce a correct answer to a general problem, for any legal input allowed for that problem.

In this course, we will be studying algorithms — how to create them, understand why and how they work, and compare them to each other. We will frequently name our algorithms so that we can refer back to them later. We will refer to the above algorithm for checking to see if an integer is prime as **Algorithm Prime-1**, and the above algorithm for map coloring as **Algorithm Color-1**.

For the purposes of this course, we will not bother to define “computer” beyond simply stating that it can execute our algorithms:

**Definition 1.4.** A **computer** is anything that can follow an algorithm.

Thus, a computer that has been given an algorithm to solve a general problem can, without further instruction, produce the specific answer for any instance of the problem.

The term “computer”, like the term “dishwasher”, has changed over time from a job title for a human being to a name for a device. The original computers were people who followed instructions to solve mathematical problems, but the term is now used primarily for digital electronic information processing machines. The configuration and use of analog computers, while fascinating in its own right, is outside the scope of this course.

A digital computer receives its input as a sequence of symbols. It is often useful to re-state problems so that their input and output use a standard finite set of symbols, such as the ten Hindu-Arabic numerals used for writing numbers, or the letters of the Roman alphabet. For example, we could re-state the map coloring problem as **Given a comma-separated list of state names, followed by**

**Box 1: A Human Computer Finds 527 Digits of \( \pi \) and 180 Digits of Garbage.**

William Shanks, an English schoolteacher who lived from 1812-1882, followed algorithms to compute the decimal representations for a variety of interesting numbers, including \( \pi \) and \( e \). He followed the algorithm for \( \pi \) at night over a period of 28 years, producing 707 digits. Unfortunately, he made a mistake, and what he actually produced was the first 527 digits of \( \pi \) followed by 180 digits that were wrong (this fact was not discovered until the 1940’s). Other human computers of Shanks’ time produced tables of values that were important for navigation. These greatly simplified navigation across the Atlantic, except of course when errors caused shipwrecks.

The classic definitions of algorithm and computer are based on the assumption that the computer will not make errors; the field of fault-tolerant computing explores the design of systems that can produce a correct final answer despite the presence of one or more errors during the calculation.

Copyright 2003-2015 David G. Wonnacott; see Page 7 for terms of free use
a comma-list of pairs of states with common borders, produce a list in which each state name appears once and is followed by the decimal expression of numbers between 0 and 1 representing the relative amounts of red, green, and blue in the color to be used for the state." In this case, a problem instance might look like “NY, Pa, Md, NY Pa, Pa Md”, and an output like “NY 0 0 1, Pa 1 0 0, Md 0.7 0 1”.

A digital computer also receives its algorithms as a sequence of symbols, arranged according to the rules of a programming language. The process of expressing the idea of an algorithm in a given language is referred to as encoding or coding the idea in the language, and the result is referred to as the code for the algorithm. Note that the word “code” here is not meant to suggest that software is necessarily cryptic (though some of it certainly is); in this context it simply means a representation engineered for a specific purpose. The software encodes our ideas in the same sense that Morse code encodes letters of the alphabet in a form suitable for sending via telegraph. In this course, we will discuss algorithms informally in English-language text, and then encode them formally in the Python programming language.

Check Your Understanding 1.1. Give an informal description of one instance of the “Given the names of two places on campus, give a route between them” problem. How might the input and output for this problem be represented as text? How might they be communicated with graphical input and output?

Check Your Understanding 1.2. Pick any information processing challenge from your everyday life and describe it as a “general problem”, giving one problem instance and its solution as an example.

1.2 Solutions and Non-Solutions

While a problem instance may have zero, one, or many solutions, a general problem can usually be solved in many ways if it can be solved at all. For example, we could solve our “prime testing” problem with the algorithms “For each integer y between 2 and \( \sqrt{x} \), find \( \frac{x}{y} \), and report “Not Prime” if one of these ratios is an integer, and “Prime” otherwise.” (we’ll call this Algorithm Prime-2), or “For each integer y between 2 and \( \sqrt{x} \), find \( \frac{x}{y} \), and report “Not Prime” if one of these ratios is an integer, and “Prime” otherwise.” (Algorithm Prime-3). General problems may also be fundamentally unsolvable, for reasons that may be clear (“Given any positive integer \( x \), find a positive integer that, when multiplied by itself, gives \( x \.”) or subtle (“Given an algorithm, determine whether or not there is some input that will cause it to run forever.”).

A general problem will also have an infinite number of non-solutions — for example, the instructions “Color all the states green.” will produce an incorrect coloring for almost any map (except, for example, a map with only one state); the instructions “Report “Not Prime” if the decimal digits of \( x \) sum to a multiple of 3, or if the last digit is even or 5; report “Prime” otherwise.” give the wrong answer for 2, 91, and an infinite number of other odd integers (we’ll call this Non-algorithm Prime-4). For the purposes of this course, we will use the academic definition of correctness (it solves all instances) rather than the economic one (it works well enough to let you sell more products than your competitors):

Definition 1.5. An algorithm is correct if and only if it solves all instances of a problem.

Much of the creative energy of computer scientists is focused on producing ideas that are correct in this sense, not just clever tricks that usually seem to work pretty well. There are two basic approaches to studying correctness: formal verification, in which mathematical reasoning techniques are used to prove the algorithm is correct, and testing, in which an algorithm is tried out on a number of sample inputs, and each output is checked for correctness.
Since an algorithm is only considered correct if it solves all instances of a problem correctly, testing can definitively prove that a proposed algorithm is incorrect — Non-algorithm Prime-4 is incorrect because it answers “Prime” when $x = 91$ (note that $91 = 7 \cdot 13$). However, testing can only definitively prove an algorithm correct if the algorithm has a finite (and reasonably small) set of possible inputs, so that every possible input can be tested. Testing is still used extensively in practice, since it can quickly provide some degree of confidence (though not mathematical certainty) that a proposed algorithm, and the code that implements it, are both correct.

As we shall see in Chapter 5, the key to effective testing is the creation of a test suite that covers a variety of different kinds of problem instances. Tests created by a program designer often focus on each aspect of the algorithm used to solve the problem, and can be used to ensure that each part of the algorithm has been tried at least once. Tests created by an independent tester may identify situations that were not foreseen by the program designer. Thus, testing is best done as a collaboration between the program designer(s) and independent tester(s). Additionally, test suites can serve as a collection of examples illustrating instances of the problem being solved, i.e., as a way of recording or communicating thoughts about the nature of the problem itself, as discussed in Chapter 2.

Formal verification can be used to definitively prove an algorithm is correct. A verification may combine a proof that the algorithm actually corresponds to certain mathematical facts (using techniques we will discuss later) with a proof of the underlying mathematics (if necessary). For example, a verification of Algorithm Prime-3 would involve a proof that our algorithm will find a factor if one exists in the range $[2, \lfloor \sqrt{x} \rfloor]$, and a proof that a number is prime if and only if it has no factors in this range.

However, formal verification has several drawbacks. First, it requires the construction of a mathematical proof that may be as hard to understand as the algorithm itself. Second, it requires a formal specification: a precise mathematical definition of the problem that the algorithm is to solve. For inherently mathematical problems, this is often straightforward, but for real-world problems, such as the piloting of an aircraft, the construction of a formal specification can be as difficult as the algorithm design that follows it. For these reasons, formal specification and verification is generally used only in limited circumstances where correctness is very important — for example, in the construction of aircraft control systems, a partial formal specification may be used to prove the control system obeys certain safety properties (we may wish to prove that the autopilot won’t engage the thrust reversers during flight, even if we don’t prove it always achieves maximum fuel economy).

Formal verification techniques, like testing techniques, can be used to both ensure correctness of a solution (Chapter 5) and describe the problem being solved (Chapter 2). We will study both testing and formal verification in this course, focusing on using these techniques for reasoning and communicating about algorithms, even if we never create a full formal verification or a complete test suite. In practice, any process that forces programmers to think hard about a program can help to find errors. Other popular techniques include code review, an examination of software by other programmers, and pair programming, in which programmers work together in every step of program development, providing a continuous review as the program is written.

Check Your Understanding 1.3. Use Algorithms Prime-2 and Prime-3, and non-algorithm Prime-4, to check whether each reports “Prime” or “Not Prime” for (a) 71 and (b) 91. (Use a calculator for the arithmetic if you wish.)

Check Your Understanding 1.4. Which technique (formal verification, testing, and peer review) do you think would most quickly detect the problem with Prime-4? Which technique would give you the most confidence that Prime-2 or Prime-3 is correct?
1.3 Comparing Algorithms

There are a variety of ways to compare the algorithms for solving a given problem. We will not worry about comparing various non-solutions to a problem, or comparing algorithms for different problems.

Some algorithms may produce better solutions, for example maps with fewer colors (which may be cheaper to print) or prettier maps. We will generally approach this issue by adjusting the problem statement to identify the solution(s) we want. For example, we could pose the problem of finding a coloring with the minimum possible number of colors.

Algorithms may vary in terms of what basic instructions we must already understand, i.e., those that are not explained in the algorithm itself. Algorithm Prime-1 requires that we know how to identify all the integers in a given range (2 to \(x - 1\), in this case), how to divide one integer by another, and how to know if the result is an integer; Algorithm Prime-3 requires us to find square roots as well. Algorithm Color-1 requires us to come up with a set of colors (either by painting on a map or by giving numerical values for the relative amounts of red, green, and blue).

We will compare algorithms in terms of their \textit{cost}, measured with their use of basic instructions. For historical reasons, this is known as the \textbf{computational complexity} of an algorithm. When two algorithms rely on the same basic operations (such as division) we can compare the number of basic instructions performed. For example, Algorithm Prime-1 does \(x - 2\) division operations, while Algorithm Prime-2 does \(\left\lfloor \frac{x}{2} \right\rfloor\) (counting the calculation of \(\frac{x}{2}\) as 1 division), so for \(x > 4\), Algorithm Prime-2 has lower complexity (it does about half the number of divisions for large values of \(x\)).

Comparing the complexities of algorithms that employ different basic instructions is somewhat trickier. Algorithm Prime-3 does far fewer divisions for large values of \(x\), but it also requires the calculation of a square root (or, to be precise, the identification of the closest integer below a square root — one algorithm for finding this involves computing the square root and rounding down). There are two approaches to comparing the complexities of algorithms with differing basic instructions. We can make a precise comparison of the actual time needed to run each algorithm on some specific computer if we know the speeds of each operation. For example, if division takes 1 nanosecond (i.e., one billionth of a second) and square root calculations take 100 nanoseconds, then Algorithm prime-2 will require at least \(\left\lfloor \frac{x}{2} \right\rfloor\) nanoseconds to test the integer \(x\) and Algorithm Prime-3 will take at least \(100 + \left\lfloor \sqrt{x} \right\rfloor - 1\). Note that \(100 + \left\lfloor \sqrt{x} \right\rfloor - 1 < \left\lfloor \frac{x}{2} \right\rfloor\) is less when \(x > 230\).

Alternatively, we may be able to make a statement about the relative complexities that is independent of any particular computer by studying the behavior of the algorithms as the input size grows without bound. This form of analysis is known as \textbf{asymptotic complexity analysis}, and is one of the fundamental tools of the computer scientist. In our primality testing example, the number of divisions increases with \(x\) in every algorithm, and the number of square root calculations stays fixed. Thus, we expect the number of divisions to be the factor that limits the speed of the calculation for large values of \(x\). On any computer, there must be some value of \(x\) beyond which the time required for the extra \(\left\lfloor \frac{x}{2} \right\rfloor - (\left\lfloor \sqrt{x} \right\rfloor - 1)\) divisions takes longer than a single square root calculation.

The asymptotic complexity discussed above is actually somewhat oversimplified, in that we have assumed that all division operations take the same amount of time, and that any square root calculation takes the same amount of time. A more precise analysis would account for the fact that some divisions (and square root calculations) are more difficult than others, by looking at the resources needed for each basic operation as a functions of the number(s) involved. We could treat each operation as requiring some number of fixed-difficulty operations (such as operations on single digits), or we could describe the time required for each basic operation. However, this sort of analysis is beyond the scope of this course (and often omitted in practice as well, except when the number of digits grows very large, as in encryption systems).
Computer scientists use asymptotic complexity analysis to classify the complexity of problems as well as algorithms — the complexity of a problem is defined as the complexity of the least complex algorithm for solving the problem. For example, finding the smallest item in a list of $n$ numbers must take at least $n$ operations, since we must at least examine each number on the list. Since it is possible to identify the smallest item in about $n$ operations (such as pairwise comparisons), we say that the complexity of this problem is proportional to $n$, even though it is also possible to create an algorithm that uses, for example, $n^2$ pairwise comparison operations.

It is also possible to compare two algorithms in terms of the difficulty of expressing or understanding them. For example, most people find it easier to understand the primality testing algorithms given above than the recently developed Number Field Sieve algorithm. There is no standard term for this property in the computer science literature, but we will refer to it as cognitive complexity (to distinguish it from computational complexity discussed above). It is difficult to make a precise measurement of cognitive complexity, as it depends on many factors, such as the language used to express the algorithm, the ability of the algorithm’s author to provide a clear explanation of how and why the algorithm works, and the mathematical background of the reader of the algorithm. In this course, we will discuss cognitive complexity primarily by example rather than with formal methods.

Unfortunately, it is often the case that algorithms with lower computational complexity have higher cognitive complexity. This often forces us to choose between two desirable properties of algorithms, though in some cases we can reduce cognitive complexity by providing a clear explanation along with an algorithm.

**Box 2: Factoring, Encryption, and Privacy.**

The field of asymptotic complexity analysis may sound like something arcane that is only of interest to academics, but our knowledge of the asymptotic complexities of various algorithms plays an important role in the design of systems that are important to the safeguarding of private information every day. For example, certain encryption systems, such as those that transmit credit card numbers to “secure” web sites, are based on certain problems for which the reverse problem seems to have much higher computational complexity.

For example, some encryption systems rely on the fact that factoring, which appears to have high computational complexity, is the reverse of multiplication, which has low computational complexity. Encryption systems based on such problems are known as “public-key” encryption systems. The term “public-key” refers to the fact that the “key” needed to encrypt a message can be publicly distributed without revealing the “private key” that is needed to decrypt a message. If our public key is the product of two large prime numbers, we can distribute it (the product) without fear of revealing our private key (the numbers we multiplied to get the product) to anyone who lacks a much more efficient factoring algorithm.

This leads to the question “How fast could factoring of large numbers possibly be?”, in other words, what is the computational complexity of factoring? The answer to these questions is not known — there is no known classical algorithm for factoring large numbers quickly, but there is also no proof that no such algorithm can exist (though many mathematicians suspect it cannot). However, the developing field of quantum computing suggests that there may be ways to factor large numbers quickly with non-classical devices, though this has not (yet) been achieved for large numbers.
In this course, we will provide informal discussions of the measurement of computational complexity and the control of cognitive complexity via clear expression of an algorithm. Details of the measurement of computational complexity are covered in a second semester course on data structures.

**Check Your Understanding 1.5.** Suppose we wish to create a public key by multiplying together two 20-digit prime numbers to produce a 40-digit number. Given the assumptions above (10^{-9} seconds per division and 10^{-7} seconds per square root operation), how long would Algorithms Prime-2 and Prime-3 take to confirm that our two 20-digit numbers are indeed prime? How long would each algorithm take to demonstrate that the resulting 40-digit product is not prime (and thereby find the factors)?

**Check Your Understanding 1.6.** Suppose a well-funded government lab purchases 1000 supercomputers, each of which is 1000 times faster than our computer, and can use these to run any of our algorithms 1000000 times faster. How long will it take this lab to use Algorithm Prime-3 to find the factors of our 40-digit number if it uses all of its computing resources entirely for this purpose? If our paranoid friend decides to use two 24-digit primes to get a 48-digit public key, how long would they need to check each 24-digit number with Algorithm Prime-3? How long will the lab of supercomputers to use Algorithm Prime-3 to find the factors from the 48-digit key?

**Check Your Understanding 1.7.** Suppose a brilliant mathematician discovers a way to find the factors of \( x \) by using \( \log_{10}(x) \) (approximately the number of digits in \( x \)) operations rather than \( x^2 \) or \( \sqrt{x} \) operations. If these operations take 10000 nanoseconds (10^{-5} seconds) on our hypothetical computer, how long would it take for this mathematician to factor our 40-digit key? Our friend’s 48-digit key?

### 1.4 Summary

In this chapter, we have introduced the basic objects of study for this course, the **general problem** and the **algorithm**. We have also introduced some of the basic tools that computer scientists use to study these objects: **formal verification** and **testing** techniques are used to study correctness; **asymptotic complexity analysis** is used to study resource usage; **pair programming** and **peer review** are among the ways groups work together to develop or understand algorithms and the programs that express them. The rest of this course will be devoted to the study of these objects and tools.

### 1.5 Further Resources

The information for Box 1 came from two sources: the first episode of WGBH’s video series “The Machine that Changed the World” (copyright 1992 WGBH Educational Foundation, available from “Films for the Humanities and Sciences, Inc.”, of Princeton, NJ), and the MacTutor History of Mathematics Archive of the School of Mathematics and Statistics University of St Andrews, Scotland (http://www-gap.dcs.st-and.ac.uk/~history/Mathematicians/Shanks.html). Cryptography references need to be added here...

### 1.6 On the Reading of Technical Writing

Some readers may not be familiar with the art of reading material with significant mathematical content. Your first instinct may be to read through all the text once, in order, and glance at the formulas and examples to see if they seem consistent with the text. However, experience shows this is merely an excellent way to create a false sense of security. In technical writing, much (or all) of the information is contained in the definitions and formulas, and the text is merely a guide to the easiest route toward understanding them.
A more successful approach goes something like this: skim the text once, without reading the formulas, to get a general idea of what the author is trying to say; then read the text again, this time paying particular attention to the formulas (or, in a computer science book, the algorithms). To “pay particular attention to a formula” means not just looking at it carefully, but trying to get a feeling for where it fits in the overall discussion. This can only be done once one is familiar enough with a formula to apply it. So, after reading a formula, stop to work out an example problem — hopefully one has been provided in the text, and if so, skim it once and then copy the statement of the problem onto blank paper and try to re-work it without referring to the original (if you really understood it, this should be easy and take very little time, but don’t be surprised if this process reveals gaps in your understanding and you have to look at how it was worked out in the original text). Repeat this process until you can re-work the example without help. At this point, try a variation on the example (hopefully one has been provided in the text, perhaps in a list of exercises at the end of a chapter). Once you can work several examples by yourself, you are familiar enough with a formula to apply it, and are then ready to try to fit it into the big picture — try working out examples that rely on a combination of this and earlier formulas and definitions. Consider what would happen if a definition (or algorithm) had been slightly simpler — which parts of what follows would still be true, and which would not? Boxes and other supplemental material can be read any time you need a break from this process.

At this point, you are probably asking “Come on, do you really read things this way? Doesn’t it take forever?”. My answer: Of course I don’t, it would take forever. But I have, through painful experience, gotten a good instinct for whether I really understand something after a quick reading (typically if I’m already familiar with the topic) or whether I’ve got a false sense of security. If I think I really understand something, I skip to a moderately complex exercise (making this up for myself if there isn’t one provided), and try it to see if I really understand. If I can’t do this, or I had an uneasy feeling about my understanding as I read something, then I either start working through the examples and exercises or (if there isn’t time) I give up and admit I don’t actually understand it. But this way I don’t get surprised by my own ignorance.
Chapter 2
Understanding the Problem to be Solved: Specifications and Test Suites

A clever solution to the wrong problem often has little or no value. One way to avoid such wasted effort is to become familiar with a problem before starting to solve it. Techniques from both testing and verification can be used to this end, and they are often combined to provide both a precise definition and a set of examples that capture the goal of an algorithm.

These techniques, like the algorithm development techniques we’ll see later, can be practiced alone or as a team. A team can collaborate on one kind of problem description, or different team members may develop different types of descriptions and then confirm that they all understand the problem in the same way.

2.1 Preconditions and Postconditions

From the field of formal verification, we borrow the idea of a problem specification. A precise specification for a problem is made up of a precondition and postcondition.

A precondition gives rules that tell us just what does or does not count as a legal instance of the problem. It makes sense to ask whether 187 is prime, but not to ask whether 3.5 is prime or whether a cat is prime; the children’s game “Pig Latin” involves playing with words, not numbers or food; we can ask for the square root of a positive number, but not for the square root of a negative number (unless we wish to allow imaginary numbers as answers) or for the square root of a word. Thus, the real-number version of the “square root” problem has the precondition that we must ask about a non-negative number; the “pig latin” problem that we ask about a word; and the “prime” problem that we ask about a positive integer.

Some problems allow flexibility in how the question is asked. In such cases, a given algorithm will be defined for a one particular way of phrasing the question, and precondition must be written in terms of this same information. For example, for the “window overlap” problem that we will see in Section 2.5, we could provide information about the horizontal location of the window in terms of the positions of the window’s left and right borders, or in terms of the position of the left border and the width of the window — either way, we would be able to tell the horizontal location the screen. In the former case, our precondition might require that the given right-hand border does not correspond to a position that is to the left of the left-hand border. In the latter case, our precondition might state that the width must not be negative.

A postcondition gives a rule or general test that lets us tell whether we have the right answer. For a mathematical problem, the postcondition can often be given as a precise mathematical formula or statement, such as \[ \sqrt{x} \cdot \sqrt{x} = x \] or “\( x \) is prime iff \( \exists y, y \neq 1 \land y \neq x \), such that \( x \mod y=0 \)” (meaning “\( x \) is prime if and only if there is no \( y \) except 1 and \( x \) such that \( x \) divided by \( y \) has a remainder of 0”, i.e., the mathematical definition of a prime number). If approximate answers are allowed, the postcondition must make this clear, and should describe the degree of approximation allowed (perhaps we want the square root computed to six digits of accuracy, or perhaps we want the most accurate answer that can easily be represented by the available computer hardware).
For a less mathematical problem, we typically use words to precisely describe the correct result, attempting to capture all situations that would be allowed by the precondition. For example, we could use a text description if asked to define the children’s game of “pig latin”, in which words are rearranged to put initial consonants at the end and the suffix “ay” is added, turning “pig” into “igpay”. Our description would need to address all questions that might arise about what counts as a correct answer, such as “How many initial consonants should be moved?”, “When should we treat the letter ‘y’ as a vowel?”, and “When we are given a word starting with a capital consonant, do we capitalize the starting letter of the result, or the letter that was initially capitalized?”.

Check Your Understanding 2.1. A palindrome is a word or phrase that has the same sequence of letters when written backwards, such as “Lived on decaf, faced no devil”, or “A man, a plan, a canal: Panama”. Write (in English) the precondition and postcondition for a function that tests whether or not a string is a palindrome.

Check Your Understanding 2.2. Give a precondition and postcondition for division — in other words, what must be true for the expression $\frac{a}{b}$ to be legal, and if $q$ is proposed as an answer to $\frac{a}{b}$, what calculation could we do to confirm it is correct? Try to express your answer mathematically (rather than in informal English), and do not use division itself in either the precondition or postcondition.

Check Your Understanding 2.3. The length of the hypotenuse (i.e., the longest side) of a right triangle whose other sides have lengths $a$ and $b$ is $\sqrt{a^2 + b^2}$. Give a precondition and a postcondition for an algorithm to find the length of the hypotenuse. How much do you need to restrict the values of $a$ and $b$? Can you state the postcondition without using the $\sqrt{}$ (square root) operation?

Check Your Understanding 2.4. The distance between two points $(x_1, y_1)$ and $(x_2, y_2)$ in a “Cartesian coordinate plane” can be computed with the formula $\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$. Give a precondition and a postcondition for an algorithm to compute distances between such pairs of points.

2.2 Test Suites

From the field of software testing, we borrow the idea of a test suite. A test suite is simply a collection of examples that illustrate all the interesting cases of a problem, along with the expected answer (or answers) for each case.

Much of the art of creating a good test suite comes in one’s interpretation of what counts as “interesting cases” — we can’t very well create a test suite for the “prime” problem by listing all the positive integers and indicating which are considered prime, but it is probably good to have at least one prime numbers (such as 7), at least one composite number (such as 12), and one or two examples that people find confusing, such as 2 (the only even prime number) and 1 (which is technically neither prime nor composite, so our algorithm should report “Not Prime”).

If a problem has only a few possible answers, a good test suite will elicit each of them at least once. For problems that allow only true/false answers (such as “prime”, “window overlap” from Section 2.5 or “palindrome” from Check Your Understanding 2.1), it is often useful to have both obvious and “near miss” examples of both true and false answers. In a test for the window overlap problem, very clear examples might include two small windows that are far apart (clearly not overlapping), or two large windows with a large overlap area; “near miss” examples might include windows with only a slight overlap, even a single point on the screen, or windows that are very close without overlapping. A test suite with a variety of “near miss” examples can sometimes identify problems in an algorithm that would pass a simpler test suite, and test suites written by experienced programmers can contain many such cases (e.g. windows with near misses for each side and each corner, in the window overlap example).
If a problem involves complex or subtle rules, a good test suite would illustrate each interesting case. A test suite for the pig latin problem should illustrate what happens to words with many initial consonants, how to deal with the letter ‘y’, and what to do with capital letters. When there are many different rules, we may need to explore how the rules interact when combined with each other, perhaps including cases with both capital ‘Y’ and lower case ‘y’. This decision of how to illustrate combinations of rules is another central challenge in the art of test suite construction: if the rules do not interact in any interesting way, we may not need to explore combinations (as in the case of pig latin); but if there are subtle interactions, we may want to explore many of the potentially huge number of ways of combining rules.

Depending on their expected use, test suites may or may not need to include examples of illegal problem instances (such as √−1 or √word for the square root problem). If the function guarantees some sort of specific response to illegal parameters, we should include a test to confirm that we get the appropriate response. However, this can also complicate automated testing, so in some cases we will simply use a comment to describe the difference between legal and illegal instances and describe what guarantees (if any) are made about detecting illegal instances. In a certain sense, this is just an argument about the definition of “illegal”: if we provide a function sqrt that guarantees not only to produce the (approximate) square root of its parameter when possible, but also to provide some specific response when there is no square root (e.g., for sqrt(‘word’)), then it is, in a sense, legal to call the function and expect a response. In other words, sqrt(‘word’) would be legal even though √word is not. A test suite and associated comments give us a chance to explain just what can and cannot be expected when the function is called in a variety of circumstances. The act of constructing a test suite provides an opportunity to think hard about these issues before we start to program.

Check Your Understanding 2.5. Provide several examples that would serve as a good test suite for a palindrome testing function (from Check Your Understanding 2.1). Give both the problem instance, and the expected answer. How many examples do you need to give a good sense for the “interesting cases”?

Check Your Understanding 2.6. Provide several examples of division, including both questions and answers. How many are needed to illustrate the interesting cases?

Check Your Understanding 2.7. Provide examples for a distance-finding algorithm as in Check Your Understanding 2.4.

2.3 Putting it All Together: Solo and Team Approaches

The above techniques can be used by an individual or a team; several popular approaches to team software development can be applied even at this early stage.

In the pair programming approach, one member of a team writes part of a computer program while another watches. The watcher checks for mistakes, but perhaps more importantly, the watcher can remain focused on how the current step fits into the “big picture”. During the development of a specification or test suite, the watcher could keep a list of other promising ideas for tests, or ideas that must be captured in the specification, while the other team member dives into the answer to the current test, or one facet of the specification.

Program development can be facilitated by a peer review process such as code review. In a code review, the software is presented to programmers who have not been working on it. This makes the author(s) present their ideas “from the top” again, possibly exposing oversights or providing fresh insight. This technique can also be used to take a fresh look at a test suite or specification.
Finally, a team may choose the classic approach of dividing the work into parts and then combining the results — some team members develop a test suite, and others a specification. Once both products are complete, the two groups meet and confirm that the questions and answers given in the test suite fulfill the pre- and post-conditions given in the specification. This last “confirmation” step may be straightforward, possibly so straightforward that the team members are tempted to skip it. However, finding an inconsistency at this point can save many hours of wasted programming time.

**Check Your Understanding 2.8.** Discuss any of your answer to any previous check-your-understanding question from this chapter with some of your classmates. Are your answers different? If so, is it just a matter of different legitimate interpretations of the question, or did one of you think of something the other missed?

### 2.4 Including Problem Definitions in Programs

Since understanding a problem is a vital part of solving it, computer programmers often provide detailed descriptions of the problem solved by their program. Most programming languages have some way to intersperse descriptive statements by the programmer with the program itself. These descriptions are known as “comments”. As we will see in Chapter 3 and Appendix A, the Python language allows several forms of comments.

Figure 2.1 shows how a programmer might describe the prime number problem. This text could be placed at the top of a computer file containing a Python program to solve this problem. The comment includes both the traditional mathematical definition (written with a regular computer keyboard rather than specialized symbols) and some examples that are formatted so that each looks like a Python function call (the notation used to make use of a Python function, which would be used to express our algorithm, within a Python program), followed by the expected result. This notation allows the use of certain automatic testing tools.

```python
""
The goal here is to determine whether or not a given integer is a prime number.

PRECONDITION:  To determine whether "is_prime(x)" is True or False, x must be a positive integer
POSTCONDITION: "is_prime(x)" is True if there is no integer other than 1 and x that evenly divides x, and False otherwise.

For example,
>>> is_prime(12)
False
>>> is_prime(7)
True
>>> is_prime(2)
True
>>> is_prime(187)
False
>>> is_prime(1)
False
"""
```

**Figure 2.1.** Defining the “Prime” Problem
Figure 2.2 illustrates a combination of precise definition and examples to describe one variant of the pig latin problem, using a combination of English text and examples. Figure 2.3 shows a specification and test suite for the real-number square root problem, including a rule (and examples) about results involving approximations of irrational numbers.

Check Your Understanding 2.9. Write your answers to Check Your Understanding 2.1 and 2.5 in Python.

```python
""
The "pig latin" problem involves rearranging the letters of a word, moving initial consonants to the end of the word and adding "ay".

For this dialect of pig latin:
* we move all consonants before the first vowel
* we always consider "y" a vowel
* we handle capitalization by converting to lower case all the consonants that we've moved to the end, making the initial letter of the result a capital if (and only if) the initial letter of the work was capital, and leaving the capitalization of all other letters alone.

PRECONDITION: To find pig_latin(x), x must be a word, for which we’ll allow any sequence of letters containing at least one vowel.
POSTCONDITION: pig_latin(x) is the pig latin form of the word x, as above.

For example,
>>> pig_latin('dog')
'ogday'
>>> pig_latin('owl')
'owlay'
>>> pig_latin('cheetah')
'eetahchay'
>>> pig_latin('thrush')
'ushthray'
>>> pig_latin('chthonic')
'onicchthay'
>>> pig_latin('python')
'ythonpay'
>>> pig_latin('yellowjacket')
'yellowjacketay'
>>> pig_latin('Dave')
'Aveday'
>>> pig_latin('McIntosh')
'Intoshmcay'
>>> pig_latin('MacIntosh')
'AcIntoshmay'
"""
**Check Your Understanding 2.10.** Write your answers to Check Your Understanding 2.2 and 2.6 in Python (be careful about cases in which real numbers must be approximated in the computer).

### 2.5 Describing the Form of the Question

When a problem instance involves more than a single number or string of text, our specification and/or test suite must also explain how we will represent the problem instance to the computer.

For example, when we raise a number to a power, we are asking questions about two numbers; when we work with computer graphics, we may need to ask about shapes, colors, lighting, or regions on the screen. Detailed issues of the representation of information are beyond the scope of this course, but we must at least touch on this subject now to provide a sufficiently rich palette of problems for study.

For the purposes of this course, we will represent all information as collections of numbers and pieces of text (the latter are known as **strings**, and are identified with in Python by either regular quotation marks, "", or apostrophes used as quotes, as in 'dog' and 'ogday' in Figure 2.2). A more advanced treatment of the representation of information is typically given in a course on **data structures** that follows this one.

Our use of numbers and strings simplifies the challenge of information representation to just knowing what each number or string means. Mathematicians often create new notation for a new problem, and provide conventions describing the meaning of each number involved: to indicate that a number is to be raised to a power, one number (referred to as the exponent) is placed above and to the right of the other (known as the base), as in $3^8$. Convention tells us that this means we are to compute the product of eight threes, $3 \cdot 3 \cdot 3 \cdot 3 \cdot 3 \cdot 3 \cdot 3 \cdot 3$.

```python
>>> sqrt(4) 2.0
>>> sqrt(2) # doctest: +ELLIPSIS
1.41421...
>>> sqrt(3.14159) # doctest: +ELLIPSIS
1.77245...
>>> sqrt(1) 1.0
>>> sqrt(0) 0.0
```

```
***
PRECONDITION: To find "sqrt(x)", x must be a non-negative number.
POSTCONDITION: sqrt(x) * sqrt(x) must give x (or be within 99.999% of x)

For example,
>>> sqrt(4) 2.0
>>> sqrt(2) # doctest: +ELLIPSIS
1.41421...
>>> sqrt(3.14159) # doctest: +ELLIPSIS
1.77245...
>>> sqrt(1) 1.0
>>> sqrt(0) 0.0
***
```

*Figure 2.3. Defining the “Square Root” Problem.*
Most computer programming languages do not allow such creation of new notation, perhaps due to their origins in programs entered via computer keyboards, or perhaps due to experiences in which the introduction of new notational conventions was found to cause confusion. Within this course, we will focus on Python’s function call notation, which we have seen already in our examples. In this notation, the function’s name (which identifies the general problem) is followed by a list of one or more parameters (which identify the particular problem instance). This leaves us with two tasks: selecting a function name, and stating the order in which parameters will be presented. For example, if we were to write a function to raise a number to a power, we might choose to give the base first, and then the exponent, writing `power(3, 8)` for $3^8$. Our description of the problem should make such conventions clear, for example with descriptive text or carefully chosen names for the parameters. Figure 2.4 shows such a description for the power problem, using a text description, and referring to `power(base, exponent)` in the precondition and postcondition. Note that some, but not all, languages let programmers indicate parameter values by name rather than, or in addition to, position. This would make it possible to express `power(3, 8)` with the more verbose but clearer code `power(base=3, exponent=8)`, if we know the names of the parameters as well as the function itself.

The question of the form of the problem becomes more challenging as more information is needed to identify an instance. The software that controls the display of information in windows on the computer screen involves numerous problems of this nature. For example, the windowing system may need to answer various questions about whether or not different objects on the screen overlap each other. If one window is drawn on top of another, as with windows on the left side of Figure 2.5, the computer may need to redraw the contents of the lower window when the top window is closed. If windows do not overlap, as on the right, it will not need to do so. If we wish to ask whether or not two windows overlap, we must have some way to communicate the relevant information to our algorithm.

We must first choose a name for this problem, such as “window overlap”, and then decide what information is needed about each instance. Some information from the figure is clearly not important for this problem — we don’t need to know what application is running in each window, or that the computer is using the Ubuntu distribution of the Linux operating system and the X window system. We do need to know that the windows are rectangular areas that line up with the borders of the screen, but since this fact is true for all windows in our system, we won’t need to communicate it as

```python

***
Raise one number (the base) to a power (the exponent).
The base must be given as the first parameter, and the exponent second.

PRECONDITION: To find power(base, exponent), our algorithm will require
that base be a number, and exponent be a positive integer.
POSTCONDITION: power(base, exponent) is base to the exponent power.
In other words, it obeys the mathematical rules
    power(base, exponent) == base, when exponent is 1
    power(base, exponent) == base * power(base, exponent-1)

For example,
>>> power(3.0, 3)
27.0
>>> power(0.9, 5)    # doctest: +ELLIPSIS
0.59049...
***
```

Figure 2.4. Defining the “Power” Problem
a parameter. We do need to identify where on the screen each window would be drawn, if it were on top. To identify this sort of rectangular region on the screen, it is sufficient to provide four numbers, giving the minimum and maximum distance from two adjacent edges of the screen in terms of some standard units. For example, we may give the distances from the top edge and left side in terms of the “pixels” that comprise the image on the screen. (The actual choices of which edges to use for reference, and what units to use for distance, are generally made by the designers of the window system). Once we have identified the key information, we try to choose clear but concise names for each value, and then clearly describe the order in which they will appear. Figure 2.6 gives one statement of this problem.

Check Your Understanding 2.11. Write your answers to Check Your Understanding 2.4 and 2.7 in Python. Since we have not seen a way to use a single value to express a Cartesian point, use two variables (e.g. x1 and y1) to represent each point.

2.6 Summary

The first step in solving a problem is understanding the problem itself. This involves consideration of unusual cases, and choices about how the information about a problem instance is to be represented. These considerations and choices can be expressed in a variety of ways. Informal text descriptions are often used to communicate the basic idea of the problem. Formal language or mathematics can be used to spell out rules for every case of the problem, and to describe which cases are to be counted as part of this problem. Examples can serve to illustrate interesting cases to someone reading the program and to provide a test suite for automatic testing, but (except in the rare case of a general problem that only has a few instances, such as adding two one-digit numbers), a test suite alone cannot give the result of every instance of the problem.

2.7 Further Resources

Many authors have explored the interplay between the understanding of a problem and the formulation of a solution. Liskov and Guttag’s classic Abstraction and Specification in Program Development focuses on formal techniques; more recently, Kent Beck’s Test Driven Development by Example focuses on the use of test suites to understand and design software. An appreciation of these works requires a greater understanding of programming, so they are described in more detail in Chapter 5.6.

Figure 2.5. Windows on a Computer Screen
Test to see if two "windows" (rectangular regions that are lined up with the border of a computer screen) overlap, i.e., see whether or not there is at least one point in both.

The parameters give the minimum and maximum values of x and y for windows 1 and 2, and are given in the order

\[\text{min}_x1, \text{max}_x1, \text{min}_y1, \text{max}_y1, \text{min}_x2, \text{max}_x2, \text{min}_y2, \text{max}_y2\]

**PRECONDITION:** Each minimum must be < the corresponding maximum, e.g. \(\text{min}_x1 < \text{max}_x1\)

**POSTCONDITION:** returns True if and only if there is a point \((x,y)\) with \(x\) in both \(x\) ranges and \(y\) in both \(y\) ranges

For example,

```python
>>> window_overlap(0, 1200, 20, 980, 750, 1550, 80, 920)
True
>>> window_overlap(0, 1020, 20, 980, 1025, 1550, 80, 920)
False
>>> window_overlap(0, 1020, 20, 980, 1020, 1550, 80, 920)
True
```

Figure 2.6. One Way of Describing the “Window Overlap” Problem
2.8 Exercises

**Exercise 2.1.** Choose a variant of pig latin and give a definition and examples of it (Wikipedia describes several variants on the basic theme, if you don’t know any).

**Exercise 2.2.** Some graphical user interfaces allow windows with shapes other than the rectangles described in Figure 2.6. The introduction of circular windows could create several variants on the window overlap problem we have seen:

a) Give a definition of and test cases for the problem of determining whether or not a point lies in a circle.

b) Give a definition of and test cases for the problem of determining whether or not two circular windows overlap.

c) Give a definition of and test cases for the problem of determining whether a circular window and a rectangular window overlap.

You may assume here that the $x$ and $y$ axes of the graphical system have the same scale — that is, that a circle that is 20 units wide in the $x$ direction is also 20 units high in the $y$ direction. This is not actually true on some systems, but dealing with this is not part of this exercise (or this course). State any other assumptions you make about the graphical system or the information provided about the graphical objects involved.
Chapter 3
Expression and Execution of Algorithms

This chapter discusses the expression of algorithms as Python functions, and the processes by which a computer applies an algorithm to a specific instance of a problem. The process of creating algorithms and expressing them in a precise language is known as programming or coding; a complete expression of algorithms for gathering input from some source (such as a computer user), processing the input, and producing an appropriate result, is known as a computer program. A computer is said to execute a program when it follows the program’s instructions.

The Python language is a complex and subtle notation for expressing algorithms and data structures. Mastering the complete language can take years. Fortunately, only a small subset of the full language is needed for this course, since we will only need to express one or two algorithms at a time, rather than develop large programs that must coordinate the work of hundreds or thousands of algorithms. The subset of Python that we will use in the first half of this course is described in detail in Appendix A.

3.1 Facts and Definitions

We will illustrate the core ideas of this chapter with two algorithms that are each based on simple mathematical facts that will hopefully be familiar. The first will solve the hypotenuse problem mentioned at the end of Section 2.1, and the latter the power problem from Figure 2.4.

The Pythagorean theorem tells us an important fact about right triangles: the square of the length of the hypotenuse is equal to the sums of the squares of the lengths of the other sides. i.e. if we label the hypotenuse length $c$, and the other side lengths $a$ and $b$, $c^2 = a^2 + b^2$. We can rewrite this fact as an equation to give $c$ in terms of $a$ and $b$, i.e., $c = \sqrt{a^2 + b^2}$. Once an equation has been solved for a given variable, we have an algorithm for finding that value, if our computer can perform all the required operations (in this case, squaring, adding, and finding a square-root) in the right order. If we have a modern pocket calculator (or a calculator app on a smart phone), we can execute this algorithm; i.e., if $a = 3$ and $b = 4$, we can find $3 \cdot 3$, then $4 \cdot 4$, then add those results, and press the $\sqrt{}$ to find the square root (which should be 5).

We can develop an algorithm for the power problem from the following two mathematical facts, as long as the exponent is a positive integer:

$$x^1 = x$$
$$x^n = x \cdot x^{n-1}, \text{ whenever } x \neq 0 \text{ or } n \notin \{0, 1\}, \text{ since } 0^0 \text{ is undefined.}$$

These equations, like the Pythagorean Theorem, can be viewed in two ways. First, they are simple statements of fact. Since $5^3$ is 125, and $5^2$ is 25, and 5 times 25 is also 125, $5^3$ is 5 times $5^2$; similarly, $17^1$ is 17. Second, we can view them as the basis for an algorithm to solve the power problem, if we can figure out when to apply each one. In this case, the selection is easy: we use the first when the exponent is 1, and the other when it isn’t (we’ll refer this as Algorithm Power-1).

Unlike our equation for hypotenuse length, the equations we have chosen for the power problem are self-referential — the second equation refers to the power problem itself. This means that we may have to apply that equation more than once for a single problem instance: to find $5^3$ this way,
we must know $5^2$, so we set aside our work on $5^3$ and set about solving $5^2$; that requires that we know $5^1$, but fortunately the first equation tell us that this is 5. We then find $5^2$ as $5 \cdot 5^1$, i.e., $5 \cdot 5$, which is 25. Now that we know $5^2$, we can go back to our original task of finding $5^3$ as $5 \cdot 5^2$, i.e., $5 \cdot 25$, and get our answer of 125. Phew. Fortunately, electronic digital computers excel at not only arithmetic, but keeping track of what they were working on and going back to finish it when they can, as we will see in the next section.

Note that a mathematical fact does not always lead immediately to an algorithm. Sometimes a simple mathematical step or two are all that’s needed: we had to solve the Pythagorean Theorem for $c$; similarly, if we want to find $x^n$ for negative integer $x$, it may help to rewrite our second power equation as $x^n = \frac{1}{x^{-n}}$. In other cases, there isn’t an easy way to turn an implicit definition, i.e. defining $\sqrt{x}$ as a number for which $\sqrt{x} \cdot \sqrt{x}$ is $x$, into an explicit definition that shows us how to compute $\sqrt{x}$. In some cases, a mathematical fact may not even implicitly define a unique answer. Our two equations for exponentiation do not define $x^n$ for non-integer $n$, even though the second rule remains true in such cases. For example, the above rules do not, by themselves, contradict a claim that $3^{\frac{1}{2}} = 17$, as long as we make consistent claims about other powers of 3, i.e., $3^{\frac{3}{2}} = 3 \cdot 17$.

### 3.2 Expressing Algorithms as Python Functions

An algorithm can be created by combining a set of equations that provide an explicit definition with a rule for selecting which equation should be used in any given case.

Our hypotenuse algorithm only had one equation, so we can convert it into Python by looking up the proper symbols for use in Python code. Figure 3.1 shows the result. Appendix A gives a detailed description of the features of Python that are used in this example and the remainder of the first half of this course; the key ideas for Figure 3.1 are:

- The function is preceded by, and includes, comments. Comments are not part of the algorithm itself, and do not contribute to the result produced by the function, but rather serve as notes for programmers who may need to read and understand the function. Comments typically focus on things that cannot be clearly expressed in the algorithm itself, such as why a certain approach is used, or what the overall goal is. A # symbol identifies all remaining text on a line as a comment; text surrounded by sets of triple quotation marks (""") can also be used to create multi-line comments, as long as they are properly indented. In this example, the comment at the top includes our specification and test suite.

- The function may also be preceded by a request for other functions with the word import, in this case requesting all the logic functions (including precondition and postcondition) with "from logic import *", and similarly requesting only the sqrt function from the math library. See Appendix C for details.

- The definition of the function itself includes the function’s name (hypotenuse) and parameters (a and b), followed by the function’s body (the lines indented below the name and parameters).

- The use of the precondition statement in the function itself allows the Python system to check the precondition and indicate an error if the condition is False. In Figure 3.1, we use the precondition statement to confirm that a and b are numbers.

- Similarly, the the postcondition statement lets us identify a legal result. In Figure 3.1, we use the confirm that the result we’ve calculated (and labeled c) does produce a value that is at least close to $a^2 + b^2$ when we square it.

- A variable such as c can be used to give a name to the result of calculation. For new, we can think of variables as we would in mathematics; the program would produce the same result.
if we simply substitute \( \sqrt{a^2+b^2} \) for each use of \( c \). As in mathematics, we introduce a variable when doing so makes the code clearer or more concise.

- A function call can be used to employ an algorithm that has been expressed as a Python function. In Figure 3.1, we use \( \sqrt{\cdot} \) to call upon the square-root finding algorithm in Python’s math library.
- The word return precedes the equation for the result.

""
Find the length of the hypotenuse of a right triangle
for which the other sides have lengths \( a \) and \( b \).

PRECONDITION: a and b must be numbers
(normally they would be expected to be >= 0,
but the function still works for negative numbers.)

POSTCONDITION: The result of hypotenuse(a, b) is the hypotenuse length.
In other words, \( a^2 + b^2 = \text{result} \times \text{result} \)
Or at least as close as the result of Python’s
square-root function allows. We’ll assume that’s
more than 99.9% accurate, and check
\( \text{result} \times \text{result} \times 0.998 < a^2 + b^2 < \text{result} \times \text{result} \times 1.002 \)

For example,
```python
>>> hypotenuse(3.0, 4.0)
5.0
```
```python
>>> hypotenuse(0.0, 4.0)
4.0
```
# Note that negatives are o.k.:
```python
>>> hypotenuse(-3.0, 4.0)
5.0
```
# Note the use of "..." and the "ELLIPSIS" option to indicate approximation
```python
>>> hypotenuse(2.0, 5.0) # doctest: +ELLIPSIS
5.385...
```

""
from logic import * # get all logic functions, e.g. precondition, postcondition
from math import sqrt # get the square root function (sqrt) from the math library
def hypotenuse(a, b):
    precondition(is_number(a) and is_number(b))
    #postcondition: the result is (about) the length of the hypotenuse
    c = sqrt(a^2 + b^2)
    postcondition(c*c * 0.998 < a^2 + b^2 and a^2 + b^2 < c*c * 1.002)
    return c

Figure 3.1. Expressing our Hypotenuse Algorithm in Python.
To express Algorithm Power-1 in Python, we'll make use of a few additional language features, as shown (along with our description of the power problem from Figure 2.4) in Figure 3.2:

- The word if can be used to select which equation will be used.
- For Figure 3.2 we have chosen to express the postcondition only in the comment.
- Figure 3.2, like Figure 3.1, involves a function call: the power function calls upon itself. This is known as a recursive call, and is used to express the need to solve a simpler instance of the same problem, as we did when solving $5^3$ by hand earlier. Note that the idea of “simpler” is quite important, and will be defined precisely in Sections 4.4 and 5.2.
- The word assert can be used to express other facts other than a precondition or postcondition, for example confirming that the smaller exponent is, in fact, smaller, and that it remains

```python
from logic import *

def power(base, exp):
    precondition(is_number(base) and is_integer(exp) and exp > 0)
    #postcondition: the result is the base raised to the exp power,
    # or a close approximation for information that
    # isn't represented exactly (such as real numbers).
    # in other words, the result is the same as Python's base**exp

    if (exp == 1):
        return base
    else:
        smaller_exp = exp-1
        assert(smaller_exp < exp) # confirm we're making progress
        assert(smaller_exp > 0) # confirm precondition for recursive call
        base_to_the_exp_minus_one = power(base, smaller_exp)
        return base * base_to_the_exp_minus_one
```

Figure 3.2. Exponentiation in Python
larger than zero.

Once an algorithm has been realized as a Python function, it can (in principle) be used as a part of a larger program to address some larger goal; a call to the `power` function would then trigger the use of this algorithm to raise a number to a power. For example `power(3.0, 3)` would compute the value 27.0; a graphics application (like the one discussed in Box 3) might include a call of the form `power(cos_of_reflect_angle, dull_matte)`; if `cos_of_reflect_angle` had the value 0.8 and `dull_matte` had the value 100, this would produce $0.8^{100}$ (about $2 \cdot 10^{-10}$).

It is usually wise to test the function before making use of it, to make sure it performs as expected (for example, by actually trying each of the tests in the test suite). In the rare but happy situation in which the function produces the desired result immediately, we may idly wonder how the computer actually got from our function definitions to the desired result. In the all-too-frequent case of a function that does not consistently do what we want, we must **debug** it: locate the flaw (or “bug”), which may lie in our original idea for the algorithm or in our translation of the algorithm into Python, and then fix it by changing the algorithm/function.

The process of debugging often relies on an understanding of what happens as the function is executed by the computer. The next sections describe two ways in which functions can be executed: via step-by-step following of instructions, or via a process of mathematical substitution. Depending on the programming language and the exact manner in which the program is started, the actual execution may be done either way or via a combination of the two. Regardless of the actual approach used by the computer, a solid understanding of each of these techniques can provide important insight into why a function does or does not work.

*Box 3: Exponentiation and Computer Generated Images.*

Exponentiation occurs “behind the scenes” in a number of applications of computers, including one algorithm for modeling reflections for computer-generated images. Light striking a perfectly smooth shiny surface is reflected perfectly (i.e., such that the angle of reflection $r$ is equal to the angle of incidence $i$). However, most surfaces are not perfectly smooth, and to produce realistic-looking reflections from surfaces like ceramics or plastic, computer graphics systems reflect some light at almost, but not quite, exactly the correct angle.

One way to model the amount of light reflected at a given direction $v$ is to raise the cosine of the angle between the direction of reflection ($r$) and the given direction ($v$) to an exponent that varies with the smoothness of the surface: shiny surfaces are given high exponents, and matte surfaces low exponents. For example, the images below show computer-generated reflections from surfaces using an exponent of 200,000 for the shiny surface on the left and 10,000 for the dull surface on the right.

![Shiny Surface: Exponent = 200,000](image1.png)  ![Matte Surface: Exponent = 10,000](image2.png)
3.3 Execution by Following Instructions Step-by-step

A Python function, such as `power` in Figure 3.2, can be viewed as a series of steps to be followed in order. If a function does not include any function calls, we can simply keep track of which line we have reached as we move from one to the next, and write down the value given to each variable or element of a computation. For example, if computing \((4+3)*(12-5)\), we could write a 7 over the 4+3, then a 7 over the 12-5, and then 49 over the entire expression; or we might choose to just write the 49. The level of detail needed will vary greatly with the number of computations in the function being investigated and the experience level of the person doing the investigation.

When investigating software that makes a few function calls, we may be able to follow the whole process of execution in a similar way, writing values next to each variable or computation in each function. However, when there are many function calls, it is easy to become lost in the maze of what is calling what; in this case, we can use a diagram known as a function call tree, or simply call tree, to show the structure of the function calls.

As with an investigation of a single function, the level of detail needed when drawing a call tree will vary from case to case — there is no precise rule for what belongs in a call tree, but it is often a good idea to show every call, with an indication of which call(s) occurred during which other call(s), and to show the value given for each parameter and to each interesting variable. Since call trees are typically used briefly during debugging and then discarded, abbreviations and other shorthands are often used liberally to keep the tree concise and easy-to-draw.

To provide an example call tree, we will consider the execution of `power(3.0, 3)`, following the step-by-step execution of the algorithm. Figure 3.3 shows the process of constructing this tree, abbreviating the variable name `base_to_the_exp_minus_one` as `bttemo`, and omitting `smaller_exp`, since it is just `exp-1`. At this point, we will not illustrate the checking of preconditions or postconditions or assertions, though it is often a good idea to confirm that the precondition holds when one writes down a function call, and to confirm that the postcondition holds when one writes down the result of a return. The program execution and call tree construction proceed according to the following steps:

1a. The computer begins by recording the parameter values and identifying the function to be called. We begin our call tree by writing the function name and parameter values, either by writing `power(3.0, 3)`, as we have done in Figure 3.3, or perhaps by writing something more general such as `power(base, exp)` and noting below it that `base=3.0` and `exp=3` (the choice is a matter of personal style preference).

1b. The computer confirms that the precondition holds, and then tests the expression from the if, i.e., `exp == 1`, using 3 for `exp`. Since the value is false, it will subsequently execute statements in the else clause of this if statement. We may choose to record this level of
detail (perhaps by writing “ELSE”) or not — we have not done so in Figure 3.3.

1c. The computer computes \( \exp - 1 \) to determine the values of parameters in the function call \( \text{power}(\text{base}, \exp - 1) \). We now add an arrow pointing from our initial call to the call \( \text{power}(3.0, 2) \), as shown in Step 2 of Figure 3.3. The computer will subsequently execute statements starting from the beginning of the \( \text{power} \) function, using 3.0 for \( \text{base} \) and 2 for \( \exp \), until it reaches a \text{return} statement, at which point it will bring the returned value back to this point for use in defining \( \text{base} _ { \text{to} \_ \text{the} \_ \exp \_ \text{minus} \_ \text{one}} \).

2a. After checking the precondition, the computer tests the expression \( \exp == 1 \), using 2 for \( \exp \), once again producing false, and moves into the \text{else} clause of the \text{if}.

2b. The computer computes \( \exp - 1 \) to determine the parameter values for the function call in the \text{else} clause, and calls \( \text{power}(3.0, 1) \), as we have indicated in Step 3 of Figure 3.3.

3a. After checking the precondition, the computer tests the expression \( \exp == 1 \), using 1 for \( \exp \); since the result is true this time, it will continue with the statements between the \text{if} and its \text{else}.

3b. The computer reaches \text{return base} and carries the value 3.0 back to the point at which

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure3.3.png}
\caption{Executing \text{power}(3.0, 3) by Following Instructions Step-by-step}
\end{figure}
power(3.0, 1) was called in Step 2b above, i.e., to define base_to_the_exp_minus_one as 3.0. This is shown in Step 4 of Figure [3.3] by an arrow labeled return 3.0, and by writing bttemo = 3.0 just below the function call receiving the returned 3.0.

4. Continuing with the calculation of power(3.0, 2), the computer multiplies base (which is 3.0) by bttemo (which is 3.0), and returns this value (9.0) to the point at which power(3.0, 2) was called, as shown in Step 5 of Figure [3.3].

5. Continuing with the calculation of power(3.0, 3), the computer multiplies base (3.0) by bttemo (9.0), producing the final result (27.0), as shown in Step 6 of Figure [3.3].

Our investigation of power(3.0, 3) ends when we have found this final result (27.0); in an actual program, this result would be communicated back to whatever had called upon power (such as the hypothetical computer graphics program of Box 3).

This view of execution as a step-by-step process is a good way to reason about a one specific execution of a program for a given set of input data. A good debugger program can help to manage the great quantity detail involved in this process. However, even with good tools or techniques, it can be difficult to apply this view of execution when trying to understand everything the program could do in response to any possible input.

![Figure 3.4. A More Complex Function Call Tree, With Associated Functions](image-url)
3.4 Execution by Substitution

Note that program execution, and our call tree diagrams, can be much more complicated than in the case of `power(3.0, 3)`. These diagrams can be not only taller (as in the case of `power(3.0, 15)`), but also wider: a function may call upon many other functions, in which case we can list them side-by-side below the function making the call. When there is only one call made in any one function, we can label all function calls simply as “call”, as in Figure 3.3; when there are many calls, we need to be more specific, using labels like “first call on line defining bttemo in the else”, abbreviations like `call bttemo #1`, or labels that we define by annotating our program. Figure 3.4 shows a hypothetical program execution diagram for such a program. In this figure we have also combined each function call/return pair into a single arrow.

Check Your Understanding 3.4. Print a copy of your function for computing distance (from Check Your Understanding 3.2), and write on it the value of each sub-computation performed in the computation of the distance from (12, 5) to (15, 9).

Check Your Understanding 3.5. Based on the function you wrote for Check Your Understanding 3.3 give a call tree for the execution of `is_a_positive_integer_power_of_two(32)`, and a call tree for the execution of `is_a_positive_integer_power_of_two(40)`. 

3.4 Execution by Substitution

In some cases, the value of an expression is completely determined by the values of the variables it contains and the functions and operations it uses to combine them, as in `exp == 1` or `power(base, exp-1)`. Such expressions are said to be referentially transparent. In other cases, such as input from the user, the result is determined by some other factor, such as what the user types.

The result of a referentially transparent expression does not depend on a particular order of execution, and it is often helpful to think about the execution of these expressions in terms of a mathematical substitution process, rather than as the following of an ordered sequence of steps. In fact, many modern programming languages and microprocessors make extensive use of this principle, providing much quicker execution by reordering the steps of the algorithm.

We can use this model of program execution to help us reason about algorithms. Instead of viewing execution as a sequence of steps that follow the order given in the program, we can view it as a process of substitution, in which each variable is replaced by its value, and each function replaced by its definition.

Figure 3.5 gives an example of this “execution by substitution”. Showing each step in a sequence of substitutions is somewhat verbose, so it shows a somewhat simpler example than that used in Figure 3.3, specifically `power(3.0, 2)`. This figure, like Figure 3.3, abbreviates `base_to_the_exp_minus_one` as `bttemo`, omits the logical statements and comments, and omits the variable `smaller_exp` (as if the lines `bttemo = power(base, exp-1)` and `return base * bttemo` were there entire body of the else clause). The full descriptions of the general rules for substitution are given in Appendix A; this particular sequence relies on our ability to:

- replace a function call such as `power(3.0, 2)` or `power(3.0, 1)` with the body of that function, after replacing the parameters names with the given values (as described in A.4.3 of Appendix A). We must also mark the beginning and end of the inserted function body, by adding \[ and \] marks. If the substitution were to introduce two variables of the same name,
as at the bottom of the first column in this example, we must distinguish between the two variables, for example by adding subscripts.

**Note.** Typeset subscripts, and the symbols [ and ], are *not* parts of the Python language, so we cannot actually type in this text and run it. However, we (or the language system) can still use this notation to keep track of a sequence of substitutions.

- replace simple arithmetic or comparison expressions such as 2-1 or 1==1 with their values (1 or True in these cases), as in [A.4.4 and A.4.5].

```python
def power(base, exp):
    
    if exp == 1:
        return base
    else:
        temp = power(base, exp-1)
        return base * temp

bttemo = power(3.0, 2)
```

Figure 3.5. Executing `power(3.0, 2)` via Rules of Substitution (from Appendices A and B).
3.4 Execution by Substitution

- replace \texttt{if True: ... else: ...} with everything indented under the \texttt{if True:} (discarding the part after the \texttt{else:}), and \texttt{if False: ... else: ...} with everything indented under the \texttt{else:} (discarding the part after the \texttt{if False:}), as in A.4.8.

- replace uses of a variable with whatever was used to define the variable (and eliminate the definition of the variable once it has no uses), as in A.4.9.

- replace \texttt{[ return ... ]}, i.e., the result of substituting a function body and then simplifying it to a single return statement, with the whatever was after the return, as in A.4.1.

Since substitution can be performed in any order without affecting the result (i.e., is associative), we could also execute \texttt{power(3.0, 2)} in the order shown in Figure 3.6. This sequence of substitutions generally follows the principles discussed for Figure 3.5 (though in a different order), but one step is new:

- We can distribute arithmetic and comparison operations over the \texttt{return}s inside \texttt{[ ... ]}, as discussed at the end of A.4.1.

Figures 3.5 and 3.6 do not provide much useful insight beyond what we would get from a good call tree. In general, viewing program execution as a sequence of substitutions at this detail is not particularly helpful in understanding one particular execution. However, this technique can help us understand aspects of the execution that are occur for all possible inputs. For example, the steps of Figure 3.6 before “Then, consider \texttt{power(3.0, 2)}” do not presume any particular values for \texttt{base} and \texttt{exp}. We could, at this step, substitute only the value 2 for \texttt{exp}, and thereby show that, \texttt{power(base, 2)} can be turned into \texttt{base*base}. This particular fact is rather obvious, but the general principle of reasoning about software with the mathematical rule of “substituting equals for equals” will play an important role in our work in Chapter 5.

\textbf{Check Your Understanding 3.6.} Fill in the gray boxes below with lines of Python to show the results of applying the substitution rules from Appendix A, using the definition of \texttt{response_to} from Figure A.1.

\begin{verbatim}
response_to(98.6+3)
... rewrite 98.6 + 3 using rules of arithmetic as in A.4.4 ...
... rewrite with “function call rule” of A.4.3 ...
... rewrite the >= using the rules of arithmetic as in A.4.5 ...
... rewrite with the “if True:...” rule of A.4.8 ...
... rewrite with the “return” rule of A.4.1 ...
"You have a fever"
\end{verbatim}
Check Your Understanding 3.7. Fill in the gray boxes below with rules from Appendix A to show that the substitution rules can transform the lines of Python at the top into those at the bottom.

**First**, transform the power function:

```python
def power(base, exp):
    if (exp==1):
        return base
    else:
        bttemo = power(base, exp-1)
        return base * bttemo
```

\[\downarrow \text{substitute body of power at the call, using subscripts to distinguish newly introduced variables}\]

```python
def power(base, exp):
    if (exp==1):
        return base
    else:
        bttemo1 = [
            if (exp-1==1):
                return base
            else:
                bttemo2 = power(base, exp-1-1)
                return base * bttemo2
        ]
        return base * bttemo1
```

\[\downarrow \text{substitute definition of bttemo1}\]

```python
def power(base, exp):
    if (exp==1):
        return base
    else:
        return base * [
            if (exp-1==1):
                return base
            else:
                bttemo2 = power(base, exp-1-1)
                return base * bttemo2
        ]
```

\[\downarrow \text{distribute * over [...], arithmetic}\]

```python
def power(base, exp):
    if (exp==1):
        return base
    else:
        return base * [
            if (exp-1==1):
                return base
            else:
                bttemo2 = power(base, exp-1-1)
                return base * bttemo2
        ]
```

\[\downarrow \text{arithmetic, if, return [...], rules}\]

Then, consider \(\text{power}(3.0, 2)\):

```python
\text{power}(3.0, 2)
```

\[\downarrow \text{substitute new body of power, w/ 3.0 for base and 2 for exp}\]

```python
\text{power}(3.0, 2)
```

\[\downarrow \text{substitute body of power at the call, using subscripts to distinguish newly introduced variables}\]

```python
def power(base, exp):
    if (exp==1):
        return base
    else:
        bttemo2 = power(base, exp-2)
        return base*base*bttemo2
```

\[\downarrow \text{substitute definition of bttemo2}\]

```python
def power(base, exp):
    if (exp==1):
        return base
    else:
        return base *
            [
                if (2==1):
                    return 3.0
                else:
                    bttemo2 = power(base, 2-2)
                    return 3.0*bttemo2
            ]
```

\[\downarrow \text{arithmetic, if, return [...], rules}\]

9.0

Figure 3.6. Executing \text{power}(3.0, 2) via a Different Sequence of Substitutions

Copyright 2003-2015 David G. Wonnacott; see Page 7 for terms of free use
Check Your Understanding 3.8. Based on the function you wrote for Check Your Understanding 3.1 and the rules of substitution from Appendices A and B, show how \( f_{to\_c}(98.6) \) can be executed by substitution.

Check Your Understanding 3.9. Illustrate the computation of the distance from \((12, 5)\) to \((15, 9)\) in terms of the substitution rules of Appendices A and B plus the rule that you can replace \( \sqrt{x} \) with \( \sqrt{x} \). How does this process compare to your work for Check Your Understanding 3.4?

Check Your Understanding 3.10. The steps of Figure 3.6 below “Then, consider \( \text{power}(3.0, 2) \)” are not shown in full detail, to save space. Write these out, using only one rule per step.

Check Your Understanding 3.11. Starting with the version of \( \text{power} \) right above “Then, consider \( \text{power}(3.0, 2) \)” in Figure 3.6 show what \( \text{power}(x, 2) \) can be transformed into \( x^x \). Show each rule of substitution you use, and show the result of each substitution. (Note that this is essentially the same as the problem above, but with \( x \) instead of 3.0 for base; however, it shows something much more general — not only do we know that \( \text{power}(3.0, 2) \) gives us 9.0, we know that for any \( x \), \( \text{power}(x, 2) \) gives us \( x^2 \).)

Check Your Understanding 3.12. Show that the Python code
\[
p = 4+2 \\
q = 2\*p + 5\*p
\]
will be simplified to just \( q = 42 \) regardless of which of the following three approaches we use:

a) first use rules of arithmetic to simplify \( 4+2 \), then substitute the value of \( p \), then more arithmetic;

b) first substitute \( 4+2 \) for the uses of \( p \) (using appropriate parentheses), then perform arithmetic;

c) first “factor out” \( p \) in the definition of \( q \) (i.e., turn it into \( (2+5)\*p \)), then substitute \( 4+2 \) for the use of \( p \), then perform arithmetic

In each approach, give the rule used and the result produced at every step.

Check Your Understanding 3.13. Based on the function you wrote for Check Your Understanding 3.3 and the rules of substitution from Appendices A and B, show how \( \text{is\_a\_positive\_integer\_power\_of\_two}(8) \) and \( \text{is\_a\_positive\_integer\_power\_of\_two}(20) \) can be executed by substitution.
3.5 Complete Programs and Program Organization

A complete Python program typically contains many algorithms, each embodied in one or more Python functions. Most programmers organize their program files by grouping several related functions into one file. (Although it is outside the scope of this course, it is worth mentioning that the implementation of a data structure as a Python class provides a natural way to group all the algorithms related to one data structure into one file; it may also be desirable to place several related classes into one file.)

In addition to functions that transform information within the computer (as our power function does), a program will also contain one or more functions to communicate with something outside the program, for example user interface functions to communicate with a human “user” of the software. In the interest of simplicity, we have grouped the function for the main algorithm of our example (power) together with its user interface (a function named simple_power_ui) into a single file, and shortened power by removing its local variable. The result is shown in Figure 3.7. Note that user interfaces are typically programmed in either the imperative style (introduced in Chapter 7) or the event-driven approach (beyond the scope of this course). Thus, we will not discuss details of design and implementation of user interfaces, despite the importance of this topic. The one principle of user interface design that we do introduce is this: keep user interface functions separate from the functions for the functions that perform the primary computation of our programs.

The separation of interface from core functionality has several benefits. First, this lets us use referentially transparent functions for our primary computation, facilitating reasoning about our algorithms (reasoning about imperative and event-driven systems is complicated by the fact that they are not typically referentially transparent and thus may not obey the usual rules of mathematics). The ability to reason about software is important to a wide range of programmers, from academics who study the field of “pure functional programming”, in which all program constructs are referentially transparent, to programmers in companies like Google — at the 2003 conference on Dependable Systems and Networks, Google’s Urs Hölzle spoke about the importance of this approach in their work to produce extremely dependable search software that runs on a very large group of computers. Second, the separation of core functionality from interface is an important principle in software design. This makes it easier to connect a single “computational engine” to a variety of interfaces for different users, allowing us to quickly adapt software for speakers of different languages, users with unusual hardware devices, etc.

Note that the user interface often contains steps that appear to be redundant with the logical statements we will use for reasoning about the program. In this example, the precondition of power states that the second parameter must be a positive integer, and simple_power_ui checks to see whether or not it is. However, these two checks serve different purposes — the precondition defines a rule that must hold for any program in which power is used; the if in the user interface ensures that the user’s input won’t violate that rule, and ensures that a comprehensible error message is provided for the user if it would.

Once a complete Python program has been written, it may be started in any of several ways. If we are using an Integrated Development Environment such as Eclipse or IDLE, we may just need to click on a button to have the computer execute our program; in other cases, we might double-click on an icon for our power1.py file on the computer’s desktop, or type a command such as python power1.py, to start this process. Once started, the Python system will proceed to execute our program: it will ignore the comments, record the function definitions for later use, and perform any other processing requested in the file. In our example, the definitions of power and simple_power_ui (the user interface for power) are noted, and then the simple_power_ui function is called — at that point, the computer actually does what was
Raise one number (the base) to a power (the exponent).
The base must be given as the first parameter, and the exponent second.

PRECONDITION: To find power(base, exponent), our algorithm will require
that base be a number, and exponent be a positive integer.

POSTCONDITION: power(base, exponent) is base to the exponent power.
In other words, it obeys the mathematical rules
power(base, exponent) == base, when exponent is 1
power(base, exponent) == base * power(base, exponent-1)

For example,
>>> power(3.0, 3)
27.0
>>> power(0.9, 5)  # doctest: +ELLIPSIS
0.59049...

from logic import *

def power(base, exp):
    precondition(is_number(base) and is_integer(exp) and exp > 0)
    #postcondition: the result is the base raised to the exp power,
    # or a close approximation
    if (exp == 1):
        return base
    else:
        return base * power(base, exp-1)

def simple_power_ui():
    print('3.0 to the 3rd power should be 27.0 -- we get: ')
    print(power(3.0, 3))
    print('0.9 to the 5th power should be 0.59049 -- we get: ')
    print(power(0.9, 5))
    print('Now you try one!'
users_base = float(input('Enter a number to serve as the base: '))
users_exp = int(input('Enter a positive integer to serve as the exponent: '))

    if not (users_exp > 0):
        print('Sorry, the exponent must be a positive integer!'
    else:
        print(users_base, 'to the', users_exp, 'power is:')
        print(power(users_base, users_exp))

    simple_power_ui()
requested in the `simple_power_ui` function, i.e., print some text and call upon the `power` function to perform the interesting calculations (such as `power(3.0, 0)`) and print the results.

**Check Your Understanding 3.14.** Create a complete program for your answer to Check Your Understanding 3.1, 3.2, or 3.3; run some tests to check that it works; if it fails, run smaller tests based on simpler problems from your call trees or substitution diagrams from other check-your-understanding questions.

### 3.6 Summary

In this chapter, we have seen the development of one simple algorithm for exponentiation, from a collection of mathematical facts through a complete working Python program. We have investigated two ways of looking at the execution of a program: execution can be seen as step-by-step following of instructions — a view that is particularly well suited to considering a specific execution with specific input data; execution can also be seen as a process of substitution — this view is well suited to reasoning about all possible program executions, though it is difficult to apply to software that is not referentially transparent. We have seen the use of function call trees to record the history of a single program execution, and text rewriting to show the effects of substitution (the latter is greatly facilitated by a good word processor). In the next chapter, we will discuss techniques for the design of algorithms. Chapter 5 revisits the view of program execution as substitution, as we investigate techniques for proving that algorithms are correct.

### 3.7 Further Resources

More information about Python can be found in Appendix A, or in any of a number of published textbooks or language references (The O’Reilly reference books are usually quite good). Pure functional programming is discussed in a number of textbooks, but these employ languages other than Python or are at a much higher level (or both).
Exercise 3.1. Use the rule \( x^n = \frac{x^{n+1}}{x} \) (given in the text of Section 3.1) to extend the exponentiation program of Figure 3.2 to handle negative exponents.

Exercise 3.2. Use the rule \( x^n = \frac{1}{x^{-n}} \) to extend the original exponentiation program of Figure 3.2 (not the result of the previous exercise) to handle negative exponents. Is there any reason to consider this version better or worse than the one from the previous exercise?

Exercise 3.3. Give a function call tree for \( \text{power}(2.0, 4) \).

Exercise 3.4. Give a function call tree for \( \text{power}(2.0, -3) \) for each of the functions you wrote for Exercises 3.1 and 3.2.

Exercise 3.5. Show each step in the production of the function call tree in Figure 3.4.

Exercise 3.6. Give a function call tree for \( \text{strange}(2,48) \), where \( \text{strange} \) is the function definition from Figure 3.4.

Exercise 3.7. Show each step in evaluating \( \text{strange}(2,48) \) via substitution (i.e., give a sequence like either Figure 3.5 or 3.6).
Chapter 4
Techniques for Algorithm Design

Algorithm design is a creative process without set rules, but there are a number of strategies that can be helpful in this process:

**Recommended Algorithm Design Techniques:**

- Directly **deriving** a program from a mathematical definition given in (or implicit in) the statement of the problem, as with our **power** program of Figure 3.2. Sometimes the hardest part of solving a problem is coming up with a precise statement of the question.

- Breaking a problem down into a collection of sub-problems whose solutions can be combined to solve the problem. If necessary, the sub-problems can be broken down into sub-sub-problems, etc. This process is known as **top-down design**.

- Considering separate **cases** of the problem and designing an algorithm for each. We will call this technique **design by cases**.

- Identifying a **base case** in which certain instances can be solved trivially, and a way of expressing the solution to any other case in terms of a **simpler instance** of the same problem. We will refer to this approach as **basic recursive design**. It differs from top-down design in that we break a problem down into one or more instances of the **same** problem, rather than into instance(s) **different** problem(s). Although our **power** program of Figure 3.2 was derived from a mathematical definition, it shares the structure that arises from basic recursive design.

- When basic recursive design fails, varying it in any of several ways. We will refer to these as **advanced recursive design**.

- Writing an algorithm to convert any instance of a problem into an instance of some already-solved problem, and then making use of the existing solution. This approach is known as **reducing** a problem to a known problem. It is essentially a top-down design in which all the lower steps have already been done.

- Combining one algorithm to generate all possible answers to a problem with a second to check each potential answer to see if it is correct. This can produce algorithms of extremely high computational complexity, and is typically used if other approaches fail. This kind of algorithm is known as a **generate and filter** algorithm.

- **Solving problem instances** by hand, in the hope that the process can be generalized. Often it helps to compare “marginal cases” — for example, in the window overlap problems of Figure 2.6 and Exercise 2.2 we might look at what distinguishes a case in which two windows just barely **do** touch from a case with **almost** identical parameters in which the windows **do not quite** overlap. This approach typically does not produce a complete algorithm by itself, but it can often stimulate progress toward one of the other design techniques, in cases when nothing seems to be working.
• Having a **brilliant flash of insight**. You can’t count on this, but when it happens, it’s great. The best way to encourage it seems to be to think hard about the problem, possibly solving some instances by hand, and then ignore it for a couple of days.

All these techniques can be practiced alone or as a team; one way to get un-stuck during problem solving is to tell a colleague exactly why your approach isn’t working.

Once an algorithm has been designed, it may be helpful to construct a partial or full verification, and to implement the algorithm as a program and test it. If the verification effort or testing demonstrate an error, this error should be used to re-evaluate the basic idea of the algorithm to see if it is sound. During this process, one must be careful to avoid the temptation to find a simple way to fix each error without taking the time to understand what was wrong with the algorithm — this process of finding a quick fix for each specific failure is sometimes mistaken for a design technique, but it tends to produce non-algorithms rather than algorithms:

**Non-algorithm Design Technique (not recommended):**

• Take a guess at an algorithm, write an implementation, and try it on a few examples. If it doesn’t work for some example, adjust it so that it gives the right answer for this example and start testing it over again (for example, if we discover that Non-algorithm prime-4 fails for 91, and adjust it by adding either “or, if \( x = 91 \), report ‘Not Prime’ ” or “or, if \( x \) is divisible by 7, report ‘Not Prime’ ”). This approach to programming was known as **hacking** before the press started to use this term for computer crime.

### 4.1 Mathematical Derivation

Many of the early graphical human-computer interfaces involved windows that were always rectangles with perfectly vertical and horizontal edges. This produced very simple algorithms for common problems that arise in graphical systems, such as testing whether a given point (such as the point the user has clicked with a mouse) is in a given window. This “point is in window” problem, like the hypotenuse and power problems of Chapter 2 and 3, can be solved via mathematical derivation.

For this problem, like many others, the hardest step is coming up with the formal statement of the problem. We begin by stating what we expect to know about the objects we must manipulate — in this case, we expect to know the coordinates of the point (we’ll call them \((x, y)\), as usual) and, as in Figure 2.6, the minimum and maximum values for the \( x \) and \( y \) values of the window. We will follow the convention of using integer-valued coordinates, though our algorithm would work just as well for real numbers. After these decisions, we can come up with the function name, parameters, and **precondition** shown in Figure 4.1.

We now need to make our informal specification of the result (“return true if and only if the point is in the window”) into a **formal** specification, i.e. something mathematically precise. There is no cookbook approach this, but a little consideration should make it clear that the \( x \) value of the point must be within the range of \( x \) values of the window, and the \( y \) value must be similarly contained in the window’s \( y \) range. To spell this out mathematically, we must settle such issues as whether or not points with the exact values of the edges and corners of the window count as “in” the window or not. This entire process of formalizing the problem requires consideration of how the algorithm will be used, including consultation with other programmers who may need to use it. However, the goal of this chapter is algorithm development, not window systems, so for this example we will simply decide that such edge values should count as “in” the window. This yields the following definition: a point \((x, y)\) is in a window whose \( x \) values range from a minimum of \( \text{min}_x \) to a maximum of \( \text{max}_x \), and whose \( y \) values range from \( \text{min}_y \) to \( \text{max}_y \), if \( \text{min}_x \leq x \leq \text{max}_x \) and \( \text{min}_y \leq y \leq \text{max}_y \); otherwise it is not in the window. Thus, our **postcondition** is “the result is true if that condition holds, or otherwise false” we can add this as the **postcondition** statement of the function.
The last step, converting the precise postcondition into Python statements, is often relatively straightforward. For this example, we use Python’s “if” statement to determine whether or not the result should be true, and appropriately return true or false (re-stating the postcondition before each return). Our programming is now complete, and has produced the function in Figure 4.1.

There is an even simpler way to translate our postcondition into Python, if we consider the fact that the function returns true if and only if the expression \( \min_x \leq x \leq \max_x \) and \( \min_y \leq y \leq \max_y \) is true; in other words, its answer is the value of that expression. Figure 4.2 illustrates the resulting more concise function, though we have written out \( \min_x \leq x \leq \max_x \) as \( \min_x \leq x \) and \( x \leq \max_x \) as part of our effort to focus on programming idioms that work in a wide variety of languages. Since the formal version of the postcondition is exactly the same as the return, we have given it

""
Is the point \( x,y \) is in the given window? (Where a "window" is a rectangular region that’s lined up with \( x \) and \( y \) axes.)

Some examples:
>>> point_is_in_window(12,120, 10,20, 100,200)
True
>>> point_is_in_window(20,200, 10,20, 100,200)
True
>>> point_is_in_window(12,220, 10,20, 100,200)
False
>>> point_is_in_window(12, 20, 10,20, 100,200)
False
>>> point_is_in_window(22,120, 10,20, 100,200)
False
>>> point_is_in_window(22,220, 10,20, 100,200)
False
""

from logic import *

def point_is_in_window(x, y, min_x, max_x, min_y, max_y):
    precondition(min_x < max_x and min_y < max_y)
    #postcondition: returns true if and only if the point \( (x, y) \) is in the window
    # (counting border values as part of the window).
    if min_x <= x and x <= max_x and min_y <= y and y <= max_y:
        result = True
    postcondition(result==(True if min_x <= x and x <= max_x and min_y <= y and y <= max_y else False))
    return result
else:
    result = False
    postcondition(result==(True if min_x <= x and x <= max_x and min_y <= y and y <= max_y else False))
    return result

Figure 4.1. Testing whether a point is in a window
as a comment rather than an actual function call; if we choose to change the code again, we can reinstate the test of this simpler statement of the postcondition.

**Check Your Understanding 4.1.** The “quadratic formula” can be used to find values of $x$ that satisfy an equation of the form $ax^2 + bx + c = 0$ when $a \neq 0$. For reference, the formula is

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}.$$

Express an algorithm to solve this class of equations as a Python function.

""
Is the point $x,y$ is in the given window? (Where a "window" is a rectangular region that’s lined up with $x$ and $y$ axes.)

For example,

```python
>>> point_is_in_window(12,120, 10,20, 100,200)
True
>>> point_is_in_window(20,200, 10,20, 100,200)
True
>>> point_is_in_window(12,220, 10,20, 100,200)
False
>>> point_is_in_window(12, 20, 10,20, 100,200)
False
>>> point_is_in_window(22,120, 10,20, 100,200)
False
>>> point_is_in_window(22,220, 10,20, 100,200)
False
""

```python
from logic import *

def point_is_in_window(x, y, min_x, max_x, min_y, max_y):
    precondition((min_x < max_x and min_y < max_y))
    #postcondition: returns true if and only if the point (x, y) is in the window
    #i.e., return true iff min_x <= x <= max_x and min_y <= y <= max_y
    return min_x <= x and x <= max_x and min_y <= y and y <= max_y

# postcondition indentical to the above:
# (result == (min_x <= x and x <= max_x and min_y <= y and y <= max_y))

Figure 4.2. More concise test of whether a point is in a window
4.2 Top-down Design (and Bottom-up Implementation)

We could also have approached the point-is-in-window problem by observing that it is made up of two tests, each of which checks to see if a value is in a given range. Thus, we have simplified our two-dimensional problem into two one-dimensional problems. This is an example of top-down design. This problem is so trivial that it really doesn’t require this technique, but we will follow it through to a solution to illustrate the process of writing code for a top-down design and to illustrate that a problem may have many solutions.

As we identify a simpler problem, we apply the practices of Chapter 2, considering it on its own and writing a specification and/or test suite for it. For example, must the first value in a range always be less than the second? Is a value at the end of a range considered part of it? The test suite and pre- and post-condition for the `value_is_in_range` function of Figure 4.3 illustrate one way of answering these questions. We can then write the function body, in this case using design by derivation, and use the test suite to check it, after which we write the `point_is_in_window` function that makes use of it.

Figure 4.3’s `point_is_in_window` function always produces the same result as the one in Figure 4.2 (as we will see in Exercise 5.1). The one interesting design choice we have made here is to let the `value_is_in_range` function accept ranges with only one value, even though our hypothetical graphical system does not allow windows with equal minimum and maximum values. This choice does not make `value_is_in_range` any less useful or efficient for our window system, but it does ensure that `value_is_in_range` could be used in other programs that had to deal with one-value ranges (we could even choose to weaken our precondition further to produce a more general function, if we could decide what it means for a point to be in a range whose minimum is greater than its maximum).

Note that we could have written `point_in_window` before `value_is_in_range`, but this wouldn’t let us fully test either function until both were completed. Since finding mistakes in two functions is often more than twice as hard as finding them in one, coding the functions in this order and testing each as it is written (i.e., bottom-up implementation) is typically a good use of time.

Top-down design works well for programming teams that prefer loose collaboration — part of the team can work on the simpler problem, while the rest writes the main algorithm in terms of the simpler. This approach only works well if both parts of the team share the same vision of the simpler problem, so a clear and complete test suite or specification of the simpler problem is critically important in this environment.

To investigate top-down design on a less trivial problem, we now turn to the question of whether two windows overlap, as introduced in Figure 2.6. We will formalize this problem as follows: Given the minimum and maximum x and y values for each of two windows w1 and w2, return true if and only if w1 and w2 overlap, that is, if there exists some point that is both in w1 and in w2 (using the definition of whether a point is in a window that was given in Section 4.1). This formalization does not directly lead to an algorithm, though if the graphics system always uses integer coordinates for window boundaries, we could create a generate-and-filter algorithm to check each point to see if it is in both windows (or, better yet, we could check only those points that lie in one of the windows). However, top-down design will yield a faster algorithm: Unlike the generate-and-filter algorithm, it will do only a fixed small number of operations, rather than a number of operations that grows with the screen size or window size.

We start by considering the possibility that we could break this two-dimensional problem down into two one-dimensional problems, as we did with the `point_is_in_window` function of Figure 4.3.
Is the point \(x, y\) is in the given window? (Where a "window" is a rectangular region that’s lined up with \(x\) and \(y\) axes.)

First, some tests for our sub-function "value_is_in_range":

```python
>>> value_is_in_range(12,10,20)
True
>>> value_is_in_range(25,10,20)
False
>>> value_is_in_range(10,10,20)  # Count the end points as part of the range
True
>>> value_is_in_range(20,10,20)
True
>>> value_is_in_range(10,10,10)  # A range with only one value is o.k.:
True
>>> value_is_in_range(20,10,10)
False
```

Now, some tests for our main "point_is_in_window":

```python
>>> point_is_in_window(12,120, 10,20, 100,200)
True
>>> point_is_in_window(20,200, 10,20, 100,200)
True
>>> point_is_in_window(12,220, 10,20, 100,200)
False
>>> point_is_in_window(12, 20, 10,20, 100,200)
False
>>> point_is_in_window(22,120, 10,20, 100,200)
False
>>> point_is_in_window(22,220, 10,20, 100,200)
False
```

```python
from logic import *

def value_is_in_range(v, min, max):
    precondition(min <= max);
    #postcondition: return true if min <= v <= max else false
    return min<=v and v<=max

def point_is_in_window(x, y, min_x, max_x, min_y, max_y):
    precondition(min_x<max_x and min_y<max_y)
    #postcondition: returns true if min_x<x<=max_x and min_y<y<=max_y else false
    result = value_is_in_range(x,min_x,max_x) and value_is_in_range(y,min_y,max_y)
    postcondition(result == (min_x<=x and x<=max_x and min_y<=y and y<=max_y))
    return result
```

Figure 4.3. Top-down Design of a Test of Whether a Point is in a Window
A little thought suggests that, for the rectangular window system we have described, two windows will overlap if and only if their $x$ ranges overlap and their $y$ ranges overlap. This leaves us with the question of whether or not two ranges of numbers overlap, i.e., of determining if there is a value that lies within each of two ranges, given minimum and maximum values of the two ranges. Once again, a generate-and-filter algorithm is possible if only integer values are allowed, but we can do better, this time by deriving the algorithm from the definition. The mathematical translation of “is there a point in both ranges”, $(\exists v \text{ such that } \min_1 \leq v \leq \max_1 \text{ and } \min_2 \leq v \leq \max_2)$, is equivalent to the simpler test $\min_1 \leq \max_2 \text{ and } \min_2 \leq \max_1$. This design gives us the Python functions shown in Figure 4.4.

The revised point_is_in_window function and the new window_overlap function both illustrate the three essential steps of top-down design, namely

1. break the problem into well-defined sub-problems,
2. solve each of the sub-problems, and
3. combine the sub-problem solutions into one complete solution.

When the sub-problems themselves are solved by top-down design, all three steps must of course be done at each level of the design. Problems and sub-problems, and the preconditions and postconditions at each level of a design, are reminiscent of the practice of subcontracting out parts of a complex task. If I hire someone to perform a task, we may draw up a contract, stating my responsibilities (the precondition for the contract) and the contractor’s responsibilities (the postcondition). I expect the contractor to honor the contract, as long as I’m fulfilling my part. The contractor may then hire subcontractors, writing contracts (and having similar expectations) with each of them.

The key to applying top-down design successfully seems to be having the ability to envision all of the levels of the design at once, before you even start to formalize them, or having excellent instinct for what simpler problems can be solved before you actually know how to solve them.

One strategy for exploring top-down design involves “brainstorming” many different top-level designs, and then thinking about which one has the most-easily-solved sub-problems. For example, Figure 4.5 shows two top-level designs for the pig latin problem of Figure 2.2 (note that we have simplified the problem slightly in terms of the response to capital letters). The first design requires that our sub-contractor solve problems we’ve called from_vowel and initial_consonants. The from_vowel function is supposed to extract all letters starting from the first vowel, e.g., from_vowel(‘thrush’) == ‘ush’; the function initial_consonants is supposed to extract the consonants before the first vowel, so initial_consonants(‘thrush’) == ‘thr’. If we put these two pieces together in the order shown in pig_latin_1, and add “ay”, we’ll get the expected ‘ushthray’. The second design of Figure 4.5 only requires a solution to the initial_consonants_at_end problem. This must give us the word with the initial consonants placed at the end, e.g., initial_consonants_at_end(‘thrush’) == ‘ushthr’. If we put “ay” after that modified string, as is done in pig_latin_2, we’ll get the expected result, e.g. ‘ushthray’.

Check Your Understanding 4.2. The palindrome testing problem of Check Your Understanding 2.1 can be approached via top-down design. Write a list of sub-problems whose solutions could be combined to solve the palindrome testing problem, a brief test suite to give examples of each sub-problem, and a Python function to solve the palindrome question by making use of the functions illustrated in your test suite.
Test to see if two "windows" (rectangular regions that are lined up with the border of a computer screen) overlap, i.e., see whether or not there is at least one point in both. The parameters give the minimum and maximum values of x and y for windows 1 and 2, and are given in the order min_x1, max_x1, min_y1, max_y1, min_x2, max_x2, min_y2, max_y2

PRECONDITION: Each minimum must be < the corresponding maximum, e.g. min_x1 < max_x1
POSTCONDITION: returns True if and only if there is a point (x,y) with x in both x ranges and y in both y ranges

First, some tests for the sub-problem "range_overlap":
>>> range_overlap(1,10, 2,20)
True
>>> range_overlap(1,2, 10,20) # miss entirely
False
>>> range_overlap(1,10, 10,20) #touch at one point
True

>>> window_overlap(0, 1200, 20, 980, 750, 1550, 80, 920)
True
>>> window_overlap(0, 1020, 20, 980, 1025, 1550, 80, 920)
False
>>> window_overlap(0, 1020, 20, 980, 1020, 1550, 80, 920)
True

from logic import *

def range_overlap(min1, max1, min2, max2):
    precondition(min1<=max1 and min2<=max2)
    # This function allows empty ranges, but not min>max.
    # postcondition: return true if and only if the ranges overlap, i.e.
    # if there exists v such that min1 <= v <= max1 and min2 <= v <= max2
    return min1<=max2 and min2<=max1

def window_overlap(min_x1, max_x1, min_y1, max_y1,
    min_x2, max_x2, min_y2, max_y2):
    precondition(min_x1<max_x1 and min_y1<max_y1 and
    min_x2<max_x2 and min_y2<max_y2)
    # postcondition: return true if and only if the windows overlap, i.e.
    # if there exists (x,y) such that (x,y) is in both windows
    # (as before, counting border values as part of the window).
    return (range_overlap(min_x1, max_x1, min_x2, max_x2) and
            range_overlap(min_y1, max_y1, min_y2, max_y2))

Figure 4.4. Solution to Overlapping Rectangular Window Problem

Copyright 2003-2015 David G. Wonnacott; see Page 7 for terms of free use
The "pig latin" problem involves rearranging the letters of a word, moving initial consonants to the end of the word and adding "ay".

For this dialect of pig latin:
* we move all consonants before the first vowel
* we always consider "y" a vowel
* leave all letters in their original case (to make it easier)

For example,
>>> pig_latin('dog')
'dogday'
>>> pig_latin('owl')
'owlay'
>>> pig_latin('cheetah')
'cheetahchay'
>>> pig_latin('thrush')
'thrushay'
>>> pig_latin('chthonic')
'chthonicthay'
>>> pig_latin('python')
'pythonpay'
>>> pig_latin('yellowjacket')
'yellowjacketay'

```
def has_a_vowel(word):
    return ('a' in word or 'e' in word or 'i' in word or
            'o' in word or 'u' in word or 'y' in word or
            'A' in word or 'E' in word or 'I' in word or
            'O' in word or 'U' in word or 'Y' in word)

# Top-Level Design #1
def pig_latin_1(word):
    precondition(is_string(word) and has_a_vowel(word) and not ' ' in word)

    return from_vowel(word) + initial_consonants(word) + 'ay'

# Top-Level Design #2
def pig_latin_2(word):
    precondition(is_string(word) and has_a_vowel(word) and not ' ' in word)

    return initial_consonants_at_end(word) + 'ay'
```

Figure 4.5. Two Possible Top-Level Designs for the Pig Latin Problem
4.3 Design by Cases

Some problems can be most easily addressed by considering a finite number of possible cases that, when taken together, include all possible problem instances. Sometimes we identify different ways in which different parameters could relate to each other: we could look at the point_is_in_window problem by considering cases in which the x value of the point is less than, equal to, or greater than the minimum x value of the window. In other cases, we may wish to look at different classes of values of one parameter: for pig latin, we could consider separate cases based on the number of initial consonants, using one rule for words with a single consonant, and a different rule for those starting with three consonants.

Once we have identified the various cases and rules to apply for each, we simply use if statements to distinguish among the cases, and provide each rule in its proper place. Figure 4.6 shows an approach to the pig latin problem in which we consider five cases (for example, if a word’s third letter is a vowel, but neither of the first two is, our rule is to concatenate all letters from the third on with those before the third and then the final “ay”). A good test suite for an algorithm designed by cases should include at least one test for each case, as in Figure 4.6.

One inherent risk of design by cases is the possibility that some program may contain a case that is not considered in by the designer, such as a word (or name) starting with five consonants in the case of pig latin. When an algorithm has such a limitation, it should be well documented with a comment (as in the figure) and checked against available real-world data (such as the spelling dictionary). Sometimes programmers will resort to “defensive programming” by adding a few extra cases that should not occur in practice. A better approach is to abandon design-by-cases for such problems, reserving it only for situations where one has confidence that only a finite number of cases are possible. We will see a less limited approach to the pig latin problem in the next section.

Figure 4.7 shows a case-based algorithm for the point_is_in_window problem. Such an algorithm might stem from the observation made above, that the point’s x must be less than, equal to, or greater than the window’s minimum x. Since our problem statement tells us that points on the border are considered in the window, we can combine the equal and greater cases, involving the minimum x only in the x < min_x test, in which case the point cannot possibly be in the window, and we give the answer False. A similar observation can be made about the maximum x value of the window, resulting in an initial identification of three cases (x to the left of the window, x within the window’s range of x values, and x to the right of the window). When the point’s x value is within the range, we must consider the relationship of the point’s y to the window, resulting in three sub-cases. Further thought about a completed case-based design sometimes yields an insight about a more concise approach — in this case our algorithm is equivalent to the one shown in Figure 4.12.

Check Your Understanding 4.3. Design an algorithm to produce one real-number solution to an equation of the form ax^2 + bx + c = 0 for any values of a, b, and c that allow a real-number solution (do not try to produce answers in cases for which which only complex number solutions are possible, such as a = 1, b = 0, c = 1). Hint: this is not the same as Check Your Understanding 4.1; you may want to think about the precondition of your algorithm, and the preconditions of any functions you want to call.
The "pig latin" problem involves rearranging the letters of a word, moving initial consonants to the end of the word and adding "ay".

For this dialect of pig latin:
* we move all consonants before the first vowel
* we always consider "y" a vowel
* leave all letters in their original case (to make it easier)

WARNING --- This algorithm only works if there are no more than four initial consonants, which is true for all words in my spelling dictionary!

(from logic import *)

```python
from logic import *

def has_a_vowel(word):
    return ('a' in word or 'e' in word or 'i' in word or 'o' in word or 'u' in word or 'y' in word or 'A' in word or 'E' in word or 'I' in word or 'O' in word or 'U' in word or 'Y' in word)

def pig_latin(word):
    precondition(is_string(word) and has_a_vowel(word) and not '' in word)

    if has_a_vowel(word[0]):
        return word + 'ay'
    elif has_a_vowel(word[1]):
        return word[1:] + word[0:1] + 'ay'
    elif has_a_vowel(word[2]):
        return word[2:] + word[0:2] + 'ay'
    elif has_a_vowel(word[3]):
        return word[3:] + word[0:3] + 'ay'
    else:
        assert(has_a_vowel(word[4]))
        return word[4:] + word[0:4] + 'ay'
```

Figure 4.6. Solving the Pig Latin Problem for English Words by Cases (NOT RECOMMENDED)
Is the point \( x, y \) is in the given window? (Where a "window" is a rectangular region that’s lined up with \( x \) and \( y \) axes.)

For example,

```python
>>> point_is_in_window(12, 120, 10, 20, 100, 200)
True
>>> point_is_in_window(20, 200, 10, 20, 100, 200)
True
>>> point_is_in_window(12, 220, 10, 20, 100, 200)
False
>>> point_is_in_window(12, 20, 10, 20, 100, 200)
False
>>> point_is_in_window(22, 120, 10, 20, 100, 200)
False
>>> point_is_in_window(22, 220, 10, 20, 100, 200)
False
```

```python
from logic import *

def point_is_in_window(x, y, min_x, max_x, min_y, max_y):
    precondition(min_x <= max_x and min_y <= max_y)
    #postcondition: returns true if and only if the point (x, y) is in the window
    # i.e., return true iff min_x <= x <= max_x and min_y <= y <= max_y
    if (x < min_x):
        return False
    elif (x > max_x):
        return False
    else:
        assert(min_x <= x and x <= max_x)
        if (y < min_y):
            return False
        elif (y > max_y):
            return False
        else:
            assert(min_x <= x and x <= max_x and min_y <= y and y <= max_y)
            return True

Figure 4.7. Design by Cases for Test of Whether a Point is in a Window
```
4.4 Basic Recursive Design

Basic recursive design is like a version of top-down design in which the simpler problem(s) we generate are actually instances of the same problem. For example, we could derive our power function this way even if we had not been given the mathematical facts $x^1 = x$ and $x^n = x \cdot x^{n-1}$. If we think of $x^n$ as $n$ copies of $x$ multiplied together, we could observe that finding $x^{n-1}$ is, in a certain sense, a simpler version of the same problem as finding $x^n$. Recursive design is often useful when a problem can easily be broken into an infinite number of cases that somehow all seem to be the same thing (as in the pig latin problem, if we wish to avoid Figure 4.6's restriction that there cannot be more than four initial consonants).

The great power of recursive design stems from our ability to call upon the function we’re already writing. If we can use this approach to solve the sub-problem(s) we generate in a top-down design, or to find a common pattern in an infinite design-by-cases, we can often produce a concise algorithm. However, this technique only works if we always produce simpler versions of the same problem that eventually reach one or more cases that are so simple we can solve them some other way.

The essential steps of basic recursive design are:

1. Identify, within the general problem, one or more smaller instances of the same problem. This is often the most challenging step in recursive design. One way to approach it is to look for a common pattern in several examples: $3^5$ can be expressed as $3 \cdot 3 \cdot 3 \cdot 3 \cdot 3$ or as $3 \cdot 3^4$, and $8^3$ as $8 \cdot 8 \cdot 8$ or $8 \cdot 8^2$; the second way of expressing each of these examples can be seen as a use of the general pattern of $x^n = x \cdot x^{n-1}$. In other words, finding $x^n$ includes a simpler problem of the same form, i.e., finding $x^{n-1}$.

2. Find a way to express the solution to the general problem in terms of the solution(s) to the smaller instance(s) and something involving the “other stuff” (e.g., we simply multiply the result of $x^{n-1}$ (the smaller instance of the same problem) by the leftover $x$ to find $x^n$). If this does not go well, consider advanced recursive design (Section 4.5).

3. Identify cases in which the “smaller instance” that would be produced by your general rule would produce something that is too trivial to bother rewriting, or too simple to pass the precondition for your problem. For example, rewriting $x^1$ as $x \cdot x^0$ is unnecessary, and would violate the precondition that the exponent must be positive. (We could, if we wished, extend the problem to allow exponents of zero, but in that case we still would not want to rewrite $x^0$ as $x \cdot x^{-1}$.) These trivial cases are known as the base cases of the algorithm.

4. Write down the (hopefully simple) algorithm for each base case (e.g., $x^1 = x$).

5. Check to make sure that the method for extracting the smaller instance(s) must eventually produce one of your base cases for any legal problem instance. For our algorithm for the power problem, note that by repeatedly reducing $n$ to $n - 1$, we must eventually hit 1, as long as we start with an integer that’s bigger than 1).

The following programming steps can be used to translate a basic recursive design into a program in a cookbook fashion:

a) Use an if (or if/elif sequence) to identify the base case (or cases) from Design Step 3. The final else will correspond to the recursive case(s). (An if inside the else can be used if there are several distinct recursive cases.) Thus, Figure 3.2’s if (exp == 1) distinguishes the base case from the recursive.

b) Use a return statement to produce each base case’s answer as described in Design Step 4. Figure 3.2’s return base expresses the base case solution of our power algorithm.
c) Within the else clause, create a variable for the smaller instance you identified in Step 1. Figure 3.2’s smaller_exp corresponds to this programming step.

d) Within the else clause, create a variable for the answer to the smaller instance by calling the function and passing it the variable from Programming Step c as in Figure 3.2’s computation

\[
\text{base_to_the_exp_minus_one} = \text{power}(\text{base}, \text{smaller_exp})
\]

e) Within the else clause, use a return statement to produce the solution to the original instance from Programming Step d’s solution to the smaller instance and any “other stuff”, as in Design Step 2. This is illustrated in the last line of the power function of Figure 3.2.

f) Although Design Step 5 is a check of the algorithm, rather than a step to be performed during the algorithm, we can often express part of it in the program. Specifically, we can confirm (e.g., via an assertion) that Programming Step c’s “smaller instance” variable is, in fact, smaller than the original instance. Note, however, that this is not a full check of Design Step 5, in that it shows the instance gets smaller in some way, but the shrinking could still “miss” the base case. For example, the assertion labeled “confirm we’re making progress” in Figure 3.2 confirms that smaller_exp is, in fact, smaller than exp, but if we had been subtracting 2 rather than 1 the exponent would get smaller and still miss the base case of 1 if we were to start with an even number.

As noted above, our exponentiation function can be derived in this way. We can also address the from_vowel and initial_consonants problems from Figure 4.5 via simple recursive design, as shown in Figure 4.8. Other examples will arise in our discussion of generate-and-filter algorithms. When programming a function created in this manner, it is often best to test the base case(s) first, since an error in the base case typically breaks every test.

The central insight for from_vowel is that the result of finding everything from the first vowel would be the same if an initial consonant were missing; in other words, from_vowel(‘thrush’) is the same as from_vowel(‘hrush’), since “t” is a consonant, and we have found a general pattern to make big instances of the from_vowel problem into smaller instances. When we encounter a word (or part of a word) that starts with a vowel, then there is no need to reduce it — we have our answer (from_vowel(‘ush’) is just ’ush’). Similar insights guide the design of initial_consonants: initial_consonants(‘thrush’) is a “t” followed by initial_consonants(‘hrush’), and initial_consonants(‘ush’) is nothing at all (written ‘’ in Python). Expressing each of these algorithms in Python involves capturing the above ideas using the language features of Appendix A, for example the notation word[1:] yields all letters of the word but the first, while word[0] yields just the first letter. Note the use of variables in the else clause of initial_consonants to make clear the origin of this algorithm in the design steps and programming steps listed above.

Basic recursive design also works for the initial_consonants_at_end problem, but in a slightly different way. At first, it may seem difficult to formulate the result of one instance in terms of the result of a simpler instance of the same problem. For example, expressing initial_consonants_at_end(‘thrush’) in terms of initial_consonants_at_end(‘hrush’) would require us to put the “t” into the middle of ”ushhr” without any easy way to know where within this string it belongs. However, the design process moves along smoothly if one notices that initial_consonants_at_end(‘thrush’) and initial_consonants_at_end(‘hrusht’) have exactly the same answer (see Exercise 4.3). Note that basic recursive design is essentially a process of formulating a self-referential definition for a concept that is not obviously mathematical, and then following the “mathematical derivation” design technique from that definition.

Check Your Understanding 4.4. Write the letters a through f and the numbers 1 through 5 next to specific elements of the initial_consonants function of Figure 4.8 to show how the programming steps and design steps
correspond to this algorithm.

**Check Your Understanding 4.5.** Give design notes for a basic recursive design to solve a simplified version of the palindrome testing function of Check Your Understanding Questions 2.1 and 4.2 in which the testing function is only required to work with strings made entirely of lower-case letters, such as “risetovotesir”. In other words, write answers to Steps 1-5 of the design process given in this section for this simplified palindrome testing problem. If this seems tricky, remember to start with Step [1] focus on one example, and imagine subcontracting out the smaller instance of the same problem. If you wanted to do as little work as possible to determine if “risetovotesir”,

""
The "pig latin" problem involves rearranging the letters of a word, moving initial consonants to the end of the word and adding "ay".

For this dialect of pig latin:
- we move all consonants before the first vowel
- we always consider "y" a vowel
- leave all letters in their original case (to make it easier)

(the examples are omitted on this page to make the new material fit)
""

```python
from logic import *

def has_a_vowel(word):
    return ('a' in word or 'e' in word or 'i' in word or 'o' in word or 'u' in word or 'y' in word or 'A' in word or 'E' in word or 'I' in word or 'O' in word or 'U' in word or 'Y' in word)

def initial_consonants(word):
    precondition(has_a_vowel(word))
    if has_a_vowel(word[0]):  # first letter _is_ a vowel
        return ''
    else:
        simpler_word = word[1:]
        assert(len(simpler_word) < len(word))
        simpler_words_consonants = initial_consonants(simpler_word)
        return word[0] + simpler_words_consonants

def from_vowel(word):
    precondition(has_a_vowel(word))
    if has_a_vowel(word[0]):  # first letter _is_ a vowel
        return word
    else:
        return from_vowel(word[1:])

def pig_latin(word):
    precondition(is_string(word) and has_a_vowel(word) and not ' ' in word)
    return from_vowel(word) + initial_consonants(word) + 'ay'
```

Figure 4.8. A Complete Algorithm for Pig Latin
is a palindrome, and could sub-contract out any palindrome testing problems that are smaller than the full 17-character string, what would you sub-contract out to make your life easy?

**Check Your Understanding 4.6.** Are any of the sub-problems you developed in Check Your Understanding 4.2 amenable to basic recursive design? If so, give design notes (answers to Steps 1-5) for an algorithm.

### 4.5 Advanced Recursive Design

Sometimes slight variations on the recursive design process can produce a simpler design. These variations are invented as they are needed; there is no set approach to producing them. As an example, we will illustrate the classic variant we’ll call “provide a hint as a parameter” by designing a new algorithm for the familiar problem of raising a number to a power. With this variant, instead of requesting progressively simpler instances of the power problem, we will produce better and better hints to help with the original instance. For example, our original power function from Figure 3.2 would respond to a request for $3^5$ by asking itself about $3^4$, and then multiplying the result by 3. Alternatively, we could have asked ourselves what would make a nice hint for someone trying to find $3^5$, and provide that hint as an extra parameter.

Certainly being told $3^4$ in advance would simplify the task of computing $3^5$; if we always provide the value of $b^{e-1}$ as a hint to a function that must compute $b^e$, as shown in Figure 4.9, the job of finding $b^e$ is greatly simplified! However, this leaves us wondering how to come up with the hint: just what could we write where it says “??? something clever here ???”? If only we had been given $b^{e-2}$ as a hint to the `power` function, we could find $b^{e-1}$ easily ... but then we’d wonder how to find $b^{e-2}$ to provide that hint, and so on. At first, this might sound like an unending problem, until we note that the hints are getting simpler. A process that seems to go on and on, but actually gets simpler, can be the basis of a recursive design if we can reliably find a simplest case to use as the base. So, when is the problem so simple that we do not need a hint, or the hint so trivial we can just write it down without requiring another hint? When the hint is $b^1$, of course.

Thus, we can construct the algorithm shown in Figure 4.10. The function `power_with_hint` finds $b^e$, given the values of $b^{e-s}$ and $e-s$. Once we have solved the “power with hint” problem, the power problem is easily solved by calling upon it with $b^1$ as the starting hint. Note that the precondition of `power_with_hint` requires that the given hint be correct, and the test suite only performs tests for which this is the case.

One might well wonder why we would ever want this version of the function rather than the one from Figure 3.2 — this one always gives the same result! In this example, the only advantage provided by this design comes from the efficiency of the algorithm produced. There is sometimes a minor speedup and saving of computer memory from the way programming languages can deal

```python
def power_with_hint(base, exp, base_to_the_exp_minus_one):
    precondition(is_number(base) and is_integer(exp) and exp > 0 and
                  b_to_the_exp_minus_one == base**(exp-1))
    return base * base_to_the_exp_minus_one

def power(base, exp):
    precondition(is_number(base) and is_integer(exp) and exp > 0)
    return power_with_hint(base, exp, ???? something clever here ???) # FIX THIS!
```

**Figure 4.9.** A First Try at Creating a Power Function that Needs a Hint (not complete).
with *tail recursion* — recursion in which the value returned is exactly what was returned from the recursive call. In Figure 4.10, `power_with_hint(3.0, 5, 4, 81.0)` returns exactly what it is given

```python
>>> power_with_hint(3.0, 3, 2, 9.0)  # find 3.0^3 given that 3^2 is 9.0
27.0
>>> power_with_hint(3.0, 3, 1, 3.0)  # find 3.0^3 given that 3^1 is 3.0
27.0
>>> power_with_hint(3.0, 5, 4, 81.0)  # find 3.0^5 given that 3^4 is 81.0
243.0
>>> power_with_hint(3.0, 5, 3, 27.0)  # find 3.0^5 given that 3^3 is 27.0
243.0

>>> power(3.0, 3)
27.0
>>> power(3.0, 5)
243.0
```

```python
from logic import *

def power_with_hint(base, exp, exp_minus_something, base_to_exp_minus_something):
    precondition(is_number(base) and is_integer(exp) and exp > 0 and
        exp_minus_something >= 1 and exp_minus_something <= exp and
        base_to_exp_minus_something == base**(exp_minus_something))
    progress(exp - exp_minus_something)
    # postcondition: same as "power": result is base ** exp
    if (exp_minus_something == exp):
        return base_to_exp_minus_something
    else:
        return power_with_hint(base, exp, exp_minus_something+1,
            base*base_to_exp_minus_something)

def power(base, exp):
    precondition(is_number(base) and is_integer(exp) and exp > 0)
    # postcondition: the result is the base raised to the exp power,
    # or a close approximation for information that
    # isn’t represented exactly (such as real numbers).
    # i.e., result is base ** exp

    return power_with_hint(base, exp, 1, base)
```

Figure 4.10. A *power* Function that Needs a Hint
by `power_with_hint(3.0, 5, 3, 27.0)`, but in Figure 3.2, `power(3.0, 5)` must multiply the result of `power(3.0, 4)` by 3 before returning it.

However, we will generally not concern ourselves with such minor speedups in this course. We have introduced this “with a hint” design approach because there can be more compelling reasons to use it. Specifically, we may sometimes find it easier to develop a design based on better and better hints, or find that the resulting algorithm avoids an enormous amount of redundant work, resulting in a program that is dramatically faster, as we will see in Chapter 6.

Note that “provide a hint” is one of many possible variants on basic recursive design. We’ll see an example of another variant, which we’ll call “break down the solution”, in Section 4.6.2.

**Check Your Understanding 4.7.** Illustrate the execution of `power(3.0, 3)` using the function of Figure 4.10 via a call tree or via a sequence of substitutions.

**Check Your Understanding 4.8.** The `power_with_hint` function solves, not the power problem, but the “power with hint” problem. Show how `power_with_hint` can be designed by using basic recursive design on this “power with hint” problem. (Hint: think carefully about what counts as a “simpler instance”.)

### 4.6 Generate and Filter

If we can define a finite set of potential solutions to any given problem instance, and a way to test each potential solution to see if it actually works, we can create a “generate and filter” algorithm (also known as an “enumerate and test” algorithm), such as Algorithm Primes-1 from Chapter 1. Algorithms produced in this way can be very slow (if there are many possible solutions to consider), but they can be effective when the number of possible solutions is small, either because we know we’ll only have to handle small problem instances, or in some rare cases because the number of possible solutions does not grow quickly with the size of the problem instance.

We will illustrate this approach using a problem we’ll call the “dinner party problem”: suppose we want to have a dinner party with at least `k` of our `n` friends, but ensure that all `k` like each other. Given the list of `n` friends, the number `k`, and information about which pairs friends like each other, list all suitable groups who could be invited to dinner.

This is related to a problem from the field of mathematics known as graph theory, which is concerned with information that can be represented in terms of sets of values (e.g., the set of friends) and relationships or connections among those values (e.g., the information about friendships). The values are referred to as the nodes of the graph, and the connections as edges or arcs of the graph. A collection of nodes, each of which is connected to every other node in that collection, is known as a subclique; a complete clique is a subclique that can’t be expanded by adding another member, e.g., a set of people who all like each other, but to which we can’t add another without then having two who aren’t friends. Our dinner party problem would thus be called the problem of finding all subcliques of size at least `k` in a given graph. As with the square-root problem, the mathematical definition may help us identify a valid solution, but it doesn’t immediately lead to an algorithm.

The generate-and-filter design strategy is essentially a version of top-down design in which we produce a generate problem, a test problem, and a typically-straightforward filtering problem (using the test on each potential solution produced by the generator). Each of these sub-problems is typically amenable to various combinations of top-down and recursive design, so there is nothing fundamentally new in this section. However, this case is important enough to have its own name, and subtle enough that it deserves some explanation.

As is often the case, the idea behind our generate-and-filter solution is quite simple. We start writing down subsets of the `n` people, and then examine each subset to see if it is at least large enough and if everyone there considers everyone else a friend. We collect all such subsets to produce our
answer. Before we can work on our algorithms for our sub-problems of generation and testing, we'll need to choose a way of representing sets (or subsets) of people and information about friendships.

As there is no easy way to record sets of people with numbers, we'll use strings of letters, with each letter representing a different person (e.g., "Y" for Yolanda, "H" for Henry, "M" for Mary, and "V" for Vince). We can identify a group of people with a string of letters (e.g., "HMV" for Henry, Mary, and Vince), or a collection of friendships with a string made up of pairs of letters separated by spaces (e.g., if Yolanda and Vince are not friends, but Henry is friends with each of them, we would use the string "HY HV"). Thus, the problem of identifying whether or not a given set of people form a clique has been turned into a problem of manipulating strings. To keep things from getting any more complicated, we will assume that all friendships go both ways — we will not consider cases in which Henry considers Vince a friend but Vince does not consider Henry a friend, assuming that any string containing "HV" means Henry likes Vince and Vince likes Henry.

4.6.1 Testing if a Group is a Subclique

We will discuss the “test” part of our generate-and-filter algorithm first, producing a function we’ll call is_a_subclique, which we’ll give a string of letters corresponding to people and a string giving friendships. For example, our algorithm will ensure that is_a_subclique("HMV", "HM HV MV MY") is true, but is_a_subclique("YMV", "HM HV MV MY") is false.

This problem can be solved with basic recursive design. Using the numbering of Section 4.4:

1. Any is_a_subclique instance with more than two people could be made into a smaller is_a_subclique instance by simply removing one person.
2. Such an instance is a subclique if and only if the group with one person removed is itself a subclique, and the person who was removed is a friend of everyone else.
3. Groups with two or fewer people are not described in Step 1 above.
4. A group of two people is a subclique if and only if the people are friends. One might wonder whether it even makes sense to ask about a subclique of one or fewer people. This could be ruled out via a precondition, or we could follow mathematical convention and accept any group of zero or one people as a valid subclique.
5. Removing one person at a time from a group of more than two people must eventually give us a group of just two.

Note also that Step 2 is actually a top-down design hidden within a basic recursive design. We need some way to know if “the person who was removed” is a friend of everyone else. We thus set aside the subclique testing problem, and take up the “friend of everyone” problem, starting with some quick test suites. This too is amenable to basic recursive design:

1. Any friend_of_everyone instance with two or more people in the group of potential friends of the person being checked can be made into a smaller instance of the same problem by removing one of the potential friends.
2. The solution to such an instance is True if and only if the person being checked is a friend of the remaining potential friends, and also a friend of the one potential friend who was removed.
3. Any instance with only one person in the list of potential friends is not covered above.
4. The instances from Step 3 should yield True if and only if the person being checked and the single potential friend are, in fact, friends.
5. Removing one person at a time from the list must produce a list of size one.

Once again, Step 2 includes a top-down design — we must be able to check if two people are friends. Fortunately this can be done easily with Python’s “in” operation. The test suites and solutions for the friends and friend_of_everyone problems are shown in Figure 4.11.
Test to see if person1 and person2 are listed as friends.
To see if A and B actually are friends, we must test A to B and B to A,
since we only require that friendship be listed one way
(two is no way to show that A likes B but not the other way around).
Names are abbreviated to one (unique) letter;
thus, a friendship is just a pair of letters,
and a list of friendships a list of such pairs.

# First, a sub-function we'll need later
>>> friends('M', 'V', 'HM HV MV MY') # does Mary like Vince?
True
>>> friends('V', 'M', 'HM HV MV MY') # does Vince like Mary?
True
>>> friends('V', 'Y', 'HM HV MV MY') # does Vince like Yolanda?
False

# Now, the main event:
>>> friend_of_everyone('M', 'HVY', 'HM HV MV MY') # does Mary like H, V, and Y?
True
>>> friend_of_everyone('V', 'HMY', 'HM HV MV MY') # does Vince like H, M, and Y?
False

# The first of the above will produce the following; the last is the base case
>>> friend_of_everyone('M', 'VY', 'HM HV MV MY') # does Mary like V, and Y?
True
>>> friend_of_everyone('M', 'Y', 'HM HV MV MY') # does Mary like Y?
True

"""
from logic import *

def friends(person1, person2, friendships):
    precondition(len(person1) == 1 and len(person2) == 1)
    order1 = person1+person2
    order2 = person2+person1
    return (order1 in friendships) or (order2 in friendships)

# Now a function to see if one person is a friend of everyone in a group:
# return true if and only if "one_person" is a friend of everyone in "others"
def friend_of_everyone(one_person, others, friendships):
    precondition(len(one_person)==1 and len(others) >= 1)
    if (len(others) == 1):
        return friends(one_person, others, friendships)
    else:
        first_of_others = others[0]
        rest_of_others = others[1:]
        return (friends(one_person, first_of_others, friendships) and
                friend_of_everyone(one_person, rest_of_others, friendships))

Figure 4.11. Posing and Solving the “Friend of Everyone” Problem (friend_of_everyone.py).

Copyright 2003-2015 David G. Wonnacott; see Page 7 for terms of free use
Once we have completed and tested the `friend_of_everyone` function, we turn our basic recursive design for the `is_a_subclique` problem into the function in Figure 4.12. The file `clique_logic.py`, which contains functions to check the preconditions, is shown in Figure 4.13.

**Check Your Understanding 4.9.** Give a call tree for `friend_of_everyone('V', 'HMY', 'HM HV MV MY')`, based on the code of Figure 4.11. Do not write out the calls to precondition functions.

**Check Your Understanding 4.10.** Give a call tree for the `is_a_subclique('HMV', 'HM HV MV MY')`, based on the code of Figure 4.12. Do not write out the calls to precondition functions, or write out the details inside each call to `friend_of_everyone`.

```python
""
Check to see if a string (such as 'YCV' or 'YCVM') is a subclique (that is, if all pairs of letters correspond to friends)

>>> is_a_subclique('HMV', 'HM HV MV MY')
True
>>> is_a_subclique('YMV', 'HM HV MV MY')  # Yolanda and Vince aren't friends!
False
""

from logic import *
from clique_logic import *
from friend_of_everyone import friends, friend_of_everyone

def is_a_subclique(possible_subclique, friendships):
    precondition(len(possible_subclique) >= 2)
    precondition(is_valid_name_list(possible_subclique))
    precondition(is_valid_friendship_list(friendships))

    if len(possible_subclique) == 2:
        # two people total --- just see if they're friends.
        return friends(possible_subclique[0], possible_subclique[1], friendships)
    else:
        # if we pick one person, and that person likes the others, 
        # and the others all like each other, we have a subclique.
        one_person = possible_subclique[0]
        others = possible_subclique[1:]
        return (friend_of_everyone(one_person, others, friendships) and
                is_a_subclique(others, friendships))

Figure 4.12. Testing a proposed subclique (clique_test.py)
Various functions to check validity of lists of people and friendships
People’s names must be a single upper-case letter.
Friendships must be made of peoples names,
but these don’t have to be people in the list of candidates for subcliques
A valid list of people to be considered must not contain duplicates,
but the friendship list can.

```python
from logic import *  # for is_string

def is_valid_name(string):
    names = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
    return len(string) == 1 and string in names

def is_valid_name_list(list):
    if not is_string(list):
        return False
    elif list == '':
        return True
    else:
        return (is_valid_name(list[0]) and
                not list[0] in list[1:] and
                is_valid_name_list(list[1:]))

def is_valid_friendship(pair):
    return is_valid_name(pair[0]) and is_valid_name(pair[1])

def is_valid_friendship_list(friendships):
    if not is_string(friendships):
        return False
    elif len(friendships) == 0:
        return True
    elif len(friendships) == 1:
        return False
    elif len(friendships) == 2:
        return is_valid_friendship(friendships)
    else:
        return (is_valid_friendship(friendships[0:2]) and
                friendships[2] == ' ' and
                is_valid_friendship_list(friendships[3:]))

Figure 4.13. Functions for Testing Preconditions of Subclique Functions.
```
4.6.2 Generating all Possible Subcliques

We now return to our overall goal of solving the dinner party problem, having completed the “test” part of our generate-and-filter design. Our goal now is to list all possible subcliques, without regard to whether or not they actually are subcliques of the proper size (in mathematical terms, we will generate all subsets of our set of friends).

After our success in solving the clique testing problem through a combination of basic recursive design and top-down design of some steps, we might first approach generation this way. This approach will serve to motivate our next (and most complicated) example of advanced recursive design. We begin once again with the five steps of Section 4.4:

1. A subset generation problem with at least two people (in the set) contains within it a smaller problem of finding all subsets of all but one of the people.

2. The solution to such a problem can be made by combining all the subsets of the smaller set (with all but one of the people) with a copy of that list of subsets with the removed person added. For example, the eight subsets of Yolanda, Henry, and Mary can be formed by combining the four subsets of Henry and Mary (specifically, "HM", "H", "M", and "" (the empty set)) with a copy of these four subsets with Yolanda added ("YHM", "YM", "YM", and "Y").

3-5. (The base case and ensuring that we reach it is straightforward with this design.)

Once again, Step 2 has produced a new sub-problem in addition to the new instance of the problem being solved. This time, we’ll need to combine the “removed person” string (e.g., "Y") with a string giving a list of subsets (e.g., if we separate the subsets by spaces, "HM H M "). In principle, this is not any harder than some of our other basic recursive design problems, but it would involve a rather tedious exploration of string manipulation in Python, after which we would encounter another rather tedious string manipulation problem as we extract the subsets to hand them to our is_a_subclique function. Thus, we will set aside this workable but tedious approach (until the exercises), and explore a different design strategy.

When we encounter difficulty with building the final answer from the answer(s) to the sub-problem(s), we can often produce a solution more easily by trying to break down the answer instead of the question. We look at examples of questions and their final answers, and consider how we could break each answer down in some sort of organized or methodical way. We then consider whether we could envision each piece of each answer as the answer to an instance of some different question that would also include an instance answered by our final answer. If so, we may be able to solve this different question more easily than the original question, and simply and cast any instance of the original question in these terms. This idea can be hard to grasp at first, so if the previous sentences are confusing, follow the example below, and then try again to make sense of the big picture. As with basic recursive design, the biggest challenge often lies in finding the pattern of the recursion, and we may be able to get un-stuck by writing out some examples on paper and marking them up.

For our subset generation problem, this approach would work as follows: we write an answer to an example problem instance — we expect to see an answer like "YHM YM YM Y HM M " to the “collect subsets” problem instance with Yolanda, Henry, and Mary, i.e., collect_subsets("YHM"). We may see several ways to divide this specific answer into sub-parts. One such division would split those subsets with Yolanda from those subsets without Yolanda, in other words, the two
different sub-answers "YHM YH YM Y " and "HM H M ". Note that the latter is an answer to a 
collect_subsets question; it was the first of these that created tedious string manipulation problems. However, "YHM YH YM Y " is not the answer to any collect_subsets question, so we can’t apply basic recursive design here. We therefore ask ourselves “for what question would that be the answer?” We could consider "YHM YH YM Y " to be the answer to the question “What are all the subsets of Henry and Mary, with Yolanda added to each of them?”. We could design a function collect_subsets_with to handle this kind of question — note that "HM H M " is also an answer to this question if we allow the option of adding nobody to the subsets of Henry and Mary.

Having identified a question whose answers can easily be combined into the answer we want, we now attempt basic recursive design of this new “collect subsets with” problem:

1. Any collect_subsets_with problem with \( m \) people to be added to subsets of \( n \) people, in which \( n \geq 1 \), contains two smaller collect_subsets_with problems, based on taking out one of the \( n \) people. In one problem, we add the \( m \) people to subsets of the remaining \( n - 1 \), and in the other, we add \( m \) people and the one we removed from the group of \( n \), to the subsets of the \( n - 1 \). In other words, collect_subsets_with("Y", "HM") contains the smaller problems collect_subsets_with("Y", "M") and collect_subsets_with("YH", "M"), if we consider Henry to be the one person we remove from the group of \( n \) (2).

2. The solution to the whole collect_subsets_with problem is just the concatenation of the solutions of the two smaller instances.

3. Any instance with \( n = 0 \) people to be considered is not handled by the rule above.

4. Any instance with \( n = 0 \) is simply the group of people to be added.

5. Removing one person at a time from the people being considered will eventually give us \( n = 0 \) people left to be considered.

Figure 4.14 shows the collect_subsets_with function that results from this design, along with a simple collect_subsets function that restates any collect_subsets question in terms of the new collect_subsets_with — this function is important because we still need to be able to answer the original question.

Check Your Understanding 4.11. Look ahead to Exercise 4.11 to see the definition of the map coloring problem. Write down one instance of this problem with three states and the three colors “r”, “g”, and “b”, and one instance with two of those two states and same three colors. Then write down the set of all possible colorings that should be produced by the generator for each instance, starting with the smaller instance first. Note that it will probably be tedious to construct the answer to the larger instance from the smaller. Then try to find a way to break down the answer to the larger instance so that it contains several components that could all be seen as answers to a question that is slightly different from the “generate all possible colorings for these states using those colors” problem. When reading this problem, it may seem straightforward, but it is typically worth actually writing this step down: write out all 27 possible colorings for the bigger instance, and try to group them into a small number of groups, in some way that actually makes sense and could be generalized. If this actually is straightforward, you can quickly feel encouraged to move on with your design; if it is not straightforward, you should spend more time thinking about the design before actually trying to code up an algorithm!
collect_subsets:
return a string of subsets of letters, with a space after each subset

>>> collect_subsets('YHM')
'YHM YH YM Y HM H M '
>>> collect_subsets_with('Y', 'HM') # subsets of HM, with Y added to each
'YHM YH YM Y '
>>> collect_subsets_with('', 'HM') # subsets of HM, nothing added to each
'HM H M '
>>> collect_subsets_with('YH', 'M') # subsets of M, with YH added to each
'YHM YH '
>>> collect_subsets('YHMV')
'YHMV YHM YHV YMV YM YV Y HMV HM HV H MV M V '

from logic import *

# collect subsets (as in the main problem), but only those that
def collect_subsets_with(subset_so_far, to_be_considered):
    if len(to_be_considered) == 0: # no one to be considered--is subset_so_far good?
        return subset_so_far + ' ',
    else:
        who = to_be_considered[0]
        others = to_be_considered[1:]
        # concatenate results of groups without "who" and those with "who"
        return (collect_subsets_with(subset_so_far+who, others) +
                collect_subsets_with(subset_so_far, others))

def collect_subsets(people):
    # just start the function above with group_so_far='', to_be_considered=people
    return collect_subsets_with('', people)

Figure 4.14. Generating All Subsets of a Set of People
4.6.3 Filtering

Of course, some subsets of our friends might not make for a pleasant party — we only want to consider groups that are compatible (i.e., subcliques) and large enough. There are a number of ways to approach the programming for this conceptually simple “filtering” task.

We could compute the complete set of subsets as one string, as is done by the \texttt{collect_subsets} function, and write an additional function (or functions) to filter the result. The new code would have to take a string like "YHM YH YM Y HM H M ", separate out the individual solutions (into "YHM", "YH", "YM", etc.), and then check each solution and re-assemble a complete string. This task is not conceptually difficult, though includes a fair bit of programming that seems redundant, since we have to why break up the collection and then re-assemble it.

Alternatively, we could filter the subsets as they are produced, changing \texttt{collect_subsets} so that it only produces the subsets we want, as shown in Figure 4.15. Note that we have chosen to do the filtering in the base case, changing the \texttt{return subset_so_far + " "} of \texttt{collect_subsets_with} into an if-else that will either return the desired subclique or an empty string (the latter contributing nothing to the final result). This change turns out to be considerably easier than doing the filtering in the lines that perform the recursive calls, or (as in the paragraph above) in \texttt{collect_subcliques}. The parameter list of \texttt{collect_subcliques_with} is considerably longer than that of \texttt{collect_subcliques}, and of course this increases the size of the calls as well as the definition.

Many modern programming languages, including Python, provide additional features that can simplify one or both of these approaches. While advanced coding techniques are not the primary focus of this course, it is worth mentioning at least three ways of making this code more concise and clear:

1. We can define \texttt{collect_subcliques_with} inside \texttt{collect_subcliques}, thus giving it access to the variables of \texttt{collect_subcliques} without requiring additional parameters. While this approach can produce more concise and clearer solutions, it can also interfere with testing, and so should be applied selectively.

2. We can replace our \texttt{collect_subsets} function with a generator, which yields one solution at a time, and write a separate \texttt{filter} to examine them, and then combine the generator and filter in our \texttt{collect_subcliques} function. See the Python documentation for details on how to express generators and filters in Python.

3. We can represent the collection of subcliques as a list of short strings (each being one subclique) rather than one big string that includes all the subcliques, and then use list-processing functions to do the filtering. This approach is appropriate once one has studied lists, e.g. in a second course on computer science. (The Python documentation also includes information about lists, for anyone who can’t wait.)

Our generate-and-filter solution to the subcliques problem is now complete. Note that there are several ways in which it causes the computer to do unnecessary work — for example, \texttt{collect_subcliques_with} comes up with all subsets and then checks their sizes and examines the friendships. But if we want a party of at least 20 people, and \texttt{group_so_far} has eight, and \texttt{to_be_considered} has 11, then we might as well just give up and \texttt{return " "} without considering all 2048 possible groups and rejecting each on the grounds that it has fewer than 20 people. Similarly, if \texttt{group_so_far} contains a pair of people who don’t like each other, we can’t pos-
sibly get any legal subcliques, and might as well stop considering ways to expand this already-
doomed-to-fail group. These considerations do not shed any new light on the design process, so
we leave them for the exercises, and return to our study of design techniques.

"""
collect_subcliques:
Return a string of all legal subcliques of at least a certain size,
given the desired size, the set of people, and the friendships.
In other words, if I want to have a party with at least "minimum size" of my
friends, what are the ways in which I can invite compatible groups?

>>> collect_subcliques(3, 'YHMVC', 'HM HV MV MY CY')
'HMV '  
>>> collect_subcliques(4, 'YHMVC', 'HM HV MV MY CY')
''
>>> collect_subcliques(4, 'YHMVC', 'HM HV MV HY MY CY CV CM CH')
'YHMC HMVC '
>>> collect_subcliques(3, 'YHMVC', 'HM HV MV HY MY CY CV CM CH') # any order is o.k.
'YHMC YHM YHC YMC HMVC HMV HMC HVC MVC '
"""

from logic import *
from clique_test import *
from clique_logic import *

# collect subcliques that contain everyone in "group_so_far"
def collect_subcliques_with(group_so_far, to_be_considered, min_size, friendships):
    if len(to_be_considered)==0: # no one to be considered--is group_so_far good?
        if (len(group_so_far)>=min_size and is_a_subclique(group_so_far,friendships)):
            return group_so_far + ' '  
        else:
            return ''
    else:
        who = to_be_considered[0]
        others = to_be_considered[1:]
        # concatenate results of groups without "who" and those with "who"
        return (collect_subcliques_with(group_so_far+who, others, min_size, friendships) +
            collect_subcliques_with(group_so_far , others, min_size, friendships))

def collect_subcliques(min_size, people, friendships):
    precondition(is_integer(min_size) and min_size >= 0)
    precondition(is_valid_name_list(people))
    precondition(is_valid_friendship_list(friendships))

    # now, collect_subcliques: start with group_so_far='', to_be_considered=people
    return collect_subcliques_with('', people, min_size, friendships)

Figure 4.15. Generating Subcliques via Generation and Filtering
4.7 Reducing a New Problem to a Solved Problem

In some cases, problems that seem quite different at first may be related in some subtle way. Recognizing this relationship between a new problem and a familiar problem can be extremely valuable, especially if there is a known efficient solution to the familiar problem (in which case this solution can be adapted for the new problem), or if there is a proof that no efficient solution can exist for the known problem (in which case we won’t waste time looking for an efficient solution to the new problem, if we can prove they really are related).

Consider, for example, the following problem involving the testing of electronic components on a “printed circuit board” such as those used in computers. We are given information about the locations of components on the circuit board to be tested, and information about the sizes of of test “probes” that can determine whether or not a given component is working. We must figure out how many components can be tested in a single test (without making probes bump into each other).

Each instance of this testing problem corresponds to an instance of the “maximum clique” problem: we treat each component as a person, and treat as friends any pair of components that are far enough apart to ensure their probes would not bump — a subclique in this hypothetical group of friends corresponds to a set of components that can be checked simultaneously. Thus, the largest clique is the largest group of components we can check at once.

Check Your Understanding 4.12. We didn’t need to use basic recursive design for both the initial_consonants and from_vowel functions of Figure 4.8. Show that one can be used to provide a simple solution to the other.

4.8 Summary

While there is no general algorithm for designing algorithms, there are a number of techniques that often work. If the problem has been stated in mathematically precise language, it may be possible to derive an algorithm directly from the problem statement and relevant definitions. In other situations, it may be possible to reduce a problem to one or more problems that have already been solved, or to a set of problems that can be solved independently via top-down design. When all non-trivial instances of the problem can be reduced to smaller instances whose solution quickly leads to the full solution, basic recursive design is often helpful; we use the term advanced recursive design for any of several variants on the basic approach of Section 4.4, such as those of Sections 4.6.2 and 4.5. Some problems are best approached by dividing the general problem into several cases and producing different algorithms (possibly using different design techniques) for each.

When in doubt about which technique to use, or when no technique seems to lead to a general algorithm, try solving a few problem instances by hand, either alone or in groups. Sometimes the process of just talking with a friend about the problem and why it can’t be solved easily, can help. These strategies may identify an appropriate design technique, show how to succeed in the use of a strategy, or in rare instances help to inspire a flash of insight that leads to an easy solution or a new approach. If none of these approaches work and there are only a finite number of possible answers for any given problem instance, it may be necessary to resort to a generate and filter algorithm, though (as we will see in Chapter 6), this often produces code that is quite slow.
4.9 Further References

Further work with algorithm design generally involves the design of data structures as well, and is covered in (and after) the second semester introduction to computer science. The circuit board testing problem was taken from Michael Trick’s web pages about graph coloring and cliques. These can be found on web servers at both Carnegie-Mellon University (http://mat.gsia.cmu.edu/COLOR/color.html) and the DIMACS research organization (http://dimacs.rutgers.edu/Volumes/Vol26.html).
4.10 Exercises

**Exercise 4.1.** Give an algorithm (expressed as one or more Python functions) to solve each of the “circular window” problems of Exercise 2.2. Include a test suite and an appropriate **precondition** and a postcondition (unless your postcondition is exactly the same as your algorithm). Also include a comment explaining which design technique you think is most appropriate for this problem.

**Exercise 4.2.** (*) Consider the problem of determining if two line segments intersect, given the Cartesian coordinates (i.e., x and y values) for each end of each segment. We can begin to produce a formal statement of this problem by rewriting it as follows: "Given the coordinates of four points, \( (x_1, y_1), \ldots, (x_4, y_4) \), determine whether or not there exists a point \( (x, y) \) that lies on both the segment connecting \( (x_1, y_1) \) to \( (x_2, y_2) \) and the segment connecting \( (x_3, y_3) \) to \( (x_4, y_4) \). (A segment’s endpoints should be considered part of the segment.)". This leaves open the questions of (1) whether or not we want to accept a degenerate "line segment" defined by two identical points (i.e., is there a line segment from \((1, 7)\) to \((1, 7)?\)), and (2) how to formalize the question of whether or not a point lies on a segment, which we can answer in at least two ways.

a) One way to define a non-vertical line is by its slope and "y-intersect", i.e., as the set of all \((x, y)\) pairs satisfying an equation of the form \( y = mx + b \) (the slope of a vertical line is undefined, and vertical lines do not intersect the y axis). For example, \( y = 2x + 5 \) describes a line through the points \((0, 5)\) and \((1, 7)\). Use this form of equation to precisely state what it means for a point to be on a non-vertical line segment, and give a separate statement of what it means for a point to be on a vertical segment (sometimes problem statements, like algorithms, can be broken into cases). **State whether or not your definition allows degenerate line segments.**

b) A line can also be defined by a set of parametric equations, i.e., as the set of all \((x, y)\) pairs satisfying a pair of equations of the form \( x = nt + x_0 \) and \( y = nt + y_0 \). For example, \( x = t + 0 \) and \( y = 2t + 5 \) also describe the line through the points \((0, 5)\) and \((1, 7)\). Use this form of equation to precisely state what it means for a point to be on a segment.

c) Give an algorithm (expressed as one or more Python functions) for determining whether or not two line segments intersect. You may use your definitions from parts a and b or start from scratch, but beware: this problem is great for illustrating the trouble that can be caused by trying to design an algorithm by hacking. Also note that it is not possible to use a generate-and-filter algorithm for this problem, since the number of points on a non-degenerate segment is not finite. You may choose to allow or disallow degenerate segments of length zero, but your algorithm should work for all other cases. If you disallow degenerate segments, you should indicate this with a **precondition**. If you allow degenerate segments, you should indicate this by stating **precondition(true)** at the start of your function.

**Exercise 4.3.** Complete the design for the “initial consonants at end” problem from Figure 4.5 and implement your design as a Python program. Include a test suite for this problem and confirm that it works. Also confirm that it produces correct answers for the “pig latin” problem when combined with the second approach for Figure 4.5.

**Exercise 4.4.** Design an algorithm to solve the pig latin problem, and try to make use of the “provide a hint” version of advanced recursive design. Is this approach helpful for this problem?

**Exercise 4.5.** A **palindrome** is a word or phrase that has the same sequence of letters when written backwards, such as “Lived on decaf, faced no devil”, or “A man, a plan, a canal: Panama”. Develop an algorithm that takes a string of lower-case letters and returns true if and only if the string corresponds to a palindrome. In other words, **palindrome(“livedondecaffacednodevel”) == True**, and **palindrome(“nocaffieneforme”) == False**. Express your algorithm as a Python function, and include a test suite or specification (possibly from Check Your Understanding 2.1 and 2.5) and a comment describing your design process. Give a complete implementation that does not rely on Python features not specifically mentioned in Appendix A (i.e., if you choose to reverse a string, write your own function to do so).

**Exercise 4.6.** The **factorial** function takes an integer \( n \) and yields the product of all integers from 1 to \( n \). The factorial function is usually abbreviated by placing an exclamation mark (!) after the number, as in “\( n! \)”, so \( 1! = 1 \), \( 2! = 2 \), \( 3! = 6 \), \( 4! = 24 \), \( 5! = 120 \), and \( 17! = 356587428096000 \). This function can be defined by two rules that lead directly to a recursive design: (a) \( 1! = 1 \), and (b) \( n! = n*(n-1)! \), when \( n > 1 \). In Python, we would use a name, such as **factorial**, as in **factorial(5) == 120**.

a) Apply the “mathematical derivation” technique to translate these self-referential rules into a recursive Python function, complete with a test suite.
b) Apply the “provide a hint as a parameter” form of advanced recursive design to this problem, producing a recursive Python function, complete with a test suite.

Which design approach was easier? Are there any obvious advantages to one function over the other?

**Exercise 4.7.** The Fibonacci Sequence is defined by the following two rules: (a) the first two elements are both 1, and (b) every other element is the sum of the two elements before it. In other words, the 1st and 2nd elements are 1, the 3rd is 2, the 4th is 3 (2+1), the 5th is 5 (3+2), the 6th is 8 (5+3), and the 9th is 34 (which is 21, the 8th element, plus 13, the 7th). In Python, we might name this function Fibonacci, as in `Fibonacci(9)==34`.

a) Apply the “mathematical derivation” technique to translate these self-referential rules into a recursive Python function, complete with a test suite.

b) Apply the “provide a hint” form of advanced recursive design to this problem, producing a recursive Python function, complete with a test suite.

Which design approach was easier? Are there any obvious advantages to one function over the other? Does your test suite include any elements of the sequence past the 50th?

**Exercise 4.8.** Develop a subclique generation algorithm through basic recursive design, as outlined in the 2nd paragraph of Section 4.5. In other words, create a version of `collect_subcliques` that calls upon itself (rather than `collect_subcliques_with`) and then builds the solution to the n-person subclique problem from the solution to the n − 1-person subclique problem. How does your solution compare to Figures 4.12 and 4.15 in terms of the number of lines of Python program the resulted and in terms of the difficulty of understanding the program?

**Exercise 4.9.** Modify the subclique algorithms of Section 4.6 to avoid unnecessary work whenever possible, as discussed at the end of Section 4.6.2.

**Exercise 4.10.** Do some additional reading of the Python documentation investigate one (or more) of the three approaches mentioned in Section 4.6.2 for producing clearer, more concise code. Then modify the the subclique algorithms of Section 4.6 to use one (or more) of these approaches.

**Exercise 4.11.** (*) Give a generate-and-filter algorithm to find all ways to color a map with a given set of colors without giving any two neighboring states the same color. Your algorithm should not be restricted to the states of the U.S.A., but should work for any map that can be represented in the way described below. Note that, if we allow states that are split into two or more pieces (such as Michigan), the states can border each other in very complex ways that are not possible on simpler maps (e.g., we can in principle have a “clique” of six states, each of which borders the other five).

For this problem, we will represent:
- each state with a single capital letter (e.g., C for Connecticut, M for Massachusetts, V for Vermont, H for New Hampshire, Y for New York),
- each color with a single lower case letter (e.g., r for red, b for blue, g for green, y for yellow),
- any set of states or colors with a string (e.g., “CMWY” or "rgb"),
- the list of bordering states as a string of space-separated letter pairs (e.g., “CM CY MV MH MY VH VY”), and
- a potential coloring as a string of space-separated pairs of one state letter and one color letter (e.g., "Cr Mb Vr Hg Yy").

You will need to do the following:

a) Develop a way of testing to see if a possible coloring really is legal.

i. Write a test suite and/or specification for an `is_a_legal_coloring` function to solve this problem.

ii. Choose a design strategy and outline a design for `is_a_legal_coloring`. You may want to start with basic recursive design, or by trying to make a mathematical definition of what counts as a legal coloring and then applying the mathematical derivation approach of Section 4.1. In either case, if your design strategy produces other problems, write a test suite and/or specification and design an algorithm for each of them too (as we did for `friend_of_everyone` in Section 4.6.1).

iii. Translate your design into a Python function named `is_a_legal_coloring`, which should have two string parameters (for the coloring and the list of bordering states). Include a description of your design as a comment before the function. Also write functions for any sub-problems you identified. You may make use of functions from this chapter, either verbatim or after slight modification.

iv. Test your `is_a_legal_coloring` function and debug as necessary.
b) Develop a way of generating all possible colorings, and collecting those that are legal (in your answer, separate the colorings with commas, since each coloring will include spaces). If you find this difficult, try doing it once in a way that works only for the three colors "r", "g", and "b", and then consider ways of generalizing your algorithm to any set of colors.

i. Write a test suite and/or specification for this problem.

ii. Choose a design strategy and outline a design. You may find that the discussion of the limits of basic recursive design, at the start of Section 4.6.2, is relevant here.

iii. Translate your design into a Python function, complete with appropriate comments. You may give this function any name and parameters you like, but make sure to provide a function named `collect_legal_colorings`, with only three parameters (for the set of states, the set of colors, and the border information) that can be called to produce the complete list of colorings (just as our example has a function `generate_groups_collect_subccliques` that embodies our definitions and a separate function `collect_all_subccliques_of_size_at_least` that can be called to produce the final answer).

iv. Test your function and debug as necessary.
Chapter 5
Correctness of Algorithms

There are two main approaches to determining whether or not a proposed algorithm is correct. **Formal verification** involves the use of mathematical techniques to reason about all possible executions of an algorithm, whereas **testing** involves the execution of the algorithm on each of a carefully chosen set of examples (known as test cases). Techniques developed for each of these approaches can be used in a variety of circumstances: To check an existing algorithm, to guide the development of a new algorithm, to facilitate the combination of algorithms written by different members of a large software project, or to communicate about how an algorithm works. This chapter discusses the application of mathematical “direct proof” techniques to computer algorithms written with the “starting set” of Python features given in Chapter A and the development of test cases for a given problem or algorithm — more information about “direct proofs” can be found in Appendix B or most books on discrete mathematics; techniques for actually running the program on each test case are left for the lab section.

5.1 Defining Correctness

An algorithm is correct if and only if it produces the correct answer to any problem instance that meets its precondition. Thus, we cannot discuss the correctness of an algorithm without having both an understanding of the algorithm’s precondition and a way of knowing whether or not a given answer is correct (i.e., the postcondition). The precondition and postcondition can be seen as a contract between the calling and called function — if the caller (perhaps a user interface function) ensures that the parameter values obey the precondition (e.g., the user interface might ensure that the user provides a positive integer exponent for the `power` function), then the called function must produce a result that obeys the postcondition (e.g., return the value `base^{exp}`). If the caller does not obey the precondition, then it cannot count on the called function in any way — it could produce a result that is reasonable or unreasonable, or it could halt the program (possibly after warning the user), or just run forever without returning anything — there are no guarantees if the precondition is violated. Thus, the precondition and postcondition must be used together to determine whether an algorithm is correct. These two conditions are referred to as the **functional specification**, or simply **specification**, of the algorithm.

In some cases, it is possible to provide a mathematically precise formal specification, and perhaps even check it with Python statements like our `precondition` and `postcondition` functions. In other cases, a programmer may wish to provide a comment in the program describing restrictions on parameters and expectations of the result of the function — such comments can be quite effective, especially if they include examples of both the “usual” uses of a function and strange cases such involving things like negative numbers or empty strings. Sometimes a very clear function name can communicate much of this information, though can be difficult to create a concise name that captures both the essential meaning and the response to strange cases.

77
5.2 Formal Verification

An algorithm that can be expressed in the terms we have used so far (i.e., the “starting set” of Python features covered in Appendix A, in which we combine mathematical calculations and/or references to other algorithms (or itself), possibly in several cases) is correct if and only if these four correctness conditions hold for it and the other algorithms it calls upon:

1. For any use of a function (or operator symbol), the arguments will obey the called function’s precondition (including argument types).

2. The value computed for each return statement fits the algorithm’s postcondition.

3. If the algorithm is recursive (i.e., it refers to itself or to another algorithm that refers back to it), then there must be some form of progress that ensures the algorithm will eventually reach a case in which it does not refer to itself. This is typically proved by finding some non-negative integer progress expression (such as \( \text{exp} \) in our \( \text{power} \) function) that serves as an upper bound on the remaining “depth” of the recursion (i.e., the number of additional calls to the function that could be simultaneously active), and showing
   
   i. that if the progress expression is \( \leq 0 \) the function cannot call itself (or call another function that in turn calls it), and
   
   ii. that, at every point at which the function does call itself (or another function that in turn calls it), the progress expression will be smaller (by at least 1) when evaluated with parameters given in the call.

4. The program will not exhaust the resources of the computer or language system software, for example it will not exceed any system-defined limit on the number of recursive function calls, or on the number of digits needed to represent a number (we will generally ignore this requirement in this course, but it is included here for completeness).

Each applicable correctness condition can be addressed via a series of substitutions (as in Section 3.4) using the rules of Appendix A (and B, when transforming expressions involving mathematics such as +, <=, and, etc.), as in Figures 3.5 and 3.6. To address Correctness Condition 1 we perform substitutions until the precondition statement of each called function becomes \( \text{precondition}(\text{True}) \); for Correctness Condition 2 we perform substitutions until each postcondition statement in the function we’re verifying becomes \( \text{postcondition}(\text{True}) \); for Correctness Condition 3.i we assume the progress expression is negative and substitute until there are no more recursive calls; for Correctness Condition 3.ii we substitute until we produce an inequality of the form \( p_{\text{called}} < p_{\text{calling}} \), where \( p_{\text{called}} \) is the progress expression with the values being passed to the function being called and \( p_{\text{calling}} \) is the progress expression with the values from the function making the call; techniques for Correctness Condition 4 are beyond the scope of this course, and we will omit it from our proofs.

As a simple example of a formal verification, consider the function \( \text{square} \) in Figure 5.1. This function makes use of our \( \text{power} \) function from Figure 3.2 to compute \( x^2 \) for any number \( x \) (as noted in the comment, this is perhaps not the best way to write this function, but the point here is to provide something to illustrate several aspects of formal verification). One formal verification for
this function is shown in Figure 5.2 though after “warming up” with this version, we will move to a more general and realistic version.

Note that Figure 5.2 makes use of our work from Figure 3.6 to rewrite \texttt{power(x, 2)}. If we had not done (or had forgotten) this earlier transformation of the body of the \texttt{power} function, we could of course have used those steps directly in Figure 5.2 (but the resulting larger sequence would not have fit nicely on one page). Also noteworthy in Figure 5.2 is our brief mention of Correctness Condition 3 — when a function has no recursive calls, there is nothing to do for this step, but it is worth mentioning this step to show we have not simply forgotten it.

Figure 5.2 includes a mix of steps that rewrite lines from the \texttt{square} function and steps that rewrite our earlier \texttt{power} function. While this approach is valid and works well for a simple example, it has several drawbacks when it comes to larger software projects. If a function (such as \texttt{power}) is called in several places, we may need to repeat substitution steps involving that function’s body. If a function changes (perhaps the \texttt{power} function from Figure 3.2 is replaced with that from Figure 4.10), verifications involving that function must be re-done. If a problem is found during an unsuccessful verification effort, this approach may not yield insight into which function (or programmer) is to blame. These drawbacks all stem from the fact that mixing steps from different functions “breaks the function abstraction”: when using a function, one should think about \textit{what} that function is supposed to produce, not how it goes about producing it. When we write \texttt{power(x, 2)}, we should only need to think about the fact that this will produce \(x^2\), not about the details of the algorithm used. In other words, we should think about pre- and post-conditions, not about algorithms.

The problems with the verification technique described above all vanish if we organize our verification efforts around each function in isolation. Each time we write or edit a function, we verify that function. This proves that any call with parameters that obey the precondition must produce a result that obeys the postcondition. Once we know that, we can rely on the postcondition of any function we call if we show we’ve called it with acceptable parameters. For the verification of our

```python
""
"square" function --- square(x) returns x squared, for any number x
There are several ways this could be done. A simple
    return x*x
would be more appropriate, except the point here is to use 'power'.

>>> square(7)
49
>>> square(-0.5)
0.25
""
from logic import *
from power1 import power

def square(x):
    precondition(is_number(x))
    result = power(x, 2)
    postcondition(result == x*x)
    return result

Figure 5.1. A Somewhat Over-complicated Function for Producing \(x^2\)
**Correctness Condition 1** (precondition holds for the one function call (to `power`).
```
 power(x, 2)
↓ substitute body of function, as in Appendix A.4.3, with ... for stuff after precondition
precondition(is_number(x) and is_integer(2) and 2>0)
...  
↓ arithmetic ([A.4.4])
precondition(is_number(x) and True and True)
...  
↓ is_number(x) is given in the precondition of square
precondition(True and True and True)
...  
↓ boolean arithmetic ([A.4.5], [B.2.12])
precondition(True)
...  
Q.E.D.
```

**Correctness Condition 2** (postcondition holds for `return`)
```
 postcondition(result == x*x)
↓ substitute definition of result (Appendix A.4.9)
postcondition(power(x, 2) == x*x)
↓ substitute body of power as above “Then, consider power(3.0, 2)” in Fig. 3.6 ([A.4.3])
postcondition(∥ if (2==1):
     return x
 else:
     if (2==2):
         return x*x
 else:
     bttemo2 = power(x,2-2)
     return x*x*bttemo2 ∥ == x*x
↓ arithmetic for 2==1 and 2==2 (A.4.5); if True and if False rules (A.4.8)
postcondition(∥ return x*x ∥ == x*x)
↓ return rule (Appendix A.4.1)
postcondition(x*x == x*x)
↓ arithmetic (x = x rule, [B.3.15])
postcondition(True)
Q.E.D.
```

**Correctness Condition 3** (every recursive call makes progress)

*This function has no recursive calls, so this step is not required (“vacuously true”)*

Figure 5.2. Formal Verification of the `square` Function of Figure 5.1.
function `square`, once we show that `square` calls `power` in accordance with `power`'s precondition (i.e., Correctness Condition 1), we can treat `power`'s postcondition as another axiom, i.e., \( \text{power}(x, y) \equiv x^y \).

This one-function-at-a-time approach to verification ensures that, if we find a problem during verification, we immediately know which function must be fixed. It also ensures that our verification of `square` will not involve work on the body of `power`, and that changes to `power` won’t require re-verification of `square` (unless the pre- or post-condition of `power` has changed). This produces the much simpler verification shown in Figure 5.3 (Correctness Conditions 1 and 3 are unchanged from Figure 5.2), though of course this is only valid if we (or someone) also provides a formal verification of the `power` function (see Section 5.2.1).

The remainder of this section provides additional examples of formal verification (Sections 5.2.1 and 5.2.2) and further elaboration of this topic (Sections 5.2.2 and 5.2.3). Note that verification is sometimes referred to as creating a “correctness proof” of software, though this term can suggest more confidence than is really appropriate: although we make use of mathematical proof techniques, we must be careful in drawing conclusions about the actual execution of a Python program, which relies on not only our algorithm, but also on the correctness of the Python language system and the computer itself.

In the context of formal verification, we often apply substitution rules to a term that is inside (and thus controlled by) an `if` statement. In such cases, we will adopt two shorthand rules that are consequences of our main substitution rule for `if`: for any statement that will be executed if the condition is `True`, we can treat the condition tested in the `if` as an axiom; within the “else” part of an `if-else` structure, we can treat the negation of the condition tested in the `if` as an axiom. This will be important in our verification of the `power` function, which will also illustrate Correctness Condition 3.

Preconditions, postconditions, and progress may be expressed in many ways: they may be informal text descriptions, mathematically precise statements that cannot be easily checked (e.g., the postcondition for our `window_overlap` function that states that it returns true if and only

---

**Correctness Condition 2 (postcondition holds for return)**

\[
\text{postcondition}(\text{result} == x \cdot x) \\
\downarrow \text{substitute definition of result (Appendix A.4.9)} \\
\text{postcondition}(\text{power}(x, 2) == x \cdot x) \\
\downarrow \text{substitute postcondition of power} \\
\text{postcondition}(x^2 == x \cdot x) \\
\downarrow \text{arithmetic for } x^y \equiv x \cdot x^{y-1} \ (B.3.14) \\
\text{postcondition}(x \cdot x^1 == x \cdot x) \\
\downarrow \text{arithmetic for } x^1 \equiv x \ (B.3.13) \\
\text{postcondition}(x \cdot x == x \cdot x) \\
\downarrow \text{arithmetic (x = x rule, B.3.15)} \\
\text{postcondition(True)} \\
\text{Q.E.D.}
\]

*Figure 5.3. Correctness Condition 2 of the square Function of Figure 5.1 Assuming power is Correct*
if “there exists (x,y) such that (x,y) is in both windows” without actually finding x and y, or they may be simple Python expressions (e.g., exp>0 as the precondition for our power function). Conditions that are simple Python expressions can be written with our precondition, postcondition, and progress functions (see Section A.4.10). More complicated conditions are typically given as comments, though in some cases a programmer may choose to write a separate function whose only purpose is to check a precondition or postcondition, or to measure progress.

We will not emphasize Correctness Condition 4 in this course, but limitations in representation of abstract values can affect the correctness of an algorithm. These limitations typically stem from the techniques used to represent numbers in computer hardware, and depend on the type of value involved and the programming language and computer hardware being used. Languages designed for high-speed computation, such as C++ and FORTRAN, choose use representations that can be manipulated very quickly by the hardware, despite the limitations of such approaches; languages such as Python and Scheme represent some types (such as Integers) in a less limited way, but other types (such as Real Numbers) as C++ and FORTRAN would. Limitations on Integers typically involve upper and lower limits that have to do with powers of 2, for example allowing the range -2147483648...2147483647 (2^31...2^31-1), in which case we might try to calculate 2147483643+9 and get -2147483644. Limitations on Real Numbers typically involve upper and lower limits and limits on precision. These limits can affect our ability to reason about a program. For example, the rules of mathematics tell us that the expression \((x \neq 0) \land (1 + x = 1)\) cannot be true (there is no way for \(x\) to be different from 0 and still have \(1 + x\) be 1). However, in almost all programming languages, when \(x\) is a very small real number, such as \(10^{-43}\) (i.e., \(x = 1e-43\)), the expression \((x != 0)\) and \((1.0 + x == 1.0)\) is true: Typical computer hardware does not record real numbers with enough precision to represent the real number \(1 + 10^{-43}\). This issue is discussed in Section A.4.2 and Chapter 9 but elsewhere in this course we will generally assume that the values in our programs correspond to the actual mathematical quantities.

Check Your Understanding 5.1. Verify only Correctness Condition 1 for the function below:

```python
from logic import *
from power1 import power

def strange(i):
    precondition(is_integer(i))
    if i > 0:
        return power(i, i)+i
    elif i < 0:
        return power(i, -i)+i
    else:
        return 0
```

Check Your Understanding 5.2. Verify only Correctness Condition 2 for the function below:

```python
from logic import *
from power1 import power

def square_difficult(x):
    precondition(is_number(x))
    if (x != 0):
        x_squared = power(x, 4)/x/x
        postcondition(x_squared == x*x)
        return x_squared
    else:
        x_squared = 0
        postcondition(x_squared == x*x)
        return x_squared
```

Copyright 2003-2015 David G. Wonnacott; see Page 7 for terms of free use
Check Your Understanding 5.3. Explain why the if is needed in Check Your Understanding 5.2 both informally and in terms of formal verification. Specifically, if the function had contained only the precondition and the three lines under the if, but neither the if itself nor anything at or below the else, what would be wrong with the function, and what step in our proof would uncover the problem?

5.2.1 Verification of our Exponentiation Algorithms

As our power function from Figure 3.2 is recursive, we must include progress in our correctness proof. Figure 5.4 repeats this function and adds a call to our progress function with exp-1 as our progress expression (this value gives a limit on the amount of recursion that could occur, which is just what we need for a progress expression). In our proof, we will need to distinguish the values of the variables in a recursive call from the values where the call is made. We will do this with the same notation used in Section 3.4, using exp1 where the call is made and exp2 for the values in the

---

Raise one number (the base) to a power (the exponent).
The base must be given as the first parameter, and the exponent second.

PRECONDITION: To find power(base, exponent), our algorithm will require that base be a number, and exponent be a positive integer.

POSTCONDITION: power(base, exponent) is base to the exponent power.
In other words, it obeys the mathematical rules
power(base, exponent) == base, when exponent is 1
power(base, exponent) == base * power(base, exponent-1)

For example,
>>> power(3.0, 3)
27.0
>>> power(0.9, 5)        # doctest: +ELLIPSIS
0.59049...

from logic import *

def power(base, exp):
    precondition(is_number(base) and is_integer(exp) and exp > 0)
    progress(exp)
    #postcondition: the result is the base raised to the exp power,
    # or a close approximation for information that
    # isn’t represented exactly (such as real numbers).
    if (exp == 1):
        return base
    else:
        smaller_exp = exp-1
        assert(smaller_exp > 0) # confirm precondition for recursive call
        base_to_the_exp_minus_one = power(base, smaller_exp)
        return base * base_to_the_exp_minus_one

Figure 5.4. power Function from Figure 3.2
called function. For example, if we were considering only the call \(\text{power}(3.0, 3)\), \(\exp_1\) would be 3, and when \(\text{power}\) calls upon itself via the expression \(\text{power}(\text{base}, \exp_1 - 1)\), \(\exp_2\) would be 2. The proof is given in Figure 5.5.

Correctness Condition 1 (precondition holds for the one function call).

\begin{align*}
\exp_1 &> 0 & \text{power precondition} \\
\exp_1 &> 0 \text{ and } \exp_1 &\neq 0 & \text{condition in which \(\text{power}\) calls itself (negation of if condition)} \\
\exp_1 &> 1 & \\
\exp_1 - 1 &> 0 & \text{subtract 1 from each side} \\
\exp_2 &> 0 & \text{substitute parameter name for called function} \\
& & \text{(giving precondition for the called function)}
\end{align*}

Correctness Condition 2a (postcondition holds for \(\text{return}\) in \(\exp = 1\) case)

\begin{align*}
\text{result} &= \text{base} & \text{return expression in this case} \\
\text{result} &= \text{base}^1 & \text{mathematical definition of } x^1 \\
\text{result} &= \text{base}^{\exp} & \exp = 1 \text{ in this case} \\
& & \text{(postcondition holds in this case)}
\end{align*}

Correctness Condition 2b (postcondition holds for \(\text{return}\) in \(\exp \neq 1\) case)

\begin{align*}
\text{result} &= \text{base} \times \text{base}_\exp & \text{return expression in this case} \\
\text{result} &= \text{base} \times \text{power}(\text{base}, \exp - 1) & \text{definition of } \text{base}_\exp \\
\text{result} &= \text{base} \times \text{base}^{\exp - 1} & \text{postcondition of call to \(\text{power}\)} \\
\text{result} &= \text{base}^{\exp} & \text{(postcondition holds in this case)}
\end{align*}

Correctness Condition 3.i (\(\exp - 1 \leq 0\) causes recursion to stop)

\begin{align*}
\exp - 1 &\leq 0 & \text{assumption for C.C. 3.i} \\
\exp &\leq 1 & \text{add 1 to both sides} \\
\exp &< 1 \text{ and } \exp > 0 & \text{precondition of \(\text{power}\)} \\
\exp &= 1 & \text{the only integer that's > 0 and } \leq 1 \\
\text{power function is just } \text{return base} & & \text{substitution rule for if (True)}
\end{align*}

Correctness Condition 3.ii (\(\exp - 1\) gets smaller by at least 1 at the point of the call)

\begin{align*}
1 &\geq 1 & \text{definition of } =, \geq \\
(\exp_1 - 1) - (\exp_1 - 1) + 1 &\geq 1 & x = 0 + x, \text{ also } x - x = 0 \\
(\exp_1 - 1) - (\exp_2) + 1 &> 1 & \text{definition of } 2^{\text{nd}} \text{ parameter in call} \\
(\exp_1 - 1) - (\exp_2 - 1) &> 1 & \text{callee's progress } \geq 1 \text{ definition of progress expression}
\end{align*}

Figure 5.5. Verification of \(\text{power}\) from Figure 5.4

Although we do not focus on Correctness Condition 4, it is worth noting that \(\text{power}\) can exceed the maximum real number value allowed in a double-precision floating-point real number (as in the case of \(\text{power}(10, 100000)\)), or produce numbers that are less than the maximum but cannot be represented exactly (as in the case of \(\text{power}(1.001, 100)\)). Thus, all we have proved is that \(\text{power}\) will produce the same approximation of the correct result that would be produced by repeatedly multiplying the value the appropriate number of times.

Copyright 2003-2015 David G. Wonnacott; see Page 7 for terms of free use
The proof of the version of power from Figure 4.10 is somewhat different — note that we must make extensive use of the detailed precondition of \texttt{power\_with\_hint}; any omission in this precondition will make the proof difficult or impossible.

**** NEED TO ADD PROOF HERE FOR TAIL RECURSIVE POWER ****

\textbf{Check Your Understanding 5.4.} Give a progress expression for the function you wrote for Check Your Understanding 3.3, and prove only Correctness Condition 3 for this function.

\section*{5.2.2 Of Proofs and Proofs}
Proofs come in several levels of detail; the form a proof takes in a high-school or first-year college math course is quite different from that given in published research on mathematics. Unfortunately, each of these forms is simply called a “proof”, leading to confusion. We will use the following terms to describe the “level of detail” of a proof.

\begin{itemize}
  \item Fully detailed: for every single substitution step, give the axiom used and show the result of the substitution — if the axiom could be applied in more than one place, identify what part of the whole is being substituted how; if the axiom is only applicable in certain conditions (e.g., an axiom involving division may require a non-zero denominator), state why those conditions must hold. This is the level of detail at which proofs are usually taught, but the level of detail in the resulting proof may obscure the key insights.
  \item Semi-detailed: Discuss key insights for every required step, and list the facts that support each step. For proofs of recursive functions, this means listing each of the three proof steps, with each sub-step, and stating which facts are critical to the sub-step under discussion (e.g., does it depend on function’s precondition? or on the “if” test? or on the result of a called function? etc.).
  \item Very abstract: Just state the most significant insights.
  \item Inadequate detail: A statement of the “This is obvious” (this shouldn’t be called a proof).
\end{itemize}

For example, consider a proof of correctness of the original \texttt{power} function of Figures 3.1 and 5.2:

\begin{itemize}
  \item A “fully detailed” proof is given in Figure 5.6, though it could use a brief note in “condition 1” that the \texttt{==}, \texttt{-}, \texttt{>,} and \texttt{*} operations all work fine because their preconditions allow any numbers.
  \item For a “semi-detailed” proof, the following would be plenty of detail:
    \begin{itemize}
      \item **C.C. 1 - preconditions.** The arithmetic operations \texttt{==}, \texttt{-}, \texttt{*}, and \texttt{>} are all fine since they are applied to numeric values (note that the result of power must be a number); the precondition for the call to power, in which we require a non-negative new \texttt{exp}, must be met because the current \texttt{exp} is a non-negative integer (from the precondition) and \texttt{exp \texttt{== 0}} has been ruled out in the “if”.
      \item **C.C. 2 - postcondition.** These follow directly from the corresponding parts of the definition of exponentiation: $x^1 = x$ and $x^y = x \cdot x^{y-1}$.
      \item **C.C. 3 - progress.** When “\texttt{exp-1}” reaches zero, the function’s single “if” prevents further recursion; values less than zero are ruled out by the precondition. The value of the progress expression falls by one as the value of \texttt{exp} is reduced by one at the one-and-only recursive call.
    \end{itemize}
\end{itemize}
• Given that this function follows directly from the recursive definition, there is not much need for a very abstract proof stating the significant insights. If one were demanded, the best answer might be “The program follows directly from the mathematical definition, with progress corresponding to the expression ‘exp’ and any positive integer exponent causing only recursion that will still fit the precondition.”

• Inadequate detail: “This is trivial”.

In a professional setting, the level of detail should be selected based on the background of the audience of the proof. Mathematics research publications may contain proofs at the semi-detailed and very abstract levels, since the audience (research mathematicians) can be expected to fill in the details as necessary. When submitting work as a student, however, one should ask what level of detail is expected.

One sub-field of mathematics and computer science involves the development of algorithms to fill in the “obvious” details in a fairly abstract statement of a proof. Since this would be impossible for an incorrect attempt at a proof, such a system could free us to focus on important concepts rather than annoying details, while remaining free of the fear of introducing an error.

5.2.3 Of Programs and Proofs

While one might think that a mathematical proof would provide absolute certainty that a computer program will execute correctly, there are several reasons why this is not the case. It is possible that a correct program will be executed incorrectly due to flaws in the hardware or system software (such as the software used to implement a language such as Python, or the operating system that moves data to and from the program). It is possible that typographical errors in the program will go unnoticed as one constructs a proof. And, of course, there is always the possibility that a conceptual mistake in the program is hidden by a corresponding mistake in a proof. In the context of computer security, one must also be alert to the possibility that a program has been modified or replaced since the proof was performed.

In principle, one could identify problems from typographical errors, and even check the steps of a proof for conceptual flaws, by writing a program to verify the correctness of proofs and the correspondence of a proof to a program. If this program were also applied to the system software, and a similar check were performed on hardware, it could increase our confidence in the correctness of the system. The construction of such programs remains an area of active research.

5.3 Testing

Testing may be used in conjunction with, or instead of, formal verification. Like verification, testing may be applied to a complete system (i.e., the top-level algorithm), to the individual components (sub-algorithms), or both. Testing may be done in an ad-hoc fashion, with a programmer trying out a few cases to see if they work. Testing may also be done in a more systematic way: first a set of test cases is produced, and each is annotated with the desired answer or answers; then the system (or component) can be checked against the entire test suite each time any part of it has changed, and the test suite itself can grow as the programming continues.
Approaches to testing are generally described as either **black-box** or **glass-box** testing — in the former, the test cases are produced without reference to the algorithm being tested (it is treated as a “black box” whose internal details are not visible). The test case generator typically refers primarily to the specification, and may try to come up with problems that produce different kinds of answers. For example, we might wish to test our `power` function’s ability to produce various kinds of answers — positive answers resulting from positive values for `base`, negative numbers resulting from negative `bases` to odd `exp`’s, positive numbers resulting from negative `bases` to even `exp`’s, non-integer numbers resulting from non-integer `bases`, large numbers (though not past the maximum value for type `double`) resulting from high `exp`’s, numbers close to one resulting from numbers very close to one raised to high `exp`’s, etc. Black-box testing is often done by people who are not in communication with the programmers who developed the system being tested, so ensure that hidden assumptions made by the developers will not be shared by the testers (and thus go unnoticed during testing).

In glass-box testing, the tester looks “inside the box”, at the internal structure of the software being tested (the terms “clear-box” and “white-box” have also been used). The tester may try to produce tests that **cover** (cause the execution of) the internal structure of the software in various ways. For example, if the goal is **all-statements coverage**, the set of tests should cause each statement in each function to be executed. If the goal is **all-paths coverage**, the set of tests should cause each combination of outcomes in each `if` statement (this may require an extremely large set of tests). To produce tests that give all-statement coverage for our `power` function, we would need one test in which `exp==1` is true (e.g., `power(3.0, 1)`), and thus the statements before the `else` are executed, and one test in which `exp==1` is false (e.g., `power(3.0, 2)`), and the statements after the `else` are executed (technically, the latter test would cover all statements, since the condition is true in the recursive call).

It is often valuable to test the program with a variety of inputs, even if these do not trigger the execution of a particular statement. The variety of inputs may be chosen from the program’s specification or the implementation (in black-box or glass-box testing, respectively). For example, any software to manipulate numbers may produce interesting behavior for zero or for cases in which various parameters are equal; text-manipulation software may respond in unusual ways to empty strings, or strings of only blank spaces. A look at an algorithm’s implementation may reveal potentially dangerous operations which inspire a particular test case: software containing a division by \(2n-8\) should probably be tested for some cases in which \(n=4\) (hopefully in this case the division is only performed when \(n \neq 8\), perhaps due to a carefully constructed `if` statement; one could discover this by careful examination of the program, i.e., by doing the steps to prove the program is correct). Sometimes an algorithm will work for all but a very particular set of parameter values — one could imagine an algorithm for window manipulation that worked unless the two windows had exactly the same minimum and maximum values in the horizontal dimension, or unless they touched at exactly one point. Specific combinations of values that might cause an algorithm to act in an unusual way are often referred to as **corner cases**, whether or not they involve the corners of rectangles.

When the software being tested communicates with the user, the tester must try various incorrect inputs, as well as correct inputs, to make sure the system gives an appropriate response. For example, the software should give a clear indication of what has gone wrong if the user enters a negative or non-integer exponent in software that is about to call our `power` function. One surprisingly effective test of user interfaces involves closing one’s eyes and pushing lots of keys on the keys while moving and clicking the mouse, to see if the software crashes.

**Check Your Understanding 5.5.** Give a test suite that provides all-statements coverage for the function from Check Your Understanding 5.1.
Check Your Understanding 5.6. Give a test suite that provides all-statements coverage for the function of Check Your Understanding 5.2. Would this test suite uncover the bug in the hypothetical function from Check Your Understanding 5.3?

5.4 Other Uses of Verification and Testing Techniques

Verification and testing are often presented as ways of ensuring software correctness of existing software, but they also have uses in designing software or communicating about it.

Simultaneous development of specifications and software has been used as the foundation for several variants of top-down design: One can start with either a formal specification or a test suite for the complete software, and try to deduce a design that combines components. The specifications for the components (either formal statements or test suites) are then constructed to fit the needs of the overall design, and the process is repeated until the components are simple enough not to require sub-components. Once the top-down design is completed, the components can be written and tested in a bottom-up fashion (since one typically cannot test the high-level components until the low-level ones are functional). We will revisit one such approach in Chapter 8. The “Test-Driven Design” approach involves the simultaneous development of software and test suites, but is based on adding one feature at a time, rather than working from high-level software to low-level.

Verification techniques may be applied in a limited way to gain some confidence about a particular aspect of a system, without requiring a full proof of correctness. For example, we may choose to construct a full proof of only certain safety-critical elements of a large system. The construction of a formal specification can provide valuable insights even if the programmer never follows through with a proof. In some cases, the correspondence between the program and proof is straightforward; an uneasy feeling that one is not quite sure how one would go about proving one component often indicates a problem in the software itself.

Even when no attempt is made to check correctness, specifications can be valuable tools for a programmer (or team of programmers) creating a large program. Errors often occur not because a specific algorithm is incorrect by itself, but because a higher-level algorithm is based on a misunderstanding of a lower-level algorithm it calls upon. The higher-level algorithm may provide parameters that do not fit the precondition, or rely on some aspect of the result that is not always part of the postcondition. Actually writing out the specification can help programmers avoid making such mistakes, and automatic checking of both precondition and postcondition can help to identify the location of errors in a large program. When such mistakes are found in an existing program, the specification can help identify which component must be changed, or even assign blame for a failure (though the true fault could lie in the lack of communication). In some cases, software development tools can make use of specifications as they automatically search for potential problems in software.

Specifications are often most useful when they are simpler to understand than the underlying algorithm, for example when they correspond to some abstraction that the programmer(s) already understand. For example, it is much easier to state the specification of a square root calculation algorithm than to give the algorithm itself.

5.5 Summary

The two primary approaches to checking the correctness of an algorithm or program are formal verification and testing. Both require some definition of the context in which the algorithm is applicable (the precondition) and some definition of what counts as the correct answer (the postcondition). The precondition and postcondition are sometimes referred to as the specification of an algorithm.
Verification can be used to prove that an algorithm is correct for all possible input, but verification can be extremely difficult for large programs. Furthermore, the proof or specification may contain errors, or the proof may not correspond to the program that is actually running (for example, a proof that is written on a printed copy of a program may not detect the fact that a 1 (one) has been mis-typed as an 1 (the letter after k)).

Testing requires a set of test cases: essentially a partial specification described in terms of examples. A good test suite includes both common cases and possible corner cases; it may be designed to include a variety of kinds of input, or to exercise various elements within the program. Testing can prove that a program is incorrect, but it typically cannot be used to prove that a program is correct (unless there are only a finite number of possible inputs that can occur).

In practice, some combination of testing and verification is often used. Verification is often most difficult, but most valuable, when a programmer is having trouble explaining exactly why an algorithm should work.

Verification and testing have a variety of uses, including investigation of existing software; software design and construction; and communication, either among programmers, between programmers and their tools, or between programmers and management. Testing and verification typically fail to identify problems with a specification itself, though an attempt at verification may uncover contradictions in the specification, and an alert tester may notice that an answer looks wrong, even though it agrees with the specification.

5.6 Other Resources


Books about “Extreme Programming” discuss the use of test suites and pair programming in software design. Kent Beck’s Extreme Programming Explained: Embrace Change is a classic reference, and his Test Driven Development by example focuses on the use of test suites. The Extreme Programming approach has been criticized in several ways, for example with claims that it encourages our non-algorithm design “hacking” approach. Matt Stephens and Doug Rosenberg’s Extreme Programming Refactored: The Case Against XP gives a long, perhaps overly sarcastic, critique of this approach.

The difference between correctness and safety is discussed in detail in Nancy Leveson’s Safeware: System Safety and Computers (1995, Addison-Wesley). This book also discusses specific techniques for improving the safety of systems that depend on software, and includes detailed descriptions of safety practices and accidents in several industries.
5.7 Exercises

Exercise 5.1. Show that the point_in_window function of Figure 4.3 can be transformed into the function of Figure 4.2 and vice versa, i.e., prove that the two functions are equivalent.

Exercise 5.2. Show that any one of the point_in_window functions of Figures 4.1, 4.2, and 4.3 is correct by showing that its postcondition can be transformed to \texttt{postcondition(True)} if we assume the precondition is true.

Exercise 5.3. Prove that the tail-recursive power function from Section 4.5 gives the same answer as our original power function.

Exercise 5.4. Prove that the two Python functions you wrote to compute $n!$ in Exercise 4.6 always give the same answer.

Exercise 5.5. Prove that the two Python functions you wrote for Exercise 4.7 to find elements of the Fibonacci sequence, always give the same answer.

Exercise 5.6. Write and check the correctness of a function to return a real number solution to an equation of the form $ax^2 + bx + c = 0$ (do not try to produce answers in cases for which only complex number solutions are possible).

a) Write the body of the function itself, using techniques from earlier chapters. You may want to make use of the “quadratic formula”, $x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$.

b) Give the function’s postcondition (as a comment) and precondition (using the definition of \texttt{precondition} from our logic.h). Do not use the quadratic formula from part (a) in the postcondition. Your precondition should be as general as possible (i.e., don’t rule out any values for $a$, $b$, and $c$ for which the question can be answered). If you identify any problems with your function while working on this, fix them.

c) Give a proof that your program is correct. If you wish, you may assume the quadratic formula given above is correct. If you identify any problems with your function while working on this, fix them.

d) If you assumed the quadratic formula was correct in your previous answer, prove that it is, i.e., that the values of $x$ that it produces really do solve the equation $ax^2 + bx + c = 0$. If you identify any problems with your function while working on this, fix them.

e) Create a \texttt{main} function that uses some sample values (for which you know have checked the answer) to test your function. You should ensure that each statement in your function has been run by at least one of your tests, but you do not need to include tests that violate your function’s precondition. If you identify any problems with your function while working on this, fix them.

Exercise 5.7. (*) Prove that your modified subclique function from Exercise 4.9 produces the same answer as the algorithm in the text.

Exercise 5.8. Show that each of the first three correctness conditions given in Section 5.2 is necessary. That is, for each condition, show a function that violates this condition, but neither of the other two conditions, and is incorrect.

Exercise 5.9. (***) Either prove that the correctness conditions of Section 5.2 are sufficient for the programs that can be written with the Python features we have seen (that is, that any such program that meets these conditions must be correct), or show an example program that meets all four conditions but is incorrect.

Exercise 5.10. (***) Note that the point_in_window functions of Figure 4.2 and Figure 4.3 have the same specification, and the function bodies can be transformed into each other via our rules of substitution. Is it always the case that functions with the same specification can be transformed into each other? And, is it always the case that functions that can be transformed into each other should have the same specification?

Exercise 5.11. Some computer scientists have observed that it is possible to create correct algorithms by combining sub-algorithms that are not fully correct, by relying on only some aspects of each sub-algorithm’s result, or by using each sub-algorithm only in restricted cases where it works.

a) Give an example algorithm that violates one of the first three correctness conditions of Section 5.2 but is still correct (optional: prove that your algorithm is correct).

b) Discuss the value of testing and verification when using this approach. What if we revise each sub-algorithm’s specification as we investigate it?
Chapter 6
Computational Complexity

For an algorithm to be useful, it must not only give the right answer, but it must give it within an acceptable amount of time. An accurate simulation of a hurricane is much less useful if we need to run the program for a month to get the prediction of the next three day’s storm activity! The study of computational complexity is often seen as a way to measure the speed of an algorithm, but it is both more, and less, than this.

Computational complexity is a general approach to describing the resources required by an algorithm, as a function of some measure of the parameters given to it — we will refer to this as a complexity function for an algorithm, to distinguish it from the Python function that we use to express our algorithm. For example, our power function from Figure 3.2 needs the microprocessor to execute \( \exp(-1) \) multiplication operations. Complexity analysis is more than just a way to measure speed, in that it can also be used to describe not only resources that consume time (such as operations to be performed by the microprocessor), but also memory use, power consumption, or other things we might care about; furthermore, a complexity function describes not just resource needs, but the way these needs grow as we apply the algorithm to larger and larger problem instances.

A complexity function is also less than a measure of speed, because computing speed depends on more than just the number of operations to be performed — speed also depends on the characteristics of the hardware (such as the number of operations that can be performed per second, a measure often quoted in advertisements), the characteristics of the software delivering the program to the hardware (the programming language and operating system), and the way in which the expression of the algorithm fits with the system software and hardware.

A dramatic difference in the complexity function, such as doing \( n \) operations or \( 2^n \), typically makes any speedup from hardware or system software insignificant. Furthermore, a complexity function tells us something fundamental about the algorithm, rather than something that may change on the next computer we use. We will therefore focus on the study of algorithm complexity, giving only a brief survey of a few other factors (Section 6.5).

6.1 Describing Complexity

Computer scientists describe the computational complexity of an algorithm by giving the function relating some aspect of the problem instance (e.g., the value of a numeric parameter, or the length of a string parameter) to the quantity of some resource (e.g., executions of multiplication operations) required to solve that problem instance. In some cases, we can give a precise mathematical formula for the complexity function; in other cases, we may provide a pair of formulas proving upper and lower bounds on the true function. We usually focus primarily on the function giving an algorithm’s asymptotic complexity (i.e., the growth of its complexity as the input size grows without bound) in terms of the “worst case” (requiring maximum resources) for each size. We are satisfied if we can classify the “shape” of this function — is it linear, quadratic, exponential, etc. (this idea of the “shape” of the function is generally formalized in the 2nd semester of study of computer science).

When many resources impact on our final goal, we normally try to describe any of the resources that have the complexity function that grows most quickly with the problem. For example, to get an idea of the speed of our power function (Figures 3.2 and 5.4), we might consider measuring the
number of multiplications, the number of recursive function calls, the number of if statements executed, or the number of times we execute return base. Since the first three all grow as linear functions of exp (see below) and the last does not grow at all, we could measure any of the first three.

It is often easy to make a quick guess at the overall complexity of a function by looking at the structure of the recursive calls in a few example call trees, as in Box 4. We can confirm such a guess by applying our proof techniques of Chapter 5 to a modified version of the function. Imagine

**Box 4: How to Recognize Different Kinds of Recursion From Quite a Long Way Away. Number 1: Linear Growth.**

A guess about a function’s complexity can be proved using the techniques described in the main text. If we aren’t sure what to guess, or just need a guess and don’t need mathematical certainty about the result, we can often “eyeball” the complexity by just looking at the structure of the recursion (or, in Chapter 8, the structure of the loops).

For example, for the original power function of Figure 3.2 and the variant in Figure 4.10 in our discussion of advanced recursive design, the number of multiplication operations grows linearly with the exponent (i.e., doubling the exponent always roughly doubles the number of multiplications). Linear complexity is typical of recursive functions that call upon themselves at most once (unlike the generation functions of Section 4.6.2) and reduce the progress expression by a standard amount each time.

This pattern can be recognized in the call tree — if you draw the call tree for, say, exp(3,8), you will get a sequence of recursive calls that can be put in a line, with each level of the recursion doing the same number of multiplications (one). This call tree is reminiscent of the linear pattern of most evergreen trees, such as the Nordman Fir shown here and on the front cover: from the side, we see a main trunk that is a straight line, with layers of branches that are (about) the same size.

This description gives a good, but not perfect approximation of the structure of the fir tree or the recursive power function. A more precise analysis would take note of the fact that this tree’s branches are actually somewhat shorter near the top, or the fact that some multiplications in the power function may be harder than others: multiplying $3 \times 3^{58}$ to get the final answer of $3^{59}$ may be more difficult than multiplying $3 \times 3^{3}$ near the beginning of the work on this answer. However, accounting for this level of subtlety is often difficult, and its impact minor, so we stick with simpler measures such as the number of multiplications (or branches).
Return the number of "*" operations use to compute power(base, exp). This is just body of the original "power" function with
+ A new name
+ All calculations replaced with the number of *'s used
(The original calculations are left as comments)

Note that it always returns exp-1:

```python
>>> mults_for_power(12, 7)
6
>>> mults_for_power(3, 1)
0
>>> mults_for_power(0.99, 123)
122
```

```python
def mults_for_power(base, exp):
    if exp == 1:
        return 0  # 0 multiplications for "return base;"
    else:
        mults_for_exp_minus_one = mults_for_power(base, exp-1)
        # base_to_the_exp_minus_one = power(base, exp-1)
        return 1 + mults_for_exp_minus_one
        # return base * base_to_the_exp_minus_one;
```

Figure 6.1. power Function from Figure 3.2, Modified to Return the Number of Multiplications.

modifying a function so that, instead of returning the answer it is supposed to give, it instead returns the number of times a certain operation would have been applied to produce that answer. For example, Figure 6.1 shows how we could modify the power function to produce the number of multiplications used in the calculation of power(base, exp).

A few runs of the mults_for_power function (or a brief look at the function itself) will show that this function returns exp-1 for any parameters meeting its precondition. Thus, we hypothesize that the postcondition is mults_for_power(base, exp) === exp-1, and a direct proof by cases will confirm that this is indeed true.

Thus, to test a hypothesis about a function’s complexity, we employ the techniques that we used to confirm a function’s correctness. Advanced techniques for formulating a hypothesis about complexity are beyond the scope of this course; for now, we rely on the informal techniques of Boxes 4 and 5.

Our choice of what to measure, and how to measure it, can affect the complexity function we find. We may need to select among various resources (such as executions of multiply operation), and also among various ways of measuring the "problem size". For example, we chose to express the complexity of our power functions in terms of the value of exp, but ignored base. If we desired a measure of the total number of digits to be multiplied (rather than the number of multiplications), we would have had to incorporate both exp and base in our function. Or, we might desire to express the number of multiplications as a function of the number of digits needed to express exp — in this
case we would say the number of multiplications is approximately \(10^{\text{number-of-digits-in-exp}}\) (since
the value of \(\text{exp}\) grows exponentially with the number of digits).

Sometimes the choice of the most appropriate measure can be obscured by the use of other
functions in an algorithm. For example, consider the \texttt{is_valid_name_list} function of Figure 4.13
(repeated for reference in Figure 6.2). A few call trees, for example for the strings "YH", "VYYH", and

```python
"VVYH",
```
suggest this a “pine tree” pattern (Box 4), with a use of \texttt{is_valid_name} at each level. Recall
that we usually measure the complexity of the worst case — here that means we ignore any “easy
answer” that might arise when we can return true at the first call, as for "VVYH", and focus on valid
name lists, for which the number of recursive calls equals the length of the list of letters. We might
then be tempted to conclude that the complexity function is best described “the number of calls
to \texttt{is_valid_name} is no more than \texttt{len(list)}”. While this statement is true, a good complexity
function should take into account the work done inside all functions (or operations) used.

To account for all the work done in a call to \texttt{is_valid_name_list}, which could call itself
\texttt{len(list)} times, we must include the work done for:

- 2*\texttt{len(list)} function calls (to \texttt{is_valid_name_list} itself and to \texttt{is_valid_name}),
- 2*\texttt{len(list)} uses of == and in (one of each in \texttt{is_valid_name} and \texttt{is_valid_name_list}),
- \texttt{len(list)} definitions of the variable \texttt{names} and the parameter \texttt{string},
- \texttt{len(list)} uses of \texttt{len}, \texttt{is_string}, and the not operation,
- \texttt{len(list)} times through the if/elif/else sequence, and
- 3*\texttt{len(list)} used of the and operation.

Since 3*\texttt{len(list)}, 3*\texttt{len(list)}, and \texttt{len(list)} all grow linearly with \texttt{len(list)}, we might
expect the time needed to run this algorithm would also grow at worst linearly with \texttt{len(list)}.
However, we would be wrong, since some basic Python operations may themselves do more work
for some operands than others.
The actual complexity functions vary with the algorithms (and data structures) used in the Python language system, and these choices are often not well documented. For now, we will simply guess that string comparison and \texttt{in} operations are based on character-by-character comparisons, and that the \texttt{len} function takes a small fixed amount of time to look up the length of any string in the computer's memory (length information can be kept in memory along with the text of the string itself). Thus, the time needed to compare strings grows with the length of the shorter string, the time needed to perform an \texttt{in} operation grows with the length of the string being searched (if this is not the longer string, the result is clearly false), and the time needed for a \texttt{len} operation is constant. Based on these guesses, we can find the number of character comparisons inside the \texttt{==} and \texttt{in} operations (for non-empty strings, these character comparisons will be more numerous than the number of uses of \texttt{==} and \texttt{in}, and thus grow at least as quickly as anything else in the program).

The \texttt{is_valid_name} function uses \texttt{in}, but note that the second parameter is always a 26-letter string. Thus, the number of character comparisons for any one call to \texttt{is_valid_name} is bounded by 26, and the total done in \texttt{is_valid_name} for any call to \texttt{is_valid_name_list} is at most $26 \times \text{len(list)}$ — based on this, our total time might still be bounded by a linear function! However, \texttt{is_valid_name_list} also calls upon \texttt{in}, and it checks for one letter in the rest of the list! Thus, the number of character comparisons at any one level of the recursion could be as high as \texttt{len(list)}-1 — if we drew a call tree in which each branch from the main “trunk” of \texttt{is_valid_name_list} calls is as long as the remaining list (i.e., it equals the number of character comparisons), we would end up with a “pine tree” shape in which each branch is as long as the remaining height of the tree: an equilateral triangle! Thus, the total number of comparison operations (i.e., the total area of the tree we drew) grows with $\frac{1}{2} \times \text{len(list)}^2$ (recall that the formula for the area of a triangle with base length $b$ and height $h$ is $\frac{1}{2} \times b \times h$; the growth rate can also be proved by mathematical induction, or through the approach we showed with \texttt{mults_for_power}).

The choice of what to measure in terms of what must be made in a way that is appropriate for the task at hand, and the description must be communicated clearly alongside any result. In some cases (especially advanced theoretical work) this is clear from context, and many computer scientists use shorthand descriptions such as “this is a linear algorithm” or “this algorithm takes order $n$ time”, even if there is no $n$ in the problem (typically $n$ is used as a shorthand for the total number of binary digits needed to express the entire input; this is particularly confusing when there is a parameter $n$).

These shorthand descriptions of complexity functions are not acceptable in the context of this course. A “linear time” algorithm and an “exponential” algorithm sound dramatically different, but as we saw above, our original power function can be described by either of these complexity functions, depending on our choices of what to measure!

### 6.2 Improving Complexity

Figure 6.3 shows a \texttt{power} function that does far fewer multiplications than our previous \texttt{power} functions (technically it does far fewer multiplications only when \texttt{exp} is large — when \texttt{exp} is 3, we can’t very well do far fewer than the two multiplications of our other \texttt{power} functions). The new algorithm is based on an additional fact about exponentiation that will let us reduce the exponent by more than one with each recursive call. It clearly does no more multiplications than the original, since each “level” of the recursion calls itself at most once and always reduces \texttt{exp} by at least 1. The fact that \texttt{exp} is reduced by larger amounts when it is large is a hint that we may have changed the shape of the complexity function, and we will confirm this hint below. However, it must do other work before it gets to the recursion, calling upon the \texttt{even} function, which must perform a remainder operation (using Python’s \texttt{\%} operator), so it is not obvious that this function will be faster, and a detailed analysis and/or experimental work is called for.
Compute base to the exp power, for integer exp>0, using a the rules

\[ \text{base}^1 = \text{base} \]
\[ \text{base}^\text{exp} = \text{base} \times \text{base}^{(\text{exp}-1)} \]
\[ \text{base}^\text{exp} = \text{base}^{(\text{exp}/2)} \times \text{base}^{(\text{exp}/2)} \]

We use the last rule only when it yields an exponentiation that

can be handled by the function we are writing (i.e., when exp is even,
so that exp/2 is a integer, suitable for our second parameter).

Our "even" function checks if exp/2 is an integer, in other words
if exp is evenly divisible by two, i.e., the remainder of exp/2 is 0.
Python uses the % sign to find remainders, i.e. exp%2.

```python
>>> even(17)
False
>>> even(42)
True

>>> power(3.0, 3)
27.0
>>> power(3.0, 5)
243.0
>>> power(0.99, 123) # doctest: +ELLIPSIS
0.290488...
```

```python
from logic import *

def even(i):
    precondition(is_integer(i))
    return (i%2 == 0) # true if and only if i%2 is 0

def power(base, exp):
    precondition(is_number(base) and is_integer(exp) and exp > 0)
    progress(exp)

    if (exp == 1):
        return base
    elif (even(exp)):
        smaller_exp = exp//2
        assert(exp - smaller_exp >= 1) # reduce exp by at least one
        assert(smaller_exp > 0) # confirm precondition for recursive call
        base_to_the_half_exp = power(base, smaller_exp)
        return base_to_the_half_exp * base_to_the_half_exp
    else:
        smaller_exp = exp-1
        assert(smaller_exp > 0) # confirm precondition for recursive call
        base_to_the_exp_minus_one = power(base, smaller_exp)
        return base * base_to_the_exp_minus_one
```

Figure 6.3. Alternate power Function with Lower Complexity Function

Copyright 2003-2015 David G. Wonnacott; see Page 7 for terms of free use
Before investigating its complexity in detail (Section 6.3), we'll confirm that the algorithm is correct — our algorithm results from a “mathematical derivation” design based on three facts about exponentiation: the two we used in Figure 3.2 and the fact that $b^e = b^{e/2} \times b^{e/2}$. This is a consequence of the general rule $b^x+y = b^x \times b^y$ (just substitute $e/2$ for both $x$ and $y$), which can be found in algebra texts or Appendix B, or proven (for positive integer $x$ and $y$) from our original pair of rules and the technique of mathematical induction, which is beyond the scope of this course (but closely related to our techniques for proving the correctness of recursive functions). As is usual for algorithms resulting from mathematical derivation technique, Correctness Condition 2 (the postcondition holds for each return statement) can easily be proven from the set of mathematical facts (that $b^1 = b$, that $b^e = b \times b^e$, and that $b^e = b^{e/2} \times b^{e/2}$).

Correctness Condition 1 (the precondition of every called function or operation must hold) must be confirmed for the mathematical operations ($=, >=, >, -, \text{ and } \ast$), the call to even, and the recursive call to power. The operators all work with numeric values (according to the precondition of power), which is all they require. We pass exp (a integer) to even, satisfying its precondition. In the final else clause, we know exp-1 > 0 from power’s precondition (that exp is a positive integer) and the fact that we’re looking at the else clause associated with the test “if exp == 1”; since base is unchanged and exp-1 is a positive integer, our call to power here must satisfy power’s precondition. The case of the even exponent is the same, except that we rely on the fact that $e/2$ is a positive integer when $e$ is an even integer larger than 1.

Correctness Condition 3 (the function makes progress as it calls itself recursively)

To complete the proof that power works, we must also prove that even is correct: Condition 1 must be proved for the use of integer division operation (//) and for ==. Division requires numbers and a non-zero denominator — conveniently, exp is an integer and the denominator is 2; the == operator can compare the resulting integer to the integer 0. Condition 2 is trivial if we take “has no remainder when divided by 2” as the definition of even. Condition 3 is not needed since there is no recursion.

Now that we know the function works, we return to our discussion of complexity.

6.3 Giving Upper and Lower Bounds on Complexity

Formulating a hypothesis about the complexity of the revised power function of Figure 6.2 is more difficult than our previous examples, as the parameter that determines the amount of work changes in different ways in the two possible recursive calls. We start by looking at each of these by itself. In the first recursive call, the function reduces the exponent by a factor of two. In this case, the function will need $n$ calls to reduce the exponent from $2^n$ to 1. Since the log$_2(n)$ function is defined as the inverse of $2^n$, this means the function would need log$_2$ (exp) calls to reduce the exp to 1 if it called itself only at this point (as it would when exp is a power of two).

In the second recursive call, exp is reduced only by one. By itself, this kind of recursion would require about exp recursive calls to get down to the base case. However, when exp is odd, exp-1 must be even, so next recursive call must involve an even exponent, and cannot end up in the second clause of the if. In other words, in the worst case, two recursive calls have reduced the exponent by a bit more than a factor of two. If we needed two steps to reduce exp by exactly a factor of two, we would have 2log$_2$(exp) recursive calls to reduce exp all the way to 1. Since each recursive call also involves one multiplication, the number of multiplications needed to find power(base, exp) should be between log$_2$(exp) and 2log$_2$(exp).
Since these functions have the same “shape”, we say we have bounded the complexity — this notion will be formalized in a later course — sometimes all we need is an upper bound, if we simply want to show that a proposed algorithm will be faster than an algorithm for which we have a lower bound.

Note that the revised power function also uses one modulo operation (for \( \exp \% 2 \)) and possibly one integer division operation (for \( \exp / 2 \)) for each call, so we can’t assume that the relative speeds of the two functions will agree exactly with \( \frac{\exp}{\log_2(\exp)} \), since the numerator and denominator have different units (namely, multiplications vs. multiplications plus mod operations plus divisions). However, these units differ by a constant factor, so for a sufficiently large \( \exp \), the revised function should be faster, and as \( \exp \) increases past this critical value, the speed advantage of the revised function will only grow.

### 6.4 Functions With Higher Complexity

The call trees for some functions we have seen look quite different from the “pine tree” structure of Box 4. For example, call trees for the subclique generation function of Chapter 4.6, and the strange function of Figure 3.4, each feature branches that sprout smaller branches that themselves could sprout still smaller branches, etc., looking much more like the branches of an oak than a member of the pine family. This sort of recursive structure typically indicates a program with an exponential complexity function.

**Box 5: How to Recognize Different Kinds of Recursion From Quite a Long Way Away.** Number 2: Exponential Growth.

Some algorithms, like some trees, branch more and more as they grow.

For example, our subset and subclique generation algorithms (Figures 4.14 and 4.15) branch into two recursive calls at each level. This suggests, not a pine tree, but this Willow Oak.

When an algorithm exhibits this branching structure, and the parameter that controls size (e.g., the size of the set) is reduced by a constant amount (e.g., 1), we typically see a complexity function that grows exponentially with the size parameter, for example \( 2^n \) where \( n \) is the number of elements in the set. (For the oak, the widths of the branches decrease more quickly, and the tree does not require an exponential amount of wood or growing space).

A quick list of the powers of two,
or some experiments with runs of the subset generation function, will illustrate the problems with using an exponential algorithm for a problem of any significant size.

6.5 Complexity, Models of Execution, and Program Speed

While the complexity function for an algorithm is often the most significant factor controlling program speed, sometimes it can be affected by the language system. Furthermore, knowledge of the language system and hardware can be used to further speed up the program once a fast algorithm has been selected. This is a tricky art, however, since the actual steps taken vary with the language system and hardware design. Here, we survey a few interesting factors.

6.5.1 Redundant Function Calls and Speed

So far, we have described computational complexity as if it were a fundamental property of an algorithm or Python function, claiming that the original \texttt{power} function performs about \texttt{exp} multiplications, and the modified version of Figure 6.3 performs about \texttt{log}$_2$\texttt{(exp)}. This is something of an oversimplification: the complexity depends both on the algorithm and the manner in which that algorithm is executed on the computer.

We have discussed execution of programs as a process of mathematical substitution, and as a process of step-by-step following of instructions. From the point of view of program correctness, these were two equally good ways to look at program execution. However, these models of execution have very different implications when it comes to complexity.

For example, our substitution rules allow the substitution of a variable’s value for any use of the variable. Thus, we could substitute the text \texttt{power(base, \texttt{exp}/2)} for each of the two uses of the variable \texttt{base_to_the_half_exp} in Figure 6.3 (and then remove the now-unnecessary definition) without changing the result of the function. However, the Python language system will typically execute the new program by actually calling the \texttt{power} function twice in this clause of the \texttt{if}, resulting in a change to the basic shape of the call tree (it can now branch at every level!). This can change the “shape” of the complexity function, i.e., make a dramatic impact on the program speed when \texttt{exp} is large.

The choice of when to execute in terms of mathematical substitution, and when to perform step-by-step following of instructions, is a key part of the art of implementing a programming language such as Python. One might hope that the language system would automatically pick the fastest way to execute a program, avoiding the transformation suggested above, or even doing the reverse if presented with a program containing two calls to \texttt{power(base, \texttt{exp}/2)}. Unfortunately, this choice interacts with many other important design goals, and one cannot always rely on the language system to make this sort of choice (though certain choices are made quite well in certain implementations of certain languages).

Techniques for avoiding redundant calls to functions include simply looking for redundancy (e.g., two calls to \texttt{power(base, \texttt{exp}/2)}) and rewriting functions to reuse previously-computed values. A full investigation of the latter approach is beyond the scope of this course, but Exercise 6.2 is essentially a simple version of the technique known as “dynamic programming” (this and “memoization” are methodical ways of dealing with redundancy across different calls to a function; they can be applied manually, or (in some cases) automatically by a language system).
6.5.2 Tail Recursion and Speed
A recursive function typically requires an amount of memory that grows with the “depth” of the recursion (the number of not-yet-completed calls at any one time). A clever implementation of a language can re-use the memory space for a tail-recursive function, so the memory use never gets more than what’s needed for a single call. For example, the “G++” compiler for the C++ language, and any legal implementation of the language Scheme, can do this. However, there are several aspects of the Python language that interfere with this sort of “optimization”, and it will probably be considered impossible in Python until someone actually does it. Thus, there is little benefit to using the the tail-recursive style in Python unless it improves the complexity function. Python, like other languages, does provide loops (see Chapter 8) as a concise clear notation for certain types of tail recursion. When this notation is used, Python implementations can easily provide the proper optimal memory use.

6.5.3 Operation Selection and Speed
Computers (like people) typically perform addition more quickly and easily than multiplication, especially for large numbers. Sometimes a programmer or language system can take advantage of this fact to speed up a program — if we have a recursive function with a parameters x and y, in which x increases by 1 in each recursive call, and in which we compute x*y, we could avoid the multiplication by introducing another parameter (perhaps called x_times_y), and simply add y to this parameter at each call.

However, the benefit of this approach depends on the relative speed of addition and multiplication, and it may make the program harder to read, preventing people or language systems from performing other optimizations.

6.5.4 Memory Systems and Speed
Sometimes the order of access to a computer’s memory can significantly affect the speed; this can be important when searching through a string or other value comprised of sub-elements (e.g., a vector or matrix, which will be introduced later). The significance of this effect, and sometimes even the choice of the best memory access pattern, vary with the design of the computer’s memory system.

6.5.5 Concurrency, Speed, and Heat
The claim that more operations mean more time consumed rests upon the assumption that operations are performed one after the other rather than simultaneously. In some cases an algorithm can make use of multiple “cores” on a multi-core microprocessor, or of multiple microprocessors on a network of computers, to perform many operations simultaneously. In such cases, the critical factor in determining program speed may not be the total number of operations, but the number of operations that must be performed in order. In terms of the classic “computing as baking” metaphor, we might hope to make a cake faster if one person mixes batter while another prepares baking pans, but we can’t expect one person to put frosting on the cake while the batter is being mixed!

Estimating or improving algorithm speed in the presence of multiple processors or processor “cores” is difficult — it remains an area of active research in both theoretical and applied computer science. The significance of various effects, and even the structure of the best algorithm, can vary with the details of the computer design. Furthermore, a program that runs very quickly on a very large number of processors may be impractical for reasons other than time: it may not be possible to purchase the required number of processors, or the processors may require too much electricity, or produce too much waste heat, for the system to be practical.
6.5.6 How to Make Programs Fast

We recommend thinking about the complexity function of each algorithm involved in the program, and writing the program clearly — if it is not fast enough, find the parts of the program that contribute most to the slowness, and figure out how to improve their interaction with the particular language system and computer you are using, or switch to a different language system or computer. If you know in advance that speed will be important, consider choosing a programming language that is known for good automatic optimizers, such as C, C++ or Fortran, so that you won’t have to re-write things later. Such optimizers may be able to account for many of the system-specific speed effects, such as choices of when to use multiple processor cores, choices of memory access patterns, or choices among different operations that produce the same effect. If you like Python, you may be able to write only the critical parts in a different language, and couple the pieces together via one of the Python/C hybrid systems. Or, better yet, you may be able to find a library of functions written for Python with such a system.

6.6 Summary

We usually use the term **computational complexity** to refer to the way an algorithm’s resource needs (typically for computation time) change with some property of its parameters. However, the resource needs actually arise from an interplay between the algorithm and the underlying software and hardware that implement the language in which the algorithm is expressed. In practice, we make certain assumptions about the language system, for example in Python we expect that an expression that uses a function name twice will result in two calls to that function.

The most significant features of a complexity function are its **overall shape**, the **property of the parameter** we care about (e.g., the value of the \texttt{exp} parameter for the \texttt{power} functions), and the **resource being counted** (e.g., the number of multiplication operations for various \texttt{power} functions). A description of complexity is not complete unless all three of these are made clear.

6.7 Further References

The topic of measuring computational complexity is usually treated in greater detail in a second-semester introductory computer science course.
6.8 Exercises

**Exercise 6.1.** Compare the number of multiplication operations performed by the two Python functions you wrote to compute \( n! \) in Exercise 4.6. Do these functions have the same “shape”?

**Exercise 6.2.** Compare the number of addition operations performed by the two Python functions you wrote for Exercise 4.7 (to find elements of the Fibonacci sequence). If you can’t come up with an exact formula for one of your functions, you may be able to find an “upper bound” and “lower bound” formula that have the same shape (i.e., both linear, both quadratic, or both exponential). Do the complexity functions for your two different Python functions have the same “shape”? What assumptions have you made about the language system’s ability to identify (and avoid) redundant function calls? Based on your formulas, how many additions would be needed for each Python function to find the 20th, 25th, 30th, 35th, 40th, 45th, and 100th elements?

**Exercise 6.3.** Measure the times needed for each of your functions from Exercise 4.7 to compute several elements of the Fibonacci sequence, such as the 20th, 25th, 30th, 35th, 40th, and 45th elements (leave the computer otherwise idle while you make these measurements). Then compute the ratios of these run times to the number of additions required for each (from Exercise 6.2). If the run time grows with the number of additions, the ratio should be roughly the same for all uses of the same function — is this the case? Based on the average ratio for each function, estimate the amount of time each would need to find element 100 of the sequence. If the run time depends only on additions, but not on any other steps performed, then the two Python functions should have the same ratio of run time to number of additions — is this the case? Based on the average ratio for each function, which do you think does more “other work” in addition to additions?

**Exercise 6.4.** Based on your answer to Exercise 6.3, discuss the merits of the “find the shape of the complexity function” approach to describing complexity. For this problem, is it more important to understand (or control) this shape, or to understand (or control) exactly which operations are used (the “other work” beyond just doing additions).

**Exercise 6.5.** Do your improvements to the subclique generation algorithm from Exercise 4.9 improve the worst-case complexity of this function? If you have implemented your improvements, measure and compare the speeds of the original algorithm and your modified version.
Chapter 7
Imperative Programming

The programs we have written so far have been closely related to mathematics, focusing primarily on what the answer is, rather than what the computer does to produce it. This facilitates the use of mathematical techniques on computer programs, but it is not the only way to write or think about software. In this chapter, we will focus on programming and thinking techniques for telling the computer what to do to produce an answer. Any program can be expressed in either way, but some programs are more easily expressed in one than the other.

The new approach will, at first, seem to abandon our use of mathematical techniques and our image view of program execution as a sequence of substitutions. However, there are ways of re-establishing these connections, as we will see later in this chapter.

These different conceptual models of computer software are referred to as programming paradigms. Previously, we have used the pure functional paradigm, in which each Python function corresponds to the mathematical (“pure”) notion of a function: a consistent way of relating parameters to results using techniques that correspond to mathematics. We now move on to the imperative paradigm, in which each function is thought of as a sequence of orders the computer must follow.

We begin our study of the imperative paradigm by returning to the now-familiar task of raising a number to a positive integer power. Our original power function from Figure 3.2 corresponds to a set of statements about what is true: “

\[ b^e = b \cdot b^{e-1}, \text{ and } b^1 = b \]

". We could explain what to do to compute \( b^e \) for a positive integer \( e \): “Start by writing down \( b \) as result_so_far; if \( e \) is bigger than 1, replace \( e \) with \( e - 1 \) and result_so_far with result_so_far*b, and keep doing so until \( e \) becomes 1. At this point, result_so_far must be \( b \) raised to the original value of \( e \).". Expressing this algorithm concisely in Python will require a both some new tools and some new ways of using the tools we’ve seen already; as we explore each tool, we will discuss its impact on testing and mathematical reasoning; the new approach has no impact on code reviews, except of course that reviewers should be familiar with the paradigm and language features used in the code they are reviewing.

7.1 Order of Execution

The imperative paradigm forces us to revise our thinking about the variables in our computer

103
programs. This is most easily done by (temporarily) limiting ourselves to the “step-by-step execution of instructions” model of Chapter 3.3 (though, in fact, imperative programs and pure functional programs are typically executed using a mixture of these techniques).

As we focus on this execution model, we will spell out the order of execution in more detail. Specifically, the “usual” order of the statements in a function is from top to bottom. If a function starts with the statement x=1 and has y=2 on the line below, we think of the computer associating 1 with x before it associates 2 with y. We can also ask Python to do two things in order by putting them on the same line with a semicolon (;) in-between, as in x=1; y=2. Regardless of the notation, we will refer to a collection of statements to be performed in order as a sequence of statements.

The order of execution of different parts of and or expressions are also precisely defined: the parts are executed from left to right, and Python stops executing once the answer has been determined. For example, suppose we had started with x=1 and y=2 and checked if x>0 and x>y and y>0. In this case, Python would determine that x>0 is True and go on to check x>y. Since this is False, and False and anything is False regardless of the value of anything, Python concludes the remaining and is irrelevant and stops checking. Similarly, Python does not evaluate the second part of an or expression when the first part is True.

7.2 What is a Variable? Sequences and Updates

At first, the “order of execution” rule just seems like a re-statement of the variable use rule we’ve already seen: each variable can be used only below its definition, within the same structural element (see Section A.4.7). However, it also gives us a way of making a variable change over time. The sequence x=1; y=x; x=2; z=x sets x to 1 and then 2, thereby setting y to 1 and z to 2 in a rather roundabout way.

Note that this sort of change to a variable’s value does not occur in mathematics, and thus we will have to envision variables in a way that differs fundamentally from that of mathematics (and earlier chapters of these notes). Our prior use of mathematical techniques such as “substitution of equals for equals” is not valid for imperative programs. If we take out the x=1 and replace all other x’s with 1 in the sequence above, we get 1=1; y=1; 1=2; z=1 — the third statement is nonsense, and the fourth is not correct, since z will actually have the value 2.

Instead of thinking of a variable as a name for a value, we think of a variable as a name that can be connected (in sequence) to one or more objects that may themselves change in value. There is an important distinction between changing the association of a name to an object, or on the other hand changing the value of an object. This subtle distinction is vitally important to understanding program execution in the imperative model. Furthermore, this distinction is made differently in different programming languages, creating further potential for confusion when programmers switch language or engage in discussions with programmers using other languages.

We will present a simplified model of Python variables that is adequate to understand the execution of Python programs, and then discuss techniques for regaining our ability to think about programs as mathematical entities. Note that an actual implementation of the Python language may vary from this model as long as the program produces the same result. Our discussion will take begin with a simplified variant of our power function that finds a number cubed, as shown in Figure 7.1.
"""
Compute base to the 3rd power, imperative style

>>> power(3.0, 3)
27.0

>>> power(0.5, 3)
0.125
"""
from logic import *

def power(base, exp):
    precondition(is_number(base) and is_integer(exp)
            and exp==3)

    # "start with result_so_far = base"
    result_so_far = base

    # replace exp with exp-1,
    # and result_so_far with result_so_far*base
    exp = exp-1
    result_so_far = result_so_far*base

    # keep doing this until exp is 1
    exp = exp-1
    result_so_far = result_so_far*base

    assert(exp == 1)
    postcondition(result_so_far == base**3)
    return result_so_far

Figure 7.1. Changing a Variable to Compute \(b^c\) for \(c = 3\).

Check Your Understanding 7.1. Write a sequence of statements that exchanges the values of two variables while changing only one at a time. For example, if \(x\) is 1 initially and \(y\) is 2, and then we execute the sequence of statements, then \(x==2\) and \(y==1\) would be true afterwards. Note that the sequence \(x=y; y=x\) will not work because \(x\) and \(y\) would both be 2 in our example.

7.2.1 Testing Functions with Sequences of Variable Updates
The possibility of repeatedly updating a variable has little impact on our approach to developing or evaluating a test suite. A glass-box test suite should still aim for good “statement coverage”; in the absence of conditional statements, any call to the function will execute all statements (unless it ends early with an error). Later, we will see that the presence of conditional statements in imperative program complicates our thinking about test coverage.

7.2.2 Reasoning About One Example with a “Box and Arrow Diagram”
If we only need to think about one specific execution of our function (perhaps because there is one case in our test suite for which it produces the wrong answer), we can go through the function one
line at a time and draw a picture showing both the association of variables to objects, and the values of the objects. Each time the program changes something, we update the picture.

These diagrams will denote objects with values in boxes, and variables with arrows from names to objects. We will refer to them as “Box and Arrow Diagrams” or by the unfortunate acronym B.A.D.’s.

The key to drawing B.A.D.’s is remembering that assignments change whatever is on the left side of the =. So far, this has always been a variable name, though later we will see other possibilities. When a simple name appears on the left side of an =, we make sure that name is in our B.A.D., and connect its arrow to the object given on the right. If the right hand side is just a variable name, as in the case of \texttt{result\_so\_far = base}, we connect the left hand variable’s arrow to the object. When the right-hand side computes a new value (or otherwise creates a new object), as in the case of \texttt{exp = exp-1}, we add the new object to our set of objects, and connect the variable’s arrow to it. As we work, we repeatedly add objects and change arrows until we reach the return, at which point an arrow to the object being returned is taken back to the point of the function call. Thus, we explore the execution of \texttt{power(0.5, 3)} by drawing and updating a B.A.D. as follows

a) We begin with the parameters and their values;

b) for \texttt{result\_so\_far = base}, we add a variable \texttt{result\_so\_far}, connected to the same object as \texttt{base};

c) for the first \texttt{exp = exp-1} line, we use the object referred to by \texttt{exp}, i.e., 3, in the evaluation of \texttt{exp-1}, i.e., 3-1, to produce the object 2, and then connect \texttt{exp}’s arrow to that (in the figure, the old arrow and old object 3 are shown as shadows, as they are not needed any more);

d) for the first \texttt{result\_so\_far = result\_so\_far*base} line, we use the objects referred to on the right in the evaluation of \texttt{result\_so\_far*base}, i.e., 0.5*0.5, to produce the object 0.25,
and then connect result_so_far’s arrow to that;

e) this process of adding objects and reconnecting variables continues; leaving us with exp connected to 1 and result_so_far connected to 0.125 before the final assertion, postcondition, and return statement.

This all seems like quite a lot of work to figure out a very simple program. These diagrams are most useful for sorting out the actions of confusing programs, but it’s easiest to get the process right with a simple example to avoid confusion later.

Note that our example involved many cases in which variables were attached to new objects, but none in which an object changed — in Python, there is no way to change an integer or string object, so changes to objects will have to wait for Section 7.5.

Check Your Understanding 7.2. Create a “box and arrow diagram” for an example use of the “swap the values of two variables” sequence from the Check Your Understanding earlier in this section.

7.2.3 Recording Facts for Each Program Point in Assertions

We can reason about the general properties of an imperative program, rather than just the execution of a specific example, by making claims about the values of variables that are true at a specific point rather than the entire function. For example, if we create a variable original_exp that retains the original value of exp, then the claim exp == original_exp - 1 is true after the execution of the first exp = exp - 1 but not after the execution of the second exp = exp - 1 line. If we abandon hope of substituting anywhere but the immediately following line, and update our list of claims after each change, all our substitutions will be correct.

When our program modifies variables but not objects (as in our running example), we can make all claims about the values of variables, and express these claims as Python assert statements, as is done for Figure 7.1 in Figure 7.3. When a program modifies objects, we also need to make claims like “a and b refer to the same object”, which may need to be expressed as a comment rather than an assertion.

Note that the sequence of assertions constitutes a proof of our postcondition — this “updating of assertions” approach is considered a legitimate way of making general proofs about imperative programs.

Check Your Understanding 7.3. Add appropriate assertions to your “swap the values of two variables” sequence from the “test your understanding” earlier in this section.

7.2.4 Converting to “Static Single Assignment Form”

Our ability to apply mathematical reasoning to a Python function was lost for imperative programs because the whole idea of a variable in an imperative program doesn’t correspond to the mathematical idea of a variable. We can regain our ability to reason mathematically if we relate each mathematical variable to a Python assignment rather than a Python variable. When each variable is assigned only once, as in previous chapters, this distinction is irrelevant. For a function like that
Compute base to the 3rd power, imperative style

```python
>>> power(3.0, 3)
27.0

>>> power(0.5, 3)
0.125
```

```python
from logic import *

def power(base, exp):
    precondition(is_number(base) and is_integer(exp) and exp==3)
    original_exp = exp  # just needed for assertions, etc

    # "start with result_so_far = base"
    result_so_far = base
    assert (result_so_far == base and exp == original_exp)

    # replace exp with exp-1,  
    # and result_so_far with result_so_far*base
    exp = exp-1
    assert (result_so_far == base and exp == original_exp-1)
    result_so_far = result_so_far*base
    assert (result_so_far == base*base and exp == original_exp-1)

    # keep doing this until exp is 1
    exp = exp-1
    assert (result_so_far == base*base and exp == original_exp-2)
    result_so_far = result_so_far*base
    assert (result_so_far == base*base*base and exp == original_exp-2)

    assert(exp == 1)

    # since original_exp == 3, base**original_exp is base*base*base, so...
    postcondition(result_so_far == base**original_exp)
    return result_so_far
```

Figure 7.3. Reasoning About Imperative Programs via Assertions

shown in Figure 7.1 a Python variable that is assigned multiple values will correspond to multiple mathematical variables.

To retain some ability to connect our mathematics back to the Python program, we adopt a convention of naming our mathematical variables after our Python variables but using distinct subscripts for each definition. Since the `power` function of Figure 7.1 includes three assignments to `result_so_far`, we will use the variables `result_so_far1`, `result_so_far2`, and `result_so_far3` for the values in the three assignments. When a function is a sequence of assignments without `if`, we can easily match up the mathematical and Python variables by giving consecutive subscripts to the
various assignments to a variable, and choosing the subscript from the most recent assignment above each use of a variable. The result of such a matching is shown in Figure 7.4, though to save space the variable names have been shortened, for example to rsf\textsubscript{1} instead of result\_so\_far\textsubscript{1}. This form of

""
Compute base to the 3rd power, imperative style

```python
>>> power(3.0, 3)
27.0

>>> power(0.5, 3)
0.125
```

from logic import *

```python
def power(base, exp):
    precondition(is_number(base) and is_integer(exp)
    and exp==3)

    # "start with result\_so\_far = base"
    result\_so\_far = base

    # replace exp with exp-1,
    # and result\_so\_far with result\_so\_far*base
    exp = exp-1
    result\_so\_far = result\_so\_far*base

    # keep doing this until exp is 1
    exp = exp-1
    result\_so\_far = result\_so\_far*base

    assert(exp == 1)
    postcondition(result\_so\_far == base**3)
    return result\_so\_far
```

(a) Imperative

```python
def power(base\textsubscript{0}, exp\textsubscript{0}):
    rsf\textsubscript{1} = base\textsubscript{0}

    # replace exp with exp-1,
    # and result\_so\_far with result\_so\_far*base
    exp\textsubscript{1} = exp\textsubscript{0}-1
    rsf\textsubscript{2} = rsf\textsubscript{1} * base\textsubscript{0}

    # keep doing this until exp is 1
    exp\textsubscript{2} = exp\textsubscript{1}-1
    rsf\textsubscript{3} = rsf\textsubscript{2} * base\textsubscript{0}

    assert(exp\textsubscript{2} == 1)
    postcondition(result\_so\_far == base\textsuperscript{3})
    return rsf\textsubscript{3}
```

(b) Pure Functional

Figure 7.4. Relating Imperative and Pure Functional Programs via Static Single Assignment

a program is known as Static Single Assignment Form or SSA form. We can convert a program into static single assignment form and enter it as a different Python program (adopting some convention for entering subscripts, e.g., exp\textsubscript{1} for exp\textsubscript{1}). Alternatively, we can leave the Python program alone but write SSA form alongside it, as in Figure 7.4, and apply our conclusions about the SSA form to the original function. For example, we can easily prove Figure 7.4b returns base\textsuperscript{3}, so Figure 7.4a must as well.

Check Your Understanding 7.4. Convert your “swap the values of two variables” sequence to SSA, and prove it is correct.
7.3 if without else or return

Previously, we have used Python’s if statement in a very specific way. Every if and if/elif sequence ended with an else. In most cases, we ensured that each indented list of statements within an if ended with a return (the only exception is the use of if/else to define one of two values for a single new variable).

Python allows the use of if in other ways that only make sense in the imperative paradigm. For example, Figure 7.5 shows the use of if to change the values of variables only under certain conditions. This allows us to create an imperative power function that works for positive integer

""
Compute base to the exp power, for integer exp 1...3,
using the imperative approach

>>> power(3.0, 1)
3.0

>>> power(3.0, 2)
9.0

>>> power(3.0, 3)
27.0

>>> power(0.5, 3)
0.125
""
from logic import *

def power(base, exp):
    precondition(is_number(base) and is_integer(exp) and exp>0 and exp<=3)
    original_exp = exp  # just for testing the postcondition
    result_so_far = base

    if exp > 1:
        exp = exp-1
        result_so_far = result_so_far*base

    if exp > 1:
        exp = exp-1
        result_so_far = result_so_far*base

    assert(exp == 1)
    postcondition(result_so_far == base**original_exp)
    return result_so_far

Figure 7.5. Conditionally Updating a Variable to Find $b^e$ for $e = 1, 2, \text{or } 3$. 

Copyright 2003-2015 David G. Wonnacott; see Page 7 for terms of free use
exponents up to three, but it will force us to modify some of our techniques for understanding program execution. We will also introduce some new vocabulary: we refer to each if, if/elif, if/else, or if/elif/else sequence (including the indented statements it controls) as a conditional sequence; we refer to the groups of statements controlled by (indented under) each part of a conditional statements as the statements in that branch. Figure 7.5 has two conditional sequences (each is a single if with two statements in the if branch and no else or elif branch).

Check Your Understanding 7.5. Modify your “swap the values of two variables” sequence from the previous section so that it swaps only when x has the larger value. So if we start with either x=17 and y=42 or with x=42 and y=17, we will end with x==17 and y==42.

7.3.1 Testing Imperative Programs with Conditionals

The presence of conditional expressions creates new ways to measure the effectiveness of a test suite. We still design black-box test suites to explore an interesting variety of problem instances and elicit a variety of answers. However, for glass-box test suites for imperative functions, we will aim for more than just the full “statement coverage” that we desired for test suites of “pure functional” Python functions.

When a function contains one conditional sequence after another, different parameter values may trigger different combinations of results from the conditionals. This sort of structure is quite common in imperative programs, and it can hide bugs that are only found on one particular combination of conditional branches.

Consider the contrived set concise example of Figure 7.6. The first three tests provide full “statement coverage”, but do not reveal the bug that is found by the fourth test. Test suites for functions written in the imperative style are often measured by both their coverage of statements and of possible paths through the program. Unfortunately, the number of possible paths can grow extremely quickly with the size of the function, and it may not be realistic to test every possible path for even a medium-sized function.

7.3.2 Drawing Box and Arrow Diagrams for Conditionals

Our approach for drawing box and arrow diagrams works fine with this new use of if. We simply only make updates for those statements that will actually be executed.

Check Your Understanding 7.6. Draw a box and arrow diagram for Figure 7.5 executing the example power(5,2), or for Figure 7.6 executing the fourth test in its test suite.

7.3.3 Detailed Assertions and Conditional Statements

If we wish to introduce assertions into imperative programs that include if statements, we must ensure that the assertions we place after the end of each conditional sequence are correct regardless of which part of the conditional sequence gets executed. Figure 7.7 shows such a set of assertions.
Some examples of a contrived function, showing different paths and path coverage.

This SKIPS the first if’s body, and executes the second if’s IF-BRANCH
```python
>>> combine_paths(2, 5)
9.0
```

This EXECUTES the first if’s body, and executes the second if’s ELSE-BRANCH
```python
>>> combine_paths(12, 5)
0.0
```

This SKIPS the first if’s body, and executes the second if’s IF-BRANCH
```python
>>> combine_paths(2, 20)
10.25
```

The above three tests have executed all statements, BUT ...

This EXECUTES the first if’s body, and executes the second if’s IF-BRANCH ... a bad combination!
```python
>>> combine_paths(13, 20)  # uh-oh :-(
```

---

```python
def combine_paths(a, b):
    # precondition: a and b are positive numbers

    if a >= 10:
        a = 0

    if b > 10:
        c = (b+0.5)/a
    else:
        c = (b-0.5)*a

    return c
```

**Figure 7.6.** A Bug in Only One Path Through A Sequence of Conditional Statements

for the function of Figure 7.5.

The assertion following the first conditional sequence illustrates two basic techniques that can be used to combine the outcomes of different paths through the conditional. Our assertion about the value of `result_so_far` is a generalization of the facts that hold after execution of the statements controlled by the `if` (i.e., that `result_so_far == base*base`, which occurs when
def power(base, exp):
    precondition(is_number(base) and is_integer(exp) and exp>0 and exp<=3)
    original_exp = exp  # just for assertions

    result_so_far = base
    assert (result_so_far==base and exp==original_exp)

    if exp > 1:
        exp = exp-1
        assert (result_so_far == base and exp==original_exp-1)
        result_so_far = result_so_far*base
        assert (result_so_far == base*base and exp==original_exp-1)
        assert (result_so_far == base**(original_exp-(exp-1)) and
                ((original_exp==1 and exp==original_exp) or
                 (original_exp>=2 and exp==original_exp-1)))

    if exp > 1:
        exp = exp-1
        assert (result_so_far == base*base and exp==original_exp-2)
        result_so_far = result_so_far*base
        assert (result_so_far == base*base*base and exp==original_exp-2)
        assert (result_so_far == base**(original_exp-(exp-1)) and
                ((original_exp==1 and exp==original_exp) or
                 (original_exp==2 and exp==original_exp-1) or
                 (original_exp>=3 and exp==original_exp-2)))

    assert (result_so_far == base**(original_exp-(exp-1)) and exp==1)

    postcondition(result_so_far == base**original_exp)
    return result_so_far

Figure 7.7. Assertions for an Imperative Program with Conditional Sequences

exp>1, and thus exp==original_exp-1) and those that hold when skipping those statements
(i.e., that result_so_far == base, which occurs when exp==1). The assertion result_so_far
== base**(original_exp-(exp-1)) captures both possibilities in a single equation.

When we can’t find a single equation to generalize all outcomes of a conditional sequence, we
can embed the idea of “if this happens, we get that” by using or to combine statements about the
individual possibilities. In this example, either execution skips the statements controlled by the if
because exp==1 (in which case exp is unchanged, so exp==original_exp), or execution includes
these statements (in which case exp is reduced by 1, so exp==original_exp-1). Since one of these
must occur, we use or to combine the possibilities on the second line of the assertion that follows
the first conditional sequence.

A similar assertion follows the second conditional sequence. Since our function’s precondition
tells us that exp must be 1, 2, or 3, we can combine the possible outcomes regarding exp by noting
that exp==1 in all possible cases, as shown in the assertion that follows. By substituting this
value of exp into result_so_far == base**(original_exp-(exp-1)), we produce the function’s
postcondition.
Check Your Understanding 7.7. Add appropriate assertions to the “exchange the values so that x is smaller” statements you wrote at the beginning of this section (you may need to add new variables to keep track of the original values). Introduce uses of Python’s \texttt{min} and \texttt{max} functions (which can be imported from the math library) as soon as possible. Does your final assertion capture the meaning of your statements?

7.3.4 Static Single Assignment and Gated Single Assignment Forms

The introduction of conditional sequences complicates our use of static single assignment: when a variable is used after a conditional assignment, we can’t just label that variable use with the subscript from the closest assignment above it. For example, although we could label the three definitions of \texttt{result\_so\_far} in Figure 7.5 \texttt{rsf1}, \texttt{rsf2}, and \texttt{rsf3}, it is not correct to assume that the name \texttt{result\_so\_far} in the return statement refers to the value of \texttt{rsf3} (the closest assignment); if this were true, \texttt{power} would always return \texttt{base}.

Since this problem only occurs when variables are used after the conclusion of a conditional sequence, we can eliminate the problem by simply moving all statements that follow the final conditional sequence to the end of every branch of that conditional sequence (adding a final \texttt{else} to include those statements if no \texttt{else} already exists). If we repeat this process until no statements follow a conditional sequence, we will achieve a program with sequences \textit{contained within} the branches of conditionals, but \textit{no sequence containing} a conditional anywhere but the end. The modified program can then be converted to SSA form by giving consecutive subscripts to the definitions of each variable and subscripting each use of a variable with the closest preceding definition that is not contained in a separate branch. This approach is conceptually simple, but can greatly increase the size of the program; Figures 7.8 and 7.9 show the first step (moving the final assert/postcondition/return sequence into the second conditional sequence) and final result of this conversion for Figure 7.5. Note that the second if \texttt{exp\_0 > 1} test (below the leftmost \texttt{else})

```python
def power(base, exp):
    precondition(is_number(base) and is_integer(exp) and exp>0 and exp<=3)
    original_exp = exp  # just for testing the postcondition
    result_so_far = base

    if exp > 1:
        exp = exp-1
        result_so_far = result_so_far*base

    if exp > 1:
        exp = exp-1
        result_so_far = result_so_far*base

    assert(exp == 1)
    postcondition(result_so_far == base**original_exp)
    return result_so_far

    else:
        assert(exp == 1)
        postcondition(result_so_far == base**original_exp)
        return result_so_far
```

Figure 7.8. Working Toward SSA form by Moving Subsequent Statements into Conditionals
def power(base_0, exp_0):
    precondition(is_number(base) and is_integer(exp_0) and exp_0>0 and exp_0<=3)

    result_so_far_1 = base_0

    if exp_0 > 1:
        exp_1 = exp_0-1
        result_so_far_2 = result_so_far_1*base_0

        if exp_1 > 1:
            exp_2 = exp_1-1
            result_so_far_3 = result_so_far_2*base_0

            assert(exp_2 == 1)
            postcondition(result_so_far_3 == base_0**exp_0)
            return result_so_far_3
        else:
            assert(exp_1 == 1)
            postcondition(result_so_far_2 == base_0**exp_0)
            return result_so_far_2
    else:
        if exp_0 > 1:
            exp_3 = exp_0-1
            result_so_far_4 = result_so_far_1*base_0

            assert(exp_3 == 1)
            postcondition(result_so_far_4 == base_0**exp_0)
            return result_so_far_4
        else:
            assert(exp_0 == 1)
            postcondition(result_so_far_1 == base_0**exp_0)
            return result_so_far_1

Figure 7.9. Conversion to SSA by Repeated Copying and Addition of Subscripts

is unnecessary, since it must be false if we are in the else branch of the outer if. This test can thus be removed (using the substitution rule for if False from Appendix A); it corresponds to a redundant test performed by the original imperative algorithm.

An alternative to the repeated copying approach is the introduction of Gated Single Assignment (or GSA) form. In this form, we use a special notation to capture the idea that a variable will have one value if execution includes one branch and a different value if it does not. The notation $\gamma(condition, true-value, false-value)$ corresponds to a function that produces true-value if condition is true and false-value if it is not. Figure 7.10 illustrates gated single assignment form for our running example from Figure 7.5.

The use of $\gamma$ for various other forms of if can be understood based on the insight that $\gamma$ is just selecting each value in the condition that it would have been in the variable in question. For an if without elif or else (as above), the $\gamma$ function selects the updated value from the controlled statements indented under the if or the value that existed before the if; for an if/else that
def power(base, exp):
    precondition(is_number(base) and is_integer(exp)
    and exp>0 and exp<=3)
    original_exp = exp
    result_so_far = base
    if exp > 1:
        exp = exp-1
        result_so_far=result_so_far*base
    if exp > 1:
        exp = exp-1
        result_so_far=result_so_far*base
    assert(exp == 1)
    postcondition(result_so_far==base**original_exp)
    return result_so_far

rsf1=base0
if exp0>1
    exp1=exp0-1
    rsf2=rsf1*base0
    exp2=gamma(exp0>1,exp1,exp0)
    rsf3=gamma(exp0>1,rsf2,rsf1)
if exp2>1
    exp3=exp2-1
    rsf4=rsf3*base0
    exp4=gamma(exp2>1,exp3,exp2)
    rsf5=gamma(exp2>1,rsf4,rsf3)
return rsf5

Figure 7.10. Relating Imperative and Pure Functional Programs via Gated Single Assignment

updates a variable in both branches, it selects the value from one of the branches; for an if/else
that updates a variable in only one of the branches, it selects the updated or original value; for an
if/elif or if/elif/else sequence, γ expressions can be used as arguments to other γ's — for
example, if each of the four cases in Figure [A.5] had assigned a value to a variable response, such
as response = "You should see a specialist" when temperature >= high_fever, we could
combine the results after the conditional sequence with a complicated γ expression
γ(t ⩾ high_fever,response1,γ(t ⩾ fever,response2,γ(t ⩾ believable,response3, response4))).

The introduction of γ expressions yields a set of equations that define the result of the Python
function in a mathematically meaningful way. One might imagine that we could convert gated single
assignment into a Python function by simply creating another Python function gamma (which would
take three parameters and returns the second or third based on the truth value of the first), and
using that for each γ expression. Unfortunately, this does not work in cases where one computation
may produce an error or infinite recursion.

The gated single assignment form of a program is thus primarily useful as a tool for proving
properties about imperative algorithms. This form lends itself to the “proof by cases” technique, in
which a separate proof is constructed for each case (for Figure 7.10 we would consider the cases of
exp0>1 and exp0⩽1, for example). These proofs are often similar to those that would have been
created for both the copy-statements-after-conditionals-and-use-SSA and add-assertions approaches
to reasoning about imperative functions; in some cases one approach may take more or less writing
than the others, but generally the choice of approach can be left to personal preference — each
approach can in principle be applied to any imperative program.

Check Your Understanding 7.8. Create a Python function larger_minus_twice_smaller that includes your
exchange the values so that x is smaller statements and then, based on the results of these statements, returns the
maximum minus twice the minimum (use Python’s min and max functions in the postcondition but not the actual
computation performed by the function). Then explore both single-assignment forms: first, move statements that follow conditional sequences into all branches, convert to SSA, and prove your function is correct; then convert your original function into GSA and prove that version correct. For this example, do you prefer assertions, SSA, or GSA?

7.4 Loops

We began this chapter with an English-language description of an imperative algorithm for raising a number to a positive integer power. The versions of an imperative power function that we’ve written so far only work for exponents up to 3 — the original plan was to keep updating exp and result_so_far until exp got down to 1. This could be achieved by having the function call upon itself, but languages like Python that support the imperative style offer a more concise approach: the use of a loop statement.

A loop statement, like a conditional statement (if, if/else, etc.), has the job of controlling the execution of other statements indented below it (these statements are known as the loop body). Unlike a conditional statement, which executes each branch zero or one times, a loop statement can execute statements repeatedly. Each execution of the statements controlled by the loop is referred to as an iteration of the loop. Loops, like conditional statements, are basic building blocks that can be combined in many ways. In this chapter, we will see loops that contain conditional statements; later we will see loops that contain other loops; the combinations are endless, and must be chosen carefully to orchestrate the way the other statements in the program perform their actions.

Python’s primary loop statements are the while statement and the for statement. Each can be illustrated with our power example.

7.4.1 while loops

Python’s while loop (like that of other programming languages) is designed to repeatedly execute a loop body as long as some condition remains true. Figure 7.11 shows our imperative power algorithm written with a Python while loop. The execution of a while loop begins with a test of the condition that follows the word while, which is known as the loop test or loop condition. If the loop condition is true, the computer executes the loop body and returns to start the process over again by checking the loop condition. Once the loop condition is found to be false, the while loop is done, and the statement below the while and its body is executed next. Note that, if the loop condition is false when execution first reaches the while, the body is not executed at all. So in the power(3.0, 1) example, the program starts result_so_far at 3.0, then since the test exp > 1 is false it skips the changes to exp and result_so_far, and goes right to the final assertion/postcondition/return sequence (in other words, the computer executes zero iterations of the loop).

Check Your Understanding 7.9. Describe (in words) an imperative algorithm for the palindrome problem of Exercise 4.5, and then convert your description into an imperative-style Python program that uses while and if.

7.4.2 for loops and range expressions

A for loop is used when we wish to start by thinking about the set of iterations, rather than about the condition under which execution continues. For example, instead of saying “multiply result_so_far by base while you keep reducing exp by 1 until it reaches 1” (as we did in our original description of the imperative power algorithm), we might say “multiply result_so_far by
"""
Compute base to the exp power, for positive integer exp,
using the imperative approach

>>> power(3.0, 1)
3.0

>>> power(3.0, 2)
9.0

>>> power(3.0, 3)
27.0

>>> power(0.5, 3)
0.125

>>> power(3.0, 5)
243.0

>>> power(0.5, 5)
0.03125
"""

from logic import *

def power(base, exp):
    precondition(is_number(base) and is_integer(exp) and exp>0)
    original_exp = exp  # just for testing the postcondition

    result_so_far = base

    while exp > 1:
        exp = exp-1
        result_so_far = result_so_far*base

    assert(exp == 1)
    postcondition(result_so_far == base**original_exp)
    return result_so_far

Figure 7.11. Updating Variables in a Loop to Find $b^e$ for Positive Integer $e$.

base once for each value in the range 2...exp (inclusive)”. This would produce the same result, but sometimes this approach is easier to visualize.

Python’s for statement executes a loop body once for each item in a given set of iteration values. As it does so, it associates a variable (known as the loop variable here) with each iteration value in turn. When we wish to construct a range of numbers, we can do so with Python’s range expression: range(1, u) creates a sequence of values from 1 to u-1 (for example, range(0,10) is the numbers 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9 — one common beginner’s mistake is to forget that range(1, u) stops at u-1 rather than u). Thus, if we begin a loop for i in range(2, exp-1), the body will be executed for each value from 2 to exp (as we desire in this example).
Figure 7.12 shows the use of Python’s `for` statement and `range` expression to control our imperative `power` function. The loop index variable `i` takes on values from 2 to `exp` (inclusive), and each time through the loop `result_so_far` is updated to be `base`.

We will usually use `for` and `range` together, though this is not required. The `for` statement can also be used with other sequences, such as strings. If we write a `for` loop that starts with the line `for letter in "aeiouy"`, the body will be executed once for each lower-case vowel, with the variable `letter` set to the vowel.

We can draw box and arrow diagrams or row and column diagrams for `for` loops as well as `while` loops.

Check Your Understanding 7.10. Rewrite your imperative palindrome function to use the `for` statement with a `range` expression. Instead of removing letters from both ends of the string during each loop iteration (as you might have done in your `while` example), use your loop variable as an “index” to find a letter near the front of the string, and the string length minus your loop variable as an index to find the corresponding letter near the end of the string.

7.4.3 Testing of Functions With Loops
A single loop statement can create the possibility of an unlimited number of paths through a function, if we consider every possible number of iterations a different path. When evaluating path coverage of test suites of programs with loops, we typically look for one test that skips the loop body entirely, one that executes it exactly once, and one that executes it at least twice. Testing, as always, is an art of compromise between the desire to add tests that might expose a new bug and the desire to stop testing and move on to something else.

7.4.4 Loops, Box and Arrow Diagrams, and Row and Column Diagrams
We can draw box and arrow diagrams to illustrate the execution of a loop on a particular example. However, if the loop proceeds through many iterations, these diagrams can get extremely sloppy.

We therefore take advantage of the organization that arises from repeated execution of the same statements: the loop body typically updates the same set of variables in each iteration. We will therefore create a table in which we provide one row for each variable in one column for each iteration — for obvious reasons, we call such a diagram a `row and column diagram`.

Figure 7.13 shows a row and column diagram for the execution of `power(3.0, 5)`. We start each row with the variable’s name and the value it has before the loop is started. The boxes defining this first column of values are left open on the left to visually illustrate that these values came in from outside the loop. We then work our way through the program, as we would when drawing a box and arrow diagram, but in this case adding columns of variable values. Each time the loop condition is true we add a vertical line to separate the values so far from the next column — note that the final column has no right-hand boundary, a visual cue that these values are open to the statements that happen after the loop.

Row and column diagrams will play a central role in establishing insights that will guide our approach to reasoning about loops in Chapter 8.

Check Your Understanding 7.11. Illustrate the use of either box and arrow diagrams or row and column diagrams for the execution of one or two simple examples of your imperative palindrome function.

7.5 Lists and Modification of Objects

KEY PART OF THIS: * you can change a variable that refers to a string, but not the string itself; you can change the elements of a list (show diagram) — “the expression on the left side of the equals sign tells us which arrow will be changed”.

Compute base to the exp power, for positive integer exp, using the imperative approach

```python
>>> power(3.0, 1)
3.0
>>> power(3.0, 2)
9.0
>>> power(3.0, 3)
27.0
>>> power(0.5, 3)
0.125
>>> power(3.0, 5)
243.0
>>> power(0.5, 5)
0.03125

from logic import *
```

def power(base, exp):
    precondition(is_number(base) and is_integer(exp) and exp>0)

    result_so_far = base
    for i in range(2,exp+1):
        result_so_far = result_so_far*base
        assert(result_so_far == base**i)

    postcondition(result_so_far == base**exp)
    return result_so_far

Figure 7.12. Imperative Programming Using for and range.

<table>
<thead>
<tr>
<th>base</th>
<th>3.0</th>
<th>9.0</th>
<th>27.0</th>
<th>81.0</th>
<th>243.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>exp</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

Figure 7.13. A Row and Column Diagram for power(3.0, 5).

7.6 Creating and Using Procedures

Show a “swap these elements in a list” procedure and “capitalize the String”, which much be a function not a procedure (unless for a list of characters).

INCLUDE list.append notation and semantics
7.7 Correctness of Imperative Programs

DISCUSS testing as well, all def-use vs. all statement coverage

also code review

*** MENTION PROCEDURES TOO!!! ***

preconditions and postconditions with “original” and “final” versions of variables (but can’t test these without holding onto the values — which can be tricky with lists).

How do people reason about object identity here???

7.7.1 Code Reviews

7.7.2 Verification

SSA

Copy post-loop code vs. GSA and proof-by-cases
will return to loop proofs in next chapter

7.7.3 Software Testing and Coverage Metrics

define all-statement coverage vs. all-defs vs. all-uses vs. all-def-use-links. But what about combinations of one def for one variable and another def for another variable (see example). Also come back to inter-functional testing — do proofs improve on this by making and using abstractions?

7.8 Summary

A new way of thinking about variables

BAD’s are good for confusion about sharing when a test case fails
SSA/DSA we don’t use for mutable objects

How to choose assertions vs SSA, SSA w/copy vs GSA? Style, specific program...

7.9 Further References

David Gries’ “The Science of Computer Programming” shows how to create algorithms by using formal verification techniques to design imperative programs (rather than to analyze existing programs). It gives a thorough treatment of the “write facts that are true at each program point” approach to thinking about imperative programs.
7.10 Exercises

**Exercise 7.1.** The imperative paradigm creates the possibility of new kinds of bugs that arise from doing things in the wrong order. Write a sequence of statements that exchanges the values into variables, and demonstrate with an example that your sequence would not work in a different order.

**Exercise 7.2.** Create a function `difference_of_differences` that takes three numerical parameters, finds the difference between the two highest numbers and the difference between the two lowest, and returns the difference between these two differences. For example, `difference_of_differences(12, 2, 5)` would find the difference between 12 and 5 (i.e., 7), and the difference between 2 and 5 (i.e., 3), and return the difference between these differences (i.e., between 7 and 3, i.e., 4). Your function should work regardless of the order of the parameters, e.g., `difference_of_differences(2, 12, 5)` should also return 4. Make use of techniques and language features from this chapter whenever possible, but write only one function.

**Exercise 7.3.** Create a function `difference_of_differences` (as above), but this time write and make use of a procedure `sort3` that takes a list of three numbers and puts them in order. In other words, if `numbers` is the list `[12, 2, 5]` or the list `[2, 12, 5]` before a call to `sort3(numbers)`, then `numbers` will be the list `[2, 5, 12]` after this procedure call.

**Exercise 7.4.** Create a function `difference_of_differences` (as above) making use of only “pure functional” language features (i.e., never change the value of a variable, give every if a matching else, and make sure the statements indented under any if or else end with a return). Which version do you like best? Is it easier to answer this question by changing your earlier programs or by starting over?

**Exercise 7.5.** Prove one of your imperative `difference_of_differences` functions is correct.

**Exercise 7.6.** Discuss the challenges of getting an appropriate test suite for one of your imperative `difference_of_differences` functions. What coverage metric do you think is appropriate? How many tests are needed to achieve full coverage with this metric? What about with other metrics? How does this relate to testing the pure functional version from Exercise 7.4?

**Exercise 7.7.** It is often said that recursion is slower than iteration. (This is typically true in current implementations of Python (and many other languages) as long as we restrict ourselves to two recursive and loop-based implementations of the same algorithm; However, this is not a fundamental law of nature.) Find a way to measure the relative speeds of the `power` functions from Figures 3.2, 5.6, 6.3 and 7.11. Which is fastest for small values of `exp`? What about big values of `exp`?
Chapter 8
Reasoning About Loops

We will discuss the use of loops to clarify the expression of functions with certain simple recursive structures, and the importance of loops as a tool for thinking about repetition without considering recursion.

8.1 Algorithm Design with Loops

(Lots of metaphor stuff, go back to “raise to a power” example; also re-do Fibonacci; relate these to tail recursive forms)

Relate while-based options to to power_with_hint options from Figure 4.10? Or leave for exercises?

8.2 Correctness of Loop-Based Algorithms

8.2.1 Loop Pre- and Post-Conditions and Invariants

For raise to-power example:

1) what’s really necessary — logical statements work for next section’s proof technique
2) how to get insight — (a) Bill Pugh’s “evil debugger” metaphor; (b) draw columns of variable values at start of each iteration, ask “could this column have come from this process?” for some random values — “why not” — add answers to “why not” to loop invariant .... then 1st column is loop precond, last is loop postcond

LEAVE FIBONACCI PROOFS FOR LAB; DON’T SHOW THIS IN THE NOTES

Things we need to do a loop proof:

You will need five pieces of information to prove a loop correct (i.e., to show that if it’s precondition holds, it will reach the loop postcondition with something that passes the postcondition) and that the function with the loop is correct. The steps are listed as a-e below:

a) The function expressing the algorithm, including its pre- and post-condition (this is already provided in the file if we have a specification for the function).
b) The loop postcondition (the function’s postcondition is given as a postcondition just before the return — typically the loop postcondition is very similar or even, in some cases, identical — though we do know that the loop test was False when we reach the postcondition). In this course, we will use our Python loop_postcondition procedure to state the loop’s postcondition.

c) The loop invariant — this is given as a loop_invariant statements at the top of the loop body (sometimes it may be placed at the end, or at the beginning and end, for emphasis).

d) The loop precondition (the function’s precondition is given as a precondition at the start of the function body — the loop precondition is given with loop_precondition immediately before the loop).

e) The loop progress condition — this is given with a loop_progress statement, typically right by the loop invariant at the start of the loop body. Our definition of loop_progress (like progress for recursive functions) requires an integer-valued expression that gets smaller with each execution of the loop body and would force the loop to stop if it became zero or negative.

8.2.2 Choosing Appropriate Logical Statements for Loops

It is often difficult for beginners to know what counts as a “good” invariant, precondition, or postcondition for a loop. Here are three ways to think about this:

8.2.2.1 Logical statements that allow a successful proof

The ultimate test of the appropriateness of logical statements about a loop is whether or not they allow the creation of a fully detailed proof based on the techniques that we will see in Section V. Unfortunately, until one has done enough successful proofs to have the “feel for it”, it is hard to use this standard to come up with logical statements.

8.2.2.2 Imagine an “evil debugger”

As easier way for beginners to develop these logical statement is to think about them in terms of an adversary that can modify your variables. I first learned this technique from Bill Pugh, who has an amazing talent for describing good ways to think about tricky technical topics. I’m not sure if Dr. Pugh came up with this himself or learned it elsewhere:

For a function’s precondition, we imagine that some nasty person is attempting to feed our function some parameters that will prevent it from returning a value that fits its postcondition. The precondition that we write will catch these nasty parameters, and our adversary knows this and does not want to get caught. Thus, if we can write a precondition that rules out everything bad while still letting the function do its job for appropriate parameters, the adversary can’t cause any trouble. This is (hopefully) easy to imagine, because you can’t always control who calls your function.

For a loop invariant, we imagine that the nasty person is using a debugger to run our program, and that our adversary can view or change the values of our variables, but (to make it a fair contest) the adversary can only do so at the start of end of the body of the loop (i.e., the statements indented under while). For example, in our loop based power function, we have:

```python
while exp_so_far < e:
    # NASTY PERSON CAN CHANGE VARIABLES HERE
    exp_so_far = exp_so_far + 1
    result_so_far = result_so_far * b
```

Copyright 2003-2015 David G. Wonnacott; see Page 7 for terms of free use
The loop invariant statement that we write will catch any changes that don’t fit the loop invariant, and the adversary knows this and doesn’t want to get caught. So, without a loop invariant, the adversary could just add seven to result_so_far, causing our function to produce the wrong answer (or report a failed postcondition check). Now suppose we add the following invariant:

```python
while exp_so_far < e:
    loop_invariant(orig_power(b, exp_so_far) == result_so_far)
    exp_so_far = exp_so_far + 1
    result_so_far = result_so_far * b
    # (Technically, we could repeat the invariant at the end of the
    # loop body too, but people usually don’t bother to do so.)
```

At this point, adding seven to result_so_far might throw off the invariant, so the adversary couldn’t get away with this. The adversary could still make some changes, however — suppose we had done several steps of the calculation of power(3,8), and had exp_so_far up to 4 and result_so_far up to 81 (i.e., 3^4). Without getting caught, the adversary could:

a) Change exp_so_far to 6 and result_so_far to 729 (which is 3**6), but this wouldn’t hurt us at all!
   
b) Change exp_so_far to 2 and result_so_far to 9 (which is 3**2); this would set us back, but unless the adversary did it repeatedly, we would still get the right answer — our "loop_progress" expression can catch attempts to set us back in this way, and prevent the adversary from trying this.
   
c) Change exp_so_far to 10 and result_so_far to 59049 (which is 3**10); this would make us produce the wrong answer, since it is _past_ our goal of 3**8. We can fix this by adding "exp_so_far <= e" in our loop invariant, so we can’t go "past the end".

Making the changes suggested in b and c, we have a version of the loop-based power function that the adversary can’t mess up:

```python
while exp_so_far < e:
    loop_invariant(orig_power(b, exp_so_far) == result_so_far and exp_so_far <= e)
    loop_progress(e - exp_so_far)
    exp_so_far = exp_so_far + 1
    result_so_far = result_so_far * b
    # (Technically, we could repeat the invariant at the end of the
    # loop body too, but people usually don’t — note that, in
    # this case, we could use "exp_so_far < e" at the top and
    # "exp_so_far <= e" at the bottom, if we wanted to do so.
```

This way of thinking about a loop invariant is often easier than the formal rule: a loop invariant and progress condition are correct if they always hold during the execution of the program and allow us to use the five-step method to prove the function is correct. The formal rule is a good check on your loop invariant — if you find that your Lab 7 proof is really hard, it typically means that you have the wrong loop invariant (or precondition, for the tail recursive power function).
So, if you are having trouble with the Lab 7 proof, you should think about the nasty person changing variables with the debugger. In particular, if you haven’t mentioned all of the variables that you use in the loop body without first putting a value into them, then you almost certainly have left an opportunity for the adversary to make a change that will throw off your calculation.

8.2.2.3 Identify plausible or implausible sets of variable values

Write the names of the variables in a column (as if they were the titles of rows of data in a spreadsheet). If you like, you can leave out “temporary variables” that hold a value that is only needed briefly within the loop body. Write some columns of values next to them, with each column corresponding to the values taken on at the beginning or end of an iteration of the loop as it processes some simple example — the first column should be the values just before the first execution of the loop body; the last column the values after the last execution of the body, and the columns in between should be the values between consecutive iterations.

Writing out an example should give you a feel for what does or does not “make sense” as a column. Now imagine that someone is going to write down a column without telling you what example it came from, and ask you whether it could plausibly have come from any real execution of that loop. (SHOULD HAVE AN EXAMPLE HERE.)

Your job is to articulate a rule that would identify columns that could possibly have been the first column in a “real” diagram (this rule is the loop’s precondition), and a rule that identifies something that could be the last column (the loop postcondition), and a rule that identifies something that could be a legal column anywhere on the diagram (this is the loop invariant).

This is often the easiest way for beginners to get the right loop invariant. I should probably describe it more clearly here.

Also, relate this to the “row of values” we’d see if we looked at the parameters in a recursive function.

8.2.3 Correctness Proofs for Loops

Steps of proof (also from Lab 7):

STEPS FOR PROVING THAT A LOOP IS CORRECT (i.e., for showing that, if we arrive at the top of the loop with the loop precondition true, then we must eventually get to the loop postcondition statement, and it will be true when we get there):

I. Prove that the loop postcondition must hold if the loop is skipped — i.e., assume the loop precondition is True and the loop test is initially False, and show that the loop postcondition must be True.

II. Prove that the loop invariant must hold when the loop is first entered — i.e., assume that the loop precondition is True (and, if necessary, that the loop test is True too) and prove that the loop invariant must be True.

III. Prove that the loop invariant must remain true — i.e., assume that the loop invariant is True and the loop test is True (so the loop body will be executed again), and show that it will still be True after performing the steps taking by the loop body (typically this is done by converting the loop body to SSA, assuming that the invariant and loop test holds with the initial subscripts on the variables, and showing that the loop invariant holds with the final subscripts for the variables).
IV. Prove that the loop must terminate (i.e., that having progress \( \leq 0 \) makes the loop stop, and that the progress expression gets smaller with each execution of the loop body).

V. Prove that the loop postcondition must hold when the loop ends — i.e., assume that the loop invariant is True and the loop test is False, and show that the loop postcondition must hold.

Also show that the correctness of the loop ensures the correctness of the function, i.e., that

- the loop_precondition must hold (based on the function’s precondition and any steps before the loop_precondition), and
- the function’s postcondition must hold (based on the loop_postcondition and any steps after the loop).

8.3 Loops and Tail Recursion

Show power with tail-rec, including logical statements, and the proof too (mention this in Lab 7 accordingly?); relate the two pieces of code, discuss “rows” of values for variables, and relation of the proofs.

8.4 Anagrams and Sorting Algorithms

8.5 Summary

8.6 Further References
8.7 Exercises

**Exercise 8.1.** Write a loop-based function to compute any requested element from the Fibonacci sequence. Include appropriate pre- and post-conditions for each loop (as well as for the function itself), and include an invariant for each loop.

**Exercise 8.2.** Prove your function from Exercise 8.1 is correct.

**Exercise 8.3.** (*)& Write a loop-based power function that uses the algorithm from Figure 6.3. This will probably be easiest if you start with the loop/function postcondition, and then think carefully about a progress expression and loop invariant, and only try to write the function once you think those are right.

**Exercise 8.4.** Prove your function from Exercise 8.3 is correct.

**Exercise 8.5.** Re-do Exercise 7.6 using the result of Exercise 8.3. Which version of the power function provides the best combination of clarity and speed?

**Exercise 8.6.** What coverage metric do you think is appropriate for evaluating a test suite for your functions from Exercises 8.1 and 8.3? How many tests are needed to achieve full coverage with this and other metrics?

**Exercise 8.7.** Repeat the steps of Exercises 6.2 and 6.3 for your function from Exercise 8.1, and consider Exercise 6.4 again, in the context of imperative programming.

**Exercise 8.8.** Compare the clarity and speed of your answers to Exercises 4.7 and 8.1. Which provides the best combination of clarity, easy-to-confirm correctness, and speed?

**Exercise 8.9.** Look back at the functions you wrote for Exercise 4.11. In which cases would your program be clearer and/or more concise if a loop had been used in place of recursion? In which cases would your program have been more confusing or difficult/impossible to write without the use of recursion? Would any of these changes have significant impact on the “shape-of-the-function” complexity, or on the running time of your program?

**Exercise 8.10.** Based on your answers to Exercises 7.7, 8.5, and 8.8, can you make any general statement about the relative advantages of recursion and loops when it comes to combining clarity, easy-to-confirm correctness, and speed?
Appendix A

A “Starting Set” of Python

This appendix describes the fraction of Python that will be used in the first part of this course. Our intent is not to give a definitive treatment of Python (which would only make sense once to someone with significant programming background), but rather to explore just enough of Python to let us express a wide variety of algorithms and explore various ways of thinking about them. We will refer to this limited part of Python as our starting set of the Python language. For this subset of the full Python language, we have also chosen elements of Python that are closest in meaning to mathematical expressions, to facilitate analysis of our algorithms. The discussion of each element concludes with a rule for performing substitutions involving that element. Note that these rules are valid as stated only for programs written using just the starting set; they must be adjusted for programs using other features of Python. Starting with Chapter 7 we will introduce other features of Python that let us express certain kinds of algorithms more clearly or concisely. Techniques for reasoning about these features (via substitution or in other ways) will be introduced with each new feature.

Our discussion of the starting set of Python will begin with the examples in Figures A.1 and A.2. These two figures together constitute a complete (but small) Python program that will greet the user by name (as long as the user is named Pat), ask the user to enter their temperature, and print a response. Figure A.3 illustrates how the program can be run from a command prompt — the elements typed by the user are in the italicised program font, the responses from the computer in the regular program font. It may also be possible to run the program by double-clicking on an icon in a file browser or an Integrated Development Environment (IDE), in which case all but the first line should appear in a window.

```python
from logic import *

# Algorithm to come up with a response to a reported temperature
def response_to(temperature):
    precondition(is_number(temperature))
    if temperature >= 99.8:
        return 'You have a fever'
    else:
        return 'You do not have a fever'
```

Figure A.1. The file diagnosis.py: A trivial algorithm illustrating some elements of Python
from logic import *
from diagnosis import *

A Text-based user interface to ask about temperature and respond. The my_name() function can be changed to allow the doctor to see a patient with a name other than Pat.

def patient_name():
    return 'Pat'

def doctor_interface():
    print('Hello', patient_name(), 'I hope you are feeling well.')
    entered_temp = float(input('Tell me your temperature in degrees Fahrenheit '))
    if is_number(entered_temp):
        print(response_to(entered_temp) + ',', patient_name())
    else:
        print("Sorry, I didn't understand you."))

doctor_interface()
line is a comment. Comments can serve as a form of communication among the members of a software team, or as a way for a programmer to make notes that will come in handy when returning to the program later to make changes (it is remarkable how alien one's own program can seem after a few weeks spent working on something else). Comments are ignored when the computer executes a program, and thus they play no role in determining the result — they are primarily for the benefit of people reading the program, though in some cases automated systems may extract information from comments to automatically produce documentation or test a program (e.g., Python's "doctest" system).

Comments are no substitute for clear organization and expression of ideas: one should strive to make the program as clear as possible without comments, and use comments to explain anything that is not evident from reading the program itself. For example, a function's name should explain what the function does, but a comment can be used to add detail that would be overly verbose in a name: a function to compute the square root of a number could be given the easily-typed name \texttt{sqrt}, with a comment used to add detail (perhaps noting that \texttt{sqrt} guarantees to always produce a non-negative value, e.g. 2 rather than -2 for \texttt{sqrt}(4), or alternatively stating that there is no guarantee about which of these is returned).

The need for comments depends somewhat on the use of other techniques to make the program clear: comments are sometimes used to provide information about restrictions on parameters (e.g., noting that the parameter to \texttt{sqrt} must be a non-negative real number), but if a \texttt{precondition} statement provides this information, there is no need for a redundant comment. However, any time the central purpose and limitations of a function are not clear, or the essential insights behind an algorithm are not immediately evident from a quick read of the function, a comment is appropriate. Other appropriate uses of comments include giving the author of a program or a copyright notice.

Note that comments should not be used to explain elements of a programming that will be obvious to an experienced programmer, except perhaps in books that teach programming. A real program would not include a comment "# the word if selects one result or another" right before an \texttt{if}.

If misused, comments can make a program harder to understand, for example if a programmer fails to edit them appropriately while changing a program. Comments that correspond only to an older version of the program can be much worse than no comments at all.

Python programmers embed comments in several ways. The hash mark (\#) is used to start a true comment— all text from the hash mark to the end of the line counts as a comment and is ignored during program execution (for example, the second non-blank line of Figure B.1 is a comment). If a comment line ends with a backslash (\) then the entire next line is also a comment.

In some contexts, Python programmers embed strings in parts of the program where they will be ignored, e.g. between function definitions. This is frequently done with multi-line strings (see Section A.4.6), which are identified by triple-quotes (""") at the start and end. The text between the sets of triple-quotes near the top of Figure A.2 is an example of one such "comment". Since strings are only ignored in certain contexts, so we will use them only for comments outside of function definitions (where they are ignored).

\textbf{Substitution Rule}: For any true comment (starting with \#), blank space can be substituted without affecting the meaning of the program. A triple-quoted comment string can be replaced with blank space when it appears by itself between function definitions (and in some other situations not used in this course).
A.3 Functions and Parameters

In this course, we will write programs by defining a collection of functions, such as `response_to`, `patient_name`, and `doctor_interface` in Figures A.1 and A.2. Functions are the basic elements from which we construct algorithms; typically each function expresses one algorithm for one general problem, though sometimes (as with top-down design) several functions are used together for one multi-part algorithm. Thus, functions (and their cousins procedures and methods) play a central role in the construction of software in almost every modern programming language.

The definition of a function does not directly impact the result of the program unless the defined function is actually used. When a function is used (or “called”), it is given information about the problem instance to be solved, which may vary if the program is run more than once or the function is called at several points during a program — we might use the `response_to` function to respond to a temperature of 99.2 one day, and 101.5 the next. The information about the problem instance is referred to as the function’s parameters or arguments. The result that is produced by the function is known as the function’s return value, e.g. the return value for `response_to(101.5)` would be ’You have a fever’.

In Python, a function definition starts with one line made up of the word `def` followed by the function’s name (such as `response_to`), an open parenthesis, a list of zero or more parameters (such as `temperature` — if there are two or more parameters they must be separated by commas), a close parenthesis and a colon. Indented below this first line comes the body of the function — the body expresses the algorithm to be used to produce the result of the function in the terms outlined in Section A.4 below.

The names given to functions, parameters, and local variables (see A.4.7 below) can be any combination of letters, digits, and underscores (the “_”, which can be used to separate words in multiple-word names like `respond_to`), as long as they do not start with a digit. Perhaps the most important technique for making a program easy to read is the choice of concise yet clear function names. Since clarity is an important tool for avoiding mistakes, and finding mistakes is somewhat harder and much more unpleasant than writing programs, many experienced programmers put great effort into choosing names.

The parameter list shows what information must be provided for the function to be used, and provides a name that will be used to refer to each piece of information within the function. For example, the `response_to` must be given one piece of information, which we will refer to as the `temperature`; the `power` function from Chapter 3 must be given two pieces, the first will be called the `base` and the second the `exp` (meaning exponent).

Substitution Rule: The definition of a function that is not used anywhere else in the program can be replaced with blank space — typically this is possible only after we have replaced each use of the function with copies of the body, as in Section A.4.3.

A.4 Statements and Expressions

The bodies of the Python functions we will write are comprised of statements, which may include other smaller program elements. The term expression is used to refer to a program element that has a value, such as ’Pat’, `temperature`, `temperature >= 99.8` (which has a value of `True` or `False`), `exp-1`, or `power(base, exp-1)`. A statement shows what is to be done with a value — for example, to state that a value should be produced as the result of a function, we use a return statement, as in `return ’Pat’` in the `patient_name` function.
A.4.1 return statements

A return statement, which has the form return <expression>, is used to define the result of a function, i.e., the final answer to the algorithm expressed in the function. For example, the body of the patient_name function consists of the single statement return 'Pat', so the function always produces the string 'Pat'. If we want to have the program use a different name, we must change this function.

Substitution Rule: For a block of text of the form [[return <expression>]] (i.e., something arising from the substitution of a function call (as we will see in Section A.4.3), and in which other statements such as if have been removed by substitution already), we can substitute (<expression>). For example, the text

\[
\text{print 'Hello', [return 'Pat'], 'I hope you are feeling well.'}
\]

(which would result from applying the rules of Section A.4.3 to Figures B.1 and A.2) would become

\[
\text{print 'Hello', ( 'Pat' ), 'I hope you are feeling well.'}
\]

Note that it is also legal to distribute arithmetic over [], as long as it is applied to every return from that [] (but not any other [] contained within it), and parentheses are added as necessary. In other words, 3 * [return 4+5] can be rewritten [return 3*(4+5)], just as 3 *(4+5+6) can be turned into the equivalent 3 · 4 + 3 · 5 + 3 · 6.

A.4.2 Literal value expressions

Some algorithms refer to a specific value, such as 99.8 in our response_to function. When such values are typed directly into the function, they are known as literal values. Literal values can represent various types of information, such as the integer 157, the real number 7.125, the boolean constants True and False, or the string of text 'You do not have a fever'.

Python, like most programming languages, uses a limited-precision floating-point approximation to record real numbers. This is like “scientific notation” for large numbers, where 60220000000000000000000000000000 is written 6.022 · 10^23, except that superscripts aren’t available in Python so we write 6.022e23 (as in most other programming languages). However, there are limits to the number of digits recorded for the exponent (e.g., the “23” part of 6.022 · 10^23) and mantissa (e.g., the “6.022” part of 6.022 · 10^23). If a calculation produces a truly huge number like 3.71 · 10^2260199, Python simply records inf, meaning something too large to record. For a number that requires many digits of accuracy in the mantissa, such as 3.0000000000000000000012 (i.e., 3+1.2e-21) Python simply records some of the digits (i.e., 3.0 in this case). Furthermore, the digits are recorded in the binary system rather than base ten, so some numbers that could be written exactly with a limited number of digits in base ten, like \(\frac{1}{3}\), end up being approximated (as \(\frac{1}{3}\) would be, in either base ten or binary). These limits stem from the manner in which the computer hardware itself records real numbers, and can be found in most programming languages and calculators. This approximation is suitable for most applications; overcoming its limitations is possible but beyond the scope of this course.
Substitution Rule: We will not generally be substituting anything for literals, since the point of our substitution process will often be to produce a single literal value. We could, of course, make any legal replacement, such as turning 17 into 42-25 or \[ \text{return } 17 \].

A.4.3 Function call expressions

A function that has been defined (or imported) can be used by giving its name and filling in expressions for each of the parameters. This kind of expression is known as a function call; it is the basic mechanism by which our program makes use of an algorithm that we have expressed as a function. When a function is called, the values given for the parameters are run through the algorithm expressed by the function’s body, and the return value is provided to the point in the program that made the call. For example, the function doctor_interface of Figure A.2 makes use of the fever-diagnosing algorithm response_to with the function call response_to(entered_temp); depending on the value provided for entered_temp, the function will return either ‘You do not have a fever’ or ‘You do have a fever’.

Functions can call themselves, as shown in Chapter 3’s power function, which calls upon power(base, exp-1). This sort of self-referential function call is known as a recursive call.

Substitution Rule: For any use of a function (e.g. the function call response_to(entered_temp) in the doctor_interface function of our example), we can substitute in a copy of the function’s body (i.e., everything indented below its def line) after first substituting the given values for the parameters in the copy of the body as per Section A.4.9 (i.e., changing temperature into entered_temp for the call above, or changing base into 3.0 and exp into 3 for the call power(3.0, 3) in Chapter 3). To keep track of the text that came from the function body, we surround the body with the symbols [ ] and [ ], and make use of these symbols when substituting for return statements in Section A.4.1. Note that this substitution can be made when a person (or the Python language system) is reasoning about/processing the program, but if we make this substitution in our editor and give it back to the Python system, it will not be accepted (this is why we have chosen the [ ] and [ ] symbols, which cannot be easily entered into a Python program).

For example, we could substitute the body of the patient_name function into the first line of doctor_interface, turning

\[
\text{print } 'Hello', \text{ patient_name()} , 'I hope you are feeling well'\]

into

\[
\text{print } 'Hello', [ \text{return } 'Pat' ], 'I hope you are feeling well'\]

and apply the return rule (Section A.4.1) to produce

\[
\text{print } 'Hello', ( 'Pat' ), 'I hope you are feeling well'\]

and drop the redundant parenthesis

\[
\text{print } 'Hello', 'Pat', 'I hope you are feeling well'\]

A simplified version of the call to response_to illustrates a function call with a parameter:
print response_to(entered_temp)

becomes, with our function call rule,

print [  
    precondition(is_number(entered_temp))  
    if entered_temp >= 99.8:  
        return 'You have a fever'  
    else:  
        return 'You do not have a fever' ]

Then, if entered_temp had the value 103.7, we would use the rules for variables (Section A.4.9), comparison operations (Section A.4.5), if (Section A.4.8), preconditions (Section A.4.10), and return to turn this into print 'You have a fever'.

If the body of the called function defines a variable with the same name as a variable in the calling function, this substitution would make us violate our rule about having only one definition of any given variable name in any one function (Section A.4.7). To avoid this, we will distinguish two variables of the same name by adding subscripts. For example, if the response_to function, like the doctor_interface, had a variable named entered_temp, we would write entered_temp₁ for the variable from doctor_interface and entered_temp₂ for the variable that came in from response_to, when making the substitution above.

An alternative substitution rule for function calls involves turning each parameter into a local variable (Section A.4.7) defined to have the value given in the call. For the example call of response_to above, the parameter temperature is given the value entered_temp, so the alternative rule would produce

print [  
    temperature = entered_temp  
    precondition(is_number(temperature))  
    if temperature >= 99.8:  
        return 'You have a fever'  
    else:  
        return 'You do not have a fever' ]

A.4.4 Arithmetic expressions

Numeric values can be combined with addition (+), subtraction (-), multiplication (*), division (/), and exponentiation (**). The result is usually what you would expect based on the rules of arithmetic, so for example a program in which we had written ((3+4)*2) would produce the same answer if we replaced that expression with 14. Python, like most programming languages, follows the “PEMDAS” order of evaluation of arithmetic, so 3+4*2 would be 11, not 14.

Note that, for some versions of Python (and many other languages), when the numerator and denominator of a division are both integers expressed without a decimal point, the division operation always produces an integer result. For example, 14/5 is 2, not 2.8. Traditional results can be ensured by introducing a non-integer number or writing an integer with an explicit decimal point: 14.0/5 yields 2.8 on all versions of Python, as do 14./5 (though this is generally avoided because it looks too much like 14/5) and (14*1.0)/5. This third form is particularly useful if numerator and denominator are both variables, e.g., x/y is often written (x*1.0)/y if x and y might both be integers and “true division” is required.

Python’s // operation always produces the integer part of the quotient, and the % operation produces the remainder (e.g. 14.0//5.0 is 2.0 and 14.0%5.0 is 4.0).
Note that Python does not understand the mathematical shorthand of writing values next to each other to multiply them, so \(2a\) (i.e., \(2 \cdot a\)) must be written out \(2*a\).

**Substitution Rules:** Except for the subtleties of integer division, traditional mathematical substitution can be performed except for the addition or removal of a decimal point, e.g. “.0”. When either operand of an operation is a non-integer or an integer written with an explicit decimal point (e.g. \(3.0\)), then the result must include the decimal point to ensure that other division operations will work as expected.

### A.4.5 Comparison expressions and Boolean operators

Numeric expressions can be compared for equality (==), disequality (written either <> or !=), or magnitude (<, <=, >=, >). For example, \(1==1\), \(4<5\), \(7.1>=3\), and \(1<>2\) all produce True, while \(1==2\), \(6<=3.14\), and \(1<>1\) all produce False. Do not confuse ==, which is used for comparison, with =, which is used to provide values for variables. Beware of comparisons involving real numbers, especially == and !=, since rounding and representation errors can affect the result (see Section [A.4.2]).

The Boolean values (True and False) can be combined with the logical and and logical or operations, or negated with not.

Python’s `isinstance` and `type` functions can be used to make comparisons involving the type of a value (e.g., whether a value is an integer, a floating-point representation of a real number, a string etc.), but in this course we will use the simpler `is_integer`, `is_number`, and `is_string` functions from `logic.py`. For example, `is_int(17)`, `is_number(100.3)`, and `is_string('cat')` all produce True, and \(\text{is_int}(17.5)\), \(\text{is_number}('100.3')\), and \(\text{is_string}(100.3)\) all produce False. Note that Python (like most programming languages) does not consider an integer that is being represented inside the computer by a floating-point approximation of a real number to be an actual integer, so `is_integer(17.0)` will give False!

**Substitution Rules:** Comparison expressions can be replaced with True or False and transformed according to the usual mathematical rules, including among other things: not False can be replaced with True and not True with False; True or <expression> can be replaced with True, and False or <expression> with the <expression>; and False and <expression> can be replaced with False, and True and <expression> with the <expression>.

### A.4.6 Strings and string operations

Strings of text are used represent a literal sequence of characters (e.g., letters, numerals, punctuation). Literal string values are indicated by quotation marks at the beginning and end; text between the quotation marks is not evaluated in the usual way, so ’1+1’ is not the same as ’2’, and neither of those strings is the same as the integer 2 (which is the same as 1+1). Strings are often used to represent text that will be printed to communicate with the user of a program, as with the string ’You do not have a fever’ in Figure [A.1].

Python allows several types of quotation marks, for example the traditional marks, e.g. "cat". Python also allows the use of apostrophes as quotation marks, e.g. ’dog’ (depending on the font in use on a computer, the apostrophe may be more vertical and less curved than in this textbook); when apostrophes are used in this way they are generally referred to as ‘single quotes’. Strings can also be surrounded by sets of three regular quotes, e.g. """mouse"""" — this last notation can be used for multi-line strings.
Regardless of the notation used, a string counts as a value in the program (see A.4.2), just as a literal number like 5 or 99.8 would. Strings can be passed as parameters to functions, returned from functions, or named with a local variable. Many of the operators that can be used with numbers can also be applied to strings, though some of these have slightly different meaning, for example '1'+'1' is '11', not '2'. Python also provides some operators and functions that can be applied to strings but not numbers.

The quotation mark itself is not considered part of the string, so if we print a string (i.e., by writing print 'dog'), the quotation marks are not printed. Thus, 'dog' and "dog" are the same string, and the expressions 'dog' == "dog", 'dog' == """dog"""", etc. produce True. The different marks are useful to allow apostrophes or quotation marks in strings (as in "Bob's dog" or 'I said, "Ice cream," not "I scream."'), or to distinguish multi-line strings (with triple quotation marks). In this course, we will use triple quotes primarily as a kind of multi-line comment: if we simply put a string in the middle of a Python function or file, but don’t incorporate this string into any other value, it will not have any impact on the result (as would be the case if we wrote any other value, such as 17, on a line by itself in a Python function).

**Substitution Rules:** The + operation, when applied to strings, concatenates the strings (i.e. combines them end-to-end). For example, 'dog'+'cow' is equivalent to 'dogcow'. Note that concatenation does not automatically introduce any space, so if we want the answer 'dog cow' we could write 'dog'+ 'cow', or of course just 'dog cow'. Concatenation is usually applied to variables or other more complicated expressions rather than specific strings (except in examples illustrating how concatenation works). For example, if we had variables first_name and last_name (perhaps with the values 'Dave' and 'Wonnacott'), we could define a variable greeting = 'Hello, ' + first_name + ' ' + last_name to produce the single string 'Hello, Dave Wonnacott'. (This approach could be used to modify the doctor_interface function to eliminate the unwanted blank space before the comma on the last line of Figure A.3.)

String expressions can also be compared with the comparison operators of Section A.4.5. The == operator returns True if the two strings are the same (including letters, numerals, punctuation, etc., and even the case of letters (e.g. 'Dave' is not the same string as 'daVe')). Magnitude comparisons (via < and the related operators) are defined in terms of “lexicographical order” (i.e. dictionary order), except that all lower-case letters come after all upper-case letters — for example, the word cat comes before dog in the dictionary, so 'cat'<'dog' produces True and 'dog'<'cat' produces False, but 'Cat'<'dog' and 'Dog'<'cat' both produce True.

The length of a string can be determined with Python’s len function. The length is the number of characters (letters, numerals, punctuation, spaces and other formatting information) not counting the quotation marks used to indicate the ends of the string, so if the variable greeting were defined as above, len(greeting) would be 21.

Sub-parts of a string can be extracted with the subscript operator, i.e. [ ], which is placed after the string and contains a number or range of numbers between the “[” and the “]”, indicating which elements of the string are to be chosen. A single non-negative integer is used to identify a single character, starting with 0 for the first element of the string. For example, first_name[0] would be 'D' given the definitions above, and first_name[2] would be 'v'; a range is indicated with a colon between numbers giving the starting element and one more than the final element. For example, first_name[0:2] would be 'Da'. When one end of a range is omitted, it runs all the way to the end of the string, so first_name[:2] would be 'Da' and first_name[1:] would be 'ave'. Finally, negative integer indices can be used to identify elements of a string from the right end rather than the left, with s[-x] being equivalent to s[len(s)-x] when -x is a negative integer, for example first_name[-2] is 'v'. Note that -0 is not a negative integer, so first_name[-0] is 'D'.

We can also check to see if a character (or sequence of consecutive characters) is contained within a string, using the word in: The expressions 'v' in 'Dave' and 'ave' in 'Dave' both produce True, and 'w' in 'Dave', 'ae' in 'Dave', and 'va' in 'Dave' all produce False.

A.4.7 Definitions of parameters and local variables

Variables are names that are used to label values. The variables we use in our starting subset of Python can be classified as parameters and local variables. Parameters, such as temperature in Figure B.1 and base and exp in Figure 3.2, are listed in the function’s definition and are used to communicate information from the calling function. Local variables, such as entered_temp and base_to_the_exp_minus_one, are used to name information within the body of one function with the = sign (as with entered_temp in doctor_interface).

The value of a variable can change from one use of a function to another. For example, if the user enters 83.5 for the program of Figures B.1 and A.2, then the variables entered_temp (in doctor_interface) and temperature (in respond_to) will both have the value 83.5. If the user runs the program again and enters 102.3, then these variables will have the value 102.3 during that run. Similarly, if a function is called several times during a single run of a program, as is the case with Chapter 3’s power example, variables will have different values for each call (e.g. base and exp being 3.0 and 3 for the call power(3.0, 3), and 0.9 and 5 for power(0.9, 5)). This principle holds even when the function calls upon itself: During the execution of the call power(3.0, 3), the values of base and exp will be 3.0 and 3. In the middle of this calculation, power calls upon itself to find 3.0², and during this calculation the name base will refer to 3.0 and exp will refer to 2. When the computer arrives at the answer 9.0 for 3.0² and returns to complete the work on power(3.0, 3), it will once again use the values of 3.0 and 3 for base and exp. This distinction between several variables is reflected in our use of subscripts to avoid confusion. In this example, we would use exp₁ for the outer exp variable (with the value 3) and exp₂ for the inner exp variable (with the value 2).

Although the full Python language allows us to provide multiple different definitions for a single name, this practice will hinder our substitution rules for reasoning about programs. Thus, in our starting subset, we will provide only one definition for any given name in any particular function (names from other functions are not considered, so it doesn’t matter whether two different functions have a variable of the same name). For the purposes of this “only one definition” rule, we do allow conditional definitions as in Section A.4.8.

Some programmers like to define each variable as early in the function as they can; others prefer to define each variable as late as possible (perhaps right before the first use). Looking ahead, Figure A.4 shows the “define-late” style, and Figure A.5 shows the “define early” style (these figures also differ in their use of if). We will mix these styles freely in an effort to make each individual program as clear as possible.

Python requires that the variable being defined must be on the left side of the =, that the definition must precede any use of the variable (Section A.4.9), and that all uses must be contained in the same program element as the definition (e.g., indented under the same function definition, or under the same part of the same if, etc.). This means that several common mathematical styles are not legal in Python. Specifically, in mathematics the equations \( x = 3 \) and \( 3 = x \) are equivalent, but Python would accept only the \( x=3 \) version; in mathematics we could write either \( c=3.0 \cdot 10^8 \) in \( E=mc^2 \) or \( E=mc^2 \), where \( c=3.0 \cdot 10^8 \). Python requires that the \( c=3.0e8 \) appear on a separate line before the \( E=mc^2*2 \). In mathematics and Python alike, if we happen to refer to the parameter of a function \( f \) as \( x \), and also to the parameter of another function \( g \) as \( x \), it is understood that those are not necessarily the same \( x \) (for example, if \( f(x) \) is defined as \( 2 \cdot g(x+1) \)). Our one exception to the “in the same program element” rule is that we can use a vari-
able after an if-else sequence if every possibility either defines that variable or returns a final result.

Substitution Rule: Variable definitions are important because they tell us what value to substitute for uses of the variable (Section A.4.9). Once we have performed this substitution for all uses of a local variable, we can remove the definition (i.e., substitute empty space for it). Definitions of parameters are removed during our function call substitution rule (Section A.4.3).

A.4.8 Conditional statements and expressions (if)
The words if, elif, and else can be use to select among several possible sets of statements that may be appropriate. The response_to and doctor_interface functions of Figures B.1 and A.2 illustrate the basic use of if and else: the lines indented immediately below the if are selected if the expression is True, and the lines indented immediately below the else are selected if the expression is False. Don’t forget the colons at the ends of the if and else lines, since Python will give an error if they are not there.

Note that an if-else pair, together with their indented statements, themselves constitute a statement. Thus, a complete if-else structure can be used in either branch of an outer if-else, to further refine the choice made by the outer test. Figure A.4 shows a variation on Figure B.1 in which fevers are further classified as high or mild, and non-fever conditions are checked for plausibility.

""" Come up with a response to a reported temperature (fancy version)
    This version detects absurdly low temperatures and extreme fever. """

from logic import *

def response_to(temperature):
    precondition(is_number(temperature))
    fever = 99.8
    if temperature >= fever:
        high_fever = 103
        if temperature >= high_fever:
            return 'You should see a specialist'
        else:
            return 'You have a mild fever'
    else:
        believable = 92 # See Ch.3 of K.S. Robinson’s "Antarctica", Bantam 1998
        if temperature < believable:
            return 'Come on, quit kidding around'
        else:
            assert(temperature>=believable and temperature<fever)
            return 'You seem fine'

Figure A.4. A more subtle diagnosis using if within if, with variables defined only when needed
It is also possible to arrange a collection of tests linearly rather than hierarchically, by combining \texttt{if}, one or more \texttt{elif}'s, and a final \texttt{else}. Figure A.5 illustrates this option. Only the statements indented below the first test that produces \texttt{True} will be selected (or those below the else if none of the tests is \texttt{True}). For the purposes of this course, we will choose between the heirarchical and linear notations primarily for purposes of clarity, trying to make the structure of the program reflect the thinking style (heirarchical or linear) that best fits the approach to solving the problem.

\begin{verbatim}
""" Come up with a response to a reported temperature (fancy version)
    This version detects absurdly low temperatures and extreme fever. """

from logic import *

def response_to(temperature):
    precondition(is_number(temperature))
    fever = 99.8
    high_fever = 103
    believable = 92    # See Ch.3 of K.S. Robinson's "Antarctica", Bantam 1998

    if temperature >= high_fever:
        return 'You should see a specialist'
    elif temperature >= fever:
        return 'You have a mild fever'
    elif temperature >= believable:
        return 'You seem fine'
    else:
        assert(temperature < believable)
        return 'Come on, quit kidding around'

Figure A.5. A more subtle diagnosis using \texttt{if} and \texttt{elif}, with variables defined early

The Python languages allows a number of uses of \texttt{if} that fall outside our starting set; these other forms will be introduced in later chapters (e.g. Chapter 7), but will require a new approach to reasoning about programs (introduced in Chapter 8). To remain within the starting set of Python features:

- we \textit{always} include the final \texttt{else} the end of each \texttt{if-else} or \texttt{if-elif-else} sequence, and
- we ensure that all the “branches” (sets of indented statements) end with a \texttt{return} statement.

(Note that, since we do not reason about our user interface functions in the same way, they don’t always follow these rules.)

One consequence of the above rules is that the value given to variable in one branch will not be available in other branches. For example, the \texttt{high\_fever} variable in Figure A.4 will not be available if \texttt{temperature < fever}.)
It is also possible to use the word if to choose between two values for a variable, just as a mathematician might define the absolute value via the equation

$$abs(x) = \begin{cases} x, & \text{if } x \geq 0 \\ -x, & \text{otherwise} \end{cases}$$

or the sentence “the absolute value of $x$ is $x$ if $x$ is positive, or $-x$ if $x$ itself is negative”. The Python equivalent would be to produce the value $x$ if $x \geq 0$ else $-x$. This use of the word if is referred to as an “if expression” or “conditional expression” rather than an “if statement” or “conditional statement”. Conditional expressions can appear anywhere a value can appear in a Python program; they can contain only other values or expressions, not statements such as variable definitions or if statements. Combining multiple decisions into one very complex conditional expression typically produces programs that are almost impossible to figure out, so we will illustrate (in Figure A.6) only the simple diagnosis function in this form.

**Substitution Rule:** A statement of either the form

```python
if True:
    <statements>_1
else:
    <statements>_2
```

or

```python
if True:
    <statements>_1
elif <expression>:
    <statements>_2
...
else:
    <statements>_n
```

can be replaced with just

```python
from logic import *

# Algorithm to come up with a response to a reported temperature
def response_to(temperature):
    precondition(is_number(temperature))
    return 'You have a fever' if temperature >= 99.8 else 'You do not have a fever'
```

**Figure A.6.** Diagnosis Function of Figure B.1 Rewritten to use a Conditional Expression

```python
from logic import *

# Algorithm to come up with a response to a reported temperature
def response_to(temperature):
    precondition(is_number(temperature))
    return 'You have a fever' if temperature >= 99.8 else 'You do not have a fever'
```
A statement of the form

```
if False:
    <statements>₁
else:
    <statements>₂
```

can be replaced with

```
<statements>₂
```

A statement of the form

```
if False:
    <statements>₁
elif <expression>:
    <statements>₂
```

can be replaced with

```
if <expression>:
    <statements>₂
```

Finally, `<expression>₁ if True else <expression>₂` can be replaced with `<expression>₁`, and `<expression>₁ if False else <expression>₂` can be replaced with `<expression>₂`.

### A.4.9 Variable use expressions

When the name of a variable appears in an expression, such as `temperature >= 99.8`, the value of that variable is used in the calculation of the expression’s value. Note that both parameters and local variables can only be used within the function body in which they were defined — for example, the name `temperature` cannot be used in the `patient_name` and `doctor_interface` functions, and `base`, `exp`, and `base_to_the_exp_minus_one` (from Figure 3.2) cannot be used in `simple_power_ui`. Furthermore, variables can only be used after the point of definition, and a variable defined inside one branch of an `if` statement cannot be used in a different branch.
Substitution Rule: For the programs that give only one definition for any given name in a particular function and do not use input, we can replace any use of a variable name with the expression used to define it. For example, in Figure A.4, we can transform

```
high_fever = 103
if (temperature >= high_fever):
```

into

```
high_fever = 103
if (temperature >= 103):
```

and, since there are no other uses of `high_fever`, this can be turned into just

```
if (temperature >= 103):
```

A definition that occurs within a branch of an if statement can be substituted in this way for other statements that follow in the same branch. (Alternatively, the if statement can first be removed via the substitution rules of Section A.4.8 and then the rule for non-conditional definitions can be applied.)

A.4.10 Preconditions, postconditions, and progress conditions

When a function is only defined for some parameter values, it is appropriate to identify the acceptable parameter values in a precondition. For example, our `power` function requires positive integer exponents, and thus begins with `precondition(is_integer(exp) and exp > 0)`.

The precondition statement can only be used if it has been imported from the logic library, e.g. via `from logic import *`.

Preconditions provide information for other programmers about the correct usage of a function, in a form that can be checked in the running program. By default, the running program checks the appropriate precondition each time a function is called, and stops and prints an error if the precondition is not true. It is also possible to turn off precondition checking to avoid wasting time on useless checks for programs that have been formally verified, or when the program is being distributed to users who would not know what to do about the error message anyway.

Functions can also define what counts as a correct answer by declaring a postcondition, using our postcondition function. For example, if we were writing a function to compute the square root of a parameter `x`, to an accuracy of 1%, our algorithm could produce the answer in a local variable named `sqrt_of_x` and then end with the lines

```
postcondition(sqrt_of_x * sqrt_of_x * 0.99 <= x and
              x <= sqrt_of_x * sqrt_of_x * 1.01)
return sqrt_of_x
```

Functions that are recursive (i.e., that call upon themselves) can use our progress function to describe the manner in which the recursive calls progress toward a solution. Our function requires a specific form of progress measure — the parameter to progress must be a nonnegative integer quantity that is always smaller in the calling function than the caller (such as `exp` in our `power` function). Our function does not actually check progress in all cases, but it can still be used to express the progress expression for other programmers.

Functions with loops (see Section A.4.14) can use our loopInvariant and other statements related to loop correctness, as described in Chapter 8.
Finally, functions may use the standard Python `assert` to indicate facts that are crucial for correct operation of the function and must be true because of the parts of the function that have come before the assertion. For example, given the precondition for `power`, and the fact that the `else` clause of the `if` will not be run when `exp` is 1, we can assert `exp > 1` in the `else` clause. Similarly, in Figure A.4, we know from the surrounding `if` structures that temperature must be at least the `believable` temperature and less than the `fever` temperature at the point where we return the string “You seem fine”. The assertion re-states these facts on a single line right before the `return` statement, for the benefit of anyone trying to read and understand the program.

In some cases, these logical statements cannot be expressed easily in Python, especially since it is not appropriate to use a function to check itself (e.g., we can’t write `postcondition(base * base_to_the_exp_minus_one == power(base, exp)` as the postcondition for `power`, since it doesn’t really make sense to use a function to check itself, and furthermore this statement would cause an infinite recursion). In such cases, the logical statement should be expressed as a comment.

**Substitution Rule:** For program written using the Python features from this appendix, we can remove `assert(True)`, `precondition(True)`, or `postcondition(True)` without affecting the result of the program. If we get to `assert(False)`, `precondition(False)`, or `postcondition(False)`, we need to stop substituting and figure out what is wrong with the program.

If these logical statements are combined with certain Python features not within our starting set, such as a call to a function that changes a “global variable”, very subtle problems can occur. (But if you stick starting set, that won’t be a problem.)

### A.4.11 Input and output

Parameters and return values are used to communicate information among the functions of a program. In most cases, the program also needs to communicate with the user who is running the program. There are a variety of ways for a program to communicate with a user, including both text-based interfaces (in which the program prints text and the user types on the keyboard) and graphical interfaces (in which the program produces a combination of text and diagrams, and the user may type on the keyboard or draw or point with a mouse).

In this course, we will primarily build text-based interfaces, though some lab exercises may be distributed with pre-made graphical interfaces. The `doctor_interface` function of Figure A.2 is an example of a text-based user interface (for the algorithm expressed in the `respond_to` function).

Information can be printed for the user by calling upon Python’s `print` function. Print is given a comma-separated list of things to be printed, and prints them for the user. The items are printed in order on one line, with spaces separating them (as in Figure A.3).

A program can wait for the user to enter information by calling Python’s `input` function. The `input` function takes one parameter, known as a `prompt`, which is automatically printed to let the user know it’s time to enter some information. After the prompt is printed, the computer waits for the user to type some text, such as a number, and then press the return key; the value of the `input` expression is then that text (in Python 2 and earlier, the text is automatically interpreted as a Python value). For example, the line `entered_temp = float(input('Tell me your temperature '))` would first produce the prompt “Tell me your temperature” on the user’s computer screen, and then wait for the user to type something (such as 99). The value of the entered text (e.g., the string “99”) is returned by the `input` expression, and the function `float` interprets as a number. Thus, after the completion of `input` and `float`, the Python system treats this line as it would have treated `entered_temp = 99.0`.
**Substitution Rule:** Substitution for `input` is rather tricky — if an input is used to define a variable (as for `entered_temp` in Figure A.2), this can interfere with our substitution rules. The problem will be most dramatic if the variable is used repeatedly — if we were to substitute the use of `input()` for each use of a variable, the user would have to type their input each time the variable is used. This can create confusion for the user and break an otherwise-correct program. A perfectly acceptable mathematical substitution could thus break the program! This problem gets at the heart of the difference between software and mathematics, a topic that we discuss in detail in Chapter 7.

To prevent this problem in functions involving input or output, we will do all input and output in our user interface functions, and perform substitution only on functions that are not part of the user interface.

### A.4.12 Separating input and output from computation

In our examples, we have placed all the interesting computation in one function (e.g. `respond_to` or `power`), while all of the input and output is performed by another. This separation is not required by the Python language, but we will follow the convention of using some functions for computation and other functions for the interface. This convention both facilitates reasoning about the program (as noted above, substitutions can be a problem when used with input), and also makes it easier to re-use computational functions in other programs.

### A.4.13 Errors

Some operations, such as division by zero (e.g., `3/0`), arithmetic involving inappropriate or incompatible operands (e.g., `'Hello' * 'Bob'` or `'Bob' / 17`), a failed logical statement such as a precondition, or infinite recursion without any return or output, do not produce a legal answer (most Python systems will produce an error message, possibly after a long delay in the case of infinite recursion).

Typically the use of a function containing such an error is itself an error (for example, `4*x/0`, `x/0-x/0`, and a call to a function that computes `x/y` with 0 for the y parameter are all errors, because they all contain the erroneous `x/0`). However, there are contexts in which a possibly erroneous expressions can exist within a Python program without causing an error. It is fine to have such an expression

- within a branch of an `if` with a `False` condition (e.g., `x/y` is legal inside an `if` that tests `y<>0`),
- within the second operand of “`True or`” or “`False and`” (e.g., `y<>0 and x/y<17.5`), or
- in a function that is never called

Note that most errors trigger a Python `exception`, an important tool for the development of large-scale computer software. Exceptions are beyond the scope of this course, but anyone who plans to do serious Python programming should research and learn to use Python’s `try/except` and `raise` statements.
A.4.14 Loops and multiple definitions

Our starting set only includes a tiny fraction of the full Python language, covering just enough to let us write functions expressing a wide variety of algorithms. Experienced programmers may be surprised by the lack of loop statements, such as Python’s for and while statements, and by the restriction of only one definition per variable. These valuable programming tools are essentially a shorthand to provide a concise expression for certain restricted forms of recursive functions. These statements are less general than recursion (in that any loop can be converted to a recursive form, but many forms of recursion cannot be turned into loops without the use of programming techniques and language features that are beyond the scope of this course). Furthermore, the formal verification techniques for working with programs that involve loops or multiple definitions of a single variable are much more complicated. For these reasons, we begin with a general discussion of recursive functions, and return to the special cases corresponding to loops and multiple definitions (and the tools for verifying them) in Chapters 7 and 8.
Appendix B
Proof Techniques

This appendix is meant to review techniques for developing logical “direct proofs”, for anyone who has not worked with these techniques recently. The relationship of direct proofs to algorithms and functions is discussed in Chapter 5; for a full discussion of techniques for mathematical proofs in the context of computer programs, refer to Chapters 0-3 of David Gries’ “A Logical Approach to Discrete Math”, or Chapters 2 and 3 of Gries’ “The Science of Computer Programming”. For more information about proofs involving the properties of integers, see Section 3.2 of James Anderson’s “Discrete Mathematics with Combinatorics”. We will not rely on the mathematical properties of real numbers, since we will not see any way to accurately reproduce these properties in a Python program (since limited-precision floating point numbers do not have all the properties of real numbers, as per Section A.4.2).

B.1 What is a proof?

A logical proof connects one set of given facts (the premise) to another (the conclusion) using a set of agreed-upon reasoning rules. Proofs can be constructed in several ways, including direct proofs, indirect proofs, and inductive proofs. In this course, we will focus almost entirely on direct proofs. A direct proof is made of a sequence of steps, each of which involves the use of a rule to rewrite the set of facts — the step should state which rule was used, and (if it is not clear from context) how the rule was used.

For example, consider the axiom of “distribution of multiplication over addition for integers”, written \( i \cdot (j + k) \leftrightarrow (i \cdot j) + (i \cdot k) \) and appearing as Rule B.3.7 below. This axiom can be seen as a rule that lets us rewrite something of the form on the left into the form on the right (or vice-versa). We could convert \( 3 \cdot (n + m) \) into \( (3 \cdot n) + (3 \cdot m) \) by matching \( i \) with \( 3 \), \( j \) with \( n \), and \( k \) with \( m \). Rules written with \( \leftrightarrow \) below can be used in both directions, so this same rule, with the same matching of variables, also shows that \( (3 \cdot n) + (3 \cdot m) \) can be converted into \( 3 \cdot (n + m) \).

Variables in rules can be matched with individual terms (such as \( 3 \) and \( n \) above) or more complicated expressions — this Rule B.3.7 lets us convert \( (r - 7 \cdot s) \cdot ((4 \cdot r) + (9 \cdot t - 5)) \) into \( ((r - 7 \cdot s) \cdot (4 \cdot r)) + ((r - 7 \cdot s) \cdot (9 \cdot t - 5)) \) by matching \( i \) with \( (r - 7 \cdot s) \), etc.. However, care must be taken to group complete parts of an expression when unnecessary parenthesis have been omitted. In this course we will not usually write parentheses that are redundant with the traditional rules of precedence (the “PEMDAS” order (“parenthesis, exponents, multiplication and division, addition and subtraction”) extended so that these arithmetic operations are followed by comparison operations (\(<,=,\text{etc.}\) then the Boolean operations \( \land \) (logical conjunction, a.k.a. “and”) and \( \lor \) (logical disjunction, a.k.a. “or”)), so \( 3 + 4 \cdot 5 > 1 \lor 6 < x \lor x + y \geq z - 17 \) will be taken to mean \( (((3 + (4 \cdot 5)) > 1) \land (6 < x)) \lor ((x + y) \geq (z - 17)) \). It is surprisingly easy to slip up when trying to keep track of unnecessary parenthesis when transforming expressions via rules, for example accidentally...
turning $3 \cdot (n + m) \cdot 5$ into $(3 \cdot n) + (3 \cdot m) \cdot 5$, rather than $((3 \cdot n) + (3 \cdot m)) \cdot 5$. It is often safest to simply put extra parenthesis around whatever matches each variable in a rule, and around whatever matches the entire rule, before applying it.

In the context of this course, the substitution rules (or “axioms”) for Python programs will be taken from Appendix A. Where Appendix A refers to “traditional mathematical rules” you may consult the later sections of this appendix or other texts such as those cited below, but with traditional mathematical symbols converted into Python notation, e.g., “\(\geq\)” into “\(\geq\)”, etc. Writing out the fully-detailed description of the proof (see Chapter 5.2.2) can be quite tedious, but it does allow checking of the proof without any use of insight or guessing.

The example in Figure B.1 gives, at the fully-detailed level, one proof that \((1 < a) \land (a < b)\) or \((1 < b) \land (b < a)\) can be turned into \(((1 < a) \land (1 < b)) \land ((a < b) \lor (b < a))\) (or, more informally, that “either \(a\) is between \(1\) and \(b\) or \(b\) is between \(1\) and \(a\)” means \("a\) and \(b\) are different numbers

\[
((1 < a) \land (a < b)) \lor ((1 < b) \land (b < a))
\]

... rewrite with rule B.2.2 used "right – to – left", with \(x\) matching \((1 < a) \land (a < b)\) ...

\[
(((1 < a) \land (a < b)) \land ((1 < a) \land (a < b))) \lor ((1 < b) \land (b < a))
\]

... rewrite with rule B.3.2[1], matching \(i\) to \(1\), \(j\) to \(a\), and \(k\) to \(b\) in the \(2^{\text{nd}}\) \((1 < a) \land (a < b)\) ...

\[
(((1 < a) \land (a < b)) \land (1 < b)) \lor ((1 < b) \land (b < a))
\]

... rewrite with rule B.2.6 used right – to – left, matching \(x\) to \(1 < a\), \(y\) to \(a < b\), and \(z\) to \(1 < b\)...

\[
((1 < a) \land ((a < b) \land (1 < b))) \lor ((1 < b) \land (b < a))
\]

... rewrite with rule B.2.4 matching \(x\) to \(a < b\) and \(y\) to \(1 < b\)...

\[
((1 < a) \land ((1 < b) \land (a < b))) \lor ((1 < b) \land (b < a))
\]

... rewrite with rule B.2.6, matching \(x\) to \(1 < a\), \(y\) to \(1 < b\), and \(z\) to \(a < b\)...

\[
(((1 < a) \land (1 < b)) \land (a < b)) \lor ((1 < b) \land (b < a))
\]

... and now, do the same sequence of rewrites on the right of the \(\lor\)...

... rewrite with rule B.2.2 used "right – to – left", with \(x\) matching \((1 < b) \land (b < a)\) ...

\[
(((1 < a) \land (1 < b)) \land (a < b)) \lor (((1 < b) \land (b < a)) \land (1 < a))
\]

... rewrite with rule B.3.2[1], matching \(i\) to \(1\), \(j\) to \(b\), and \(k\) to \(a\) in the \(2^{\text{nd}}\) \((1 < b) \land (b < a)\) ...

\[
(((1 < a) \land (1 < b)) \land (a < b)) \lor (((1 < b) \land (b < a)) \land (1 < a))
\]

... rewrite with rule B.2.6 used right – to – left, matching \(x\) to \(1 < b\), \(y\) to \(b < a\), and \(z\) to \(1 < a\)...

\[
((1 < a) \land ((1 < b) \land (a < b))) \lor ((1 < b) \land (b < a))
\]

... rewrite with rule B.2.4 matching \(x\) to \(b < a\) and \(y\) to \(1 < a\)...

\[
(((1 < a) \land (1 < b)) \land (a < b)) \lor ((1 < b) \land (1 < a)) \land (b < a))
\]

... rewrite with rule B.2.6, matching \(x\) to \(1 < b\), \(y\) to \(1 < a\), and \(z\) to \(b < a\)...

\[
(((1 < a) \land (1 < b)) \land (a < b)) \lor ((1 < b) \land (1 < a)) \land (b < a))
\]

... and now, merge together the two \((1 < a) \land (1 < b)\) terms...

... rewrite with rule B.2.4 matching \(x\) to \((1 < b)\) and \(y\) to \((1 < a)\)...

\[
(((1 < a) \land (1 < b)) \land (a < b)) \lor (((1 < a) \land (1 < b)) \land (b < a))
\]

... rewrite B.2.6 right – to – left, matching \(x\) to \((1 < a) \land (1 < b)\), \(y\) to \((a < b)\), \(z\) to \((b < a)\)...

\[
((1 < a) \land (1 < b)) \land ((a < b) \lor (b < a))
\]

Q.E.D.

**Figure B.1.** An example direct proof, in full detail.

Copyright 2003-2015 David G. Wonnacott; see Page 7 for terms of free use
B.1 What is a proof?

bigger than 1", assuming the use of the informal “i is between j and k” means i isn’t the same as either j or k as well as coming between the lower (j) and upper (k)). This example is based only on the rules from the sections below, and uses mathematical symbols rather than Python notation; descriptions of what has been changed are identified with “...”.

Note that this is one of many possible sequences of steps that can be used to prove this fact using the axioms from the sections below; a different sequence, or a sequence based on a different set of axioms, can produce a different proof of the same result. Note that insights leading to a proof typically do not come in the order of the steps in the finished proof ... Figure B.1 might be the result of a first insight that we’ll want to group together 1 < a and 1 < b in one place, and a < b and b < a in another; this would then lead to an observation that we can introduce a term like 1 < b when we know 1 < a and a < b (from axiom B.3.21), after which the connecting steps are put in place (writers of fiction sometimes describe a similar process, in which a story starts with an idea or image of a key scene that may appear near the end, and the rest of the story is then constructed around this scene).

A proof can be condensed in various ways if it is intended for a human reader rather than an algorithmic proof-checker. For example, the five steps below “and now, do the same sequence...” of Figure B.1 could be condensed into a single step if we can rely on the human reader to actually apply those steps if desired.

A slightly more formal way to avoid repeating a standard sequence of steps is to define a lemma, essentially creating a new substitution rule from one or more axioms that are already accepted. In the context of Figure B.1, we would introduce a lemma to help in our goal of introducing terms like 1 < b when we know 1 < a and a < b; the most general form of this rule would be \((r < s) \land (s < t) \rightarrow (((r < s) \land (s < t)) \land (r < t)))\). Once this lemma has been proved (via axioms B.2.2, B.3.21, B.2.4, and B.2.6 again), the main proof just consists of two applications of the lemma, followed by the last two steps shown in Figure B.1 (see Figure B.2).

When writing for a human audience, it is often acceptable to use simple combinations of two or three rules without formally introducing a lemma. For example, the above use of steps B.2.6, B.2.4, and B.2.6 again simply re-orders three terms connected by ∧, despite the fact that the parenthesis weren’t where we wanted them, turning \(((x \land y) \land z)\) into \(((x \land z) \land y)\).

In this course, we will employ one variation on the traditional direct proof described above, specifically proof by cases. For example, if we wish to prove something involving variables x and y, we might wish to make three simple proofs, one for \(x < y\), one for \(x = y\), and one for \(x > y\), rather than one proof that works in all three situations (this technique can actually be achieved via direct proof and the proper combination of rules of boolean algebra and the definitions of <, =, and >, but it is sometimes easier to think of it as a proof technique). The central point of proof by cases is that it is only valid if you list all the cases — it is not valid to consider only \(x < y\) and \(x > y\) unless we have proven some how that \(x\) can not be \(y\) in the context of the proof we’re creating.

The process of creating a proof may be time-consuming and involve subtle insight into the facts, but the process of checking the proof simply involves making sure that each step really does use the stated rule in the stated way. If two parties agree on the premise and reasoning rules, one of them can create a proof to convince the other of the conclusion. For example, imagine a case in which a client requests that a vendor write a Python function for a specific purpose — if the two can agree on the definition of what counts as correct (i.e., the precondition and postcondition for the function), and on rules for reasoning about Python functions (such as those presented in Appendix A), then the client could also ask the vendor to supply a proof that the function is correct. If the rules for reasoning accurately describe the meaning of a Python function, then such a proof can only be constructed if the function is, in fact, correct. Furthermore, the client can check the proof to see if it is valid without having to understand all the insights that were involved in its creation.
Lemma: \(((r < s) \land (s < t)) \rightarrow ((r < s) \land (s < t) \land (r < t))\)

... rewrite with rule [B.2][2] used "right – to – left", with \(x\) matching \(((r < s) \land (s < t))\)...

\(((r < s) \land (s < t)) \land ((r < s) \land (s < t))\)

... rewrite with rule [B.3][21], matching \(i\) to \(r\), \(j\) to \(s\), and \(k\) to \(t\) ...

\(((r < s) \land (s < t) \land (r < t))\)

... rewrite with rule [B.2][6] used right – to – left, matching \(x\) to \(r < s\), \(y\) to \(s < t\), and \(z\) to \(t < r\)...

\(((r < s) \land (s < t) \land (r < t))\)

... rewrite with rule [B.2][4] matching \(x\) to \(s < t\) and \(y\) to \(r < t\)...

\(((r < s) \land (s < t) \land (r < t))\)

... rewrite with rule [B.2][6] matching \(x\) to \(r < s\), \(y\) to \(r < t\), and \(z\) to \(s < t\)...

\(((r < s) \land (s < t)) \land (s < t)\)

Main Proof:

\(((1 < a) \land (a < b)) \lor ((1 < b) \land (b < a))\)

... rewrite with Lemma above, matching \(r\) to 1, \(s\) to \(a\), and \(t\) to \(b\), in the term to the left of the \(\lor\)...

\(((1 < a) \land (1 < b)) \land (a < b)) \lor ((1 < b) \land (b < a))\)

... rewrite with Lemma above, matching \(r\) to 1, \(s\) to \(b\), and \(t\) to \(a\), in the term to the right of the \(\lor\)...

\(((1 < a) \land (1 < b)) \land (a < b)) \lor ((1 < b) \land (1 < a)) \land (b < a))\)

... rewrite with rule [B.2][4], matching \(x\) to \((1 < b)\) and \(y\) to \((1 < a)\)...

\(((1 < a) \land (1 < b)) \land (a < b)) \lor ((1 < a) \land (1 < b)) \land (b < a))\)

... rule [B.2][9] right – to – left, matching \(x\) to \((1 < a) \land (1 < b))\), \(y\) to \((a < b)\), \(z\) to \((b < a)\)...

\(((1 < a) \land (1 < b)) \land (a < b)) \lor ((1 < a) \land (1 < b)) \land (1 < b) \lor (b < a))\)

Q.E.D.

Figure B.2. Proof from Figure B.1 still in full detail, condensed by using a Lemma.

The creation of a full proof of correctness for an entire computer program can be extremely time consuming, so the scenario described above probably lives only in our imagination. However, the same techniques can be useful in demonstrating that a critically important component of the software does obey a specific safety property. For example, the purchaser of control software for an aircraft might wish to provide a specification that captures a property like “the software will not call the “turn on thrust reverser” control function while the “is there weight on the landing gear” sensor function returns false”, and ask the vendor to prove this property.

The remaining sections of this appendix provide some of the most useful rules about formulas involving Boolean values (true and false), Integers, and quantifiers (such as “for all \(x\), ...”). Rules for Python functions are given in Appendix A.

Check Your Understanding B.1. Fill in the gray boxes below to show the result of applying each axiom of Section B.3 in the manner described, completing the following proof that \(n + 0\) can be rewritten \(1 \cdot n\) (assuming \(n\) is an integer, so the rules of Section B.3 can be applied). Note that this expresses the basic insight that \(n + 0\) and \(1 \cdot n\) are each just \(n\).

Copyright 2003-2015 David G. Wonnacott; see Page 7 for terms of free use
... rewrite with rule B.3.8, matching \( i \) to \( n \) ...

... rewrite with rule B.3.9, matching \( i \) to \( n \) ...

... rewrite with rule B.3.4, matching \( i \) to \( n \) and \( j \) to 1 ...

Q.E.D.

Check Your Understanding B.2. Fill in the gray boxes below to show which axioms of Section B.3 must be used, and how they would be applied, in the following proof that \( m \cdot (n + o) + p \) can be rewritten \((m \cdot n) + ((m \cdot o) + (m \cdot p))\), assuming that \( m, n, o, \) and \( p \) are all integers.

\[
m \cdot ((n + o) + p)
\]

\[
m \cdot (n + (o + p))
\]

\[
(m \cdot n) + (m \cdot (o + p))
\]

\[
(m \cdot n) + ((m \cdot o) + (m \cdot p))
\]

Q.E.D.

Check Your Understanding B.3. Construct a fully-detailed proof, using the axioms of Section B.3, that \((n + (o + p))\) can be rewritten \((p + (o + n))\), assuming that \( n, o, \) and \( p \) are integers.
B.2 Axioms of Boolean Algebra

The following rules can be used for rewriting expressions involving values that are true or false (known as “Boolean values” after the mathematician George Boole, who developed precise rules for dealing with true and false values). The rules below can be found in Joel Friedman’s 1996 “Mathematical Logic and Foundations, 1847-1947”, in the aforementioned texts by Gries, or in most textbooks about discrete mathematics or logic. The traditional mathematical symbols are somewhat different from those found in Python, specifically “and” is written $\land$ rather than and, “or” is written $\lor$ instead of or, and logical negation (making true into false and vice versa) is written by drawing a line above something rather than preceding it with not (i.e., the negation of $x$ is written $\bar{x}$ rather than not $x$). In the rules below, the variables $x$, $y$, and $z$ refer to boolean variables or boolean-valued expressions (such as a variable with the value true or false, or a comparison such as $2 \leq 3$).

1. $x \land y$, $x \lor y$, and $\bar{x}$ are all boolean values.
2. $x \land x \equiv x$
3. $x \lor x \equiv x$
4. $x \land y \equiv y \land x$
5. $x \lor y \equiv y \lor x$
6. $x \land (y \land z) \equiv (x \land y) \land z$
7. $x \lor (y \lor z) \equiv (x \lor y) \lor z$
8. $x \lor (y \land z) \equiv (x \lor y) \land (x \lor z)$
9. $x \land (y \lor z) \equiv (x \land y) \lor (x \land z)$
10. $\bar{x} \land \bar{y} \equiv \bar{x} \land \bar{y}$
11. $\bar{x} \lor \bar{y} \equiv \bar{x} \lor \bar{y}$
12. $x \land \text{true} \equiv x$
13. $x \lor \text{false} \equiv x$
14. $x \land \text{false} \equiv \text{false}$
15. $x \lor \text{true} \equiv \text{true}$
16. $x \lor \bar{x} \equiv \text{true}$
17. $x \land \bar{x} \equiv \text{false}$
18. $x \land y \equiv x$
19. $x \Rightarrow x \lor y$
20. $\bar{x} \equiv x$

Copyright 2003-2015 David G. Wonnacott; see Page 7 for terms of free use
B.3 Axioms for Integers

Most of the following can be found in James Anderson’s “Discrete Mathematics with Combinatorics” and many other sources. The variables $i$, $j$, and $k$ refer to integer variables or integer-valued expressions (such as $2 + 3$ or $i + 6$).

1. $i + j$, $i - j$, $i \cdot j$, and $i^j$ are integers
2. $i < j$, $i \leq j$, $i = j$, $i \geq j$, and $i > j$ are boolean values
3. $i + j \equiv j + i$
4. $i \cdot j \equiv j \cdot i$
5. $(i + j) + k \equiv i + (j + k)$
6. $(i \cdot j) \cdot k \equiv i \cdot (j \cdot k)$
7. $i \cdot (j + k) \equiv (i \cdot j) + (i \cdot k)$
8. $i + 0 \equiv i$
9. $i \cdot 1 \equiv i$
10. $k = i + j \equiv i = k - j$
11. $(k + i = k + j) \equiv i = j$
12. $k \cdot i = k \cdot j \wedge k \neq 0 \equiv i = j$
13. $i^1 \equiv i$
14. $i^{j+k} \equiv i^j \cdot i^k$
15. $i = i \equiv true$
16. $i \leq j \equiv (i < j) \vee (i = j)$
17. $i \geq j \equiv j \leq i$
18. $i > j \equiv j < i$
19. $i \geq j \equiv \overline{i < j}$
20. $i \leq j \equiv \overline{i > j}$
21. $i < j \land j < k \equiv i < k$
22. $i \leq j \land j \leq k \equiv i \leq k$
Appendix C
Logical Assertions Program File

For reference, a simple version of the Python file that defines the functions we use for logical assertions (preconditions, etc.) is given below. This should not be necessary to understand the abstract content of this course, but it will be needed to run the examples that use `from logic import *`. More sophisticated Python programming might allow more sophisticated processing of errors, or detection of additional errors (such as a lack of progress), but this version should be sufficient to make the examples work, and to detect many errors.

""
Simple functions to let us declare pre- and post-conditions
and make various other logical statements in Python programs.

Started Summer 2006 by Dave Wonnacott (davew@cs.haverford.edu)

These produce various kinds of exceptions if they get an illegal parameter
(i.e., False for most, or a non-integer for progress, etc.).
[Except for a few "is_" functions for testing, e.g. is_integer.]

They are currently _very_ primitive, and

* The postcondition must be stated just before _each_ return
* There is no actual checking of progress

I hope some day to know enough about Python to make it better...
""

class LogicConsistencyException(Exception):
    """ Could add more here, if it were needed. """

class PreconditionException(LogicConsistencyException):
    """ Nothing """

class PostconditionException(LogicConsistencyException):
    """ Nothing """

class AssertionException(LogicConsistencyException):
    """ Nothing """
class ProgressException(LogicConsistencyException):
    """ Nothing """

class LoopPreconditionException(LogicConsistencyException):
    """ Nothing """

class LoopPostconditionException(LogicConsistencyException):
    """ Nothing """

class LoopInvariantException(LogicConsistencyException):
    """ Nothing """

class LoopProgressException(LogicConsistencyException):
    """ Nothing """

def precondition(value_of_precondition):
    if (value_of_precondition != True):
        raise PreconditionException

def postcondition(value_of_postcondition):
    if (value_of_postcondition != True):
        raise PostconditionException

def assertion(value_of_assertion):
    if (value_of_assertion != True):
        raise AssertionException

def progress(progress_value):
    # NOTE that we can't expect progress to be non-negative,
    # because some things like "fib" can skip "levels" and
    # ask for values for which the progress exp. is negative
    if not is_integer(progress_value) or False:
        # *** still need to check actual progress ***
        raise ProgressException

def loop_precondition(value_of_loop_precondition):
    if value_of_loop_precondition != True:
        raise LoopPreconditionException

def loop_postcondition(value_of_loop_postcondition):
    if value_of_loop_postcondition != True:
        raise LoopPostconditionException
def loop_invariant(value_of_loop_invariant):
    if value_of_loop_invariant != True:
        raise LoopInvariantException

def loop_progress(loop_progress_value):
    # NOTE that we can’t expect progress to be non-negative,
    # because some things like "fib" can skip "levels" and
    # ask for values for which the progress exp. is negative
    if not is_integer(loop_progress_value) or False:
        # *** still need to check actual progress ***
        raise LoopProgressException

# And now, some things that will hopefully make "isinstance" less confusing...

def is_integer(v):
    return isinstance(v, int)

def is_number(v):
    return is_integer(v) or isinstance(v, float)

def is_string(v):
    return isinstance(v, str)
# Index

<table>
<thead>
<tr>
<th>Term</th>
<th>Page(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>advanced recursive design</td>
<td>45</td>
</tr>
<tr>
<td>algorithm</td>
<td>10</td>
</tr>
<tr>
<td>arc</td>
<td>62</td>
</tr>
<tr>
<td>asymptotic complexity analysis</td>
<td>13</td>
</tr>
<tr>
<td>basic recursive design</td>
<td>45</td>
</tr>
<tr>
<td>bottom-up implementation</td>
<td>49</td>
</tr>
<tr>
<td>call tree</td>
<td>32</td>
</tr>
<tr>
<td>clique</td>
<td>62</td>
</tr>
<tr>
<td>code</td>
<td>11</td>
</tr>
<tr>
<td>code review</td>
<td>12</td>
</tr>
<tr>
<td>coding</td>
<td>11, 27</td>
</tr>
<tr>
<td>computational complexity</td>
<td>13</td>
</tr>
<tr>
<td>computer</td>
<td>10</td>
</tr>
<tr>
<td>correct</td>
<td>11</td>
</tr>
<tr>
<td>debug</td>
<td>31</td>
</tr>
<tr>
<td>design by cases</td>
<td>45</td>
</tr>
<tr>
<td>dinner party problem</td>
<td>62, 67</td>
</tr>
<tr>
<td>edge</td>
<td>163</td>
</tr>
<tr>
<td>edge</td>
<td>62</td>
</tr>
<tr>
<td>encoding</td>
<td>11</td>
</tr>
<tr>
<td>formal specification</td>
<td>12</td>
</tr>
<tr>
<td>formal verification</td>
<td>11</td>
</tr>
<tr>
<td>function call</td>
<td>20</td>
</tr>
<tr>
<td><strong>function call</strong></td>
<td>20</td>
</tr>
<tr>
<td>general problem</td>
<td>9</td>
</tr>
<tr>
<td>generate and filter</td>
<td>45, 62</td>
</tr>
<tr>
<td>graph theory</td>
<td>62</td>
</tr>
<tr>
<td>hacking</td>
<td>46</td>
</tr>
<tr>
<td>node</td>
<td>62</td>
</tr>
<tr>
<td>pair programming</td>
<td>12, 19</td>
</tr>
<tr>
<td>parameters</td>
<td>23</td>
</tr>
<tr>
<td>postcondition</td>
<td>17</td>
</tr>
<tr>
<td>precondition</td>
<td>17</td>
</tr>
<tr>
<td>problem instance</td>
<td>9</td>
</tr>
<tr>
<td>programming</td>
<td>27</td>
</tr>
<tr>
<td>programming language</td>
<td>11</td>
</tr>
<tr>
<td>recursive</td>
<td>30</td>
</tr>
<tr>
<td>reducing</td>
<td>45</td>
</tr>
<tr>
<td>referentially transparent</td>
<td>35</td>
</tr>
<tr>
<td>self-referential</td>
<td>27</td>
</tr>
<tr>
<td>specification</td>
<td>17, 16</td>
</tr>
<tr>
<td>strings</td>
<td>22</td>
</tr>
<tr>
<td>subclique</td>
<td>62</td>
</tr>
<tr>
<td>subtle</td>
<td>62</td>
</tr>
<tr>
<td>test suite</td>
<td>12, 18</td>
</tr>
<tr>
<td>testing</td>
<td>11</td>
</tr>
<tr>
<td>top-down design</td>
<td>45</td>
</tr>
<tr>
<td>user interface</td>
<td>30</td>
</tr>
<tr>
<td>diagonal problem</td>
<td>9</td>
</tr>
</tbody>
</table>