Software Craftsmanship

David Souther

2020-11-27

© 2020 David Souther

NOTE

Copies of this document may be made for your own use and for distribution to others, provided that you do not charge any fee for such copies and further provided that each copy contains this Copyright Notice, whether distributed in print or electronically.

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!

- [Intro: Python](./01_python/README.md)
- ~~[Intro: TypeScript](./02_typescript/README.md)~~ Coming soon!
- ~~[Intro: C](./03_c/README.md)~~ 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.

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)