Software Craftsmanship

David Souther

2020-11-29

Software Craftsmanship is the study and practice of using computers as a tool in our everyday lives. Today's world is deeply and critically tied to the computers we use, in subtle and varied ways few would even begin to consider. At the same time, the computer sitting on your desk offers a fantastic opportunity to parse and trove your data. People in nearly every profession could stand to benefit from the knowledge and ability to write programs to solve their own problems. This book shows the layperson how to tap this potential, taking you from being able to perform basic computing tasks, through writing your first program, on to having a basic understanding of how programs work, while being able to begin to understand the programs that are used in your profession.

This book is available online at http://davidsouther.com/software_craftsmanship, with workbooks at http://davidsouther.com/software_craftsmanship/python.html and others to come.

It is available for download as PDF in parts:

- textbook
- python workbook
- and others to come.

Table of Contents

Introduction

Human endeavors rest on the backs of the hard-working crafters. From stone- movers of the Egyptian pyramids to steel-workers on today’s wind farms, skilled workers built civilization with their hands. Stone, steel, lumber, and leather have for centuries been the foundation of human enterprise. In the 21st century, there is a new medium demanding attention: information. The data flying across the Internet is the backbone of international trade and commerce, and needs skilled craftspeople to shape it. Yet even as professional carpenters build masterwork cabinetry for law firms and movie stars, there are laymen working on the same craft with the same tools in their garage. This is no less true for computers - the relatively low cost of consumer software gives hobbyist programmers the same tools to work with as the professional.

This book is aimed at those who are interested in this new medium as a potential hobby or curiosity. The book begins assuming the reader knows how to turn on and use their computer for basic tasks — email, word processing, image editing, and video games; and takes them to a level where they will be comfortable and confident in using and controlling their computers. This book is very fast paced. Software development is a field which has undergone active development for the past 80 years, yet did not exist beyond a dream before then. There is a lot to learn about this craft, but readers who want to become truly skilled at this craft will hopefully find this book gives them many of the tools they need to feel comfortable working their computer. To get the most out of the text, I recommended readers work through the companion workbook. Like any craft, to get good at software development you need to develop software. The workbooks lead you through developing software, in a way highlights the concepts presented in the text.

Computers Are Tools

Information is a critical piece of today’s infrastructure. How people use information is part of the field of Computer Science. Computer Science is on the one hand deeply seated in mathematics, and on the other, firmly rooted in practicality. The computer itself is simply a tool, like a band saw or plasma torch, which does amazing things with this data. Whether used in large-scale data mining and predictions for financial institutions, determining exactly what websites have what content, or creating a side show of family photos. At the end of the day, the laptop or desktop sitting in front of you is just a piece of silicone, copper, and plastic, capable of doing only exactly what it is told.

The task of the computer programmer is to tell the computer what to do. This is not an easy task. When people think about a problem, we can start off being a bit loose on how we describe the problem to ourselves. We can find issues or errors in our assumptions, and change them on the fly. Computers cannot do this. Every condition must be considered before hand. Say you’re balancing your check book. You add up $12.57, $52.45, and $1.99, but miss the decimal on the $52.45, and end up with $5249.56 - clearly an error. On paper, it is obvious what you did wrong. The computer has no way of knowing this was an error. Instead, it is up to the good programmer to tell the computer to verify that any numbers typed into the financial program ends with a decimal and two numbers. Then, the program can warn the user before making the calculation, potentially avoiding a costly transaction.

Using this tool

Software craftsmen use this tool by writing programs. A program is a document both written and read by human beings, while telling the computer unambiguously what to do. Take this example: `4 + 6 / 2`. Is the answer 7 or 5? If you remembered something like "Please Remember My Dear Aunt Sally" from grade school, you would say 7. If you were a desk calculator in an accountant's office, you would say 5. This statement is **ambiguous** - it could mean more than one thing. The programmer's job is to decide if the statement should be `(4 + 6) / 2 = 5` or `4 + (6 / 2) = 7`.

This is a pedantic exercise. As software gets more complex, software engineers are responsible for deciding how the computer helps people interact with their data. Consider the difference in organ donor rates between the United States and Sweden. In the book [Nudge](http://nudges.org/), the authors contend the difference is in "presumed consent" - that is, DMVs in the United States require drivers to Opt In to being an organ donor, while Swedish DMVs require patients to Opt Out. If you were writing a web page to take DMV registration information, the difference in those conditions is eight characters - adding "checked" to the Organ Donor input. With that minuscule change in a program, the software craftsman has the potential for enormous influence on the lives of millions of people.

Sharing this tool

There is a whole world outside your front doorstep, and as much as some programmers might want to deny it, at some point every craftsman's programming will be used by someone else. It is important your code be both readable and usable by these other craftsmen. There are many ways to do this, and many great tools to facilitate this. While many software craftsman may want to hoard their code, this is simply not possible. Even at companies like Microsoft, the largest proprietary software development company on the planet, teams are used because software projects, even most hobbyist projects, are too large for one person to handle alone.

Many computer science courses ignore this aspect of programming, which I believe is a real detriment to the students. Many computer science students can go their entire undergraduate careers without ever looking at their peers' code, and only having their program reviewed by the instructors. Craftsman of all fields can drastically improve their work by meeting and sharing ideas. Building partnerships with other craftsmen is one of the most rewarding ways to practice a trade, but can also be a harrowing experience learning how to work and interact with a completely new group of people. To work past this impediment, I present tools for managing and sharing your programs early in the book (Chapter 3, Source Control), and encourage you to find other programmers and technology clubs in your area and on the Internet.

Computer Languages

Computer languages are the tie between the world of everyday common sense conversation and the exact stupidity of the computer. There are hundreds, if not thousands, of different programming languages today. Many are "toy" languages, built for a fun exercise or class project and are not intended for wide-scale use. There are other languages that are proven as the workhorses of computers, and are used everywhere from your cell phone to robots running on Mars.

Languages are often classed based on their style of programming, and their level of expressivity. There are, broadly, three styles of programming: imperative, functional, and logical. Imperative languages are similar to a cookbook recipe. They describe, one statement at a time, the "things" a computer is to do - add two numbers, print those numbers to the screen, ask the user for confirmation. C, JavaScript, and Python are all imperative languages. Functional languages embrace the mathematics of computer science. Functional languages often use similar syntactic constructs as imperative languages, but with fundamental underlying differences. Haskell, Erlang, and Lisp are functional programming languages. Logical programming languages have little to do with either imperative or functional programming. Prolog is a logical programming language. Logical programming will not be discussed in this book.

Expressivity is notion for the ratio of amount of programming code written to how much the computer does. Languages with the least amount of expressivity are referred to as "low-level" languages - they must deal with all aspects of the computer and its memory. "High-level" languages handle many of the details of working on computer hardware, and let the programmer just focus on expressing the logic of the program in question. This book is presented using three different languages. These languages were chosen because they each represent a very different approach to software craftsmanship. Each approach is correct in its own way, and it is important to know each of the approaches to be a great software craftsman. Further, each of these three languages is mature, and actively used in a variety of projects today.

C

C started its life at Bell Labs in 1973. C was written by Dennis Kernighan and Brian Ritchie as a language to write the Unix operating system. Before C, operating systems (the program enabling all the other programs to run) were written in an assembly language for each computer Unix would run on. However, no two brands of computers had the same assembly language, so any time someone wanted to run Unix on a new computer, they had to rewrite the entire operating system. At the time, this could be thousands of lines of code. Today, it would take millions of lines of assembly to code Windows or Linux. C was designed to be a consistent language that, rather than being run on a computer as is, would first be compiled into the appropriate assembly code.

C is widely considered the *lowest-level* of today’s common programming languages, meaning C is as close to running "on the hardware" as you can get. When writing C, the programmer has to deal with many aspects of computer memory management. There are few utilities to achieve all but the most common tasks (though there are a wealth of *libraries* to fill the gap). C is the only language used in this book that must be compiled before being run. This *low-level* nature of C makes it a very powerful language, especially when faced with requirements to interact directly with hardware, or when hardware is in short supply (embedded on robots or cell phones). That power comes with great responsibility for writing the program correctly.

Python

Python is a programming language developed by Guido van Rossum in the late 1980s. Python has gone through two major revisions since its first release, and now is widely available on nearly any computing platform. Today, Python is used by many Linux distributions to write a variety of their higher-level tools. Organizations from Google to NASA use Python for numerous mission-critical applications.

Python was designed to be a very flexible language, and as such is *high-level* compared to C. Python was also designed to be a fun language to use - the name Python refers not to the snake, but to Monty Python’s Flying Circus. Python has a certain culture around its use not seen in many other programming languages. In Python, the prevailing wisdom is "there should be one - and preferably only one - obvious way to do it."

Javascript

Javascript is a programming language developed at Netscape in 1994 by Brandon Eiche over the course of a week. Javascript is the de-facto standard for writing programs served over the Internet to run in a user’s web browser. While actually a functional programming language, Javascript is often (mis)represented as being an imperative language. This has lead to some very poor code being written in the 15 plus years since its inception. That said, its use in every web browser today has many people working to make Javascript a less-maligned and better respected language.

Javascript is a very high level language. The programmer has few worries about memory management, and no capabilities to access the computer’s hardware (though there are initiatives to enable such use). When combined with libraries like jQuery, javascript can be the most expressive of the three languages presented. When we start working with graphical programs later in the book, Javascript’s expressive power will really shine, in that the amount of code needed to do the same thing (click a button) can often be an order of magnitude less than a similar program in C.

TypeScript

TypeScript is a dialect of Javascript, developed by Andres Hejlsburg at Microsoft in 2010. The goal of the TypeScript project is to provide a strong "type system" on top of JavaScript. A Type System is a set of tools that allows another program, the TypeScript Compiler, to analyze your program to prove various properties and check for common errors. For instance, if you were to write in your program `5 + "hello"`, TypeScript would say you have an error combinging a number and a string. TypeScript is currently the most popular variant of JavaScript, because of its tremendous benefit and value in helping teams of all sizes manage the complexity of their JavaScript code.

Using this book

This book is meant to be a guide on your programming journey. The main textbook, which you have in your hands right now, talks about software engineering concepts and ideas. It isn’t tied to any specific programming language. Instead, it provides a discussion of the topics, definitions, and general content in a lecture style. After reading the textbook sections, there are workbooks available to put the ideas into practice for a specific programming language. There are three textbooks. When using this book for the first time, I suggest using the Python textbook. Python is the easiest of the languages to start using, with the lowest barrier to entry. After Python, TypeScript covers the same content in a slightly different approach. You will learn the concepts in the textbook even better by seeing how they are expressed in a different language. But still, you should do the Python workbook in full first. Finally, there is the C workbook. C has a level of detail above the other two, and will serve as a good final introduction to programming techniques.

Syntactic Core (1-3)

The first three chapters will cover the core of writing programs. This will get you to the point where you can have your computer talking to you and asking questions, though perhaps not gracefully. We will discuss how computers understand data, and how they operate on data in a clear and precise manner. You will also learn how to begin grouping this data and these operations into increasingly complex pieces that work in synergy with one another. You should also follow the first two appendicies, on using the terminal and source control, to begin learning the programming-adjacent skillsets that are necessary in software craftsmanship.

Programming Patterns (4-6)

With the basics down, these chapters begin to take a look at patterns common to bigger and more robust pieces of software. You will learn how to write tests for their code. Tests are small programs which verify the main program is itself written correctly. Input and output are necessary for having programs which work with data sets, when storing more complex data longer than the run of the program. Chapter 6 begins to look at longer running programs, including graphical user interfaces for highly interactive programs running on local PCs and using web servers to allow people worldwide to access your work. Another pair of appendices augment the programming itself - a look into debugging techniques to teach how to isolate problems in a misbehaving program, and a discussion of containers, a modern technique in distributing production systems to make them available for a number of users.

Building Large Programs (7-9)

The last section of the book looks at making larger programs that are fully featured. This includes making a space-invaders like video game, building out a painting program, and working as a team to create a board game. Each of these projects in the chapter should be a complete stand-alone tool that combines lessons from all the prior work in the book. The final appendicies discuss a technique for managing data called parsing, and a bibliography of future books & courses to consider as you continue your programming journey.

Practice

Like any skill, software craftsmanship takes practice. The workbooks are designed to highlight the concepts presented in the text, while giving you an opportunity to practice these skills. The workbooks are broken into lessons roughly corresponding to sections of the main text. Each lesson has two parts. The first part is a listing of code. You should type the code into your editor exactly as written. Do not copy and paste the code. Much of programming involves paying very close attention to a myriad of small details, and every character has meaning. This discipline in typing code exactly as presented will pay off in your programming future. The second part of each lesson is a few exercises to work with the new concepts introduced in the text and program listing, and ideas to combine them with what you learned and wrote previously. Some of the lesson exercises will involve conducting research on the Internet. Being able to find help with programming questions is another invaluable skill as a software developer.

What Next?

At the end of each section, there are links either to the workbook exercises in the various languages, or links from the workbook back to the textbook at the next topic section. So whenever you're ready, click on the link, or come back to the last section you worked on!

Exercise: Hello World

If this is your first time programming, I'd recommend doing the exercises in the Python workbook. If you've gotten through the book, try redoing the exercises in TypeScript, then C!

- Python workbook
- Typescript workbook (Coming soon)
- C workbook (coming soon)

Basic Types and Control Flow

Programs begin with data. They then perform operations on that data over time. This chapter introduces you to what data is in a proram, and how to control that time aspect. After we have an overview of what data is, and how we can operate on it, we’ll write a number guessing game.

Basic Types

Computers work with data. At the lowest level, this data is just 'on' and 'off' in a transistor. Software craftsmen don’t work at this level. Instead, our programming languages give us the tools to work with this data in a much more intuitive fashion. In broad strokes, this is achieved with two concepts: data types and control flow. Data types describe the data our program is working with, whether it be a number, a bank account, or an image. Control flow describes the operations that occur on the data over time, like adding numbers, checking the balance of a bank account, or cropping an image.

You can think of data types as the space of a program. At any one point in time, we will have certain sets of data we are working with. We can stop the program, look at the data, and carry on. While we think of three dimensions in space, up/down, left/right, and front/back, there are many more dimensions we could talk about for different things. A table has a color, along with its position. Our programs represent these properties in different ways, but we as programmers abstract them, and the term we use to talk about these properties is data types.

Control flow would then be the time aspect of a program. As the program runs, it does different things to the data. At some point, it might need to make a decision, and do one thing or another depending on what properties the data has. At another point, the program might need to do the same thing on a bunch of similar pieces of data. Control flow represents the logical actions our programs follow.

Syntax

Before we dive into discussing our first program, there is one more topic to cover. Syntax describes the words and symbols that make a valid program. In English, nouns, verbs and adjectives must come in a certain fashion for a sentence to make sense. In a computer program, there are similar rules for how a program can be written, and still make sense to the computer to run. We will cover the details of syntax for each programing language in the workbook, but there are some key terms to know regardless of the language.

Keywords

In any programming language, there are a few reserved **keywords**. These are words which have a specific meaning in that programming lanugage. Most programming languages only use no more than a couple dozen keywords. These will be covered in the workbooks as needed.

Variables

In a program file, **variables** are words used to refer to some piece of data. Variables let programmers use concrete concepts to refer to the more abstract realm of the mathematical values in the computer. Variables can generally be any word that isn't a keyword in a programming language.

Variables have two parts - an **identifier** which gives it its name, and a **value** which is the data that it *currently* stores. An identifier is a way to refer to a certain bucket holding a piece of data, but it is not the data itself. This distinction is really important when we start looking at data changing over time.

Over the course of a program, there are some identifiers which will assign different values to, and do those assignments many times. In all the programming languages we use, this is done with the single equals character, `=`. The `=` operator always has two sides - the left hand side will be the identifier (the variable's name), like `age` or `area`. The right hand side is some value, or some calculation. `age = 32` stores the number `32` in the variable named `age`, so if later on I wanted to know the age of something, I can refer to `age` instead of needing to know that it was `32` every time. If I want to calculate something, I also store that in a variable. `area = length * width` refers to two other identifiers, which are the names of two other variables that we've set already, `length` and `width`. The assignment takes the values that are stored in those variables, multiplies them together, and stores the result of that in the variable `area`.

Using variables like this to track data gives us two really useful benefits. Most importantly, this allows data to change over time. We can use variables to track what the data's value is, and then us it in other calculations, without alwyas having to know its specific value each time. When we deal with user input we will see how variables let us write a program that executes and generates a value that no programmer ever coded!

From an engineering perspective, variables let us attach names to concepts in our programs. Using identifiers this way lets us write programs that read almost as prose. With just a sprinkle of notation (and a rigid syntax), the names we give our variables let us be expressive in our intent for what the program does.

Variables are not algebraic

Variables in programming are not variables in algebra. You might be having a flashback to high school algebra, maybe remembering something like `3x\^2 + 5x + 7 = 0`, and being asked to "solve for x". In mathematics like this, a variable is an unknown, a part of the expression that we need to isolate and handle separately. While the word in programming comes from the same place, it's almost the opposite - in programming, a variable *always* has a value and we always know what it is. It's more like keeping track of the amount of ingredients in a recipe than it is solving a mathematical equation.

Operators

Operators are the built-in things that can happen to data. Common operators are things from elementary arithmetic - add `+`, subtract `-`, multiply `*`, and divide `/`. Other operators let the program compare two pieces of data - equals `==`, less-than `<`, greater-than- or- equal-to `>=`. Notice there are some funny things for the comparison operators. First, equals is two characters, `==`, not the single `=` used in elementary school. This is because a single `=` sign is an operator used to assign values to a variable using its identifier. Multiplication is a star, `\*`. While `x` was used in elementary school, we use `\*` for our programs because `x` could be a variable, and we need to be unambiguous in what the program means. There is no `▯` on a common keyboard, so programs use the two character combinations of `⇐` and `>=` to do comparisons. There are other operators that will be covered as they are needed.

Basic Types

Data Types fall into two distinct categories. **Primitive** data types are those that are specified in the programming language itself. **Complex** data types are defined by the programmer, and are made up entirely of primitive data types and other complex data types already defined. In this section, we will go over the basic data types in your language. In general, these basic types are applicable in any programming language in major use today, with small discrepencies between them.

Numbers

Since computers only do arithmetic, everything is at some level a number in the computer. In mathematics, numbers can be as big as they need to be. If we were counting pebbles of sand on a beach, there is a number to represent that (though I will not be the one doing the counting). Computers are somewhat more finite. The representation of a basic number in a computer is confined to a range of a few possible values. On today's computers, that range is either `2\^32` or `2\^64` possible values. The 32 and 64 refer to the number of **bits** your computer handles. When the IT guy asks whether your computer is 32 or 64 bit, this is what we're talking about.

Characters

Characters are single bytes of data that are interpreted as being human- readable in some form. For instance, the letter `a` is the integer `97`. The letter `V` is `86`. `!`, the exclamation point, is `33`. Because the computer operates on numbers, there is no distinction to it between the number `33` and the character `!`. Instead, programmers tell their programs how they intend to use the data, and the program does the "right thing". The "right thing" is exactly what is was told to do. To understand what you're telling the computer to do better, it is important to know that there is nothing special about the mappings between the numbers and characters. Indeed, the first 128 character numbers were chosen by an organization called the American Standard Code for Information Interchange, or ASCII. For this reason, the first 128 characters are called the "ASCII" character set. Of course, there is no way to fit all the characters of both the Latin and Cyrillic alphebets into 128 different numbers, much less most Asian character sets. For this reason, there is another character set called UNICODE that defines over 4 million possible characters, enough to satisfy human languages for some time. We will not go into the details of Unicode here.

![ASCII Code Chart](./800px-ASCII_Code_Chart.png)

Let's take a closer look at some features of the ASCII character set. First, notice that uppercase and lowercase characters are represented distinctly. This is why case sensitivity is important on computers. Second, 0 through 31 look really funky. These are called control characters, not printing characters. Most of these control characters are no longer used in today's computers (they were used to literally control printers and other devices in the 70s and 80s), but there are some that warrant special attention. The first is NULL, 0. NULL is used in C to represent the end of an array (see Chapter 2). 10, `\n` or Newline, is used to make the computer put a line break in a program's output; otherwise, everything would show up on one long horizontal line. Second is 15, `\r` or

Carriage Return. This goes back to when computer printers were fancy typewriters, which required a special operation to move the carriage back to the starting position. While I can't think of any printers today that need this fuctionality, it is not uncommon to see `\r\n` as a legacy pair of characters in many programs. The last control character that is interesting to us is 9, `\t`, horizontal tab (HT). The horizontal tab tells the computer that we want a lot of whitespace (usually 4 or 8 spaces worth).

Strings

Characters on their own aren't particularly interesting. What is useful is having long runs of characters to make up words, sentences, paragraphs, and so on. We achieve this by using what are called strings - which are literally just a collection of characters in a row. Usually strings are surrounded by double quotes in a program. See the workbook for a variety of examples on how strings look and work in a program.

While characters really are treated as just numbers, strings have a variety of common operations that don't make sense with numbers. Among these are operations to determine the length of a string (how many characters it has), to combine strings together, and to pull strings apart. These are also covered in detail in the workbook.

Practice

Work with the `types` program in the language of your choice.

- [Python](01_types/01_python.md)
- ~[CoffeeScript](01_types/02_typescript.md)~
- ~[C](01_types/03_c.md)~

Control Flow

In the last section, we talked about primitive data. The basic pieces of information computers work with, characters or numbers, that can combine in some interesting ways. It is sometimes described as the "space" dimension of your programs. In this section, we're going to talk about control flow, which is the "time" aspect, and how programs gain their power to compute any new piece of data, given some rules.

In broad strokes, programs will follow one sequence of operations - the sequence written in the source code. Control flow changes that in one of two ways: either it jumps forward somewhere else in the program based on some condition on the data, or it jumps back to repeat a section of code a certain number of times. These are called branches and loops, respectively.

Each branch or loop is made up of many small pieces, which we want to describe.

Blocks

A **block** of code is a logical collection of code, separated from the rest of the program by some syntactic construct in the programming language. In the "c-based" languages (languages which got their syntax from C, including C++, Java, and Javascript), blocks are grouped within '{' and '}'. In

Python, Coffeescript, Ruby, and some others, blocks are grouped together by indenting lines of code at the same level. Within a block of code, each piece is executed in order. Each statement in the block happens one at a time, independant of other statments around it. The statement is made up of expressions, which are individual calculations on pieces of data. Like statements, expressions are evaluated (or run, or calculated) independantly of one another. The only way a computer takes a value from one expression or block or statment and uses it in another is by storing the results in a variable.

Expression

An **expression** takes some data and performs an operation. Adding two numbers is an expression. Checking if one number is greater than another number is an expression. Calculating one of the roots of the quadratic equation is an expression, but storing that result in a variable is not - that is a statement.

Different types of expressions have different precedence - for example, what do you think would happen with the expression `2 + 3 < 1 * 7`? In the languages in this book, the will all evaluate to `true`, because the arithmetic `+` and `\*` have priority over the relation `<`. We introduced operators in the first half of the chapter, and here are a few more of the specific rules.

There are five common arithmetic operations - `+` addition, `-` subtraction, `\*` multiplication, `/` division, and `%` modulus. Addition and subtraction behave as you would expect for both integers and floating point numbers. When mixing and integer and a float, the result will be a float. Subtraction with floats has some finicky details, explored in the basic types exercises. Multiplication between integers results in an integer, but multiplying very large numbers may have some side effects, which we explore in the exercises. Modulus on integers returns the remainder, and between a float and an integer behaves roughly as expected, but modulus between two floats is (for me) harder to intuit. There are very, very few places in programming where the modulus of two non-integers is necessary.

There are six relational operations that compare numbers - `==`, `!=`, `<`, `<=`, `>`, and `>=`. They should be pretty obvious, but `==` is equality, the numbers must be exactly the same, `!=` the numbers must not be the same, and the number must be less than, less than or equal, greater than, or greater than or equal to. Integers will always behave exactly as expected with the relational operators, but `==` equality rarely works correctly with floating point numbers. In calculations involving several steps of arithmetic in floating point numbers, the least significant part of the numbers will start to drift slightly, thus making them strictly not equal, while still being incredibly close. In those situations, it is common to instead of testing equality, testing if the difference of the numbers is less than some small error margin. More on this is in the exercises.

When combining several calculations, grouping with `()` guarantees that some steps will execute first. In the quadratic equation example, the `(-b + discriminant) / (2 * c)` made it certain that the addition and multiplication would be performed first. Without those, the implicit order from the program would have been `- b + ((discriminant / 2) * c)`. It is always good form to use parenthesis to group operations, even if the language itself would order your operations correctly.

When working with multiple logical conditions, the three operations `!` not, `&&` and, and `||` or are useful. The and operation works as expected - provided the clauses on both sides are true, the entire expression is true. The or operation does not behave as expected, in English. If I said I was going to make eggs or bacon, then I brought you eggs and bacon, you would probably be happy with getting an awesome breakfast, but still say my original statement was incorrect. In English, or is an exclusive logic operation. Either one OR the other condition should be true, but not both. In computers, or is an inclusive operation. At least one of the conditions must be true, but if both are true, the expression is still correct. The not operation is also a bit different. The operations covered to this point take two clauses, one to the right and one to the left. The not operation is 'unary', meaning it takes a single clause, to the right of the `!` character.

The final operation, and one used in every program, is assignment `=`. Assignment takes the computation on the right side and stores the value in the variable on the left. This leads to calling the pieces the "lvalue" and the "rvalue" - the rvalue can be nearly any expression, but the lvalue must be something that can be assigned to. In C, that means a single variable. In Python and Typescript, there are ways to assign to several variables at a single time.

Statement

A **statement** is the smallest executable chunk of code. In C, making a statement is as easy as adding a `;` to the end of an expression, delineating where one expression ends and the next begins. Usually, statements group blocks of code. An expression will run to check a condition of the program, and some number of blocks of code are executed in some way. The complete collection of the expression and the blocks are the statement. Of course, the blocks in the statement are statements themselves. Branching and looping, discussed next, are the best ways to see how statements combine to form a program.

Statements are executed one at a time, in sequence, without any direct effect on other statements. Indirectly, they can only work with one another via storing and reading values in and from variables. As a statement executes, the program goes through each expression in the statement in their order of operations. Usually, that means following parentheses fron inside to outside, and then going from the left side to the right. The big exception to this is with assignments, which always are the left-most side before the `=`, and then the rest of the statement.

Let’s look at this next line of code, which is valid in all the languages we use, and see how it goes one step at a time.

```
total_cost = (item_count * cost_per_item) + shipping_cost
```

Reading from the left to the right, we see the statement starts with `total_cost =`. From this, we know that whatever follows the `=` will calculate some value, and then we will store that value in the variable named `total_cost`. After the `=`, we see our first parenthisis, so we find the ending paren and do the calculation within them, `item_count * cost_per_item`. By multiplying these together, we get some intermediate value. That sticks around for a moment longer while we look at the last piece of the expression, `+ shipping_cost`. We take that value we had from the multiplication, add it to the shipping cost, and then and only then take this final single value and store it in `total_cost`. All that in a single expression, separate from anything else in the program!

Let’s walk through it again with real values.

```
item_count = 4
cost_per_item = 4.25
shipping_cost = 2.99
total_cost = (item_count * cost_per_item) + shipping_cost
```

When this executes, it goes inside the parens to outside, left to right, and finally assigns the value to the variable. So inside the parns, we have `item_count * cost_per_item`. The program looks up `item_count` first, and finds `4`, then looks up `cost_per_item`, and finds `4.25`. At this point, it has `4 * 4.25`. Only after it's gotten the values from those variables does it do the multiplication! So it does, and we have `17`. Now, it keeps this value in mind (just a value, no variable to keep it around), and does the next operation, `+ shipping_cost`. It looks up shipping cost, `2.99`, and puts it together with the value we just got, and have `17 + 2.99`. This works out to `19.99`, and now we have the final value of the expression. We take it and store it in `total_cost`. Now, any time after this line, if the program uses `total_cost`, it will look it up and find `19.99` to use however it needs.

Branching

With the concept of a **block** of code being a logical chunk of code, combining them in interesting ways gives programs their power. The most common piece of control flow is the logical branch. A logical branch examines some condition of the program's data, and runs one or another block of code depending on whether that condition is true.

A block is just a set of statements which execute in order. Typically, later statements in the

If-Then-Else

This basic branch is called "If/Else", and often the "If" is set off in programs from the truthy block by the word "Then". Even if the "Then" is not in the programming language, it's still discussed as such.

There was an example of an "If/Else" statement in the examples for the first part of this chapter. In common English, the statement reads:

```
Check the expected result of the Transaction.
IF the result of the transaction is greater than the account balance
THEN warn the teller the transaction is too large
ELSE execute the banking transaction.
```

If/Else statements are very regularly used when interacting with users. The program will ask the user for some input - "Would you like to run again? (Y/n)" IF the user types "Y", THEN the program runs again, ELSE the program says "Bye!" and exists.

Many business requirements can be expressed with If/Else logic - the transaction example is common, but think about a security card scanner.

```
IF the scanned card has an employee ID
THEN
    IF the employee id is allowed in this building,
    THEN
        the door is unlocked for 10 seconds
    ELSE
        flash "Unauthorized Access" on the keypad
ELSE
    Do nothing (so the thief doesn't even know they can't get in)
```

Notice how the first IF clause has a second IF inside it. By nesting these statements as deep as needed, any business rule that can be expressed in terms of true/false is valid to write a program to manage.

Looping

Where branching runs a block of code based on some condition of the state of the program, looping runs the same block of code multiple times. Looping comes, broadly, in two flavors. When the number of iterations is known, the loop is a **for** loop. When the number of iterations is based on some changing condition of the program, the loop is a **while** loop.

For Range

The most common looping construct repeats a block of code some discrete number of times, usually changing the value of one or two variables each time the loop repeats. Take the example of summing several numbers:

```
Let the variable "sum" be set to 0
FOR i taking values between 1 and 3, inclusive
    Set sum to be the current value of sum plus the value of i
Print sum
```

Implementing this program would print `6`. Going through it line by line, the program would:

1. Set `sum` to 0
2. Set `i` to 1
3. Add the value of `i` (1) to `sum`, and store it - `sum` equals 1.
4. Set `i` to 2 (jumping back to the next iteration).
5. Add the value of `i` (2) to `sum`, and store it - `sum` equals 3.
6. Set `i` to 3 (the start of the next iteration).
7. Add `i` (3) to `sum` (3) - `sum` equals 6.
8. There are no more numbers to iterate over
9. Print `sum` (6)

This wouldn't be too hard to write the three additions by hand. When the operation is taking the product of the numbers, or grows to summing hundreds of numbers, loops become very attractive.

Finally, when we move to handling lists and groups of data in the next chapter, the number of iterations is unknown when writing the program but is defined when running the program. Loops are the only way to work on all those pieces of data.

While

There are other times when a program needs to perform an operation many times, but must make a decision every time it repeats. In the HiLo game example, the loop repeats as often as the user continues entering "yes" when asked if they want to play again.

Comments

There is one last piece of a program we need to mention. Comments are text in our programs that are not used by the computer, but purely for a programmer to explain a particular piece of code. Comments have a checkered position in the software development community. Because comments aren't part of the execution of a program, they have a tendency to drift from the original implementation, after programs have been in development for some time. Comments can also be useless - in C, `x = x + 1; // Add one to x`. Even with their faults, a well-written and well placed comment can be invaluable in aiding other programmers understanding some code.

Practice

Work with the `control flow` program in the language of your choice.

Tracing Program Execution

At this point, we've run a few programs. We've gotten a bit of practice with programs that can take user input, perform calculations on those inputs, and then give the user some output based on the calculation. Whether this was rug prices, checking account balances for a transaction, or converting to roman numerals, the idea is the same for each.

As we learn to program, we find ourselves in this position of having a very powerful piece of machinery that has very limited capabilities for anything outside how it was built. It has no self-reflection, and no capability to reason or intuit its own behavior. All of that is left to us, which means we need to have a very good understanding of what specifically the computer will do with all these instructions we give it.

The technique we use to determine these behaviors is called tracing. When we **trace** a computer program, we take a piece of paper $ a pencil and, step by step, record exactly what the computer would do if it were running the program. To make a trace, we want to capture three columns of information. The first two columns are side by side - "identifier" and "value". That should be familiar, as those are the two parts of a variable! We are going to go through the program line by line and track each identifier as we come across it, and the value it has at any given time. There will be a third column, separate from the first two, called "Input/Output". Whenever the program creates output or asks for input, we'll record that in this column before copying it to the apropriate row in the variables section.

Tracing types

Let's see how this looks! Take this snippet from the python workbook:

```
i = 2
j = 3
k = 5

print(i, j, k)

m = i * j
n = j + k

print(m, n)
```

As we build a trace for this, we start with our paper looking like this:

```
Id | Value       _Output_
---|------
   |
```

We then look at the first line of our program.

```
i = 2
```

We see an identifier, `i`. We look at our trace, and beccause we don't have an ID yet for `i`, we add it:

```
Id | Value       _Output_
---|------
 i |
```

We look on the right side of the `=` sign to find the value to assign to the variable, and see that it's `2`, so we add that to the value column:

```
Id | Value       _Output_
---|------
 i |  2
```

That's the entirety of that line! We can now go to the next line:

```
j = 3
```

We don't have an id `j` yet, so we add it and its value `3` to our trace:

```
Id | Value       _Output_
---|------
 i |  2
 j |  3
```

That’s the whole line, and we can do it one more time with `k = 5`:

```
Id | Value       _Output_
---|------
 i |  2
 j |  3
 k |  5
```

The next line is a bit diferent, it’s got the `print(i, j, k)` statement. For this, we take what the computer will output and put it under the `Output` column. What will it output? Well, it will print the variables `i`, `j`, and `k` which we can look up in the table of IDs to values! We look at the table, and read `2` for `i`, `3` for `j`, and `5` for `k`. Print will put those all on one line of output:

```
Id | Value       _Output_
---|------         2 3 5
 i |  2
 j |  3
 k |  5
```

Moving on to the next block, we have

```
m = i * j
n = j + k

print(m, n)
```

The first line is `m = i * j`. `m` is a new identifier, so we can add that to our table:

```
Id | Value       _Output_
---|------         2 3 5
 i |  2
 j |  3
 k |  5
 m |
```

What value does it have? `i * j` is an operation that creates a value, not a value itself. So to get the value, we need to perform the operation! To do that, first we look at the identifiers `i` and `j`, and get their values from the table: `2 * 3`. Multiplying `2` by `3` is `6`, and that’s the value we can put in `m`!

```
Id | Value       _Output_
---|------        2 3 5
 i |  2
 j |  3
 k |  5
 m |  6
```

We can do the same process for the next line - note that we have a new variable `n`, get the values `j` and `k` for the operation, add those values together, and put the result in `n`:

```
Id | Value       _Output_
---|------        2 3 5
 i |  2
 j |  3
 k |  5
 m |  6
 n |  8
```

And then we have a print statement again. Looking up the values, we have `6` and `8`, which we can add to our next line of output in our trace:

```
Id | Value       _Output_
---|------       2 3 5
 i |  2          6 8
 j |  3
 k |  5
 m |  6
 n |  8
```

While this might feel slow and tedious, that's a good thing at this point - doing these traces will help you slow down and think through what the computer is actually doing, which is rarely 100% in line with what you want it to do!

Exercise: finish tracing your `types` program in the language you're using.

Changing variables and control flow

We've seen how to track new variables in our program, and use identifiers to find the value of a variable created elsewhere to use in an operation. We will use this same approach to track changing values in a variable, looking at our control flow program.

```
sum = 0
for i in range(1, 6):
    print(i)
    sum = sum + i
print(sum)
```

We'll make a new trace for this program

```
Id | Value       _Output_
---|------
   |
```

Starting at the first line, we see a new identifier, `sum`, which gets set to `0` right away. (We call this immediate setting a variable to a value **initialization**, so that when we access it later we know what value it started with).

```
 Id | Value       _Output_
----|------
sum |  0
```

On the next line, we see a `for … in …` loop. As discussed earlier, this does create a new variable, in this case `i`, and will assign a new value to it every time we come back through the loop. We cover `range(1, 6)` in depth in the next chapter; for now, just know that it will give `1` and then `2` each time we come to it again, up to `5` when it will be done.

```
 Id | Value       _Output_
----|------
sum |  0
 i  |  1
```

Now, we go inside the loop. The first line inside the loop is `print(i)`. We look for an identifier `i`, and read its value, `1`. We print this value, so put it in the `Output` column.

```
 Id | Value       _Output_
----|------          1
sum |  0
 i  |  1
```

The second line in the loop body is `sum = sum + i`. We see that we already have an identifier for `sum`, so we don't need to make a new one. On the right side, we see that we have a `+` for `sum` and `i`. This happens **before** the `=`, so we look at the values in the identifiers `sum` and `i` *as they are right now*. `sum` is `0` and `i` is `1`, so we add those together, getting `1`, and put that in the `sum` value. But wait, sum already has a value? That's right. We're going to change the value, and we do that in the trace by crossing out the old value and putting the new value adjacent to it.

```
 Id | Value       _Output_
----|------          1
sum | ~0~ 1
 i  |  1
```

If we read this now, we see that `sum` started at `0`, and then later became `1`.

So we've finished both lines of the loop body, which means we go back to the top, `for i in range(1, 6):`. We already have an `i`, so we don't need to make a new one. We remember that last time `range` gave us `1`, so this time it will give us `2`. We assign that to the variable `i` just like we did with `sum`, and we end up with this trace:

```
 id | value       _output_
----|------          1
sum | ~0~ 1
 i  | ~1~ 2
```

The first line of the loop is the print. We see that the current value (the one furtherst to the right, that isn't striked out) is 2. Let's output that:

```
 id | value       _output_
----|------          1
sum | ~0~ 1          2
 i  | ~1~ 2
```

We then have the `sum = sum + i` line again. We just saw that `i` is currently `2`, and when we look at `sum` we see that it's `1`. Adding them together, we store `3` back in `sum`.

```
 id | value       _output_
----|------          1
sum | ~0 1~ 3        2
 i  | ~1~ 2
```

We go back to the top of the loop. This time, `range` will give us `3`.

```
 id | value       _output_
----|------          1
sum | ~0 1~ 3        2
 i  | ~1 2~ 3
```

And when we've finished the loop body, we have this trace:

```
 id | value       _output_
----|------          1
sum | ~0 1 3~ 6      2
 i  | ~1 2~ 3        3
```

Take a moment to check what the rest of the program will look like. Remember that `range` will give numbers up to 5, and then will exit the loop the next time it gets called.

When you've written this out, we should see this trace:

```
 id | value           _output_
----|------              1
sum | ~0 1 3 6 10~ 15    2
 i  | ~1 2 3 4~ 5        3
                         4
                         5
```

And we're at the end of the loop. We come back to the top, `range` already gave us 5, which is the last thing we used in the loop. `range` has nothing left, so we exit the loop body. There's one more line - `print(sum)`, and looking at the final value for `sum` we get 15!

```
 id | value           _output_
----|------              1
sum | ~0 1 3 6 10~ 15    2
 i  | ~1 2 3 4~ 5        3
                         4
                         5
                         15
```

So we looped 5 times, with an increasing value of `i` for each. We printed out that `i` each time to see where we were at in the loop, and kept adding it to sum. After the loop was finished, we look back at `sum` one last time, and output its value, `15`. Then, the program is done and ends!

Tracing input

In the Roman Numerals program, we also have input. We can handle this one of two ways, whichever is more comfortable for you, but either way basically has us write another column for input. You can either write this separate from `output` and have

```
 input   | output
--------|---------
input 1 | output 1
input 2 | output 2
        | output 3
```

and they have their own columns, or you combine them and wrte them on opposite sides of one column in order:

```
    in / out
--- --- ---
    output 1
input 1
    output 2
input 2
    output 3
```

It’s up to you which way you want to do it.

So let’s look at how to do this with the roman numerals program. We’re using the first version we typed, not the later version that you filled in with all the remaining types

```
print("Decimal to Roman Numeral")
number = int(input("Decimal integer: "))

numeral = ""
while number > 0:
    if number >= 10:
        numeral = numeral + "X"
        number = number - 10
    elif number >= 5:
        numeral = numeral + "V"
        number = number - 5
    elif number == 4:
        numeral = numeral + "IV"
        number = 0
    else:
        numeral = numeral + "I"
        number = number - 1

print(numeral)
```

The first thing it does is print out `"Decimal to Roman Numeral"`, so put that in the `_output` column of the trace.

```
 ID    | Value         input | output
-------|------       -------|---------
                             | Decimal to Roman Numeral
```

To keep this from getting too long, we’re going to skip the empty print which just gives us an extra newline for prettier output. So the next line is `number = int(input("Decimal integer: "))`. This is a new variable, so we need to record that first:

```
 ID    | Value         input | output
-------|------        -------|---------
number |                     | Decimal to Roman Numeral
```

Then we have the `input("Decimal integer: ")` part. This will do two things - first, it will print output, and then, it will take whatever input we provide so that we can use that as the value for `number`. To do this, first we'll record the output

```
 ID    | Value         input | output
-------|------        -------|---------
number |                     | Decimal to Roman Numeral
                             | Decimal integer:
```

And then, we'll decide what input we will give it, and record that in the `input` column. This value comes from the user, and isn't part of the program. Therefore, we will choose `18` arbitrarily as what our users has put in.

```
 ID    | Value         input | output
-------|------        -------|---------
number |                18   | Decimal to Roman Numeral
                             | Decimal integer:
```

Now we can finish the line of code - `number = int(input…)` means we can take the value in the input column, and put it into the value for the `number` identifier.

```
 ID    | Value         input | output
-------|------        -------|---------
number |  18            18   | Decimal to Roman Numeral
                             | Decimal integer:
```

From here, we can continue tracing the code exactly like we did in the last section. The next line starts `numeral = ""`, making a new identifier and setting it to the empty string.

```
 ID    | Value         input | output
-------|------        -------|---------
number |  18            18   | Decimal to Roman Numeral
numeral|  ""                 | Decimal integer:
```

In the `while number > 0:` decision, we look up `number`, see that it is `18`, and do the loop body. In the loop body, the first statement is the first if condition, `if number >= 10:`. In this `if` condition, we look up `number` (again), see that it is (still) `18`, and then do the first body ignoring the other `if` branches. The body sets `numeral` (`""`) to `numeral + "X"` (which with string concatenation becomes just `"X"`), and sets `number` (`18`) to `number - 10` (`8`). When the body is done, we have this trace:

```
 ID    | Value         input | output
-------|------       -------|---------
number |  ~18~ 8        18    | Decimal to Roman Numeral
numeral|  ~""~ "X"            | Decimal integer:
```

The `while number > 0:` sees `number` is `8`, which is larger than zero, and does the loop body. `if number >= 10:` is not true, though - `8` is less than `10`, so this `if` condition does not apply and we do not execute its body! Instead, it goes to the next `elif` and checks that condition, `number >= 5`. It is, so at this point the `elif` block gets started. Notice, though, that the computer did execute each `if` check individually. It did **not** just jump straight to the block we wanted to.

So it has found the `elif` condition true, executes that block, and goes back to the top.

```
 ID    | Value             input | output
-------|------           -------|---------
number |  ~18 8~ 3          18    | Decimal to Roman Numeral
numeral|  ~"" "X"~ "XV"           | Decimal integer:
```

From the top again, `while number > 0:` finds `number` to be `3`. The first if finds `3` to be less than `10`, so it doesn't do that body. The first elif then gets a shot, and finds that `3` is less than `5` so again doesn't execute it. The second `elif` is up, but `3` doesn't equal `4` so no dice. The only thing left is the `else`, which means that the `else` body has to execute. But it did still first try all of the other conditions! The computer executes in order, it does not pick the *best* option. It's up to you to put the more specific cases first, and the more general cases later.

At the end of this round we have this trace:

```
 ID    | Value                   input | output
-------|------                 -------|---------
number |  ~18 8 3~ 2              18    | Decimal to Roman Numeral
numeral|  ~"" "X" "XV"~ "XVI"           | Decimal integer:
```

It should be pretty easy for you to finish the rest of the trace. Take a minute to do so.

```
 ID    | Value                                  input | output
-------|------                                -------|---------
number |  ~18 8 3 2 1~ 0                         18    | Decimal to Roman Numeral
numeral|  ~"" "X" "XV" "XVI" "XVII"~ "XVIII"           | Decimal integer:
                                                      | XVIII
```

With the `print` at the end of our program, this is our finished trace!

Traces in the book

Through the rest of the book, we will use traces whenever we come across new and interesting control flow & concepts. To keep the traces consistent and contained we'll be formatting them like such:

```
 ID    | Value
-------|------
number |  ~18 8 3 2 1~ 0
numeral|  ~"" "X" "XV" "XVI" "XVII"~ "XVIII"


  output / input
------------------
Decimal to Roman Numeral
Decimal integer:
                       18
XVIII
```

That is, the variables will be up at the top, and we will use one column of combined output and input below. Output will be left justified, and input will be right justified. When you make traces on your own, it doesn't matter where these pieces go. Find a layout for your page that works for you!

Exercises

You guessed it - trace your roman numerals project with several other inputs.

Wrap Up

This is a pedagogical tool

Project

Let's use these to write our first big program - a video game called [HiLo](../03_hilo/README.md)

HiLo

An early tradition of computer programming was a series of video games published as source code in BASIC programming language magazines. In that tradition, the first big program we type is a game called HiLo. It's a simple number guessing game, but has a good variety of programming functions, and serves as an excellent introduction to writing a mildly complex program.

In the tradition of these early magazine games, you get the source code for a complete game here, but it's not very functional. It's up to you to type the original in, and then make any improvements or changes you feel are worthwhile. You might want to change the position of text, the colors on the screen, or give the game several difficulty levels. Like all programming, it's all to your imagination!

[Python](01_python.md)

Wrap Up

Let's take a minute to appreciate what we've covered and accomplished! We started with a code editor and a way to run a Python progam, but at the time it didn’t do much more than print "Hello, world!". Over the past chapter, we’ve learned about how computers represent data, taking just numbers and making them serve all sorts of different purposes. We’ve looked at how to control the computer, making it branch and loop based on conditions. We used that to write a program which converted our numbers to Roman numerals. You wrote your (likely) first program! That did a real thing, that is useful! (Ok… questionably useful, but the next programs will really be useful!) Then we took that knowledge and wrote a jackpot game, that lets us test our skill at picking numbers. Not content at the bare rules of the game, we added color, boxes, and maybe even a little animation to snazzy it up! Be proud of yourself, you are now a computer programmer!

Recommended: Take a detour through [Appendix 2: Source Control](../A02_source_control/README.md)

Functions, Arrays, and Strings

We write programs to manipulate our data. In the last chapter, we treated data as integers, floats, and strings. We can iterate, or repeat, an operation on data. We can take different branches in our programs, doing different things based on the value of our variables. As programs get bigger, much of programming is managing complexity, keeping different parts of the program small and logically connected. This chapter covers three tools - functions, ways to block sections of code together; arrays, to keep more than one piece of data in a variable; and more details of strings, as they are useful in many ways in programming.

Functions

In programming, functions allow us to "bundle up" a section of code and make it reusable without needing to copy and paste it anywhere we want to use it. Like mathematical functions, we can take an idea, write it out, give it a name, and use it over and over any where we need. This allows us to break our program into composable parts, and put those together like Legos or an erector set.

Functions are made up of four pieces - an **identifier**, a list of **arguments**, the **body** of the function, and some **return** value. Let's look at each of these!

First, they need a name. Similar to variables which store data, function names allow us to refer to this block of code consistently and repeatedly. In fact, in some languages, function names are variables! When defining a function, choose an identifier which accurately but succintly describes what the function will do or what it will compute. Where variables capture things we work on, their identifiers generally use noun phrases. Functions do things, and so their identifiers will often use verb phrases to convey the actions the function will perform.

Second, a function needs a list of arguments. If you think about a function we've already used, the square root function, you'll remember it has a single variable passed to it - or one argument that's either an int or a float. An argument is a variable within the function that gets its initial value from whatever outside code calls (or executes or invokes) the function. Functions can have no arguments, one, two, or as many parameters as is necessary. Generally, though, a function should be limited to a few arguments before it just gets unweildy to use and manage. Having no parameters to a function often means it will do some action based on outside data. We'll use this in our next program to get user input for the rugs program.

Third, a function needs a body. This is the computer code that will be executed when the function is run. It can create its own variables, it can use the function arguments as variables, and in some languages it can use data nearby the function. (This is covered in the TypeScript workbook.) Just like a body of a while loop or if statement, the function body will execute from top to bottom when the function gets called. Unlike the body of a while loop or if statement, the variables inside the function are in a new **scope**. A scope is a way to say that there's a new grouping of variables, independant from other groups or blocks. This means that two different functions can both have a variable named `area`, which are completely unrelated to to one another!

Finally, most functions need to produce a result. This is called returning a value. Almost all functions will return a value, but there are some cases where they work only on arguments, or only do something to cause a side effect (like writing a line of text). We'll cover these as we come to them. When a function returns a result, the function takes whatever value is on the right side of the return statement and provides it to wherever the function was called from. The function then immediatly finishes, just like when using break inside a loop!

Calling Functions

Once we have defined a function, we need to do something with it. On its own, it just sits there taking up programming space. To execute a function, from somewhere else in the code we will **call** the function. Calling a function, sometimes referred to as **executing** or **invoking** it, requires us to specify the function name and then provide a list of arguments that will be the values the function uses.

We have now used the phrase "list of arguments" to mean two differnt things. In the last section, the list of arguments was what we used to name all the variables which the function needed to have filled in from whoever asked it to do work. Here, we use the same phrase to describe the values the calling code will hand to the function for it to work on.

This process is kind of the opposite of asking for user input. When we asked for user input, the code waited for the user to provide data, then did calculations on it, and then provided the user output. For a function, we hand it all the data it will need right away, as input via its arguments. It then executes, and when it finishes it hands us data back as the function’s output via its return statement. Be careful, though, because while this function input and output is analogous to user input and output, they are still different things and we still use both in our programs!

That was a lot of theory, so now might be a good time to head to the workbook and do the first section in this chaper, getting practice with this intro to functions.

Function Tracing

In chapter 1, we explored how computers execute programs and how their memory changed over time by making traces of their memory as we pretended to execute the programs in our head. The same tools work to help us understand how function calling and returns work, as well. We’re going to walk through this using the `square_rug_cost` function from the last workbook section. We are using the python code, but the principles work the same in any other languges.

```
def square_rug_cost(size, fringe):
    area = size ** 2
    cost = area * 5
    if fringe:
        perimeter = size * 4
        cost +=  = perimeter * 1.5
    return cost

length_of_side = 5
has_fringe = True
cost = square_rug_cost(length_of_side, has_fringe)
other_cost = square_rug_cost(length_of_side * 2, not has_fringe)
```

Let's start with our trace! Because this file uses functions, in our ID and values table we're going to add a third column to the left, which will have the name of the function scope we're currently executing.

```
Scope               |   ID             | Value
 ----------------| -------------- | -----
*file*              |                  |
```

Every file starts with a top-level scope, and we indicate that by writing `file` at the top of that column. We then start reading the file top to bottom. The first thing we see is `def square_rug_cost(size, fring):`. This is the function declaration, but it isn't the function's execution! So we don't do anything with it just yet. We make a note that this function is around, and skip down past its body.

Skipping down takes us to a pair of lines,

```
length_of_side = 5
has_fringe = True
```

Which just declares two variables. This we know how to handle:

```
Scope               |   ID             | Value
 ----------------| -------------- | -----
*file*              | length_of_side | 5
                    | has_fringe     | True
```

We have two new variables, we record their IDs, and we note their first two values. The next line of code, though, gives us something new:

```
cost = square_rug_cost(length_of_side, has_fringe)
```

Starting at the left, we see that we have a new variable, `cost`. Let's mark that down.

```
Scope               |   ID             | Value
 ----------------| -------------- | -----
*file*              | length_of_side | 5
                    | has_fringe     | True
                    | cost           |
```

Note that it doesn't have a value yet! The next thing we see is a function call. We know that a function creates a new scope, so we're going to make that new scope first. We do that by drawing a solid line below what we currently have and writing the function's name in the Scope column. We're also going to **immediately** draw an arrow from the name of the function in the scope and point the arrow at the `cost` variable we just created. We do this right away to keep track of where the function's return value will go!

```
Scope               |   ID           | Value
 ----------------| -------------- | -----
*file*              | length_of_side | 5
                    | has_fringe     | True
                  /-| cost          | 155
 ---------------|-|---------------|-------
 square_rug_cost| |               |
```

This has us set up now to work within the `square_rug_cost` function, in a new scope that is completely isolated from the variables above it. In the trace, any variables between a pair of horizontal lines can interact, but none of them can cross those horizontal lines! Ever!

> If you're tracing this by hand, you might want to look ahead and make note that we do have one more variable in the file scope, so maybe we should leave room for one more line before the function's scope. Just don't fill it in yet!

So now we have our new scope, we can start working with it. The first thing we need to do is fill in the arguments. We'll go ahead and note both arguments in the ID column right now

```
Scope               |   ID           | Value
 ----------------| -------------- | -----
*file*              | length_of_side | 5
                    | has_fringe     | True
                  /-| cost           | 155
 ---------------|-|---------------|-------
 square_rug_cost| | size           |
                  | fringe         |
```

And now we need their values. To do that, we're going to go back to the call site, `square_rug_cost(length_of_side, has_fringe)`. Remember again that arguments are positional. That means that to get the value for `size` inside the function, we go go the call site and get the first value in the calling argument list. The first thing in the argument list there is `length_of_side`. That's a variable, but we want a value, so we go to the trace, copy down the value, `5`, and write that value (not the name of the variable) into `size` in the `square_rug_cost` scope.

```
Scope             |   ID           | Value
 ----------------| -------------- | -----
*file*            | length_of_side | 5
                  | has_fringe     | True
                /-| cost           | 155
 ---------------|-|---------------|-------
 square_rug_cost| | size           | 5
                  | fringe         |
```

We then do the same thing for `fringe` in the second position. We go to the call site, see it uses `has_fringe`, we read `has_fringe` from the trace, get the value `True`, and write that into the function's scope.

```
Scope             |   ID           | Value
 ----------------| -------------- | -----
*file*            | length_of_side | 5
                  | has_fringe     | True
                /-| cost           | 155
 ---------------|-|---------------|-------
 square_rug_cost| | size           | 5
                  | fringe         | True
```

From here, we go into the function body, but the trace proceeds exactly like normal just reading code from the body and getting values from the scope. Let's do that through the loop, and we'll stop before the `return` statement:

```
Scope             |   ID           | Value
 ----------------| -------------- | -----
*file*            | length_of_side | 5
                  | has_fringe     | True
                /-| cost           | 155
                | | other_cost     | 700
 ---------------|-|---------------|-------
 square_rug_cost| | size           | 5
                  | fringe         | True
                  | area           | 25
                  | cost           | ~125~ 155
                  | perimeter      | 20
```

We calculated the area, we computed the area cost, we got the size of the perimeter, and then we updated cost with its old value plus the new cost of the perimeter. Now we're ready to look at how to handle `return cost`.

For the return statement, we create a "virtual" variable and call it `return`. It will have the value of whatever is on the rest of the line, whether that's a single variable like here or a complex calculation. We then take this value, follow the arrow from the top of the function back to whatever variable we were calculating, and put the return value there as well.

```
Scope            |   ID           | Value
 ----------------| -------------- | -----
*file*           | length_of_side | 5
                 | has_fringe     | True
               /-| cost           | 155
 ---------------|-|---------------|-------
 square_rug_cost| | size          | 5
                  | fringe        | True
                  | area          | 25
                  | cost          | ~125~ 155
                  | perimeter     | 20
                  | *return*      | 155
```

Once this is done, the function is over. Finished. Done with all its things. That means that this version of the functions's scope is forever lost to us. Litearlly. After we return, we can never go back to see what its values were! To indicate this on the trace, we're going to write a final horizontal line at the bottom and then write a big X through the function.

```
Scope            |   ID           | Value
 ----------------| -------------- | -----
*file*           | length_of_side | 5
                 | has_fringe     | True
               /-| cost           | 155
 ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
 square_rug_cost| ~ size          ~ 5
                  ~ fringe        ~ True
                  ~ area          ~ 25
                  ~ cost          ~ ~125~ 155
                  ~ perimeter     ~ 20
                  ~ *return*      ~ 155
 ------------------ --------------- ----
```

So from our trace, we remember it happened, but as far as the compute is concerned that work is lost. The only thing left is a final copy of the return value, `155`.

Now that we're back to our file execution, we can look at the next line of code.

```
other_cost = square_rug_cost(length_of_side * 2, not has_fringe)
```

We have a new variable, `othe_cost`, which calls `square_rug_cost` with some new values. Let's set up the scope that we'll be using for this:

```
Scope              |   ID             | Value
 ----------------| -------------- | -----
*file*             | length_of_side | 5
                   | has_fringe     | True
                 /-| cost           | 155
/ -               | | other_cost     |
 |~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
 |square_rug_cost| ~ size           ~ 5
 |                 ~ fringe         ~ True
 |                 ~ area           ~ 25
 |                 ~ cost           ~ ~125~ 155
 |                 ~ perimeter      ~ 20
 |                 ~ *return*       ~ 155
 |----------------- --------------- ----
\square_rug_cost  | size           |
                  | fringe         |
```

We did a few things here. We made a new row for the `othe_cost` variable. We started a new scope for the function below the old scope, which we had crossed out. We gave two initial rows for the `size` and `fringe` arguments. And we drew an arrow from the name of the function at the top of the scope back up to the variable which will take its value when we return.

Time for the arguments! `size` takes the first argument from the call site, which is `side_length * 2`. Looking up `side_length` in that scope, we see it is `5` so we jot down the value of the calculation, `10`. Then we look at the second argument at the call site, `not fringe`. Since `fringe` in this scope is `True`, `not fringe` must be `False`. Let's fill those in to our table:

```
Scope              |   ID             | Value
 ----------------| -------------- | -----
*file*             | length_of_side | 5
                   | has_fringe     | True
                 /-| cost           | 155
/ -               | | other_cost     |
 |~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
 |square_rug_cost| ~ size           ~ 5
 |                 ~ fringe         ~ True
 |                 ~ area           ~ 25
 |                 ~ cost           ~ ~125~ 155
 |                 ~ perimeter      ~ 20
 |                 ~ *return*       ~ 155
 |~~~~~~~~~~~~~~~~~ ~~~~~~~~~~~~~~~ ~~~~
\square_rug_cost  | size           | 10
                  | fringe         | False
```

Now we can just go through the function like normal, making a fake variable in the trace when we get to the return statement:

```
Scope             |   ID            | Value
 ----------------| -------------- | -----
*file*            | length_of_side | 5
                  | has_fringe     | True
                /-| cost           | 155
/ -             | | other_cost     |
|~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
|square_rug_cost| ~ size           ~ 5
|                 ~ fringe         ~ True
|                 ~ area           ~ 25
|                 ~ cost           ~ ~125~ 155
|                 ~ perimeter      ~ 20
|                 ~ *return*       ~ 155
|~~~~~~~~~~~~~~~~~~ ~~~~~~~~~~~~~~~ ~~~~
\square_rug_cost  | size           | 10
                  | fringe         | False
                  | area           | 100
                  | cost           | 500
                  | *return*       | 500
```

Notice a couple things here. Because `fringe` was `False`, we never do the `if` block, which means we never define `perimeter` nor ever change `cost` to the area cost plus the fringe cost.

The last thing to do, then, is copy that value `500` back to where we asked for it and close out the frame:

```
Scope             |   ID            | Value
 ----------------| -------------- | -----
*file*            | length_of_side | 5
                  | has_fringe     | True
                /-| cost           | 155
/ -             | | other_cost     | 500
|~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
|square_rug_cost| ~ size           ~ 5
|                 ~ fringe         ~ True
|                 ~ area           ~ 25
|                 ~ cost           ~ ~125~ 155
|                 ~ perimeter      ~ 20
|                 ~ *return*       ~ 155
|~~~~~~~~~~~~~~~~~~ ~~~~~~~~~~~~~~~ ~~~~
\square_rug_cost  ~ size           ~ 10
                  ~ fringe         ~ False
                  ~ area           ~ 100
                  ~ cost           ~ 500
                  ~ *return*       ~ 500
 ~~~~~~~~~~~~~~~~~~ ~~~~~~~~~~~~~~~ ~~~~
```

There you have it - a concise visual representation of how the computer memory keeps function scopes separate, how it returns variables from functions, and how once a function has finished executing, none of the memory that went into figuring out the calculation is left to the computer.

Exercises

TODO

When you've had a chance to get more comfortable with traces, commit your work, take a break, and when you come back we'll look at strings and string formatting in the textbook to improve our user messages.

Strings

We briefly discussed strings in chapter 1. We're going to take a deeper dive in this section, exploring some a few utilities that will make our lives much easier.

A string is a (possibly empty) list of characters that are grouped together. You can create them manually, or get a string from what the user typed with `input` in Python, `readline` in TypeScript, or `scanf` in C. Once we have a string, we have seen how to compare it to match exactly the contents of another string. That does have the consequence of needing to be case sensitive, so if we want to check if the user typed a "y", we would probably need to check for both `"Y"` and `"y"`.

Once we have some strings, we've seen one way to put them together - using the `+` sign. For numbers, this adds them, but for strings, it joins them together into a bigger longer string. We can use that to put the result of a variable into a bit of text formatted for human readability. To do this, we do have to ensure all the things we're concatenating in this way are themselves strings. Python and C have specific functions to do this, while TypeScript does it implicitly. There are also functions in all our languages to take a string and parse it (read the characters and determine the appropriate value) into any of our other data types, most often a number of some sort.

Because this process of joining strings, and using functions to convert to and from strings, is tedious and error prone, we have tools that can take a "template string" and fill in the values per some specification. This achieves several benefits. For one, it means that the template string itself can be a single entity we pass around, rather than a complex expression. This lets us change the text quickly, without having to change where we `+` the strings together. The template strings have facilities to describe how we want our data formatted. When we have printed out floating point numbers, we often accidentally get things like `4.0000000000001` as a result of round off. With template strings, we can ask for at most 2 or 3 digits, and fill them in with `0` if the number is whole.

There are a few more common operations on strings that we have libraries for. These usages don't come up every day, but do show up regularly enough that we can discuss them & see some examples, so you'll at least have the memory they exist when you come across a problem that could use them.

The first is the idea of substrings. A **substring** is exactly what it sounds like - a part of a larger string. Substrings are used regularly when you need to do any of your own parsing - any time you need to write a program that looks into a string and pulls content out of its data. Substrings are always specified as the starting index within the string, and either the end index (exclusive) or the number of characters to take. The ability to change and edit a string based on its substring depends on each language, but each language does provide a mechanism to do so.

An **index** in this context is just a numerical count of number of characters into the string to look at. It's important to know and remember that indicies start at `0` - the don't give you the `1st` or `3rd` character in the string, but rather specify the number of characters to skip before it starts reading. So the `1st` character is at substring index `0`.

Finally, there are a few methods that a string lets us use to manipulate it. We can take a string and make it all **uppercase** or all **lowercase** (or get a substring and make that substring all upper or all lower). You can often **count** the number of occurances of a character or (shorter) string inside another. And there are utilities to **find** occurances of a string in another, and then **replace** those occurances with some other value.

There's a lot to cover for strings, but in the workbook, we focus on formatting strings for user interfaces and displays.

Arrays

The programs we've written so far work on one piece of data at a time. Each variable holds a single value, and while we've looped work based on what the user wants to do, we haven't done anything that uses multiple pieces of similar data. Pricing one rug is helpful, but what if I have a whole warehouse to price?

We handle this by using arrays. Arrays are themselves a data type, but unlike the data types we've encountered so far, they are comprised of multiple pieces of some other data type. Actually, there is one type we've seen like this, the string! We can think of a string as an array of characters, and in a lot of ways that will help us ground our understanding.

So an array is a list or collection of lots of data of the same type. Instead of using a different variable to access each piece of data, we use a single variable that references the array as a whole, and then we use an **index** to access individual elements of the array. We discussed indexes when we talked about substrings, but they're really important so we're going to repeat them again here!

An index is an **offset** into an array. That means that rather than saying the index is the "1st" or "2nd" or "4th" element, the index is the number of elements to skip to get to the element we want. This means that (in any language we will use) the first element of an array is actually at index 0! And if you have an array with 20 items, if you access the element at index 20, you'll actually be off the end of the list entirely! This is generally considred a Bad Thing, and you are discouraged from doing so.

The most common thing we'll do with an array of data is iterate or loop over it, doing some operation for each item in the array. This is in contrast to the looping we've done so far, which each looks for a condition to be met with a `while` loop. In the `for` loop, we give it an array and access every item sequentially, stopping when we've run out of items to check!

Later in the book, we'll look at arrays with all sorts of data types in them, and get a lot of interesting behavior. For this section, though, we're going to stick with good old fashioned numbers, and write some functions that calculate statistics about these numbers.

Changing arrays

Arrays are not static. Just like how we can assign different values to variables, so to can we change an array. We can do this by assigning to a single index. We can also change the number of items in an array, adding items by **appending** them to the end (and making the array larger) or **removing** them (and making the array smaller). Like strings, we can join two or more arrays together using concatenation. Some languages like Python and TypeScript even let us slice and dice chunks from the middle of an array!

In the workbooks, we're going to start using arrays to perform statistical analysis of some data sets.

Wrap up

In this chapter, we took our basic toolkit of programming fundamentals and added three powerful and useful items. We discussed **functions**, which allow us to bundle up chunks of code and make them generally useful in a wide variety of sitations, rather than only working for a single specific case. This includes giving **identifiers** or names to functions for a block of code, taking input as **arguments** to those blocks, and often **returning** a piece of data as the finished calculation from the function. We discussed how to augment our tracing technique to handle functions in a clear and direct way. We then discussed **template strings**, which are compact but powerful ways to format text in a more legible way for the users of our programs. Finally we took a look at **arrays**, the first of our data structures which let us work with more than a single piece of data at a time. We wrote several functions to calculate statistics about data sets — numbers stored in an array.

In the next chapter, we will discuss objects which provide a mechanism to organize larger and larger amounts of data in understandable, tractable manners.

Introduction to Objects

In this chapter, we take our understanding of programming as variables tracking values over time and begin to add structure that will help us manage complexity as our code grows. We use **functions** to manage the complexity of breaking out executable pieces of code over time. Here, we start using **objects** to manage the complexity of pieces of data with multiple facets. We begin to get an understanding of these by building out a program which prints a desktop calendar.

The second project takes us from using built-in objects to building out our own objects. Returning to the trusty Rug Shop, **classes** will describe the data of a rug, and **methods** will let us cut down yet again on the code needed to keep all the rug types around. We will gain a large amount of flexibility in how rugs are stored and priced, and can use this to quickly create many more types of rugs.

The last section will take us back to video games. We'll build out a set of classes which let us have a maze or labyrinth that our intrepid player must navigate, collecting keys for locked doors before finding the treasure room.

Objects

Variables in computer programs track data. We saw some of the kinds of data they can track in chapter 1. In chapter 2, we saw how computers can use variables to track calculations across a number of functions. Whether this is as internal variables to remember an intermediate result, in arguments to take or pass data, or in return statements to hand data back the the caller, variables are integral to making programs work.

Sometimes, though, we want a bit more structure than what we can get just from naming them differently. What we want as programs get complex is a way to bundle several related variables together. In this way, we can treat them as a single entity in our program. We can create them once, modify each part individually, and keep the whole thing together when passing it to another function. This also helps understanding - rather than talking about an area a width and a height which we keep in our heads as being about one rug, we can instead talk about a single rug, which has an area, a width, and a height. It may sound pedantic, but in practice it really makes tracking complex pieces of data much easier!

These bundles of data are called **objects**. Each piece of data bundled into an object is a **property** on an object. We can think of objects having "shape" or "structure" based on what properties it has on it, and the types of data that are stored in those properties. A property is, in many ways, just a variable that is attached to a single object in our program. The property has an **identifier** and stores a **value**. It's just that it can't appear alone, and only is part of the object.

As we work with objects, these pieces of data with many facets, we often want to talk about their "structure" or their "shape". For simple data, in the last two chapters, we talked about the type of a value. It could be a number, a string, or a boolean. For complex data, the type of the value is described by its shape or structure. "Shape" and "structure" in this context describe the properties on an object, and the types of values that go in each of those properties. Let's consider a date for a moment.

Usually, we write a date in one single block - 2020/08/12. We might consider using a string to hold the 10 characters. But when we think about the date again, we often want to think about the parts of the date itself. What year is it in? What month? Which day of the month is it? With only strings, we would be required to do some sort of matching and conversion, like "for the month, take the 6th and 7th characters, and convert them to a number." With the concept of objects, we would instead store each of those pieces separately. Instead of using a variable to store the single value `"2020/08/12"`, we can use an object to store three properties - `year: 2020, month: 8, day: 12`.

The structure of this complex data object has three properties, year, month, and day, each of which have type number. We can call this structure of three properties with these names the **type** of the data object. We've now created a new data type, beyond what the programming language originally provides. Instead of working with a combersome single string, or needing to lug around three variables with numbers, we can create a single Date object and it has all the pieces we need to do calculations on a date as a whole.

A definition of a complex object like this is called an **interface**. It is the public description of what the object is, the full description of its shape.

An object is a value, which means you can assign it to a variable. You can pass it to a function as an argument. You can even assign an object into a property of another object! Building out complex objects like this is how we'll later begin building **data structures**. That's a couple chapters away - in the mean time, just keep in mind objects being a bundle of related data, each piece of which you can access from its properties. Let's get some practice looking at dates, objects which store information about a moment in time.

Classes

In our calendar program, we used objects that we got from our language's date module. This is great for common features when we need them, but we also want to define our own object types. We do this by creating a class. A **class** is a description of the type of object that we want, as well as a group of functions which operate specifically on this data type. These focused functions are called methods, which allows a class to bundle both data and the operations that are effective on it.

Once you have a class which describes a bundle of data, we use it to get the object. A copy of an object created from a class like this is an **instance**, which has data specific to one copy of the class. Where the class describes the data in abstract, and you only have one class for any time, you can have any number of instances of the class. You create a new one any time you want data of a certain shape that matches the class, fill it in with the specific data, and can then use it any where that expects a value of this type. Just like some functions take numbers or strings, they can also take objects to operate on.

All classes start with a **constructor**, a function which specially creates and initializes an object of that type. The constructor runs any time you want to get a new instance, and it fills in all the properties on the object. Constructors, as functions, can take arguments which allow it to specify any values it needs to correctly initialize the state of the object. Once the constructor has finished running, our program has a new value of this class' type, each of its properties filled in by the constructor.

Almost all classes have methods. **Methods** are functions which are attached to the class and whenever they are called, take an argument either explicitly or implicitly of the instance to work on. Methods can also take additional arguments, and they can both change the state (values of internal properties) of the object as well as returning a value. Usually, a method will either change internal state, or it will return a value calculated from internal state.

As an example of this, in the workbooks we will create classes for our rugs. We have been dealing with square, rectangular, and circular rugs, but all of these are doing basically the same set of operations - calculate area, possibly calculate perimeter, ask for some inputs, calculate a cost at the end based on the area and perimeter. In the past chapters, we've had functions which handle each of these separately. Our programs had to keep track of the inputs, add up the total cost, print it out, just passing values over and over.

With classes, we can instead define how our general data will behave. It will handle asking for input in a contained way, using a `get_values` method which each different rug type will use to get what is important to it. We can have a `cost` method which delegates to `area` and `perimeter`, which can be written specifically to each type of rug we have.

One incredibly powerful feature of classes is the idea of **inheritance**. Inheritance lets us create a class with a reference to some other class, which it can then add to or partially modify. This is powerful because we can describe general classes which cover a wide range of cases, and then make later classes which are more specific to their cases later.

This will let us organize our rugs code even further. We'll have a **base class** (also known as a **super class**), `Rug`, which knows how to calculate cost and ask whether it has fringe. We can then make **derived classes** specific to square rugs and rectungular rugs and circular rugs to handle each specific case of getting the area and perimeter for each of those shapes.

The end result of all of this should be code that is much more self contained, concise, and understandable by ourselves and any future reader.

Tracing Objects

Now that we're starting to develop our feel for using objects as a way to bundle data, let's take a closer look at how the computer handles the data in an object under the hood. We'll do that by extending the notation we use when tracing a program.

In our traces for chapters 1 and 2, we kept a two-column table of identifiers and values, which represented variables in our program. We separated the rows by functions, and used arrows to track where return values came from and went to.

For objects, we could write out the entire set of properties and their values in the value column, but that seems difficult as objects get larger than maybe two or three properties. It also doesn't give us a reasonable picture of what is actually happening in memory. Instead, whever we create an object, we're going to draw a T in the right half of the trace. Above the T, we're going to write what type the object. And then we're going to use the left and rigt parts as identifier and value - of the properties of the object! So just like each function has its own scope in the main table, over on the side here each object will also have its own little area to track its data.

So what do we put in the value for the variable in the function? We're going to draw an arrow from the value box and point at the object we just wrote out! This lets us clearly see where each variable is pointing, when it has an object value instead of a simple value.

Software Design and Architecture

On our journey into the world of Software Craftsmanship, we've completed our first steps. We have an understanding of the way computers run and store values. We've used functions to abstract reusable blocks of executable code, and we've used objects to abstract bundles of data. These core pieces are all we need to write software. Everything else is just using them - either built in to our language, from a library we include, or that we've written ourselves. Of course, you probably aren't comfortable with that level of power yet!

The rest of this book explores how to take these tools and use them in effective ways. How to take the objects we're creating, and develop & iterate on their designs so that they effectively solve our problems. How to verify the correctness of our objects and functions, both so they meet our original requirements as well as so they don't break when someone edits them later. How to evaluate, learn, and utilize libraries from other developers to short cut lots of effort using pre-built and off-the-shelf tools. How to work in a team and collaborative environment to build tools that work for more people in a wider range of settings.

That's a lot left to cover, but in this section we're going to start small. We're going to look at a problem statement, discuss several approaches to designing code which will handle that problem, and then implementing one version based on our analysis.

Maze

We're writing a video game called "Maze". In this game, the player starts in some room of a maze. Their goal is to find the exit of the maze. They can wander the maze going through doorways which connect rooms. Some of these doors are locked, and the keys to any locked doors are place in rooms throughout the maze. Let's look at an almost trivial sample maze:

```
  012345678
6 ┌───────┐
5 │ X   │
4 ├ │ B │ ┤
3 │     ├───────┐
2 ├───╫───┤ B   │
1 │ E ╪     │
0 └───────┴───────┘
```

In this maze, the player starts in the room on the bottom left marked `E`. This room has two doors, one to the north and one to the west. The room to the west is a bit larger, and has a key marked `(B)lue`. North of the entrance room, there's an empty room with another door again to the north. This door is locked and needs the blue key to unlock it. Once through that door, the player is at the exit and has beaten the level! To solve the maze, the player clearly needs to visit the room with the blue key in order to make it out.

While we don't have much programming under our belts, we can at least start to think of a couple ways to represent this. One possibility, looking at the one drawing we have, would be to represent the maze as an array of arrays. What this means is that each row in the maze is an array of characters or strings, and each character in that array we compare against a number or the ASCII values of the drawing we have. We would then need some notion of a player, which would need to figure out how to "walk around" in this two dimensional array. Let’s see what this might look like for a single function, `move_left`, in Python:

```
def move_up(maze, player):
    if len(maze) <= player.row + 1:
        # Can't move this direction
        return
    cell = maze[player.row + 1][player.column]
    if cell == 'X':
        print("Found the exit!")
    if maze[player.row + 1][player.column - 1] == '|'
            and maze[player.row + 1][player.column + 1] == '|':
        if not player.keyring.includes(cell):
            # The door is locked
            return
    if cell == ' ' or cell == '╫':
        player.rows += 1
```

If we call this function using the above maze and a player at row 1 column 2, we can see what happens. First, we check if the cell up one row is outside the bounds of the maze. In this case, it is not. So we look into the maze, first at row 2 and then inside row 2 at column 3. That cell has a □. It’s not an X, so we skip the first condition. Then we need to look to see if the cell above us is in a door - we do that by checking it it’s the | characters. They are not, so we continue to the last condition. If it’s an empty cell or the vertical open door we change the player so that they’re in the next row up.

This… is convoluted, to say the least. One of the most important design criteria in software engineering is the "single responsibility principle". It’s like the Occham’s Razor of programming. Any function, class, or other abstraction we create should choose one thing to do, and do that one thing well. Complexity should come from the domain the program is being written for, and not from the program itself.

Looking at this sample code from the single responsibility principle, we can see several competing concerns. Each `if` block really tries to do its own thing with some checking for the exit, some checking for open doors, others checking for locked doors. On second review, we actually see that this code doesn’t do anything in the case of moving onto a square with a key - try tracing the function starting at row 1 column 6.

Rooms and Doors

Let’s try a different design. The last design tried to create a model of the maze with the ASCII diagram being the "source of truth". With our notion of classes and objects, we can take a more abstract view of the maze, which will lend itself to clearer, more sraightforward code to traverse the maze.

Instead of looking at the maze as a grid of cells, we can create a class which represents each room.

The room, then, knows which room it connects to, and which keys it has. In the future, it would provide a convenient point to logically add more generic items. So if we think of rooms, instead of blindly looking around, we can have a player object that can just look at the room object it's in for what doors are around.

As we think through this, one issue pops up quickly - how do we keep track which doors are locked? With classes and objects, this is easy! We just introduce the idea of a door. A door is an object which has two sides, each being rooms, and a property for whether it's locked (and which key unlocks is). Now, how does the `move_up` code look in python?

```
def move_up(player):
    room = player.current_room
    door = room.north_door
    if door == None:
        return
    if not door.is_unlocked_by(player.keyring):
        return
    player.current_room = door.other_side(room)
    player.current_room.on_enter(player)
```

Comparing the two functions, it's hopefully apparent this is a much clearer as to what's happening. We know that the function is working with the north door (there's a clear connection between "move up" and "north door"). The bounds checking is very clear - when the door to the north is not present, the function does nothing. The door can check itself if the player's keyring has a key that fits it. And then, with the clear setup of what's happening, the player steps through the door - and lets the room on the other side tell the player what they just found!

Beyond teaching programming, a goal of this book is to teach programming in a way that shows how to make these types of decision, or at least where these decisions will arise in a setting of a larger development process. It is not an overstatement to say that code like the first example has grown to cause projects to run over time, over budget, and even to fail, while clear architecture and design during the software engineering process has directly led to projects coming in on time, in budget, and serving a long and useful life.

Let's take a look at the full example and explore this maze in depth in our language of choice.

Wrap Up: Objects, Classes, and Object Oriented Design

In this chapter, we've looked at bundling data into objects. This allows using a single variable to track complex data, accessing pieces of the complex data via its properties. We looked at built-in objects representing dates and times, and used those to build a calendar that showes us the days in a month, holidays, weekends, and days until some event.

We then explored using classes and inheritence to describe, create, and simplify object handling. With the rugs functions we've been using, we created a class for each type of rug. Each rug type was able to do its own calculations on its own contained data sets. There was still some duplication, so we introduced the idea of inheritance - letting one class provide foundational behavior, and several variations of that behavior in subclasses. This let us have clear, concise, contained code for each type of rug, and our input and output utilities didn’t need large amounts of special casing.

With the understanding of creating and using objects, we added a technique to track objects, their properties, and their lifecycle in our program traces. …

We ended the chapter building a maze game. This gave us more practice with classes, but more importantly, we used it as a jumping point for talking about object and program design. We saw how using classes and objects let us write clear code which shows its intent.

These first three chapters have given us the basic building blocks of writing computer programs. From here, we will look at techniques for building programs that are larger and more difficult to understand at one time. These tools will aid in developing these larger programs in a systematic and verifiable way.

Unit Testing

For most of the programs in the first few chapters, you may have picked up that every time we did a bit of work, we wrote some code immediately after it to see what it did and check that it worked. We'd erase it right away as we moved on, but having those little pieces of checking code served as a useful scaffolding while we took measured steps along our trail.

In this chapter we will formalize this approach by writing unit tests. **Unit tests** are small, concise programs which execute and verify isolated pieces and modules of our larger primary programs. By writing these smaller verification programs, we gain large amounts of confidence that what we wrote is in fact doing the correct and reasonable computation.

Unit tests should be small and simple so that they are easily verified. They should be focused to constrain and contain the areas of interest. Unit tests help us debug as we're writing, and help us catch any **regressions**, changes we later make which break the program.

- Unit Testing is validating programs work
- Unit tests are focused
- Unit tests are small
- Unit tests help debug as we're writing, and catch regressions later.

Anatomy of a unit test

Unit tests are built into a test suite. A **test suite** is a collection of unit tests around a group of functionality - for insteance, a test suite for a square rug. Within the test suite, we will have a number of **tests cases** which are short functions to verify a single behavior or edge case of the square rug.

Unit test methods generally have three parts. They begin by doing **set up** to get the objects into some state that is interesting. This could be creating a new instance of a rug, or building a maze with the Player in a certain room. The test will then **evaluate** some code. This might be calculating the area or cost of a rug, or insturcting the player to move through a door. Finally, the test will **assert** that some condition (or conditions) is true. This could be checking the value of the computation of the rug cost or area, or it could check that the player did move through the door to a new room (or did not move through a locked door).

Let's see what this looks like in practice. We'll write some tests for the Rugs and Maze programs we wrote in Chapter 3: Objects, and then discuss some advanced techniques in unit testing.

Rug Tests

For the rugs program, we want to write unit tests to verify all of the parts of the rug classes. We should have at least three - `SquareRug`, `RectangularRug`, and `CircularRug`. Each uses some functionality of the base `Rug` class, and also provides its own calculations for `area` and `perimeter`.

We'll write a test case for each class, then, and within each test case write individual tests for a rug of a certain size, with or without fringe. The tests will set up by create a rug of whatever type we're verifying, evaluate the calculatoins

1. Libraries
2. Test Files
3. Square Rug Tests
    a. Area
    b. Perimeter
    c. Cost
    d. Edge cases: 0 area, 0 perimeter

Advanced Topics

1. Before Each: Block of setup
2. Mocks: Replacing complex with simple behavior
    a. Test rugs input/output
3. Test Driven Development
    a. Red/Green/Refactor