Programming: An Ultimate Guide

Mitch Feigenbaum

16 April 2024

1. 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 for free on your own time.[1] What this book is meant to do is to get one on the write track and give one the tools to succeed.

On my GitHub, I have posted many of the projects which I have made for fun.[2] In the repository labeled 'Personal', 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.[3]

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, arithmetic, 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.

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.

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

---

[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 the journey to become a hacker, are in the resources section.

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

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.[4]

---

[4] 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.

2. Tools

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.

2.1. 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™, especially older 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.[5] [6] [7] If a computer can play Minecraft with a steady frame rate, it is more than capable to write and compile programs on.

2.2. Operating System

A UNIX-like operating system is ideal for programming. They have native access to the latest GNU tools for programming.[8] 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.[9]

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 files, directories 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

---

[5] A gigabyte is 10^9 bytes

[6] A byte is eight bits

[7] RAM | Random Access Memory

[8] GNU | GNU's Not UNIX

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

the specifics of the UNIX operating system or the shell, but the AT&T Documentary "The UNIX Operating System" is a great reference for starting out in UNIX.[10]

2.3. 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 IDE.[11] 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 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.

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 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.[12] 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.[13] Once Neovim is installed, you may learn how to use it with the following shell command and configuration.[14]

```
nvim -c Tutor
```

2.4. Interpeter

An interpreter is best thought of as a filter. It takes in the input of a program's source code and then, line by line, executes that code. Interpreters can correct code on the spot and give the exact placement of where an error occurred. As a general rule of thumb, high level languages like Python use interpreters, while lower level languages like C use compilers. The features present in higher level languages such as automatic type inferencing and conversion and automatic memory management simply are not possible in compiled form.

---

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

[11] IDE | Integrated Development Environment

[12] https://neovim.io

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

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

2.5. Compiler

Program compilation is the process by which the plaintext programming languages is converted into machine readable code. Compilers consist of multiple passes. During each pass the words written by the programmer are converted into a more machine readable form. While all compilers are different and may have a different number of passes, most follow a similar form. First compiler directives are read. These are embedded into source code and act as explicit instructions to the compiler. After this the code is split into individual directions, or tokens. The tokens are then formed into a syntactic tree, which explains the flow of the program. This tree is then expressed in terms of object files. These object files are then combined into a single tree, written in assembly code. From this point the assembly code is converted into their machine instruction counterparts and the program is ready to be executed.

There are benefits and disadvantages to using compilation rather than interpretation. Interpreted programs do not need to be compiled. Because compiled programs are far larger than their source code, programs written in interpreted languages are far more economical to store. Interpreted programs may also attempt to correct errors at run time, which is impossible with already compiled programs. Interpreted languages can be more easily edited and debugged because source code can be changed without the need for recompilation.

However, there are some notable drawbacks to interpreted languages. Because interpreted programs are read one line at a time, it is difficult to make optimizations like automatically concatenating strings or adding two integer constants together. As the program must be converted into machine code in real time, interpreted programs are more resource intensive. Conversely, compiled programs must only convert their source code to machine code once, at compile time. Reading a file one line at a time and converting those instructions into byte code has other disadvantages as well. At compile time, misleading indentation, out of bounds arrays, and otherwise valid source code that could lead to runtime errors can be warned about. This feature is notably not possible in interpreted languages.

There are three basic errors that a programmer may make when writing a program. Compiler errors are caught at compile time before a program is run. They usually have to do with syntax irregularities and undefined variables and functions. Because these errors are caught by the compiler, they are typically the easiest to find and fix. The second type of error is a runtime error. These involve code that is synactically correct but nevertheless computationally wrong. a for loop which attempts to access too high an index of an array is techinally valid from a syntax standpoint but is definitely not a safe or intended behavior. Even though such a theoretical program may compile successfuly, this program would likely crash when it was run. Some programming languages display helpful messages as to why the program crashed. Others will just return an error code specifiying that the program did not terminate successfully. Smart compilers emit warnings as well as errors. These warnings can be ignored by the programmer if they so please, but warn about common errors and undefined behaviors such as dividing by zero. The third type of error is a logic error. Logic errors are typically the hardest to spot and do not result in either a compiler error nor a program crash. However, logic errors do cause a program to produce unexpected or incorrect output. These types of errors can be the result of simple mistakes like not enclosing part of an equation in parentheses. The best way to detect logic errors is to include multiple debugging (print) statements throughout a program to determine the exact point where the error occurs and then studying that point. Often it is a good idea to have someone who did not work on that particular function of the program to help fix the error as they can help bring a new perspective and eliminate the tunnel vision a programmer may have.

Compiler optimizations are ways in which source code can be simplified to both reduce the number of operations and reduce the resources needed to run a program. The amount of compiler optimizations present in a compiled program is typically given by an argument to the compiler. The level of optimization required is a trade-off. With each level, the program will be come more efficient and less resource intensive. However, the more a program is optimized, its binary size will increase and it will be more prone to bugs and errors. Some optimizations should always be enabled. One place where optimizations are most commonly used is to do addition, subtraction, multiplication, and division on integer constants. For example, a line where a variable is declared as

```
x := 2 * 3
```

A simple optimization would be to do the multiplication operation at compile time and change the line to

```
x := 6
```

A more aggressive form of optimization is loop optimization. Loops tend to be resource intensive because they repeat the same code multiple times. However, with some clever changes, a loop can be simplified or removed entirely. Take, for instance, a piece of code that declares a variable, y. This variable is to be assigned every value in a for loop, 1—100, and then printed to the user.

```
y := 0
while y < 100
        y := y + 1
print y
```

This code is unnecessarily complicated, and has an extra 100 operations more than it needs. A loop optimization could totally remove the loop and reduce the code to two lines. Because y is guaranteed to have a final value of 100, y can be declared as equaling 100 and then printed.

```
y := 100
print y
```

2.6. Streams

There are three standard ways in which a programmer and a computer can communicate with each other. *stdout* is the way in which a program can give feedback to the programmer.[15] 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 world program could be written like the following.

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

---

[15] stdout | Standard Output

Print statements may also be used to access the second stream, *stderr*.[16] 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, tabs, backspace, and return can all be represented by escapes.[17] [18] In most programming languages, escapes are represented by the backslash character.

Programs can obtain data from the user by accessing the third stream, *stdin*.[19] The stdin stream contains the keystrokes 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 stdout. Thus is the function of the echo program found in both Windows™ and UNIX shells. A simple input to output program can be displayed in pseudocode like this.

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

---

[16] stderr | Standard Error

[17] A newline is equivalent to pressing the enter key.

[18] The return goes to the first character of the current line.

[19] stdin | Standard Input

3. Control Flow

3.1. 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.

¬   The *not* operator negates the statement. It checks if the compliment of a logical statement is true.[20]

∨   The *or* operator checks if one or both of two statements is true. Only if both statements are false will it return false.

∧   The *and* operator checks if both of two statements are true. If one or both statements are false, it will return false.

By combining these operators together, two additional operators emerge. These operators are less used but nevertheless applicable and useful.

⊕   The *xor* operator returns true only if both statements have a complimentary value.[21]

↔   The *xand* operator returns true only if both statements are the same value.[22] [23]

3.2. 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.

---

[20] Compliment is used to mean opposite in theoretical logic.

[21] $p \oplus q \in (p \wedge \neg q) \vee (\neg p \wedge q)$

[22] $p \leftrightarrow q \in (p \wedge q) \vee (\neg p \wedge \neg q)$

[23] The xand operator is also referred to as the xnor operator as it is the logical compliment of xor.

```
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 as C, 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 n
```

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"
```

3.3. 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 language.[24] Goto statements consist of a label and a looping point.

---

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

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

This preceding code segment 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
```

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
```

3.4. 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 simplify 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(a^2 + b^2)
```

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 statement 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.

$$F_n = F_{n-1} + F_{n-2}$$

The sequence starts with the numbers 0 and 1. In equation form, the Fibonacci sequence is denoted as such. In pseudocode, 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. Memory

4.1. 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* values 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`. *ASCII* is a common way that text is represented.[25] ASCII 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 language they were programming in would interpret the integer 65 to represent the letter 'A' and

---

[25] ASCII | American Standard Code for Information Interchange

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 maximum number is $2^{\text{bits}} - 1$ and the minimum number is $-2^{\text{bits}} + 1$ for signed integers. For unsigned integers, the maximum number is $2^{\text{bits+1}} - 1$ and the minimum 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 interpreter have such an issue with this seemingly 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 evaluated to the repeating decimal $0.\overline{7}$.

There are several ways to combat round-off errors.[26] 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.

---

[26] In computer science lingo, issues in the accuracy of floating numbers are called round-off errors.

4.2. Arrays

There are many instances where just using primitives 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 languages will implicitly create arrays and assume their size. The simplest 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 undefined 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 the double quote character. They are terminated with a null byte.[27]

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

Which is equivalent to declaring an array of characters into s 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 theoretical 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 game board by using the newline ASCII character.

---

[27] A null byte is represented by the ASCII value zero.

```
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
```

4.3. 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 equivalent to $\rightarrow$`(arr + i)`.

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.

5. Constructs

5.1. Lists

Lists are structures that are an alternative to arrays. Unlike arrays, they do not need to be initialized as containing a certain number of elements. Lists are made up of objects that contain the value of a variable and a pointer to the next element in the list. Lists are considered to have a dynamic size because they can have as many elements as the memory available.

Because lists are dynamic structures, they require certain functions to be available to manipulate them. As standard, most lists will have functions to add, remove, and access certain elements. These functions will do the complicated work of creating objects to append or prepend to the list, or to insert at a certain "index".

```
list.append(1)
list.append(2)
list.append(3)
```

$$1 \rightarrow 2 \rightarrow 3$$

5.2. 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 hundreds 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.

PRNG are used to generate multiple random numbers.[28] 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.

---

[28] PRNG | Pseudo Random Number Generator

```
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 maximum 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 algorithm is or is not cryptographically secure. A *CSPRNG* contains a sufficient amount of unpredictability to be used in secure applications like encryption.[29] 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.

5.3. 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.

---

[29] CSPRNG | Cryptographically Secure Pseudo Random Number Generator

```
/*
 * 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 maintenance. 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 / m^2
```

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 unneeded, superfluous, or distracting.

6. Python: The First Frontier

Python is a language that is unique among established programming languages. Its quirks include using whitespace instead of brackets, extensive formatted strings, and being a solely interpreted language. It is the fastest growing programming language and used in a number of commercial and free software applications. Python is also notable for its simplicity. Programs can be made in Python with virtually no boilerplate.[30]

Because of Python's use of whitespace and lack of boilerplate, it is often used as a beginner language. Delving into Python requires virtually no knowledge of the base components of a computer and memory such as pointers. Python is also a safe language. If one were to write a faulty program in C, it could cause a segmentation fault or worse permanently corrupt the computer's memory. Python programs are interpreted and have extensive safety measures programmed in, so the risk to the programmers software and hardware are virtually zero.

One of the most unique elements of Python is its use of whitespace. This difference can be difficult to adapt to for experienced programmers. However, for beginners, this element of the language can be helpful. Parentheses still exist in the language, but their use is strictly to impact the order of operations. Brackets are nonexistent. Lines are terminated by the newline character rather than by semicolons.

```
statement 1
statement 2
```

Requiring whitespace is a good way to make sure that programmers use the best programming practices. In Python, proper indentation is not best practice: its required. Misleading indentation levels is one of the leading causes for a program to not work, as it will cause a programmer to either use too many or too little brackets. In Python, because brackets do not exist, it is easy to find and correct this error.

In Python, statements like if statements are formatted without brackets or parentheses, and contain a colon before the indentation level increases.

```
if x:
    pass
```

In most other programming languages, an if statement looks like this.

```
if (x) {
}
```

One may see that the first code example looks much simpler and allows the reader to clearly see the indentation level. It also looks far less cluttered and verbose.

Comments in Python come in two varieties. Single line comments are delimited by a newline character thus they may only take up one line. They begin with a hash character.

```
# This is a single line comment
```

Multiline comments technically do not exist in Python. However, there is a trick to make them work. If a multiline string is not assigned to a variable, it will be ignored by the interpreter.

---

[30] Boilerplate is code that must be present in a program but does not impact uniquely program execution.

This property can be used to create multiline comments. Multiline comments are delimited by three double quote characters.

```
"""
This is a
multiline comment.
"""
```

Python provides several operators to perform basic mathematical operations.

Symbol	Operation
+	Addition and string concatenation
-	Subtraction
*	Multiplication and string repetition
**	Exponentiation (eg ten squared is `10**2`)
/	Division (float)
//	Floor division (integer)
%	Modulus (ie remainder of a quotient)

Types are not explicitly declared in Python. Based on the value provided to a variable, a type will be inferred. There are six main types that will be discussed in this book: `bool`, `int`, `float`, `str`, `list`, and `dict`.

Variable assignment is practically the same as in pseudocode. A variable is simply set to equal a value. There is no variable declaration.

```
w = True # declares a Boolean
x = 2 # declares an integer
y = 0.5 # declares a float
z = "A sentence." # declares a string
```

It should be noted that a string may be between single or double quotes. However, it is best practice to maintain one style of string declaration throughout a program. Variables may also be reassigned to be different types. A variable initialized as an integer could be reinitialized to be a string and vice versa.

```
var = 9 # initialized as an integer
var = "nine" # reinitialized as a string
```

Boolean values are represented in Python with the constants `True` and `False`. All logical statements in Python evaluate to either True or False. True implicitly may be converted to one and False implicitly converted to zero. Thus the statement

```
True + True
```

yields the integer two.

Lists can be declared in Python using the left square bracket and right square bracket characters. They can contain any combination of types, which is another unique property of Python. A single list can contain integers, floats, strings, dicts, and other lists all at once. The

elements of a list are separated by commas.

```
arr = [1, 5, 3.0, "a string", 8.3, [1, 2, 3], {"a": 1, "b", 2}]
```

Dictionaries are similar structures to arrays. They contain keys and values. They may be initialized like this.

```
d = {"one": 1, "two": 2, "three", 3}
```

Walrus notation is a new element recently added to the Python language standard. It allows for variable assignments to appear within a statement. Walrus notation uses the `:=` symbol (a colon and an equal sign) to assign values to variables. All variable assignments must be within parentheses or they will cause an error.

```
2 + (x := 3) # Evaluates to five
```

Python has included in the base language methods to produce input and output. Output is produced with the print statement. At its base level, a print statement will output a string with a newline appended to the end. Thus, a classic hello world program could be produced like this.

```
print("Hello, world!")
```

Print statements can also output integers, floats, dictionaries, and arrays.

Standard input in Python is gathered from the `input` function. This function will return a string containing every character the user inputted up until they press the return key. A simple program that converts input to output could be written thusly.

```
print(input())
```

The input function takes an optional prompt parameter, which is displayed on the same line as the input. This prompt can be used to tell the user what they should enter as input. A prompt could ask the user for their name and then store it in a variable.

```
name = input("What is your name?: ")
```

Casts convert values of one type into values of another type. In order to be eligible for conversion, a value must reasonably be able to represent another type. A string containing only a whole number (eg "123456") would be eligible for conversion to an integer. However, a string containing letters and numbers (eg "ABC123DEF") would not be eligible for conversion into an integer or float. One place where casts are often used in Python is when using the input function. The input function returns a string, but oftentimes a program must ask a user for an integer or float. In these instances a cast may be used. Take for example a program which takes two numbers from a user and stores the product in an integer variable.

```
x = input("Enter a number: ")
y = input("Enter another number: ")
z = int(x) * int(y)
```

If statements in Python are represented by the `if`, `elif`, and `else` keywords. The conditional portion of the statement is terminated with a colon. A simple if statement in Python takes this form.

```
if x: #do y
```

Any code on the same line of the if statement or indented below the if statement will be executed if the conditional evaluates to true. The else statement similarly is terminated by a colon and will execute code on the same line or below if the conditional evaluates to false.

```
if x: #do y
else: #do z
```

Because Python takes whitespace into account, the else if construct found in other programming languages is impossible to implement in Python. Instead, nested conditionals are represented with the `elif` keyword. If all if and elif statements in a conditional statement evaluate to false, the code in the else statement is executed, if the else statement is part of that construct. The elif statement can be used if code should be executed based on whether a conditional has various characteristics. When calculating a letter grade from a percentage, the code must display different outcomes for all grades A-F. A conditional using if, elif, and else could represent this.

```
if grade >= 90:
    print("A")
elif grade >= 80:
    print("B")
elif grade >= 70:
    print("C")
elif grade >= 60:
    print("D")
else:
    print("F")
```

The `pass` statement is used as a placeholder for code. If there is no code to be executed in a conditional statement, Python's interpreter will not be able to recognize when the statement ends. Thus is the reason for the pass statement.

```
if x:
print("This causes an error.")
```

The above code will cause an error because there is no code on the same line or indented below the conditional. When writing code, there are scenarios in which a conditional or other construct may be created and left empty for later use. The pass statement allows a conditional to not throw an error and not execute any actual code.

```
if x:
    pass
print("This does not cause an error!")
```

The Python language contains a construct for variable assignment using conditionals on one line. Ternary structures in the Python language use the if and else keywords. Based on whether a conditional is true or false one of two statements will be assigned. Ternaries in Python take this form.

```
var = x if y else z
```

This form is different from most other ternary statements in programming languages which usually put the condition first and follow it with the assignment values. Ternaries may also be used to assign values to parameters in functions. In the print function, a ternary could be used to print one of two strings.

```
print("one" if x == 1 else "two")
```

While loops in Python take a similar form to if statements.

```
while x: pass
```

As long as the conditional is true, the code one the same line or indented in the while loop will be repeated.

For loops are enhanced forms of while loops. A Python for loops iterates through the content of a list. The next element of the list is assigned to the temporary looping variable at each iteration of the list until the list is done. A for loop could be used to print every element of a list on a separate line.

```
for x in arr: print(x)
```

The equivalent code in while loop form would be thus.

```
iter = 0
while iter < len(arr):
    x = arr[iter]
    print(x)
    iter += 1
```

For loops are also used to repeat code for a certain amount of iterations. This function is fulfilled with a function called `range`. The range function returns an array of numbers in sequence. Range takes three parameters, of which two are optional. If only one parameter is supplied, the range function will generate a list with the numbers from zero to that parameter (exclusive). This parameter is known as the 'stop'. If two parameters are supplied, the first indicates the minimum numerical value in the list called the 'start', and the second parameter is the stop. Finally, if three parameters are supplied, the first is the start, the second is the stop, and the third indicates the number by which each value from the start to the stop is incremented by, called the 'step'. The numbers 1—10 (exclusive) only containing every other number in that range could be contained in this range deceleration.

```
range(1, 10, 2)
```

Because steps can be negative, it is possible to make a decrementing list. This range declaration displays the numbers 10—1

```
range(10, 1, -1)
```

The `break` and `continue` keywords can be used to manipulate looping cycles. The break keyword will exit a loop regardless of whether the conditional is valid. The continue keyword will immediately enter the next iteration of a loop without executing any of the code below it in the loop.

The else keyword has a second meaning when applied to loops. When the else keyword is used immediately after a for or while loop, it indicates that a loop has finished successfully without the use of the break keyword. The else keyword makes logical sense in this position if a loop is simply thought of as a conditional that executes its code and then repeats itself until its conditional is false. In that case the else statement has the same meaning as in the if statement.

There exists a version of a for loop that is similar in structure to the ternary conditional statement. It fills a list with the results of one line of code executed in a for loop. The enhanced for loop is somewhat complicated especially for new programmers. It however is extremely powerful. This example of an enhanced for loop fills a list with string versions of the numbers in a range.

```
x = [str(x) for x in range(10)]
```

Functions in Python take the following form.

```
def func(p1, p2="val", *args, **kwargs):
    # code to be executed
    return value
```

A function must first be declared with the def keyword. After the def keyword is the functions name. succeeding the function name are the parameters, which are enclosed between parentheses. There are four types of parameters that may be present in function definitions. The first type, represented by p1 in the above function, must be supplied to the function and may contain variables of any type. Variables can be made optional and have a default value, such as p2 in the sample function. Args, denoted by a single preceding asterisk, are lists and can take any number of arguments, including zero. When supplying a function arguments where an args is a parameter, it can be like this.

```
f(1, 2, 3, 4, 5, 6)
```

Kwargs, denoted by two preceding asterisks, are dictionaries and may include a variable amount of arguments. The arguments are supplied with an equal sign, with the left operand being the key and the right operand being the value.

```
f(a=1, b=2, c=3)
```

After the closing parenthesis, a colon is used to denote where the function body begins. Indented is the code that is to be executed when the function is called. Following the executed code a function may optionally return a value. Values are returned using the return statement.

The Python language is specifically purposed to allow for programs to be made with virtually no boilerplate. However, it is best practice to put everything in a function. Keeping all code in functions allows code to be called multiple times and also allows code to be imported into other files without executing automatically. Previously, the hello world program for Python was established as

```
print("Hello, world!")
```

In the real world, it would be better practice to write this as

```
def main():
    print("Hello world!")

if __name__ == "__main__":
    main()
```

Now if the code from this file is imported into another file, that file could use the functions from this file without executing any code.

6.1. F-Strings

A unique structure that demonstrates the simplicity and power of Python is the "F String" (short for "format string". F-Strings allow a programmer to store types other than strings inside of strings. These strings also allow for formatting instructions, like only displaying the first two decimal points of a floating number. What adds to an F-Strings power is the fact that it can take code and format it. This allows for complexities like mathematical operations, spacial formatting, and mathematical operations all inside of a string.

Python F-Strings follow a basic form. They are formatted as regular strings with a preceding lowercase *f*.

```
print(f"This is an f string")
```

Variables or code may be inserted into the f string between left curly bracket and right curly bracket characters.

```
x := 3
print(f"The value of x is {x}")
```

Using Walrus case, a variable may be assigned within an F-String. This variable may be called at a later point in the string. Walrus case allows for the preceding code to be written on one line.

```
print(f"The value of x is {(x := 3)}")
```

F-Strings allow for the formatting of floats and integers. Formatting instructions are placed after the variable or code and are placed after a colon (:). Below is a quick reference for using F-Strings.

:   A *colon* denotes following format instructions. With the exception of the equal sign All the below symbols must appear after the colon.

<   The *left angle bracket* denotes that the variable should be output left-aligned. The amount of spaces by which the should be left aligned is determined by a following number.

```
f"{x:<20}" # left aligns variable x by twenty spaces
```

>   The *right angle bracket* denotes a variable should be right aligned. As with the left bracket, the amount of spaces is determined by a following number.

+   The *plus* sign forces an outputted number to display a preceding plus sign if that number is positive.

%   The *percent* sign displays a number as a percentage. The number is displayed multiplied by 100 and with a succeeding percent sign.

,   The *comma* operator formats a number with a comma every three digits. As an example, if one wished to place comma separate's at every three digits in an integer, the formatting instruction would be such.

```
print(f"One million comma delimited is {1000000:,}")
```

The resulting output is this.

```
One million comma delimited is 1,000,000
```

\#   The *hash* operator denotes the base that a number should be displayed in. This operator supports the following bases, which are placed succeeding the # mark and represented by an individual lowercase letter.

Letter	Name	Base
b	binary	2
o	octal	8
d, n	decimal	10
x	hexadecimal	16

When a base other than decimal is used the formatter places a zero and the base letter preceding the number.[31] If one were to attempt to print the number 16 and format it as a hexadecimal number with the hash operator, it would be displayed as `0x10`.

=   The *equal* operator is used to display a variable along with its name. It has applications in program debugging. In a previous example, the number one million was placed into an F-String and formatted. In practice, any number within code should be assigned to a variable so that it is documented in code.

```
mill = 1000000
```

The value and variable mill may be printed with the equal sign within an F-String

```
print(f"{mill=}")
```

This outputs `mill=1000000`. Whitespace is preserved around the equal sign. This means that if the f string is changed to have preceding and succeeding whitespace,

```
f"{mill = }"
```

It may now be outputted as this.

```
mill = 1000000
```

Because the equal sign is not after the colon, it allows the value of the variable to be manipulated by any format specifier. The output of

```
print(f"{mill = :,}")
```

would be

```
mill = 1,000,000
```

---

[31] Decimal is base 10.

type-letters
Type letters can be used to denote whether a variable is an integer, exponent, float, or string. Usage of an operator can prevent errors such as displaying an integer number as a float. a number preceding a type letter specifies how many digits a number should be aligned with. The type letters are d (integer), e (scientific notation), f (float), and s (string). For example, if you wanted to print out a float to only two decimal points, you could do that in the following way.

```
# declare and initialize a float called 'x'
x = 2.23014232
# print that float using an f string
print(f"{x:.2f}") # outputs: 2.23
```

F-Strings are not only capable of displaying variables. They may also be used to execute code. Mathematical operations, user input, variable assignment, and string multiplication may all occur with F-Strings. The below code takes two variables (x and y) and adds them together inside of an F-String and displays the sum.

```
x = 1
y = 2
print(f"the sum of x and y is {x+y}")
```

String multiplication is also a string operation that may be performed in an F String. The asterisk character may be used not only to multiply integers and floats but also can multiply strings. The multiplication operation,

```
"xy" * 3
```

would produce the string

```
"xyxyxy"
```

This operation may also be done in an F-String. However, there is one caveat. If the double quote operator is used within curly brackets, it may signal to the interpreter that the string is terminated and will thus crash the program. Therefore, when a string is within an F-String, the single quote character should be used. The same string multiplication operation from above could be displayed in an F-String using this technique.

```
print(f"{'xy' * 3}")
```

Multiline strings are denoted by triple single or double quotes. They are unique in that they preserve whitespace such as tabs, spaces, and newlines. They may also be made into f strings with a preceding lowercase f.

6.2. Files

When making a program there are scenarios under which data must be stored. Thus, a programming language needs facilities to read, write, and create files. Python is able to do all these file operations. The file reading operation is simple and intuitive. It uses various functions to open and close files, as well as perform operations on files as a whole.

The open function is used to open a file stream for manipulation. It takes two arguments. The first argument is a relative or absolute path to the file. The second argument is a string containing a list of file permissions. Within this string may be the letters from the following chart.

Letter	Permission
a	append
b	binary
r	read
t	text
w	write
x	create

As an example, if one wished to open a file named `file.txt` for reading, and assign the file to a variable named f.

```
f = open("file.txt", "r")
```

For safety, only the permissions needed should be specified in the string. If a file meant for reading only were to be given a write permission, it could permanently erase the file or jumble its data. All file operations are permanent, so extreme caution should be exercised.

It is extremely expensive to make changes to a file on the disk. Therefore, it is a more efficient to only write the changes made to a file after all manipulation is done. Therefore, buffers are used when performing file operations. A file stream is first open with the `open()` function. Then, the file is read from the disk into RAM The content of the file in memory is called a *buffer*. A file buffer can be created in Python using the `.read()` function. This function returns the raw characters within the file in string format. In practice, this output is assigned to another variable. This variable is then used to make any file manipulation that the programmer sees fit.

Lines do not exist in a file. Rather, they are used in editors to represent the newline character. While this varies based on operating system, Python can automatically write with to a file with the appropriate newline characters. The `.splitlines()` function converts a string into an array of strings with newline characters used as delimiters. Thus, if one wanted to read a file end and split the files content into a newline delimited array, they could use the `.read()` and `.splitlines()` functions in conjunction.

```
f.read().splitlines()
```

Because this operation is so often used, Python has a special function to replicate this behavior called `.readlines()`. The `.write()` and `.writelines()` functions work like the `.read()` and `.readlines()` functions but in reverse. The write functions take either a string or a list of strings respectively. They change the current content of the file to that specified by the string or list of strings.

Whenever a file is open, it uses computing resources from the disk. This, as previously mentioned, is resource intensive. Thus, it is important to close the file stream whenever it is not being operated on. The `.close()` function does exactly this. Once a file has been used completely, it may be closed like this.

```
f.close()
```

The need to remember whether a file is open or closed can be mitigated with a construct. Using the `with` keyword, a file may be opened and closed automatically. When the with block is completed, the file and variable it was assigned to are deallocated. Any file operations such as creating a buffer are done within the with statement. This construction using the with statement opens a file and reads it into a buffer.

```
with open("file.txt", "r") as f:
    buf = f.read()
```

This construction using the with statement writes a buffer to a file.

```
with open("file.txt", "w") as f:
    f.write(buf)
```

6.3. Libraries

In many scenarios, the task a programmer is trying to perform has already been done in an optimized form. Using existing code for functions can speed up the development process and make bugs less prevalent. Python modules allow a programmer to "import" existing code implementations into their own programs.

There exists certain keywords that allow a programmer to use existing libraries in their code. The import keyword specifies exactly what libraries will be made available. To say, import the random module, this line would do so.

```
import random
```

Once a module is imported its functions or classes can be called with the dot operator. To recall the randrange function, which generates a random number within two inclusive numbers, the name of the module random would need to precede the function name randrange. This code generates a random number between one and ten inclusive.

```
random.randrange(1, 10)
```

In some cases a programmer may need only one function from a module. Instead of importing the whole library, the from keyword allows a programmer to import a specific function or functions into the program. To only import the randrange function, one could write this.

```
from random import randrange
```

Then the module with a dot operator may be omitted.

```
randrange(1, 10)
```

Using the asterisk character, all functions, classes, and variables in a module are made accessible without using the module namespace.

```
from random import *
```

This is strongly discouraged, because it can cause naming conflicts and typically adds many variables, functions, and classes that will never be used. The as keyword changes the name of the namespace of the module. The as keyword has already been previously used to assign itself to a file. One usage of the as keyword would be to assign a shorter name to an often used

module. Because rnd is a common way programmers abbreviate the word random, a Python programmer may wish to make the random module callable by this shorter name. This request is possible with the as keyword.

```
import random as rnd
```

The math module is used to perform mathematical operations that are not available with the native math operators.[32] The math operator contains the constant variables e, inf, nan, pi, and tau. It also contains functions to perform trigonometry and algebra such as sin(), sqrt(), and gcd(). In some instances these functions add to functionality already native in Python. Using only native Python operators, the square root of a number can be found by using the exponentiation operator of the number and 0.5

```
n ** 0.5
```

The sqrt function provides this functionality as a function rather than as a clever workaround.

The random module contains functions for performing random operations. Initially it is worth noting that the random module is not *cryptographically* random. The random module is great for use in simulations that require many random numbers but do not require them to be cryptographically random. It should not, however, be used in places where security is needed such as password hashing. For applications requiring cryptographically random numbers, the secrets module should be used. The secrets module contains many of the same functions of the random module but in a cryptographically secure form. Below are listed some of the most commonly used functions in the random module.

choice(arr)

> The choice function returns a random element from a list.

shuffle(arr)

> The shuffle function takes a list as input and randomizes the order of the content in the list.

randbytes(n)

> The randbytes produces n amount of randomly generated bytes.

random()

> The random function returns a float between 0.0 (inclusive) and 1.0 (exclusive).

randrange(start, stop=None, step=1)

> The randrange function returns a random number between start (inclusive), stop (exclusive), and divisible by the step. If stop is not provided, its value is substituted by step.

6.4. Practice

For some hands-on experience with Python, complete the following challenges.

1   The first challenge involves using f-strings. The program will only be one line of code, that prints an f-string. Within that f-string two numbers will be taken as input and added together. The sum of those two numbers will be displayed as output.

2   The next challenge involves importing modules. Ten random numbers between the numbers 1 (inclusive) and 10 (exclusive) will be generated and stored in an list. Then that

---

[32] Python native math operators: +,-,/,*,**,%

list will be printed.

3   The final challenge will involve loops, conditionals, and importation. A random number will be generate in the range of 1 (inclusive) and 10,000 (exclusive). Then the user will have 20 tries to guess the number. If the number the user inputted is the random number generated, a congratulatory message is displayed and the program exits. If the user's number is greater than the generated number, a message is displayed telling the user that their number is to high. Otherwise a message is displayed telling the user that the number they generated was too low. If the user expends all twenty guesses, a message is displayed telling the user that they failed to guess the number in the allotted attempts. The generated number that the user failed to guess will also be displayed in that message.

7. Picking up: Moving to Low Level Languages

Thus far only the Python language has been discussed. The Python language is quite similar to pseudocode in its form. Types, memory, and program execution are all abstracted away. This fact, while making Python a great beginner language, also makes it extremely inefficient. Every abstraction that Python makes requires that more computing resources be used.

In any programming language, a compromise is struck between being computer friendly and programmer friendly. At their core, every coding language eventually becomes machine code, binary electrical pulses in silicon. Thus the simplest programming language would be one where a programmer were to write binary ones and zeros in a sequence. However, this is practically impossible. There just isn't enough time for humans to write machine code. Therefore, machine code is abstracted. With each abstraction more optimizations are lost. Generalizations must be made about operations. As the amount of memory and storage available to computers has increased exponentially, programming languages have become more and more abstracted. Languages that are "faster" often require the programmer to learn more about the fundamental components of a computer such as memory management and logic gates.

There exists a danger to using lower level languages. In higher level languages, because basic computing constructs are abstracted, there is less risk that a program may enact damage to a computer or result in data loss. In lower level languages these tasks are left to the programmer rather than the compiler or interpreter. Many programmers begin using heavily abstracted high level languages such as JavaScript and Python. By the time that these programmers move on to learning and using lower level languages, they already have the foundations to deal with low level obfuscations.

7.1. Explicit Typing

There are two approaches to combing the type of a variable. In weak typing, when variables are used in operations such as addition, variables implicitly change type as to avoid errors. While this results in less runtime errors, this can cause undefined behavior not specified by the programmer. In strongly typed languages, variables do not implicitly change type and type changes must be specified by the programmer. In most languages a combination of implicit and explicit type changes are used. When doing operations together on integers and floats, implicit typing may be used to allow the answer to be of either type integer or type float. There are also *casts*, which allow the programmer to specify that a variable be interpreted as another type. Casts must be to a valid type. If one tries to cast a large integer type to a small integer type, this may result in an error as the small integer type does not have enough space to store the larger integer.

Programming languages have two approaches to declaring the type of a variable. In dynamic typing, a variable is declared without its type. Some languages must still declare the variable using a keyword such as `let` or `var`. Other languages, however, use *duck* typing. Duck typing allows a programmer to declare a variable in a simple assignment. This method is the simplest way to declare a variable for a programmer, but very taxing on the machine. Duck typed languages are impossible to compile because the variable and its declaration must be created in real time. The method of typing used in most compiled languages is explicit typing. When a variable is declared, its type is also declared. This type cannot change for the duration of the variable's existence. When a variable is declared, additional information, such as whether its

value may be changed, if it is accessible from anywhere in the program, and whether it is local or global all must be specified by the programmer.[33] [34] Again, this is a trade-off of programmer effort and computing resources. An upside to explicit typing is that errors from type inferencing or declaration may be spotted at compile time. Some languages, like Microsoft's TypeScript™, exist solely for this reason. TypeScript was created to add explicit typing to JavaScript, the programming language used for internet sites. By using TypeScript, which compiles to JavaScript, companies have been able to fix many errors in their web applications that result from weak and implicit typing.

The main characteristic that separates explicit and implicit typing is speed. When types are declared explicitly, the compiler must perform less operations to allocate memory and characteristics of a variable. Because implicit typing requires that the compiler assign types to variables in real time, implicit typing is impossible to implement in compiled languages. Explicit typing puts the burden of allocating memory to variables on the programmer rather than on the compiler and by proxy the computer. In almost all cases this trade-off is worth it, especially if a program is used thousands or millions of times as would be the case in a production environment.

7.2. Difficulty

In general, compiled languages tend to be more verbose than interpreted ones. This is due to the fact that many tasks that are typically designated to the interpreter, like casting and memory management, are instead placed on the programmer. Languages that use OOP like Java and C# have added verbosity because they use many namespaces and classes.[35] OOP associates operations that might be done n other languages using operators and functions into classes and objects. This method of programming is somewhat more organized for large scale projects but in smaller scale applications is hideously overcomplicated. OOP languages also favor verbosity over short names as it makes the code self documenting.

A difficult element for new programmers being introduced to more traditional programming languages is that whitespace is optional. Statements end based on the presence of a line ending character such as a semicolon rather than a newline. Instead of indentation brackets are used to denote conditional execution. This allows for interesting concepts to become realized such as else for loops, as demonstrated below in C code.

```c
if (x == 1) {
    puts("X is 1");
} else for (int i = 0; i < x; i++) {
    printf("i is %d\n", i);
}
```

Some of the most common syntax errors are forgetting an ending bracket or a semicolon at the end of a statement, issues which are much less prevalent in languages that use whitespace as delimiters.

An argument against the verbosity and apparent complexity of compiled and low level languages is that it takes more time for the programmer to write a program than for the program

---

[33] Local variables only last for the duration of a function.

[34] Global variables lasts for the duration of a program.

[35] OOP | Object Oriented Programming

to execute. This argument is valid only in certain cases. If, as an example, one wants to find the answer to a math problem, a short program written in an interpreted language could work. However, if a program is meant for business applications and will be executed millions of times and needs to be able to run a variety of computers, it should probably be made in a faster language even if it would be more work and time for the programmer. It is up to the programmer to make this trade-off and decide if making a program in a lower level language is worth it for their needs.

7.3. Brackets

The first characteristic that is new to programmers starting out in lower level languages from a language like Python is their use of brackets. Brackets are used to enclose multiple commands to be executed. If only one statement is executed, brackets may be omitted in most structures. However, if multiple commands are executed based on a conditional brackets must be used. There are several different styles of bracketing, each with their own advantages and disadvantages.

Allman

```
if (x)
{
    statement 1;
    statement 2;
}
else
{
    statement 3;
}
```

K&R

```
if (x) {
    statement 1;
    statement 2;
} else {
    statement 3;
}
```

Lisp

```
if (x)
    {statement 1;
    statement 2;}
else
    {statement 3;}
```

In this book, the K&R style of bracketing will be used, as it is the de-facto standard in most programming languages.

8. Java: Object Oriented Programming

In Java, certain words are not available for variable assignment.[36] These are known as reserved keywords. They represent an element in Java processed by the compiler as an instruction.

In Java, variables must be declared before initialization. Additionally, they must be declared with their type specified. There are eight primitives available to the programmer in Java. The primary differences between these data types is the amount of bytes of memory they can hold. They range from the `boolean` type, which may only hold one bit, to the `long` type, which can hold eight bytes.

Java contains functions for accessing all three streams. The functions for accessing stdin are found in the Scanner module, which will be discussed later. Stdout and stderr may be accessed with the `System.out.print` and `System.err.print` functions respectively. Variables of different types can be concatenated into a string with the plus character.

```
String cat = "" + 1 + 'c' + "another string"
System.out.print(cat) // outputs "1canother string"
```

If the suffix `ln` is added to the end of either of the two above functions, a newline character will be appended.

The if and else operators operate require the logical statement to be enclosed in parentheses. If more than one statement is to be executed, those statements must be enclosed in curly brackets. The elif statement can be created by appending an if to an else statement.

```
if (con1)
    statement1;
else if (con2) {
    statement2;
    statement3;
} else
    statement4;
```

Ternaries are used to assign values based on the value of a statement. First comes the logical statement, followed by a question mark and the first value. Finally, a colon and the second value. Ternaries could be useful in a scenario in which a string variable should be assigned the value "even" or "odd" based on the value of a integer.

```
String pos = n % 2 == 0 ? "even" : "odd";
```

A switch-case statement is used to execute code based on the prospective values of a variable. It can simplify repetitive if statements that only evaluate if a variable has a certain value. A switch statement is contained within brackets and is denoted by the `switch` keyword. In parentheses is the variable which is to be tested. Within the case statement, multiple potential values for the variable are to be tested with the format `case <value>: code;`. If the variable fulfills no value, the code within the `default` statement is executed. The break

---

[36] Java Keywords: abstract assert boolean break byte case catch char class const continue default double do else enum extends false final finally float for goto if implements import instanceof int interface long native new null package private protected public return short static strictfp super switch synchronized this throw throws transient true try void volatile while

keyword is used to exit the switch statement. If the break keyword is not used, what is known as fall-through occurs. The default statement is always executed unless a break or return statement is executed beforehand. One usage of a switch statement could be to print the ordinal notation of a number 1—5.

```
switch(n) {
    case 1:
        System.out.println("first");
        break;
    case 2:
        System.out.println("second");
        break;
    case 3:
        System.out.println("third");
        break;
    case 4:
        System.out.println("fourth");
        break;
    case 5:
        System.out.println("fifth");
        break;
    default
        System.out.println("The number provided is not 1-5");
}
```

The while loop in Java should not be surprising compared to its Python or pseudocode counterpart. It takes the form of a single if statement that repeats itself.

```
while (condition) {
    statement1;
    statement2;
}
```

The for loop, however, is quite different. A for loop takes three parameters. The first parameter assigns a value to a variable.

```
int i = 0
```

The second parameter is a condition under which, if true, the loop will continue.

```
i < 100
```

The third parameter is a piece of code to be executed whenever the loop is executed.

```
i++
```

The first statement below the loop declaration (or everything in curly brackets following the loop declaration) is to be executed at each loop cycle.

```
System.out.println(i);
```

Put together, the above for loop prints every number 0—99.

```
for (int i = 0; i < 100; i++) {
    System.out.println(i);
}
```

This is equivalent to the code that would be executed if a more verbose while loop was used.

```
int i = 0;
while (i < 100) {
    System.out.println(i);
    i++;
}
```

In order for code to be executed, a main statement must be present in a program. In Java there is a condsiderable amount of boilerplate.

```
public class Hi {
    public static void main(String[] args) {
        System.out.println("Hello, world!");
    }
}
```

This small example can be used to describe the language structure elements of Java. Firstoff, everything must be inside of a class. A class is an object that contains variables and functions. Within that class, all code that is within a main statement will automatically be executed when the file is run. The `String[] args` allows a user to supply arguments to the Java program via the command line.

8.1. Methods and Classes

Java is an object oriented language. This means that everything in Java is within an object. Programmers can create objects. All objects have certain properties defined by the programmer, such as enclosed methods, variables, and protections. Classes are defined by the `class` keyword, and may contain global variables that are declared before any methods are defined.

```
class Temperature {
    char system;
    int degree;
}
```

Java has four levels of access modifiers. These modifiers decide who has access to the values of variables, methods, and classes.

public
> The *public* modifier causes the object to be available to anyone regardless of what permissions they have.

private
> The *private* modifier specifies that only methods and variables within the same class may access the object.

protected
> The *protected* modifier allows an object to be available within the package, class, and all

subclasses.

default

The *default* modifier allows the object to be accessed anywhere within the same package. The default modifier does not need to be specified.

It is best practice to make every object that is not explicitly needed outside the class to be private. This ensures that the class cannot have unintended user errors. The class should be public if it is intended to be used within another class.

```
public class Temperature {
    protected char system;
    protected int degree;
}
```

A constructor specifies code to be executed when the class is initialized. In practice a constructor is usually used to define the private global variables. Using this method ensures that any unintended input during the class initialization can be dealt with. The `this` keyword is used within a method to access a global variable that has been overridden with a local variable of the same name.

```
this.system = system;
```

Error checking should also be present to warn the programmer if a value is incorrect. Constructors take the same name as the class.

```
public Temperature(char system, int degree) {
    if (system != 'c' || system != 'f')
        this.system = 'c';
    else
        this.system = system;
    this.degree = degree;
}
```

The `toString()` method is the representation of a class that is used when it is output. It by default is a hash code where the class is stored in memory. The user can define their own `toString()` method to display what they wish to display when the class is represented.

```
public String toString() {
    return "System " + system + " Degree " + degree;
}
```

FILL THIS IN ABOUT JAVA EQUALS METHOD

In Java, methods are declared with their return type in a similar fashion to how variables are declared. Methods, unlike variables, can have a *void* type. This means that the methods will not return any value. The return statement can be used to exit the method, but it will not yield any value. A method is declared with access modifiers and return type. A common use for methods is to show the value of a variable without giving access to it to change it. These methods customarily have the prefix *get* to return the value of the variable and *set* to modify the value of a variable.

```
        public int getDegree() {
            return degree;
        }

        public void setDegree(int degree) {
            this.degree = degree;
        }
```

The `static` keyword in Java is used to denote a method or variable that is to remain the same throughout the duration of the program across all instances of a class. Static methods can only access static variables. The main method must be static because it lasts for the entirety of a program. A static variable could be used to count how many of a particular class have been initialized throughout a program.

```
        protected static int temps = 0;
        public Temperature() {
            temps++;
        }
```

Sometimes there are methods that need to perform operations on multiple types of variables. One might need a method to add together two integers and another to add two floats. Furthermore, one might need to add two integers together or concatenate three strings. Java has a convenient way of doing this through *method overloading*. Multiple methods may be created with the same name which take a different number or type of parameters. The method that is executed is determined by the compiler based on the content of the arguments rather than the method's name. The overloaded methods may also have a different return type.

```
        int add(int n1, int n2) {
            return n1 + n2;
        }

        int add(int n1, int n2, int n3) {
            return n1 + n2 + n3;
        }

        float add(float f1, float f2) {
            return f1 + f2;
        }

        String add(String s1, String s2) {
            return s1.concat(s2);
        }
```

One key element of OOP is *encapsulation*. Objects can be inherited and used by and within other objects. In Java, this is accomplished by using the `extends` keyword. When a class is extended,

```
        class Thermometer extends Temperature {
```

All attributes of the parent class are passed that have protected status or higher. Additional methods and variables can be created that are unique to the child class.

```
private String color;
```

The `super` keyword is used to access variables and methods from a parent class. It works in much the same way that the this keyword accesses global variables within local functions. The super keyword can access the constructor of a parent class, which allows for the reuse of variables in a parent class.

```
public Thermometer(char system, int degree, String color) {
    super(system, degree);
    this.color = color;
}
```

Classes in Java are treated as types. A class is initialized by using the `new` keyword, which returns the location of the class when it is initialized with the constructor method. The class name must be used as the type of the variable, and the constructor must be supplied with the right arguments.

```
Temperature t = new Temperature('F', 212);
```

Some classes such as ArrayList are initialized with the type of another object. An ArrayList that holds instances of the Temperature class would be initialized with the Temperature class in angle brackets.

```
ArrayList<Temperature> tempArray = new ArrayList<Temperature>();
```

8.2. Modules

Java has many external functionalities available under its module system. These modules are located under a certain namespace. The Scanner module is not in the standard Java base language. However, it can be accessed under the `java.util` namespace. Thus to access the scanner in Java, its namespace must be explicitly called.

```
java.util.Scanner scan = new java.util.Scanner;
```

This method of calling namespaces is clunky. If one were to open two or three scanners, the code would begin to look hopelessly obfuscated. The `import` keyword is used to access namespaces in code without explicit naming. If a module is imported, it can be called without the need to write its full path name.

```
import java.util.Scanner;
Scanner scan = new Scanner(System.in);
```

The module for accessing file streams in Java is called `java.util.Scanner`. The scanner class is initialized with the file stream being passed. To access standard input, the module will have System.in passed to its constructor.

```
Scanner scan = new Scanner(System.in);
```

The Scanner module has two basic methods: *hasNext* and *next*. The next methods return the next instance of a certain type or pattern. The hasNext methods return whether a Boolean value based

on if there is an instance of a type or pattern within the stream. This allows for type safety when gettin user input, such as assigning the value of the input to an integer variable.

```
if (scan.hasNextInt())
    int i = scan.nextInt();
```

FILL IN ABOUT JAVA METHODS TO GET CURRENT DATE AND TIME

The Math class is natively in the Java namespace. It has functions to perform multiple mathematical functions, such as floor division, logarithms, and trigonometry. The Math class contains hundreds of different functions, which may be accessed with the dot operator.

```
Math.floor(4.3) // returns 4.0
```

Another feature included in the Math class is RNG.[37] The `Math.random()` function returns a random floating point number ranging from 0.0 to 1.0 (exclusive). The float multiplication method explained in section 5.2 is used to generate numbers in range.

```
(int) (Math.random() * 10 + 1); // Integer 1-10
```

There are several operations that are commonly performed on primitive types. Some common functionalities would be checking if a char is uppercase, obtaining the string representation of an int, or converting a string representation of a double into a double typed variable. The primitive boxed typed in Java are `Boolean` (boolean), `Byte` (byte), `Character` (char), `Float` (float), `Integer` (int), `Long` (long), `Short` (short), and `Double` (double), The functions associated with the box types are accessible with the dot operator.

```
Character.isUpper('C');
```

Box types can be initialized by passing their respective primitive type to their constructor.

```
Float f = new Float(2.34);
```

Once a box type variable is initialized, it will implicitly use the functions within that type.

```
f.toString();
```

Arrays are a type of object in Java. Thus, they have implicit functions and properties that may be accessed with the dot operator. Arrays are typed as a primitive or class type followed by left square bracket and right square bracket characters. Arrays are also initialized with a certain size. This can be done implicitly by providing the contents of an array surrounded by left curly bracket and right curly bracket characters and seperated by commas at initialization.

```
int[] digits = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
```

If the array's contents are not initially provided, an instance of an array object must be created with the new keyword, passing the size of the array within the square brackets.

```
float[] decimals = new float[10];
```

Once initialized, the array's size is stored in the `.length` property. Array elements can be accessed by putting the index in square brackets after the variable name.

---

[37] RNG | Random Number Generation

```
System.out.println(digits[0]); // Prints 1
```

Using this method, a variable at a specific index can be modified.

```
digits[1] = 9;
```

ArrayLists are lists that still retain some properties of arrays. They are accessible under the namespace `java.util.ArrayList`. Like arrays, they can only contain one type. However, elements can be infinitely appended or inserted. ArrayLists can only hold classes. This requires the use of primitive boxes if primitives are to be used with ArrayLists. They can be initialized by putting the class to be stored in angle brackets.

```
ArrayList<Integer> al = new ArrayList<Integer>();
```

ArrayLists cannot be directly assigned variables. Thus, they require the use of object specific functions to be manipulated.

`.add(int index, E element)`
    Insert an element at a specific index of the list and shifts all following indices to the right.

`.add(E element)`
    Append an element to the ArrayList.

`.remove(int index)`
    Remove an element from a certain index of the list and shift all following elements to the left. Additionally, the element that was removed is returned.

`.get(int index)`
    Return the value of the element at a certain index.

`.set(int index, E element)`
    Reinitialize the value of a certain list index.

`.size()`
    Return the amount of items in the list.

ArrayLists, because of their versatility and dynamic sizing, are often used in place of more traditional arrays.

The for-each loop is a construct in Java that allows for more concise iteration through arrays and ArrayLists. It automatically iterates through a list or array object from the first to the last element. It first has a decleration of a temporary variable to store each elemenet of the list, then, after a colon character, it contains the list that is to be iterated upon. The elements passed to the temporary variable are copies of the original variable. Thus, modifing the temporary variable does nothing to the original variable's value.

```
for (int i: digits)
    i += 3
```

The functionalities of the for-each loop are useful for objectives such as finding the average of an array of integers.

```
    int sum = 0;
    for (int i: digits)
        sum += i;
    float average = sum / digits.length;
    System.out.println("The average of digits is " + average);
```

8.3. Practice

For some hands-on experience with Java, complete the following challenges.

Create a class called `Book`. Within it create a constructor to initialize two properties.

* `title`

* `author`

If the title is less than two characters long, print out a message that the title is short. If the author is greater than thirty characters, print out a message that the author's name is long. Create the following get and set methods within the class.

* `getTitle()`

* `getAuthor()`

* `setTitle(String title)`

* `setAuthor(String author)`

In addition, create a method called `amountOfBooks()`, which returns the amount of times an instance of the Book class has been created.[38]

The function of the Book class is to contain the pages of the book. A "page" is a string provided by the user. Like a physical book, pages will need to be in sequential order. Create the following methods to manipulate the pages in the book.

`addPage(String page)`
    Append a page to the book.

`insertPage(String page, int index)`
    Insert a page at a certain page number in the book, shifting all following pages to the right in the index.

`removePage(int index)`
    Remove a page at a certain index, shifting all following pages to the left in the index. Return the content of the page that was removed.

`getPage(int index)`
    Return the content of the page at a certain index of the book.

`setPage(String page, int index)`
    Replace the page at a certain index with another page, keeping all indices the same.

`pages()`
    Return the amount of pages that are currently in the book.

Extend the Book class to a subclass called `Journal`, which will add the date (provided by the user) to the first line of a page. this is done by overloading the addPage, insertPage, and

---

[38] Hint: reread the section on static variables

setPage methods to add another parameter for the day of the week (string). Additionally, the addPage, insertPage, and setPage methods should be overriden so that if no date (in string form) is provided by the user, the current date followed by a newline is added to the top of the page. Finally, make a constructor that takes a book as an argument and converts it into a Journal. This then iterates through all pages of the book and adds them to the journal, while also adding the current date with a newline appended to the top of each page with the `java.time.LocalDate.now()` method.

9. Shell Script: A Pragmatic Language

Shell script is notably different from any language discussed thus far. It is mainly used interactively, with the user entering each line in real time. Scripts can be stored in files for later use, but most time in the shell is spent manually entering commands. The shell, in its simplest terms, is a way to launch programs.

9.1. Languages

Shell script is a blanket term for a number of different langauges across a variety of systems. Each language that falls under the category of 'shell script' has its own traits. A few of the most popular shell languages are POSIX compliant shell, BASH, and PowerShell.[39] [40] The command prompt in Windows™ is also a shell, although it differs greatly from other shells currently in use.

The shell that will be discussed and used in this chapter will be POSIX compliant shell, or SH as it is commonly reffered to as. This choice is due to its universality, simplicity, and standardization. SH is available by default on all UNIX and UNIX-like operating systems. Even on Windows™, where POSIX-compliant shell is not available by default, users have the option of either using PowerShell (which can validly accept most SH commands) or WSL.[41] SH syntax consists of a relatively small amount of rules that can be easily understood and remembered. Unlike some shells such as PowerShell and Fish, SH syntax has remained the same since the twentieth century, and it is unlikely that any large changes to the standard will happen. All implementations of SH must follow the guidelines set in the POSIX standard. Therefore, the SH instructions in this book are unlikely to become obsolete in the future.

9.2. Programs

Sh programs tend to be very general. They do one job, and they do that job well. For more specific tasks several programs can be used together. Programs are not part of the shell, but are executable files. The shell only calls the executable. Using a shell command can often be quicker and easier than making a program in a traditional programming language like Java.

There are a few basic command line programs that are available across many operating systems. On a system today, there may be more than a thousand different executables available to the user. By knowing of just a few basic programs, a programmer can utilize the shell for doing work.

`cat` Takes multiple file names as input and concatenates their content as output.

`cal` Displays a calendar for a specific month, or year.

`cd` Change the current working directory.

`chmod`
Change the permissions of a file.

`cp` Copies the content of one or more files to another file or directory.

`date`
Displays the current time, day, month, and year.

---

[39] POSIX | Portable Operating System Interface

[40] BASH | Bourne Again Shell

[41] WSL | Windows Subsystem for Linux

`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.

`jobs`
>	List all tasks running in the background.

`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***.

`seq`   Output numbers in a sequence, with optional parameters for start and step.

`sleep`
>	Do nothing for a certain amount of seconds.

`sort`
>	Sort the contents of a file alphabetically.

`sudo`
>	Executes a command with root permissions.[42]

`sudoedit`
>	Edits a file with root permissions.[43]

`touch`
>	Create an empty file.

`uniq`
>	Remove duplicated lines in a file.

`wc`   Count the number of characters, words, and lines in a file.

9.3. Piping and selective execution

FILL IN ABOUT FILE PATHS LATER

A program can be executed in the shell by calling it by its name. Any file located in the system's path is callable in the shell.

---

[42] Root is also known as superuser, hence sudo (**S**uper **U**ser **DO**).

[43] The sudoedit command is equivalent to `sudo -e`.

<div align="center">

`cat`

</div>

If a program is not in the shell, it can be called by using its file location.

<div align="center">

`/bin/cat`

</div>

A file must have execution permissions to be used in the shell. This acts as a security protection, as files that have not been given execution permissions may contain malicious code. Files can be given execution permissions via the chmod command.

<div align="center">

`chmod +x prog.sh`

</div>

After being given file permissions, a file can be executed. If it is in the same working directory, it can be called using its path.

<div align="center">

`./prog.sh`

</div>

A program can be optionally customized. Sometimes, a user may want a program to behave differently from its default behavior. For example, one might want to recursively delete a directory and all of its files. Usually, this behavior is done with the rm program.

<div align="center">

`rm directory`

</div>

However, by default deleting whole directories is not allowed for safety reasons. Using the example above, the directory will not be deleted and the following error message will be shown.

<div align="center">

`rm: cannot remove 'directory': Is a directory`

</div>

So what is a user to do? Luckily, there is a way to delete non-empty directories with the rm program with the use of flags. Flags are denoted by a preceding dash for short flags (one character long) and two preceding dashes for long flags (more than one character long). Short flags are generally older and are part of the standards. Long flags are either aliases to short flags or flags added more recently that may not be part of the standards. Some flags require arguments while others do not. In the case of the rm program, two flags, $-r$ and $-f$, can allow the deletion of directories. The -r flag specifies that deletion should be recursive. The -f flag specifies that the directory should be deleted without prompting the user or instituting any protections. Now the previous rm command can be modified to produce the output that was expected.

<div align="center">

`rm -rf directory`

</div>

Or, the command can be written using the long form options (`--recursive` and `--force`).

<div align="center">

`rm --recursive --force directory`

</div>

When a program has completed its task, it will terminate. Upon termination, an exit code will be emitted. This code will be zero if the program exited successfully (ie without any errors). A program may return any other number to indicate that the program exited unsuccessfully. Using this property, the shell can perform *conditional execution*. Conditional execution causes the shell to execute a command based on whether the previous command executed with or without error. The && operator will execute a command only if the previous command exited successfully.

<div align="center">

`echo hi && echo bye`

</div>

The || operator executes a command only if the previous command exited unsuccessfully.

```
echo hi || echo bye
```

If one wishes to execute multiple commands in tandem, without regard for whether the previous commands executed successfully, the semicolon character can be used. This command uses the semicolon character to wait for three seconds and then output the phrase "Your break is over.".

```
sleep 3; echo Your break is over.
```

The test program takes advantage of conditional execution. The test program is called as a left square bracket, and the program is terminated with a right square bracket. The program uses a combination of values and flags to determine if a statement is true or false. An example of a flag used in the test program is the -z flag. The -z flag tests that a string is empty, and will return 0 (successful) if that statement is true. In the event that the string is not empty, the test program will return 1 (unsuccessful). This can be used advantageously to test whether a variable has been initialized.

```
[ -z "$string" ] && echo string is empty || echo string is not empty
```

Piping is another way that programs can interract with each other in the shell. Using a pipe, represented by the vertical bar, the output of one program can be taken as the input of another program. Multiple programs can be piped together to produce specific output. Just like a garden hose, the program output flows through until it reaches the end of the pipe. Pipes can be used to accomplish abstract tasks that would otherwise require dozens of lines of code in a typical programming language. A task that is perfect for using pipes would be to count the number of files in a directory that contain the letter f. For this task we would need ls, to list the files in the directory, grep, to search for a pattern in a line, and wc, to count the number of patterns returned by grep. Pipes can easily accomplish this task in one line.

```
ls | grep f | wc -l
```

The shell has multitasking built in. Doing multiple tasks at once can decrease the time it takes to complete an objective. Additionally, tasks can be moved to the background so that the shell can be used while another task is running. In computer science, doing multiple tasks at once is called parallelization, because many things are being done simultaneously on different threads. In the shell the ampersand character is used to specify that a task should be run in the background. Once a background task has been completed, a message will be sent to stderr indicating that the task has terminated.

Only one task at a time can be in the foreground. Thus when the sleep command is executed, the foreground cannot be used for the duration that the sleep program is running. However, with parallelization, the sleep command can execute in the background while another program executes in the foreground. This command executes the sleep command in the background while the echo command is allowed to executed in the foreground.

```
sleep 3 & echo This is the foreground.
```

Because an almost unlimited amount of tasks can execute in the background, it is imperative that tasks can be tracked. The jobs command displays all tasks executing in the background. This can be useful for checking if a process is still working or has already been completed or failed.

Most shell programs have simple input and output. They will usually get their input from a file or an argument supplied in stdin. The program will then typically output a string of text to

stdout. However, there are many instances in which a programmer would want the output of a program redirected into a file. This book, for instance, was not orignially in its current heavily formatted form. It started out as markup; plaintext written in a file. Then the file's contents were piped through multiple command line programs to format tables, equations, footnotes, and paragraphs before finally being converted into pdf form. By default at the end of the pipeline the pdf version of this book was output to stdout. While impressive, the contents of a pdf in stdout is completely useless. The output will dissapear rather than being openable by a pdf viewer to view or print. Therefore, it is necessary to *redirect* the output to a file. The shell uses the single right angle bracket operator to redirect output.

```
troff -ms proguide.ms > proguide.pdf
```

By default, the right angle bracket operator will redirect from the file stream stdout. To redirect output from stderr, the programmer may specify that only stream 2 will be output into a file.

```
troff -ms proguide.ms 2> proguide.pdf
```

The single right angle bracket operator will delete the contents of a file before replacing them with the output from the terminal. By using the double right angle bracket operator, output can be appended to a file rather than blanking it. The technique of using the numeral two followed by the redirection operator still work with the append operator. Content can easily be added to a file without the need for a text editor with the append operator.

```
echo Another Value >> values.txt
```

The left angle bracket operator outputs the content of a file to the terminal.

```
< file.txt
```

This can be a good alternative to cat, as it does not require the added overhead of calling another program other than the shell. Multiple files can be output to the terminal in succession by using multiple left angle bracket characters.

```
< file1.txt < file2.txt
```

Shell script allows for values to be assigned to variables. Variables are assigned using an equal sign with no surrounding spaces.

```
var=1
```

The variable can be called with a preceding dollar sign.

```
echo $var
```

However, these variables are unlike those found in other programming langages. They are more like placeholders than variables. When called, they will evaluate to their true form. This can cause shell variables to exhibit idiosyncratic properties. When the name of a program is assigned to the value of a variable, that variable can be called to launch that program.

```
var=cat
$var file1.txt file2.txt
```

The output of commands can be assigned to variables using *.command*substitution Command substitution is done by placing a shell command within parentheses preceded by a

dollar sign.

$$\$var=\$(ls)$$

Command substitution may also be used as input for other commands. By using the cat command and supplying it as an argument a command substitution of the ls command, every file in a directory can be concatenated.

```
cat $(ls)
```

Rudimentary arithmetic can be performed in the shell language with double parentheses preceded by a dollar sign. Addition (+), subtraction (-), multiplication (*), division (/), and modulus (%) can be performed in the shell.

```
echo $((1+2-3*4/5%6))
```

Variables can be called during shell arithmetic without a preceding dollar sign, as all non numbers are assumed to be variables by the shell. There are some limitations to shell arithmetic in that it cannot process decimal (floating point) numbers. The statement $((9/6)) will evaluate to 1 rather than 1.5. This limitation can be overcome by using an external program meant for processing mathematical expressions such as the bc program.

9.4. Complex Elements

The shell *alias* is much like a variable. Similarly, it expands to its true for once it is called. Aliases can be created by using the alias command. After the alias command is called, the name of the alias is declared just as if it were a variable. Aliases can be used to make it easier to call an otherwise long command. A common use of the alias command is to call a program with its options. In my shell startup script, I made an alias called lsp which calles the ls command using its options for color, listing format, and human readable sizes.

```
alias lsp=ls -l --human-readable --color=always
```

Unlike variables, aliases can be called without using the dollar sign.

```
lsp
```

Aliases act just like if a programmer were to type out their value manually. Therefore, it is possible to supply options after an alias is called. The lsp command could be called and told to list the contents of a directory called folder.

```
lsp folder
```

Functions allow for multiple commands to be bundled together into one. Functions are identical to any other command. They can be called, take parameters, and can even exit successfully or unsuccessfully. Functions are declared in shell by entering the name of the function proceeded by parentheses and a left curly bracket.

```
func(){
```

Functions are ended with a right curly bracket.

```
}
```

Within the function body, any number of commands can be executed. What makes shell

functions unique is that they use implicit variables. Instead of a function needing to name parameters in its definition, it uses numbers to represent the order of arguments provided to the function. $0 is always defined and will hold the name of the function. When $0 is called within the shell, it will name the binary that the shell originates from. $1—$∞ hold the arguments supplied by the user to the function. If the user does not provide an argument, the variable will still be accessible, but it will not hold any value. Functions exit automatically once all their commands have been run. However, a function may be exit prematurely using the `return` command. The return command can optionally take a number as a paramter which indicates that the function was successful (zero) or that it was unsuccessful (any number other than zero).

Now let's try a practical example. Whenever a code project is made, it is best practice to include it in a directory that has a certain structure. It should include a readme file, which explains what the project is about and how to use it. A license, to make known who owns the code and what permissions they give to people using their code. Additionally, it should include a `bin` directory for executables and an `src` directory for source code. If the project is going to be collaborated on, it should be in a *git* repository. Git is a version control system that allows for stages of a project to be tracked and modified by multiple people. When git is used, a `.gitignore` file can be used to specify files or directories that should not be committed. In the case of a code reposiory, the bin directory should not be committed because binaries tend to be very large and are not useful for code collaboration. Put together, the structure is this.

```
<ProjectName>
|    .gitignore -> "bin/"
|    LICENSE
|    README.md -> "# <ProjectName"
|
+---bin
\---src
     +---main
     |   \---resources
     \---test
          \---resources
```

Creating this code repository would require manually entering many commands. Instead of having to type out these commands every time a code project is created, these commands can be bundled into a function. This function should have a short yet descriptive name, such as `mproj`.

```
mproj() {
    mkdir $1 || return 1
    cd $1
    echo "# $1" > README.md
    touch LICENSE
    echo "bin/" > .gitignore
    mkdir -p {bin,src/{main,test}/resources}
    git init >/dev/null
}
```

All shells have what is known as a startup profile. This profile executes commands whenever a shell is launched. The profile could be used to greet a user whenever they launch the shell.

```
echo Hello, $USER.
```

More useful, the startup profile can be used to store commands that the user enters often and execute them automatically. Add about how aliases and functions persist when put in the startup profile and will die when the shell is exited if they are not in the startup profile. The file that is used as the shell's profile varies depending on the shell that is used. In POSIX shell, the file that is used as the startup profile is called `~/.profile`.

Shell wildcards are used to match files that begin or end with certain characters. The shell wild card is represented by the asterisk character. For example, to output all files beginning with la, The literal la would be present followed by an asterisk.

```
echo la*
```

To output all text files, the asterisk would be used followed by the extension `.txt`.

```
echo *.txt
```

If one were to want to use a literal asterisk in the shell, a backlash character would be used.

```
echo \* is an asterisk.
```

In addition to conditional execution, shell script has a more formal if-else statement. Instead of being terminated by indentation or brackets, shell script if statements are terminated with the word if spelled backwards (`fi`). The if statement is started by the word if, followed by a logical statement surrounded by square brackets. Afterwards, the keyword `then` is used to indicate that the code to be executed is starting. Optionally, an else statement can be included to execute when the if statement returns false. Finally, the `fi` keyword is entered to complete the construct. Let's try a simple if statement, that will output "even" if `$i` holds an even number and "odd" if the `$i` holds an odd number.

```
if [ $((i%2)) -eq 0 ]
then
    echo even
else
    echo odd
fi
```

While it certainly isn't *wrong* to use if-else statement in shell script, it also isn't necessary. Conditional logic is cleaner, shorter, and can replace almost all instances of if-else statements. The only reason to ever use an if-else statement in shell is when there are multiple statements to be executed depending on whether a condition is true or false. For most situations, simple conditional execution will suffice. The even odd program that was made with if statements can be created in just one line using only conditional execution.

```
[ $((i%2)) -eq 0 ] && echo even || echo odd
```

The `while` keyword in sh takes the form of the if statement. However, instead of starting execution with `then` and ending with `fi`, it starts execution with the `do` keyword and ends with

the `done` keyword. The following code will continue executing until $i is greater than or equal to 10, outputting the value of $i on each iteration.

```
i=0
while [ i -lt 10 ]
do
    echo $i
    i=$((i+1))
done
```

The `until` keywords functions the same as the `while` keyword except that it will continue executing as long as the statement is false.

```
i=0
until [ i -ge 10 ]
do
    echo $i
    i=$((i+1))
done
```

The `for` keyword will iterate through a list of values. The syntax consists of the `for` keyword, followed by the name of a temporary variable. The values to be iterated through are provided after the `in` keyword. Following the values, a for loop has the same `do` and `done` keywords that the while and until loop have. In its style, it is very similar to a Python for loop. Values can be manually inputted by the user, in the case of this for loop which outputs every letter of the alphabet on a seperate line.

```
for c in a b c d e f g h i j k l m n o p q r s t u v w x y z
do
    echo $c
done
```

To iterate through a range of numbers, the `seq` command is used. The seq command takes three arguments: start, step, and stop. The while loop from above that iterates through the numbers 0—9 can be replicated more compactly with a for loop.

```
for i in $(seq 0 1 9)
do
    echo $i
done
```

Strings can be declared in SH with either single or double quotes. This distinction is not meaningless. Strings declared with single quotes are literal, and all special characters such as $ and ` will be ignored. Double quoted strings will execute code between backticks and after dollar signs. This distinction is important. Double quoted strings can represent variables, execute commands, and even perform arithmetic. All statements that a programmer wishes to be read literally must be escaped with a backslash in double quoted strings. The content of strings (whether single or double) can be read as a command using the `eval` command.

```
eval 'echo code in a single-quoted string.'
```

10. C: The Final Frontier

C is the most used programming language in the world. Everything from embedded software to artificial intelligence algorithms have been written in C. The C language is renowned for its simplicity, ease, and speed. Due to C's age and low resource requirements, it ommits many of the common metaphors known to higher level langauges. C is very close to the "metal" of a computer. Some experienced C programmers can even see exactly what instructions will be sent to the machine in their C programs. Yes, there is a learning curve. It can be difficult for some developers to distance themselves from the ideas of objects, classes, and types. However, even if one does not do the majority of their programming in C, learning C can help bring a greater understanding and appreciation of what a computer actually does and how it truly functions.

C serves as a sort of middle ground between low and high level languages. It has access to advanced faculties such as structures and control flows yet has the ability to manipulate the machine on the bit level. Indeed, programmers skilled in both C and Assembly can see how their C code is directly translated into machine instructions. Thus, C has been applied and continues to be applied from simple embedded software to complex modern-day web browsers. Knowing C gives a programmer that freedom to directly manipulate their machines while using powerful high level abstractions.

Most major languages take at least some inspiration from C. C was one of the first langauges to implement control flow blocks such as for and while loops. Before C all control flow had to be done with *goto* statements. If statements could only conditionally execute one command. The result was spaghetti code, in which large code bases were nearly impossible to trace. By introducing bracketed blocks, multiple commands could be executed under one condition. C still has goto statements but, except in certain rare circumstances, they are unnecessary.

The basic constructs in C are very similar to what was in the previous Java section. The if-else, while, for, and case constructs all follow the same rules and forms. Therefore, this text will not restate how these constructs are formed or how they function. It is recommended that the analogous parts of the Java section are reread in order to refresh your memory.

Primitives in C are numeric types. The primary and only difference among them is the amount of bits they can store. This is, the highest (and lowest) number that can be possibly held. The `char` primitive is the smallest type. It can hold only a single byte. This is the amount of room required to represent the highest ASCII character value, hence the name. Thenceforth the integer types go on through `short`, `int`, `long`, and `long long`. The exact amount of bytes stored in these primitives depends on the compiler used. In general practice the char variable can be used for storing ASCII values or put together to form strings, while the int can be used to store most numbers that would need to be computed by a programmer.

C has two primitives that may be used to store floating point numbers. The `float`, which is single precision, and the `double`, which is double precision. These types are suitable for when a non-whole number is used but absolute accuracy is not needed. This applies to applications such as finding the average of a data set or finding the difference between an expected and actual result.

The void type represents nothing. Although variables can be typed as void, this is quite useless in practical programs. The void type is more meant for functions which yield no return values, and another application that will be discussed in the memory management section of this

chapter.

  The `unsigned` keyword is used to denote that a variable cannot hold negative values. By allowing for an extra bit which would otherwise denote the sign of the number, the variable can hold two times as high a number as was previously possible.

```
char a; // can hold a 7 bit number (-127, 127)
unsigned char b; // can hold an 8 bit number (0, 255)
```

  The streams are manipulated by using the standard input/output functions. In order to access these functions, this line of code must be present at the top of the C file.

```
#include <stdio.h>
```

Four functions are most used for interacting in a basic way with the streams.

  The first, `putc` outputs a single character to a stream specified by the programmer. It takes two arguments, the first being the character value and the second being the stream. The following code uses putc to print an uppercase A to *stdout*.

```
char c = 'A';
putc(c, stdout);
```

10.1. Preprocessing

10.2. Memory Management

10.3. Practice

For some hands-on experience with C, complete the following challenges.

1

11. Resources

Truth table[44]

p	q	¬p	¬q	p∨q	p∧q	p⊕q	p↔q
1	1	0	0	1	1	0	1
1	0	0	1	1	0	1	0
0	1	1	0	1	0	1	0
0	0	1	1	0	0	0	1

De Morgan's law proofs[45]

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

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

Setup

Programming setups are 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.[46]

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 WPM, the better.[47] A rate of about forty WPM is the minimum requirement to effectively be able to write programs.

---

[44] A truth table asserts the result of a logical operation in multiple circumstances.

[45] De Morgan's laws are logical proofs that stipulate that the compliment of not A or B is equivalent to not A and not B and the compliment of not A and B is equivalent to not A or not B.

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

[47] WPM | Words per minute

12. Exercise Keys

12.1. Python

1

```
print(f"The sum is {int(input('Enter a number: '))+int(input('Enter another number: '))}.")
```

2

```
from random import randrange
rlist = [randrange(1, 10) for x in range(10)]
print(rlist)
```

3

```
from random import randrange
rndnum = randrange(1, 10000)
for x in range(20):
    guess = int(input("Guess a number between 1 and 10,000: "))
    if guess == rndnum:
        print("Congratulations! you guessed the number.")
        break
    elif guess > rndnum:
        print("You guessed too high.")
    else:
        print("You guessed too low.")
else:
    print(f"You failed to guess the number {rndnum} in 10 tries")
```

12.2. Java

Book Class

```
import java.util.ArrayList;

class Book {
    private String title
    private String author;
    ArrayList<String> pages = new ArrayList<String>();
    private static int books = 0;

    public Book(String title, String author) {
        this.title = title;
        this.author = author;
        if (title.length() < 2)
            System.out.println("Wow, that title is SHORT!");
        if (author.length() > 30)
            System.out.println("Wow, that name is LONG!");
        books++;
```

```java
    }

    public String getTitle() {
        return title;
    }

    public String getAuthor() {
        return author;
    }

    public void setTitle(String title) {
        this.title = title;
    }

    public void setAuthor(String author) {
        this.author = author;
    }

    public int amountOfBooks() {
        return books;
    }

    public void addPage(String page) {
        pages.add(page);
    }

    public void insertPage(String page, int index) {
        pages.add(index, page);
    }

    public String removePage(int index) {
        return pages.remove(index);
    }

    public String getPage(int index) {
        return pages.get(index);
    }

    public void setPage(String page, int index) {
        pages.set(index, page);
    }

    public int pages() {
        return pages.size();
    }
}
```

Journal Class

```java
import java.time.LocalDate;

class Journal extends Book {
    public Journal(String title, String author) {
        super(title, author);
    }

    public Journal(Book other) {
        super(other.getTitle(), other.getAuthor());
        for (int i = 0; i < other.pages(); i++)
            addPage(other.getPage(i));
    }

    public void addPage(String page) {
        addPage(page, LocalDate.now().toString());
    }

    public void addPage(String page, String date) {
        super.addPage(date + "\n" + page);
    }

    public void insertPage(String page, int index) {
        insertPage(page, LocalDate.now().toString(), index);
    }

    public void insertPage(String page, String date, int index) {
        super.insertPage(date + "\n" + page, index);
    }

    public void setPage(String page, int index) {
        setPage(page, LocalDate.now().toString(), index);
    }

    public void setPage(String page, String date, int index) {
        super.setPage(date + "\n" + page, index);
    }
}
```

12.3. C

Table of Contents