Learning to Program (The Do It Yourself Way)

Mitch Feigenbaum

1. Where to Start

One of the most difficult parts of learning to program is knowing where to begin. There are hundreds of languages, computers, and programs that promise to teach one the magical art of giving instructions to machines. Any of these programs that promise to teach one to program "In just two weeks" or some similar timeline are definitively a scam. With the advent of the internet, one can learn to be an expert computer hacker[1] for free on your own time by yourself. This book will not teach one how to code, nor is it intended to do so. What this book is meant to do is to get one on the write track and give one the tools to succeed.

But why am I qualified? Because I have experienced firsthand the trials and tribulations involved with learning how to program. The summer before my freshman year in high school, I decided that I wanted to learn to code. I made a list of languages I wanted to learn: `HTML`, `CSS`, `JavaScript`, `Python`, `Java`, `C#`, and `C`. My goal was to learn all those languages in one Summer, with no prior programming experience. I did not even know what differentiated any of the aforementioned languages. I had only heard of them in name. Many would agree that this is too lofty a goal. However I was young, naïve, and confused.

Suffice it to say that I did not reach my goal at the end of the Summer. By the end of the Summer I had learned `HTML`, a bit of `CSS`, `JavaScript`, and `Python`. I was nowhere near proficient, but I knew of the basic if-else, while, for, and function constructs. More importantly, I had fallen in love with the sensation of programming. To me, it was as satisfying as solving a puzzle or putting together a Lego set. Just like those activities, when one create a program, one can use it themself or give it to their friends. If one get good, they can even end up selling it to people and businesses. For an amateur programmer, there is nothing more rewarding than seeing others using and benefiting from a program that they themselves have made.

On my GitHub[2], I have posted many of the projects which I have made for fun. In the repository labeled 'Personal'[3], which is reserved for miscellaneous personal code that cannot itself warrant its own repository, I have written the following description.

> These are some personal files I have. I will probably upload here periodically. There is no real structure to this. Just stuff I made. Some of it is junk, some of it is brilliant, and some of it is unexplainable.

---

[1] The word 'hacker' is often used by the media and non-technical sources to mean someone who, often illegally, infiltrates computer systems. This definition may be attributed to a 'cracker' rather than a 'hacker'. The true definition of a hacker is one who is highly skilled at programming a computer and not just knowledgeable in theoretical computer science.

[2] The link to my GitHub, as well as many other resources to help on teh journey to become a hacker, are in the resources section.

[3] https://github.com/mrf-dot/Personal.git

The best way to learn computer science is not in a classroom setting; it is by example. That is why this book is structured in such a way.

If a person were to take a computer science course at a university, they would start with the theoretic and eventually make your way into abstracted levels of computing like web design. This is a good method to learn if one only wishes to know the theoretical basis behind computer science, but not optimal if your goal is to become a good programmer. The way that I learned to program was first through true fingers to keyboard coding. It was only later when I learned of the theoretical basis behind computer science. Don’t get me wrong: theoretical computer science *is* interesting. However; it is in fact true that one may gain more appreciation of theoretical computer science by first learning to do real programming. It brings meaning to the common constructs in computer science like pointers, strings, mathematics, floating point numbers, loops, and conditionals. Learning about theoretical constructs is a big *AHA* moment. It brings a new meaning to the programs they have created. If, contrarily, they learn to program when they have already learned theoretical computer science, it takes all of the fun out of learning how your programs truly interact with the machine they are run and compiled/interpreted on[4].

As I have already mentioned, learning to program is not something that just happens overnight. It takes months, even years to truly learn and master. When you are reading through this book, do not be afraid to read slowly or reread sections. Computer science is a difficult subject contrary to what many code boot camps would have you believe. Included in the sections are interactive exercises that will assist in gaining a more in depth understanding of each language. Hopefully this book will help avoid some of the common pitfalls and errors that befall many aspiring programmers.

Source code in this book will be clearly marked. Source code may be differentiated because it is offset and centered on the page, in a constant width font. Below is some example source code which prints hello world in `C`.

```c
#include <stdio.h>

int
main(int argc, char **argv) {
    puts("Hello, world!");
    return 0;
}
```

Clearly, the code is differentiable from the surrounding prose text.

[4] Later on in this book there will be an in depth explanation on the differences between compiled and interpreted programming languages.

2. Tools

No doubt the first thing a programmer must do when setting out in their career is creating their programming environment. Your programming environment is as much about the type of programs you use as what is in your physical surroundings. In the next part of this section, I will make some suggestions on the setups that are optimal for good programming practices. If these instructions are followed, it will set the programmer up for future success. However, note that these are merely suggestions. These programs are recommended for a reason: they work. If a programmer already has an environment that is comfortable for them to program in, then making a change to the programs suggested may simply not be worth it. It is up to the hacker to decide what tools they will use.

Below are listed required items needed to program. There is no need to get fancy. Use whatever is comfortable and inspiring.

Chair

This needs to be a happy compromise between comfort and professionalism. Keep in mind that you will be spending hours in hacking sessions, so make sure that the chair you use will be comfortable to sit in for long periods of time. For those who are health-inclined, a standing/walking desk might be right. Many people who have standings desks are very content and even claim that it improves their brain stimulation. Linux and Git creator Linus Torvalds has a setup in which he uses his computer while walking on a treadmill at low speed[5].

Computer

The type of computer a hacker needs varies based on the what they intend to do with it. I am not an expert on building PCs, so I am not qualified to give recommendations on specific parts to be used to build a computer. For a simple setup it might be optimal to get a laptop instead, especially if the prospective programmer is on a budget or must get a laptop anyways for school or work. Particularly if one wishes to use Unix-like operating systems such as Linux or *BSD, the Lenovo Thinkpad (particularly old models such as the X220) are prized. This is due to the lack of proprietary firmware such as EFI and secure/fast boot which impede operating systems other than Windows™. The specifications do not matter very much, especially if light editors such as vim are used. There is no need to spring for twenty gigabytes of RAM. If a computer can play Minecraft with a steady frame rate, it is more than capable to write and compile programs on.

Editor

The editor is where a programmer spends most of their time. The method in which a program is typically written is edit→compile→debug. Most time spent in the editor is not used on strictly writing code; rather, the time is spent looking for errors, consolidating existing code, and planning out the program. Thus, a good editor must not only have functionality to type characters into a file but also to jump between different parts of the file and make repetitive edits such as deleting entire lines or changing all instances of a particular variable name. In my experience the best editor for programming is VI, as it has native functionality to quickly perform repetitive actions through muscle memory. Many new programmers start out using an integrated development environment (IDE). IDEs may be useful for large projects, but can be confusing or misleading to those new to writing code. It is best to start out using an editor such as Vim, and, if the functionality of

[5] https://www.youtube.com/watch?v=SOXeXauRAm0

an IDE is truly needed, switching to an IDE later. Many IDEs such as Microsoft® Visual Studio™ have extensions that allow use of VI keybindings. Even if a programmer is not per se using VI or a VI clone, knowing VI keybindings is a useful skill that pays dividends. As a side note, it also looks really cool for the uninitiated.

Headphones

When writing code for long stretches of time, it can be helpful to have music to listen to. Listening to music can help pass the time and make the programming process less frustrating. Headphones usage is a personal preference. Some people simply cannot focus if there is music playing in their ear. Others derive productivity benefits if they are listening to music. A good pair of over the ear headphones can make programming a lot more bearable. Note that over the ear headphones are best, as on the ear headphones or earbuds will become uncomfortable after a short period of time. Over the ear headphones are also typically better at keeping distracting outside noise out.

Keyboard

The keyboard is the primary way that a programmer interacts with a computer. It is highly recommended to get a keyboard with high key travel. Mechanical keyboards are the best, but gel keyboards may also do. Keyboards with short key travel, like those typically found on laptops, will cause repetitive strain injury. As an additional note, it is a de facto requirement for programming to be able to touch type. The higher the words per minute (WPM), the better. A rate of about forty WPM is the minimum requirement to effectively be able to write programs.

Monitor

The monitor setup is intended to reduce eyestrain and give the programmer a view of all their code. A three monitor setup is optimal, as typically a programmer will have three programs open at the same time. On one monitor the editor is open, where code is written, deleted, and modified. On the second monitor a terminal window is open, where the program is compiled and run. On the third monitor is documentation or a web browser for reference.

Mouse

The mouse is not extremely important when programming. Some editors and most IDEs rely on the mouse for a number of actions. If a keyboard shortcut based editor such as VI or EMACS is used, the mouse will almost never be used. When programming the mouse will be mostly used to switch between editing and compile/debugging windows. If using a laptop, having an external wireless mouse can be more ergonomic.

Notepad

A pen and paper should be in reach when writing code. Writing information like variable names and procedure descriptions on paper can be quicker than having to scroll through code.

Operating System

A Unix-like operating system is ideal for programming. They have native access to the latest GNU tools for programming are by default equipped with command line editors and programs. Programming on the command line in an operating system such as Windows™ requires the installation of a host of software and a bit of tweaking. If one must program on Windows™, there is a bootstrap guide which will simulate a Unix environment to the farthest extent without requiring a switch to Unix or administrator privileges[6].

2.1. Vim: The Programmer's Editor

Vim is an editor based on keybindings rather than mouse based context menus. The mouse is never used in vim. Rather, different modes are entered to modify text. In my opinion, Vim is the best editor for programming. It allows a level of modification at such high speed that it is sure to give improvements over simple editors like notepad, and its free and open source. If you wish to use VI, I recommend using Neovim[7] as this version tends to be the most up to date and features many features out of the box that must be specified with a build of Vim. It also has some features enabled by default like syntax highlighting and language servers that must specifically be enabled with Vim. Using a VI based editor is a learning curve, but those who put in the effort unequivocally do not regret it. If you wish to use Neovim, install it using the instructions for your particular operating system[8]. Once Neovim is installed, you may learn how to use it with the following shell[9] command.

```
nvim -c Tutor
```

If you wish you may use my configuration[10], which contains shortcuts for the `C`, and`Java`, `HTML` programming languages.

2.2. UNIX: The Programmer's Operating System

In computer circles oft heard are praises of UNIX and its offspring Linux, BSD, and MacOS. The UNIX environment may seem strange to newcomers. One principal of the UNIX environment which seems particularly odd is that everything is a file. Programs are a file, directories (folders) are files, and even hard drives are files. This idiom however is in the interest of simplicity. By stating that every object is a file, it allows for a programmer to specify the creation of a file without giving it superfluous information. Information such as type, permissions, and creation are stored in structures called inodes, which link to the physical file on the drive. Programs on UNIX are simple and general by design, and can yield specific input through mechanisms such as piping and conditional execution. This book will not go in depth on the specifics of the UNIX operating system or the shell, but the AT&T Documentary "The UNIX Operating System"[11] is a great reference for starting out in UNIX.

[6] https://github.com/mrf-dot/linuxize-school-pc

[7] https://neovim.io

[8] My aforementioned Linuxize PC guide for Windows™ installs Neovim and my configuration automatically.

[9] The 'shell' is the language used when you open up a command prompt.

[10] https://raw.githubusercontent.com/mrf-dot/deb-bootstrap/main/nvim/init.vim

[11] https://www.youtube.com/watch?v=tc4ROCJYbm0

3. Logic

Logic is the basis of all computer programming languages. By learning some simple rules of logic, a programmer can consolidate their code and better check it for errors. There are three basic logical operators that are used often in programming. In the next section “Operators”, these logical operators will be reviewed.

3.1. Operators

$\neg$	The *not* operator negates the statement. It checks if the compliment (opposite) of a logical statement is true.
$\vee$	The *or* operator checks if one or both of two statements is true. Only if both statements are false will it return false.
$\wedge$	The *and* operator checks if both of two statements are true. If one or both statements are false, it will return false.

3.2. Truth table

A truth table asserts the result of a logical operation in multiple circumstances. In computer science, the Boolean values true and false may be represented by the numbers 1 (true) and 0 (false). Below is a truth table of basic operations using the $\neg$, $\vee$, and $\wedge$ operators.

Statement	**Value**
$\neg 0$	1
$\neg 1$	0
$0 \vee 0$	0
$1 \vee 0$	1
$1 \vee 1$	1
$0 \wedge 0$	0
$1 \wedge 0$	0
$1 \wedge 1$	1

3.3. De Morgan’s laws

De Morgan’s laws are logical proofs that stipulate

1 The compliment of not A or B is equivalent to not A and not B and

2 The compliment of not A and B is equivalent to not A or not B. The following proofs show the validity of De Morgan’s laws.

Proof 1
let $x \in \neg(A \cap B)$
$x \notin A \cap B$
$x \in \neg A \vee x \in \neg B$
$\therefore\ \neg(A \cap B) = \neg A \cup \neg B$

Proof 2
let $x \in \neg(A \cup B)$
$x \notin A \cup B$
$x \in \neg A \wedge x \in \neg B$
$\therefore\ \neg(A \cup B) = \neg A \cap \neg B$

4. Programming Constructs

All programming languages are different. Speed, efficiency, verbosity, and comprehensibility vary wildly based on the type of language. However, across all programming languages, there are certain constructs that emerge. These constructs may be called by different names or implemented differently, but for the most part are present in multiple languages. By learning these abstract constructs, knowledge of them may be applied to speed up the process of learning a new language.

4.1. Comments

Over the course of a program it is good practice to write documentation. This is the idea behind code comments. Code comments can describe the purpose of code, clarify confusing code, and help the programmer remember what a piece of code does. Comments are identified by a certain start and end keyword. When a program is run, comments are ignored and thus not interpreted as actual code.

A tough task for beginners is deciding when and where to comment. There are a few good simple rules to follow.

1 Comment at the start of your program. This comment should state the purpose of the program and the algorithm that it uses. If a program is about an existing concept, like calculating body mass index, it is a great idea to link to a url that describes the concept. List any web links that were references for the program to give proper credit.

```
/*
 * This program calculates BMI, or body mass index.
 * More information can be found at <this> link.
 */
```

2 Comment at function definitions. The same rules apply for the comment at the top of the program. Also list what the parameters and return mean in the context of the function.

```
/*
 * This function calculates BMI based on
 * weight (in kilograms) and height (in meters).
 */
```

3 Comment lines that are excessively long or complex. If a line is doing many different operations or is hard to read, having natural language descriptions of what the line does can be a big help for maintanence. This comment should describe what the line is doing in code in the natural language, as well as a justification for the complexity of the line.

```
/* This line calculates BMI from the provided arguments. */
bmi := kg / mˆ2
```

4 Comment out lines of code that are not in use. Comments can be used to make the program not execute lines of code. If there are errors in the program, commenting out specific lines of code can help trace the error down. This is a good alternative to deleting lines of code that might not necessarily be faulty. When a program is completed, these commented out lines of code should probably be deleted.

```
/* The following line needs to be fixed. */
/* kg := 703 / (lb * 0) */
```

One danger of commenting is overindulgence. Overcommenting can have the opposite of its intended effect; complicating code. It is easy to avoid overcommenting. If a comment contains art, then it should probably be removed. Also refrain from commenting on every line. If a program contains more comments than code, then odds are that some comments are uneeded, superflous, or distracting.

4.2. Input/Output

There are three standard ways in which a programmer and a computer can communicate with each other. Standard output (*stdout*) is the way in which a program can give feedback to the programmer. A print statement gives the programmer an insight into what the program is doing. It has many uses. One use might be to print the result of a complex equation. Another common reason that print statements are used are to debug programs. If a program seems to output the wrong result, print statements may be used to find the exact point in the program where the error is present. One of the simplest and first programs one writes in a language is called *Hello world*, which simply prints the string “Hello, world!” to*stdout*. In pseudocode a hello w orld program could be written like the following.

```
print "Hello, world!"
```

Print statements may also be used to access the second stream, standard error (*stderr*). The stream *stderr* is intended for problems that may arise in a program. An example usage of *stderr* would be to warn a user that a program has been forcefully terminated.

Print statements not only take in letters and numbers but also formatting commands. There are many characters that are difficult to represent in their true form. Characters like newlines (pressing enter), tabs, backspace, and return (go to the start of the line) can all be represented by escapes. In most programming languages, escapes are represented by the backslash (\) character.

Programs can obtain data from the user by accessing the third stream, standard input (*stdin*). The*stdin* stream contains the k eystrokes a user types on the keyboard. User input is needed in interactive programs, such as search engines or games. A program can use user input to determine whether to terminate or stay alive. The simplest use of user input is to print the data from *stdin* to *stdout*. Thus is the function of the`echo` program found in both W indows™ and UNIX shells. A simple input to output program can be displayed in pseudocode like this.

```
x := input()
print x
```

4.3. If-Else

The if-else construct executes code based on the value of a logical statement. Typically, a logical statement, which may contain and, or, and not signs (which vary based on language). The *if else* statement is used implicitly in *while* and *for* loops, another type of structured programming construct. One common task done in a programming language is determining whether a number is odd or even. A program built for this task could be made with a simple *if* *else* statement.

```
if n mod 2 == 0
    print "n is even"
else
    print "n is odd"
```

Traditional *if else* statements may only take one logical statement, and execute code based on the binary true or false value of those statements. However, with a technique called *nesting*, an if else statement can compute multiple scenarios under which different code is executed.

The game ‘Fizz-Buzz’ describes a scenario under which different outcomes may be specified in more than two situations. If a number is divisible by three, then the player says “Fizz”; if by five, then “Buzz”; if by both three and five, then “Fizzbuzz”; otherwise, the player simply says the number.

```
if n mod 3 = 0 and n mod 5 = 0
    print "Fizzbuzz"
else
    if n mod 3 = 0
        print "Fizz"
    else
        if n mod 5 = 0
            print "Buzz"
        else
            print n
```

Although this code is correct, the amount of indentation required makes the code hard to follow and understand at a glance.

To correct this, many programming languages have a way around this amount of indentation. In languages that consolidate white space such asC, it is a common idiom to put both *else* and *if* on the same line separated by one space. In languages that do not consolidate white space, a special keyword such as *elif* is used. The above code to calculate the response to ‘Fizz-Buzz’ could be rewritten in this manner.

```
if n mod 3 = 0 and n mod 5 = 0
    print "Fizzbuzz"
elif n mod 3 = 0
    print "Fizz"
elif n mod 5 = 0
    print "Buzz"
else
    print "Fizzbuzz"
```

In comparison to the first code, this is much easier to follow and requires far less indentation.

An even more consolidated version of an if statement is found in multiple languages. The ternary statement does not solely execute code based on a logical statement. Instead, it returns a value based on the logical value. This feature is used in variable assignment and function return statements. In an above example an if statement executed code based on whether a number was even or odd. The code both scenarios executed was a simple print statement. Using a ternary statement, the code could be refactored into just one print statement.

```
print "is even" if n mod 2 = 0 else "is odd"
```

4.4. Loops

Besides conditional execution, there are scenarios under which a programmer would need to repeatedly execute a block of code. There exists multiple programming constructs for repeating, or *looping* through code.

The most basic of these is *goto*. All looping structures implicitly use gotos, as this is what code becomes when it is compiled down to assembly[12] language. Goto statements consist of a label and a looping point.

```
LABEL:
print "word"
goto LABEL
```

This will print the string "word" repeatedly until the program is terminated. In most real world examples, an if statement would be used to determine whether to goto the label or continue through the rest of the code. Why would a programmer wish to repeat code? One reason would be to check for input errors. If a program needs the input of a number, a scenario in which a user inputs multiple letters instead might crash the program; or at the very least produce incorrect calculations. If the code that captures the input is looped through until a number is inputted, the code will be guaranteed to produce safe and expected results.

```
NumberGet:
n := input()
if isnumber(n)
    print n + 10
else
    goto NumberGet
```

In early high level languages such as FORTRAN, goto statements were the only way to loop through code. In practice, this created code paths which could be impossible to decipher to a reader. As a result, code became unmaintained and fell into disrepair. The phrase "Spaghetti Code" is used to describe this style, as the paths the code takes are like following one noodle in a bowl of spaghetti. To rectify the situation, structured loops were introduced. These loops explicitly stated that they were present, and required a logical value to be used as an escape conditional.

A *while* loop is the most basic type of structured loop found in programming languages. A while loop is merely a goto with an if statement. Thus, it can be easily translated into goto and if statements. Now, the previous example can be written like this.

```
while not isnumber(n := input())
print n + 10
```

[12] Assembly language is the instructions that a particular CPU uses to give bytecode instructions to the central processing unit.

Loops are also conventionally used to repeat code forever. Forever, of course, means that the code will be continuously executed until the program is terminated. The UNIX program `yes` uses a loop to print the string “yes”, or a custom string provided by the user, until the program is killed. This program is simple to write using either goto statements or a while loop, like in the following code snippets

```
YES:
print yes
goto YES
```

or

```
while true
    print yes
```

4.5. Primitives

A primitive is the most basic component in a programming language. It usually refers to some type of number, although in some languages it may also be used for strings. Usually, the difference between different primitive types is the amount of memory each type takes up. The names indicate the amount of bytes taken up by each variable of the type. Some languages implicitly assume the type based on how the variable is declared, while others must have types explicitly declared with the variable.

Besides the amount of memory used, there are typically a difference between two different types of primitives. *Integers* are whole numbers which do not have the capability of storing a decimal point. *Floats* are decimal numbers which may store a limited amount of precision and a whole number. In almost all programming languages there are facilities to implicitly convert between integers and floats in order to perform mathematical operations. It should be noted that *strings*, which are arrays of characters, are not primitives. This is because a string can be broken down into its character components.

Although the integer types theoretically represent numbers, they are often used as a metaphor for other computer science constructs. *Boolean* v alues represent true and false. They can be used to store the value of a logical statement. Conveniently, the values true and false may also be represented by one and zero respectively. In most programming languages, the number zero represents false and every other number represents true. Programming languages may also supply a Boolean type. The Boolean type ensures that a variable cannot overflow and thus become a false value without the programmers intention.

Another usage of integer types that intends to represent another unit is the character, or `char`. The American Standard Code for Information Interchange (ASCII) is a common way that text is represented. It uses the values 1—256 to represent characters. For example, if a programmer wished to represent the letter ‘A’ in ASCII, they could assign the numerical value 65 to an integer variable. The print function for the langauge they were programming in would interpret the integer 65 to represent the letter ‘A’ and then print it to *stdout* or *stderr*. Words, sentences, and paragraphs might contain thousands of these integers in sequence. As with the boolean type, multiple programming languages provide an explicit type to hold ASCII characters.

The major differentiation between different integer primitives are the amount of bytes the numbers can take up. It is simple to determine the highest and lowest numbers that may be stored in a particular type if the maximum amount of bytes is already known. Because computers operate in base two, the maxium number is $2^{\text{bits}} - 1$ and the minimum number is $-2^{\text{bits}} + 1$ for signed integers. For unsigned integers, the maxium number is $2^{\text{bits}+1} - 1$ and the minum number is 0. The difference between a signed and unsigned integer is that a signed integer may represent both positive and negative values. Unsigned integers can represent larger numbers because in a signed integer one bit is reserved to differentiate between positive and negative numbers.

A float is an extension of the integer. In addition to storing a whole number, it also stores a decimal point. Because real computers have limited memory, floats need to be rounded to a certain level of precision. Note the difference here between *precision* and *accuracy*. There are many scenarios in which a floating point number may have high precision but low accuracy. Take for instance the operation of adding the floating point numbers `0.1` and `0.2`. One would assume that this operation would yield the number `0.3`. However, in many programming languages this is not the case. This is the output of adding `0.1` and `0.2` in the Python interpreter

```
>>> 0.1 + 0.2
0.30000000000000004
```

Why does the itnerpreter have such an issue with this seaminlgy simple task? It all boils down to a difference in bases. Humans for the most part use a base ten system, where the numbers 0-9 are available for counting. Computers, however, only have two numbers available for counting (0 and 1). Therefore, when numbers are stored by a computer they are rounded in binary format rather than decimal. In binary 0.3 is converted into a repeating number. It should be noted that the same situation happens in decimal numbers as well. The fractional number $\frac{7}{9}$ is evalueated to the repeating decimal $0.\overline{7}$. In computer science lingo, issues in the accuracy of floating numbers are called round-off errors.

There are several ways to combat round-off errors. One solution is to format floating point numbers to a certain level of accuracy. if the result of the equation tested in the Python interpreter are limited to only two decimal places, the equation will “correctly” yield `0.3`. Some languages provide a specific decimal type, which is immune to round-off errors. The simplest way to protect against round-off errors is to use plain integers whenever possible. In accounting programs, instead of computing pennies as the floating point number 0.01 as $\frac{1}{100}$ of a dollar, balances are computed in pennies and only displayed to the user in decimal form. Just as there are often different sizes of integers, languages can also provide float types with different levels of precision. The floating type chosen by the programmer is based on the specific needs of a variable to hold a said amount of precision.

4.6. Functions

Computer code is often repetitive. While working on a program, especially if it is extensive, many operations will be repeated a number of times. The more code that is present in a program, the more likely there is to be bugs and mistakes. Additionally, a surplus of code makes a program harder to follow and thus more difficult to maintain. This issue is the reason why functions exist.

Let’s say that a programmer is coding a program to do trigonometric operations. They notice that they compute the Pythagorean theorem multiple times throughout the program and wish to simpify their code. First they start with the equation itself.

$$c = \sqrt{a^2 + b^2}$$

The equation takes the input of the two legs of a right triangle (a and b) and outputs the length of the hypotenuse (c).

The first part of a function is its name. Depending on the language, a function may need to be initialized with its return type or by a keyword such as *func*. The name of a function should describe exactly what it does. It should be short yet descriptive. A good name for a function which computes the pythagorean theorem would be `pythag`.

```
func pythag
```

The second part of a function is its parameters. Parameters are variables that are input into the function. Parameters in most cases must be explicitly declared in the function definition. Because the output of the Pythagorean theorem relies on the variable sizes of two sides of the right triangle, there needs to be a way for the programmer to communicate those sides to the function. The parameters are variables, but they only last until the function is completed. Like any other variable they have types, names, and can be operated on. For the Pythagorean function a good name for the parameters would be `a` and `b`.

```
(a, b)
```

The third part of a function is the function body. The body is where all code operations are executed. A function body should be relatively short and only do one thing. Functions can contain other functions, and even itself. As a general rule of thumb, what a function does should be able to be described in one sentence. This sentence can double as the description of a function contained in a code comment. If it takes more than that to describe what exactly a function does, the function is too long and vague, which makes it vulnerable to mistakes and harder to read. In the case of the pythagorean theorem, the description is short.

```
/*
 * Compute the hypotenuse of a triangle
 * based on the two adjacent legs.
 */
```

Just as the description is short, so is the body of the function. First the function should check that a and b are greater than zero, and then the function should calculate the Pythagorean theorem.

```
if a <= 0 or b <= 0
    c := -1
else
    c := sqrt(aˆ2 + bˆ2)
```

After the function body has calculated a result, that result needs to become available to the rest of the program. It does no good to have simply calculated the right answer, that answer has to be revealed. This is the reason for the return statement. The return statement is the value of a function. Once a return statement has been reached, the function terminates until it is called again.

The algorithm to calculate `c` in the Pythagorean theorem has already been defined. Now, the return statmeent must return c.

```
return c
```

With this, the Pythagorean function declaration has been completed.

```
func pythag(a, b)
    /*
     * Compute the hypotenuse of a triangle
     * based on the two adjacent legs.
     */
    if a <= 0 or b <= 0
        c := -1
    else
        c := sqrt(a^2, b^2)
    return c
```

The `pythag` function can now be used throughout the program. To invoke the function, a programmer must call it by its name and give it the appropriate number and type of arguments. One application of this function would be to receive user inputted legs and then find the hypotenuse of those legs. If the function yields -1, which is impossible for a real triangle, the programmer will know that the input was incorrect. The following would be a possible implementation

```
print "Enter the first leg"
leg1 := input()
print "Enter the second leg"
leg2 := input()
hyp := pythag(leg1, leg2)
if hyp = -1
    print "The length of one or both legs is incorrect"
else
    print hyp
```

A unique property of functions is that they can contain themselves. Functions that call themselves are called *recursive*. In mathematics, there are many algorithms that are recursive. The most famous of these is the Fibonacci Sequence. The Fibonacci sequence is calculated by adding the two previous numbers in the sequence together. The sequence starts with the numbers 0 and 1. In equation form, the Fibonacci sequence is denoted as such.

$$\mathrm{F}_n = \mathrm{F}_{n-1} + \mathrm{F}_{n-2}$$

In psuedocode, the Fibonacci sequence could be written like this, where n is the index in the sequence.

```
func fib(n)
    if n <= 1
        return n
    return fib(n-1) + fib(n-2)
```

Recursion works to simplify the amount of code. However, recursion is expensive in terms of computational resources. It is almost always a better idea to use loops instead of recursion. For safety, there is a recursion limit programmed into most languages. If a function calls itself too many times, the program will crash. This prevents infinite recursive loops, which would eventually take up all resources on a computer and cause a crash.

4.7. Arrays

There are many instances where just using primitves for variables does not suffice for a programmer's needs. Often a program will need to hold hundreds of numbers from a data set. Assigning a variable to each of these numbers is inefficient and limiting. Instead, a data set can be stored in one variable.

What arrays look like vary based on language implementation. Each language varies about the complexity of what can be stored in an array. In some languages, the array must be initialized with its size and the type of primitives it holds. Contrarily, some langauges will implicitly create arrays and assume their size. The simplist initialization sequence to make an array looks similar to this.

```
arr := [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

An advantage of arrays compared to standalone variables is that each element is in sequence. Therefore, they can be looped through. Arrays are zero indexed, meaning that the first element is at index zero of the array. Operations like finding the average of an array of integers are exceedingly simple.

```
i := 0
sum := 0
sz := sizeof arr
while i < sz
    sum := sum + arr[i]
print sum / sz
```

Knowing the size of the array is important, because a program could crash or exert undefinded behavior if it goes outside the bounds of an array.

Strings are a type of array that hold integers. However, when printed in their char format, they form ASCII characters. Strings are often initialized with double quotes ("). They are often terminated with a null byte (zero).

```
s := "hello"
print s
```

Which is in fact equivalent to declaring an array of characters into s (depending on the language with or without a null byte) and then printing that array variable.

```
s := [ 104, 101, 108, 108, 111, 000 ]
print s
```

An interesting property of arrays is that they can be initialized to hold arrays. This concept is similar to Matrices in theretical mathematics. Nesting can expand beyond one level, so three dimensional, four dimensional, and beyond are all possible. However, in most real instances two dimensional arrays suffice. One common use for two dimensional arrays is to simulate a game board. A tic tac toe could be represented by a two dimensional array.

```
tictactoe := [['X' 'X', 'O'], ['O', 'X', 'O'], ['X', 'O', 'X']]
```

This array can be formatted like a gameboard by using the newline ascii character.

```
i := 0
while i < sizeof tictactoe
    j := 0
    while j < sizeof tictactoe[i]
        print tictactoe[i][j]
        j := j + 1
    print '\n'
    i := i + 1
```

4.8. Pointers

Where are variables stored? How are arrays linked together? The memory used by a program can be thought of as a one dimensional array. Within that array are stored all the memory used by a computer. At certain points in that array the computer allocates memory to store the value of variables. A variable is in reality a name attributed to a certain address in memory where its value is stored.

Earlier in the chapter the differences in the sizes of primitives was discussed. In reference to memory as a whole, the memory for primitives is allocated at an address chosen by the computer and then that address is assigned to the variable. But this fact still does not explain how arrays work.

The way that arrays are allocated and looped through in memory is actually quite simple. When the array is initialized, the computer allocates memory in a row for all the elements in the array. When array elements are accessed, what the code in reality is doing is adding index provided to the initial elements location in memory and incrementing that location by the index multiplied by the amount of memory provided to the type of each array element. This means that

```
arr[i]
```

is actually this.

```
*(arr + i)
```

Also note that pointers are commonly referred to by the asterisk, which has many uses in programming languages besides multiplication. What the asterisk means is decided based on context.

Pointers access raw memory without any safety in between. Thus, accessing and manipulating pointers can be dangerous. Great care should be exercised whenever pointers are used. If available, a construct in a language that abstracts away memory manipulation should be used.

4.9. Random

Randomness is a frequently used concept in computer science. Randomness is used to encrypt passwords and for simulations. The use of random elements is frequently used in video games as well. The applications of randomness in computing has been the subject of hunders of academic papers A working understanding of the basic principles of random number generation is needed to understand how to make a host of applications in programming languages.

The most simple (and secure) way to obtain random numbers is from a truly random source. In most operating systems, there are sources of true randomness, which compute random streams of characters based on variables like startup time, disk speed, and temperature. Unfortunately, truly random numbers are extremely expensive on computing resources and is also limited. These truly random sources do have an important purpose though. They are used as the seeds for random number generators.

Pseudo random number generators (PRNG) are used to generate multiple random numbers. Because obtaining truly random numbers is so computationally expensive, it is more efficient to use an algorithm to generate unpredictable numbers. PRNG algorithms generate numbers that, without knowing the seed, are impossible to predict. With that noted, if the seed is known, the random number can also be predicted. This is why it is important to get the number from a cryptographically random source.

Once there exists an efficient way to get multiple random numbers, one facility that is still needed is a way to get a number in a certain range. For example, in a list randomization algorithm, one needs to be able to switch the index of one element of the list with another. There are two approaches to do this. In some languages, the random number is a very large. In order to get a number in range, the modulus operator is used. To get a number in the range of 1 (inclusive) and ten (exclusive), one would have to use a modulus of ten plus one.

```
min := 1
max := 10
r := random()
print max mod r + min
```

In other languages, the random number emitted is a floating number between 0 and 1. In these languages, getting a random number between one and ten would involve multiplication and addition rather than the modulus operator

```
min := 1
max := 10
r := random
print r * max + min
```

The randomization of an array is done using the Fischer-Yates algorithm. In this algorithm, the rndint() function is a variant of one of the two above methods to get a number with an inclusive minimum value and exclusive maxium value.

```
arr := [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
i := sizeof arr
while i > 0
    j := rndint(0, i + 1)
    tmp := arr[i]
    arr[i] := arr[j]
    arr[j] := tmp
    i := i - 1
```

One critical distinction between randomization algorithms is whether the algoirthm is or is not cryptographically secure. A cryptographically secure pseudo random number generator (CSPRNG) contains a sufficient amoutn of unpredictability to be used in secure applications like encryption. A critical element of CSPRNGs is that they must be seeded with a truly random number. A novice and naïve approach to seeding PRNGs is to use the current time. This approach is not cryptographically secure because, even if the PRNG has a very extensive algorithm, that algorithm can be predicted if the seed is known.

5. Python: The First Frontier

5.1. Introduction

5.2. Pythonic Code

5.3. F-Strings

5.4. Files

5.5. Modules

5.6. Critiques

5.7. Practice

6. Picking up: Moving to Statically Typed Languages

6.1. Compilation

6.2. Explicit Typing

6.3. Speed

6.4. Difficulty

7. Java: Object Oriented Programming

7.1. Main Method

7.2. Methods and Classes

7.3. Javadocs

7.4. Compilation

7.5. Practice

8. Shell Script: A Pragmatic Language

8.1. Languages

8.2. Programs

`cat`	Takes multiple file names as input and concatenates their content as output.
`cal`	Displays a calendar for a specific month, or year.
`cp`	Copies the content of one or more files to another file or directory.
`date`	Displays the current time, day, month, and year.
`dict`	Shows the definition of an inputted word.
`echo`	Copies an input string to output.
`editor`	Launches the user’s default editor.
`find`	Searches for a particular file name or characteristic and returns all files matching that description.
`grep`	Searches for text patterns within a file.
`ls`	Lists files in a directory.
`man`	Displays documentation for a specific command.
`mount/umount`	Adds or removes an internal or external drive from the file system.
`mv`	Moves the path of one or more files to another path.
`rm`	Deletes files ***permanently***.
`sudo`	Executes a command with root (superuser) permissions.
`sudoedit`	Edits a file with root permissions (the same as `sudo -e`).
`uname`	Displays information about the operating system.

8.3. Piping and selective execution

<

>

&

|

&&

||

[]

``

$

$()

$(())

8.4. Complex Elements

9. C: The Final Frontier

9.1. Basic Properties

9.2. Preprocessing

9.3. Memory

9.4. Practice

10. Brainfuck and Turing Machines

11. Resources

12. Exercise Keys

12.1. Python

12.2. Java

12.3. C

Table of Contents