Code for Thought Documentation

Release 0.1.0

Taylor "Nekroze" Lawson

July 14, 2016

Contents

This, that I loosely call a book, is meant to guide your first steps into the world of programming and abstracting thoughts and reality into easy to understand code.

CHAPTER 1

Table of Contents

1.1 Chapters:

1.1.1 Introduction

While **Code for Thought** is designed for those who have never programmed, it is not exactly designed for the computer illiterate. There are some assumptions made by this book that I should get out of the way before we get started.

Requirements

First and foremost we recommend and assume a *POSIX* based environment. This means either a *Linux* operating system, like Xubuntu (which could be installed as a virtual machine using VirtualBox in any operating system) or on windows you could install Cygwin (an easy way to get a *POSIX* based environment in windows for a few trade offs). All instructions in this book are given as command line commands for a *POSIX* environment. There may be equivalents for your own setup that can be found on the internet with a cursory google search.

There are many many guides on the internet on how to setup these environments, many of which explain it all better then I could in the limited scope of this book.

Ontop of some kind of *POSIX* based environment you are going to need to install *Python* along with the packages called Setuptools and *Pip*. *Python* will be the language we will be learning how to code in. It is a very nice to read and concise language that is perfect for beginners learning the basic concepts of programming. Setuptools and *Pip* will help us to install and distribute *Python* packages.

In these environments there should be some kind of text/code editor you are comfortable with using. Anything will do and you can change your mind and use something else at any point. Personally I use Emacs which is a great editor that has a long tradition in the programming world, although people say it has a steep learning curve. Most programmers prefer Vim to Emacs but there are many alternatives and if it comes down to it windows notepad will work fine... However if you want to use notepad give NotepadPlusPlus a try. It is similar to notepad but with useful programming features.

This can all be done in pure windows obviously however programming in *Linux* is just much nicer. If you wish to program in windows only (without Cygwin) you will still need a text editor and be able to use the command prompt to run your code. There may be mention of tools that are different to use or nonexistent on windows. Most likely in these cases there are alternatives that should not be too hard to find with a quick Google.

If you do not want to deal with the recommended environment outlined above you can follow much of this book using the REPL website which offers a half decent python interpreter right in your browser. Much of the code examples in this book are written in *Python* because it is a clear, and easy to follow, programming language that follows many of the universal programming concepts and REPL offers a great solution without making any changes to your computer.

The website offers a *Python* 2.7 interpreter and a nice clean text editor on the side that can interact with the interpreter all inside your browser. However later chapters in this book will fall out of the scope of the REPL website.

Structure

Code for Thought will take you through learning how to write your own code from start to finish. By the end of this you will understand the major concepts behind programming and the tools that will make your code more successful. You will learn how to test your code and make sure it works, how to document it so that others can help contribute to your code, and learn how to contribute to other peoples code and make the world a better place.

OK. maybe you won't be influencing the entire world by the end of this book but today, more than ever, the world needs people who can program. Almost everyone you know uses computers far more then they would have 15, 10, even 5 years ago. Yet we are all just users, so few care to even think what is behind it all. Truth is that in general it is simpler than you think. If more people can innovate and contribute to the growing world of computers then that has to be making the world at least a little better, right?

This book will move slowly from concept to concept, focusing not just on teaching you some information about programming but actually teaching you how to teach yourself to program.

We start with an introduction to the basic concepts of programming with interactive examples. Then we move onto abstracting ideas into functions. Then on to data structures. With these basics we will begin to construct simple programs you can test out and tinker with. Then we will use example programs and code to provide real life usage of the advanced concepts you will be learning. As the projects we work with grow in size we will introduce new tools that can greatly help with programming. These include, but are not limited to, things like; version control, unit-testing and documentation.

At the end we will have a more free form discussion (albeit rather one sided) about programming concepts and tools for the future.

Dedication

This book is dedicated to, Elysha. **Code for Thought** is designed to help her and others like her to learn code and better understand the second love of my life.

1.1.2 Playground

Put on your sneakers kids, we are going to the playground. When the day is out we will rule the school... But less lame sounding, I promise.

Firstly we want to fire up the *Python Interpreter*. You can use the default *Interpreter* perfectly fine and many do. However if you want some extra features to make learning and using the *Python Interpreter* easier you might want to check out bpython or install it using the following command:

```
$ pip install bpython
```

Now open a new *Python Interpreter* using either:

```
$ python
```

or if you chose to install bpython:

```
$ bpython
```

You will be presented with *REPL* environment that you can play around. If anything goes bad or you want to start again you can close the *Interpreter* down using `CTRL+D` or executing this command on a new line in the *Python Interpreter*:

```
>>> exit()
```

Afterwards you can re-open it the same as before for a new environment.

New kid on the block

This is our stomping ground so we need to learn how to start stomping to get results!

The first piece of programming we will be learning is a simple expression. Expressions are chunks of code that do something and return results.

There are a great many things you can do with expressions but for now lets just try some simple math in our *Python Interpreter*:

```
>>> 1 + 2
3
>>> 1 / 2.0 * 20
10.0
>>> 1 / (2.0 * 20)
0.025
```

Great so we have written some simple, but boring, mathematical expressions in *Python*! But the last two examples are a bit different. First, if you didn't guess already, the symbol for multiplication in programming is the asterisk (`*`) character and division is the forward slash (`/`) character. But the last two examples are almost exactly the same except for the parenthesis around the `2.0 * 20` expression in the final one.

The reason for the parenthesis is to solve one of the largest problems in programming. OK well not specifically but bare with me for a moment. One of the largest problems for new programmers, other than the syntax of the language they have chosen, is understanding that the computer does not (and can not) think the way they do. It has no clue what you want to do with your code. This makes it very hard for a computer to figure out what the right thing to do is, so often it doesn't even try.

In programming we need to make our intentions clear and preferably concise. Not only does a computer have to understand what you mean but so do other humans. This speaks to a balance that we need to find between telling the computer exactly what to do to get it right, and being able to actually articulate, and understand those commands ourselves. Remember that while you may understand what you write today if you come back in six months will it still make perfect sense?

But we are getting ahead of ourselves a bit. We use the parenthesis in the last example above because we want to divide 1 by the result of 2.0 multiplied by 20. Whereas in the second example we are dividing 1 by 2.0 and then multiplying the result of that by 20.

Can I have a locker next to yours?

So we have some basic numbers and we can manipulate these numbers. What we need to do now is store them. In programming we use variables to store information under a (sometimes) easy to remember name. Instead of just saying `100` we can store that number in a variable called `distance` to more easily remember what the number does and what it means. Languages have many different ways to create and interact with variables. Luckily *Python* is a dynamic language (more on that in the future) and we can just give any value a name really simply:

```
>>> distance = 100
>>> distance
100
```

Now that we have stored the speed variable we can use it in calculations instead of the number and store the result.

```
>>> speed = distance / 20.0
>>> speed
5.0
```

The above is just a simple velocity calculation (I promise we will move away from maths soon) that uses the stored `distance` variable we set earlier and divides it by `20.0` (the time it took for our imaginary vehicle to travel that distance) and then stored the result in the variable called `speed`.

The important thing here is not the maths, it is the fact that you can store almost anything to a variable and use the variable instead of the actual value. Now that distance is stored in a variable all we have to do is change the distance value to something else and re-run the speed calculation and it will use the new distance! OK not that exciting yet. But it will be.

He's Just Not My Type

There are more things than numbers in the world of programming. And there is much more than maths. Actually only very few programming fields are math heavy. Mostly we deal with basic data types and manipulating them to become what we want.

Generally speaking, there are only a few basic types of data we can use and store.

Strings

A string is just text, any kind of text really. Some languages have different ways of writing these but mostly a line of text enclosed with quotation marks denotes a string.

```
>>> name = "Taylor \"Nekroze\" Lawson"
```

The above example works perfectly well in *Python* to store a string of my name. But there are some important things here. If a string is any text between two quotation marks then how do we include the same quotation mark in our text? For this we have *Escape Sequences* these are characters that have a backslash (\) before them and are read as a single letter, rather than two letters. In the case I presented we use `\"` to show that we don't want to end the string but rather to include a quotation mark inside of it.

Now in *Python* we have the ability to also use single quotation marks as well as the double so we could have just as easily done the following:

```
>>> name = 'Taylor "Nekroze" Lawson'
```

And now it would work fine without using the *Escape Sequence* `\"` because the `"` character would not close the string in this case. Which you use is up to you in *Python* however in some languages the single and double quotation mark means different things.

For example sometimes we differentiate between a string and a character. A character is just one letter and a string is a collection of characters. But, dynamic languages to the rescue once more, *Python* just takes either one and stores is for you without complaining.

In *Python* we can also easily do multi line strings by using a *Triple-Quoted String* which can use either single or double quotes and works on multiple lines of text.

Numbers

In programming we split numbers into different categories. Some languages have more categories than others. The main split is between an *Integer* and a *Floating Point Number* (which is usually called a *Float*).

An *Integer* is any whole number; `1, 2, 3, 4, 5,` etc. Whereas a *Float* is a number that has a decimal point such as `1.1, 1.2, 1.3, 1.4, 1.5,` etc.

There is a difference in these types, not just conceptually, but in the way the computer handles them. *Floats* are harder for the computer to work with and take more space to store them. Also *Floats* are a representation of a number, they are not always accurate but are usually accurate enough.

Some languages also make a distinction between small and large numbers. Many languages can have either an *Integer* or a *Long*. A long is mostly the same as an integer however its maximum and minimum values are much larger than an *Integer*. When it comes to *Float* there is a similarly larger version in many languages called *Double*, which just means double the precision thus a longer decimal point.

Once again in *Python* we don't have to worry about the differences all that much, If we want to use any type of number *Python* will just store it and keep on working. However there is one thing worth noting when working with different types of numbers. Because a both a Long and a Float have more information then a simple Integer can hold if we change the types of a value around we may end up loosing some information in the process.

Booleans

Booleans are interesting. A *Boolean* value is either `True` or `False`, that is all they can store. Think of it like a switch that is either on or off.

Some languages allow many different things to be considered in *Boolean* terms. For example in *Python* (and most languages) `0` is equivalent to `False` and anything higher then and including `1` is the same as `True`. Later we will see other ways to use many types of data as *Booleans* as well.

Collections

This is where it can get a bit crazy. A collection at its simplest is just a way of grouping other data types together to store a collection of "things".

Your basic collection is a *List*, which works exactly as you would expect. Just add in your data and it is all stored together and can be manipulated as you wish. For example:

```
>>> shades = ['white', 'black']
>>> shades.append('grey')
>>> shades
['white', 'black', 'grey']
```

This is how we make a *List* in *Python* and add an element to it. Because *Python* is a powerful dynamic programming language we can store any types we want in any given collection. However many other programming languages require collections to be homogeneous, this means that all values must be the same type.

There are many other types of collections. Another very common type is the *Dictionary* (or *Hash Table*). These allow you to make a map of one data type to another, like looking up something in a dictionary.

```
>>> favorite = {'color': 'black', 'language': 'Python'}
>>> favorite['color']
'black'
```

We have just created a dictionary, stored it in the `favorite` variable and then given it some simple mappings. On the second line we look up what the dictionary holds under the string `color` and retrieve it.

Later on we will look at classes which are kind of like collections, in that they can hold a variety of types at once, but with some tasty additions.

I Love it When a Plan Comes Together

Using just the types of data above and learning how to manipulate them we can make just about any piece of software we can imagine. No, really. Pretty much every computer program ever written uses some form of the above data types along with a series of tricks to manipulate and control them. It's kind of beautiful if you think about it. Ever single computer in the world; phones, laptops, airplanes, traffic lights. At some level these are all controlled by code that just fiddles with these basic types. This is why coding is such a powerful field, everything uses it somewhere.

The goal is for you to learn how programming works, not just *Python*. Play around with these data types in the *Python Interpreter* to get a better feel for how they work, because these things are almost entirely universal in programming. And once you get the basic concepts behind programming itself, the language you use becomes a trivial wrapper around your thoughts. Now that is what **Code for Thought** is all about!

In the next chapter we will be looking at using functions and telling the computer how to do a repetitive tasks.

1.1.3 Form and Function

Here we will be looking at functions and control flow statements that can be used to simplify common instructions and separate usage and implementation. Important concepts for any programming project.

What is your quest?

Say we wanted to do a calculation. Say we wanted to do a calculation often. It would be useful (increasingly so with added complexity) to separate the calculation into a *function*.

Functions are sets of instructions that are executed whenever the function is called by name and can optionally take in data and even provide an output.

Imagine we had a reason to count how many times the letter "a" is in a given string and return the result. Instead of writing the instructions required to perform this task each time it is needed we could write a function that does it for us. For example in python we would do the following:

```
>>> def count_a(text):
...     count = 0
...     for letter in text:
...         if letter == "a":
...             count += 1
...     return count
>>> count_a("A test sample")
1
```

The above code presents a host of new programming features that will be covered in this section.

Firstly we defined a function called `count_a` and told it to take one parameter called `text`. This parameter is our input that we will count.

In programming we have the concept of code blocks, these are a section of code that belongs to some other code. Some languages use braces such as `{ }` to say that anything in between those symbols is a code block. However *Python* uses white space indentation do denote a code block. This means that anything on the same indentation level is part of that code block and further indentations denote nested code blocks. After defining our `count_a` function we need to give it a block of code that defines what that function will do when it is called latter. Code blocks are used whenever we need to define what a specific thing should do when it is called; the code of a function, what to do in a loop, code to execute if a condition is met, etc.

The next line should be rather familiar by now. We are simply creating a integer variable to store the count, which starts at zero.

Here is where things get interesting, control flow. We need to check over each letter in the `text` variable and see if it is the letter we are looking for. To accomplish this we have employed a for loop. For loops are very common in programming however some languages have different types of for loops. In python the for loop is more of a `foreach` loop, unfortunately it is not named as such. Basically it takes any `iterable` value (more on that later but for now its anything that can be looped over) and gives you each piece of that value until there are no more pieces left. Here we have given the for loop the text value and told it to call each piece of it `letter`.

Next we introduce another new type of control flow called the if statement. An if statement evaluates a boolean expression and then allows you to execute instructions if it is true. The boolean expression we are using here is `letter == "a"`. Meaning if the piece of the text parameter we are looking at currently is the same as a string containing the letter `a` then the boolean expression evaluates to true. The if statement, now having an expression that equals true then executes its code. In this case that is to add `1` to the `count` variable.

When adding one to the `count` variable we are using the in place addition operator += because we want to store the result in the same place. we could instead use the following:

```
count = count + 1
```

this would do the calculation of whatever is stored in the count variable plus one and then store it into the count variable. This does the exact same thing but is a little longer. However we could not just say `count + 1` and thats it because that is an expression and that expression will return the sum of count plus one but has no idea that it needs to be stored.

After the if statement the for loop will return to the top and get the next `letter` from `text` and do it all again until we have gotten to the end of the input.

Once all letters have been checked the `count` variable will now be storing the exact amount of the letter `a` that are contained within the text that this function is given. But its of no use to just count the letters and then forget about it so we need to return the results using, you guessed it, `return count`. By now it should be rather obvious that this will simply return whatever is stored in the variable `count` back out to whatever has called this function.

Finally we see our brand new function in action by calling `count_a("A test sample")` return the result `1`. In our function we only checked for the letter `a` but we must remember that this is not the same as `A` as strings are case sensitive. If we wanted to check for both lowercase and uppercase `a` we could change the boolean expression for the if statement to a little to the following:

```
letter == "a" or letter == "A"
```

Then the if statement has two boolean expressions that make up the larger boolean expression. Firstly it checks for the lowercase `a` then if that is false it will continue to the uppercase `A` check and if that is also false then the entire boolean expression will be false and not execute the if statements instructions. In python you can also use the `and` keyword if both of these boolean expressions needed to be true in order to proceed. In this case however the `and` keyword would do no good because a letter cannot be lowercase and uppercase at the same time so we use `or`.

Go with the flow man

In programming we often use control flow statements to alter the way our code performs.

If your happy and you know it

For example if we want code to do one thing or another depending on a variable we use an `if` statement.

The if statement evaluates a boolean expression and executes the body of the if statement if that boolean is positive. If however it is negative the code can either; look for an else if and evaluate that expression next, look for an else statement and do that instead of any condition, or finally continue normal execution of the code.

In python we can give the following example that takes in a message code and then prints the corresponding message:

```
>>> def message(code):
...     if code == 1:
...         print("Hello world!")
...     elif code == 2:
...         print("Goodbye cruel world!")
...     else:
...         print("I don't even know what to say...")
...     return True
>>> message(1)
Hello world!
True
>>> message(2)
Goodbye cruel world!
True
>>> message(3)
I don't even know what to say...
True
```

Each part of an if statement is executed one after another until a positive boolean expression is found. An `else` block is optional as is an `else if` (called `elif` in python) however if an `else` is used it must be last in the chain as it acts as a sort of "catch all" in that if none of the if statements are executed then the else surely will be.

Do a barrel roll!

Doing something over and over again until a particular time is done using loops which are most commonly; `for`, `foreach`, and `while`. In some cases only one of the first two loops are available. In python there is a loop that is started using the keyword `for` however it behaves like a `foreach` loop.

There are some minor but important differences between each loop type.

For Usually takes 3 statements; variable to count with, how to increment the variable, and the condition in which to stop the for loop. In `c/c++` a for loop would look something like this:

```
for (int count; count >= 10; count++){
    dosomething();
}
```

This would, in order:

- create an integer variable called *count*.
- ensure the loop will stop when the number stored in *count* is greater then or equal to the number `10`
- instruct the for loop to increase the number stored in *count* by 1 each loop.
- call the function `dosomething` with no arguments.

The body of the loop is the function call `dosomething();`. The body of a loop gets called until its end condition is met and the loop as played itself out. Alternatively the body of a loop can tell the loop itself to `break` and thus the loop will stop and return to executing the code outside of the loop.

Foreach A foreach loop typically wakes two statements. A iterable object and a name to use for each element from that object.

Firstly, an iterable is anything that has multiple elements to be retrieved or iterated over. A list of names is iterable making a for loop the best way to perform some action that uses each name on the list.

The following python code (python uses the `for` keyword even though it is really behaves like a `foreach`) will iterate over a list of friends names and then call the `print` function with the current name until all names have been iterated over.

```
>>> friends = ["nekroze", "lyshkah"]
>>> for name in friends:
...     print(name)
nekroze
lyshkah
```

The `foreach` loop is a little bit of a newer concept then the `for` loop. Many programmers would use a `for` loop that had an end condition be the length of a list and just use the counter as an index to the list. This was doing essentially the same thing as the `foreach` loop does but is much more complicated. Python gets away with having only a `foreach` loop (using the `for` keyword however) because you can still get the original `for` loop functionality by generating an iterable range for example, but many languages have both.

While The while loop takes only a condition and will keep looping until that condition is met. A `while` loop is much like a `for` except it is up to the programmer to create the counter variable and implement how it is increased.

```
>>> loops = 0
>>> while loops < 10:
...     loops += 1
>>> loops
10
```

An important thing to note, especially with `while` loops, is that the condition can be any expression that can equate to a boolean. This can even be a function so long as it returns something that can be considered a boolean.

1.1.4 The Object of my Desire

Now we have data that we can store and functions to easily and repeatable manipulate that data but sometimes it is not enough. Many programming languages use a paradigm called Object Oriented Programming (commonly referred to as *OOP*) that allows us to do many cool and complex things in a relatively elegant way.

The primary feature of *OOP* is... you guessed it, Objects. An object is a named collection of variables that is used as a template to create a data structure that performs as specified on demand.

This may sound quite complicated but it just needs to be explained better. Objects are a great way of conceptualizing real world objects in programming. Say we wanted to create a representation of a cube in our code. We could use collection data types like a dictionary or some such to store the data about our cube. What would be even better, however, is to create an object that defines how a cube should act and what kind of data it would store.

The first thing that needs to be done is to think about what kind of data we want to store. For a cube it should have at least a size variable to store, well, its size. Because we are defining a cube and each side should have the exact same length we can use one size variable for all of the sides. This is a fine representation of a cube, albeit very simplistic. Lets also decide we want to store its position in a three dimensional space. For this we simply need to store an; X, Y and Z variable to describe its position.

Now enough theory lets have a look at this object in some real code, namely *Python*.

```
>>> class Cube(object):
...     def __init__(self):
...         self.size = 0
...         self.posx = 0
...         self.posy = 0
...         self.posz = 0
>>> companion = Cube()
>>> companion.size = 10
>>> companion.size
10
```

OK, some basic things to get out of the way. In *Python* objects should inherit from the base object, this is why after we name our new "class" (the common name for an object definition) we place `(object)` to denote that this class acts like an object.

Objects often have "constructors" and sometimes "destructors" these are functions (or "methods" as they are called when they are part of an object's definition) that are called when, you guessed it again, the object is constructed and/or destroyed.

Also often when defining classes/objects and their methods we use the terms `self` or `this` to mean this instance of an object.

In the above example we use the *Python* object constructor `__init__` that takes an object instance as an argument (`self`) and will give its variables their default values, in this case the integer `0`.

Next we assign the variable *companion* as a new instance of the *Cube* object by calling the object as if it where a function. Finally we set the *size* variable of our new *Cube* object to `10` and finally we show that the change worked.

Now we can create any number of *Cube* objects each with their own values by creating a new instance just as we did above with *companion*.

Other languages employ different methods and keywords for using and creating objects, classes, instances, etc. and is usually very easy to find on the web.

The Layer Cake

Another very useful feature of *OOP* is Inheritance. What this means is that one object definition can be based on another, taking all its variables and methods and building on top of them.

Lets just go straight to an example this time.

```
>>> class InSpace(object):
...     def __init__(self, posx=0, posy=0, posz=0):
...         self.posx = posx
...         self.posy = posy
...         self.posz = posz
>>> class Cube(InSpace):
...     def __init__(self, size, posx=0, posy=0, posz=0):
...         super(Cube, self).__init__(posx, posy, posz)
...         self.size = size
>>> destination = InSpace(1,posz=5)
>>> destination.posx
1
>>> destination.posy
0
>>> companion = Cube(10)
>>> companion.posx
0
```

This time we are doing things a little different.

We start off with similar thing to before, we are just creating a new class to define things that exist in a three dimensional space. However here we are using default arguments to allow the constructor to optionally take the position of an *InSpace* object only if it is given, otherwise that dimension will be `0`.

Next we define a new *Cube* object, this time instead of inheriting directly from *object* we inherit from *InSpace*. This means that our new object will have everything that *InSpace* has and can be used anywhere an *InSpace* object is expected. For this objects constructor we tell it that we want the size argument to be required and have the position arguments to default to `0` upon creation/initialization of this object.

In some languages, *Python* included, you will need to explicitly call the constructor of the "parent" object if you want it to be executed. *Python* uses the `super` function to make this a bit easier in *Python* 3 it is even easier as `super` can be called with no arguments to do exactly the same thing as above, but people are still using both so I show what works everywhere.

This is more language specific rather then general programming and so is not something I will go into too deeply. Suffice to say that above we use `super` to get the object definition of the parent of *Cube* and then call its constructor appropriately.

After we have defined our object hierarchy I have just done some example usages of both classes including different ways to use the optional positional arguments.

The Method to my Madness

Now we can go about doing cooler things like giving special methods that only cubes can use or even better adding methods to *InSpace* that allows every object definition that inherits it to easily move around without having to update its "children" such as *Cube*. In fact lets do just that!

Using the above example, again, any changes in the code to the *InSpace* class will be reflected in any class that inherits from it (it's children) accordingly. Because of this we can easily abstract the concepts behind a class in its base components. So if everything exists in a three dimensional space it might be a good idea to implement things specific to being in such a space in a class such as *InSpace* so each object that derives from it does not have to implement such things over and over again. This leaves each object inheriting from *InSpace* to focus on what it specifically needs to accomplish it's job.

With this in mind let us redefine the *InSpace* class with some methods to help us move around in a space.

```
class InSpace(object):
    def __init__(self, posx=0, posy=0, posz=0):
        self.posx = posx
        self.posy = posy
        self.posz = posz

    def move_x(self, distance):
        self.posx += distance

    def move_y(self, distance):
        self.posy += distance

    def move_z(self, distance):
        self.posz += distance
```

With this as our new base class we can use the `move_` methods from any object that inherits from *InSpace*.

This means that we can use the *Cube* class as it was defined above and do `companion.move_x(10)` to move `10` units forward in space and `companion.move_x(-10)` to move `10` units backwards. Note that in the function call to move backwards we use `-10` for a specific reason.

We could have a method for moving forwards and backwards on each axis but that may get a little messy. Instead we use a more general approach. When we add the distance to a variable we use the += operator which adds `distance` to the current value of the variable on the left and then stores the result in the same place. Basically the final two statements are identical.

```
>>> position = 0
>>> position = position + 10
>>> position += 10
```

Now comes the part that we abuse to make the movement three simple methods instead of six. When you add a negative number (`-10` in our case) to another number it will actually perform a minus operation. By using this we can just hand the move methods positive numbers when we want to move forward on that axis and a negative integer when we want to move backwards. Neat huh!

This Isn't Even my Final Form

It doesn't end here. Depending on you needs and what you language of choice provides you can create powerful base classes and object hierarchies or even interfaces that you can use to make your code easily re-usable and even extendable.

Some languages allow a class to inherit from multiple classes at once. In statically typed languages there is often *Templating* which allows for you to make a generic class that can be used with any object type. There are very few problems that cannot be solved using an *OOP* approach.

It sounds complex but this can be super helpful. However just the basics outlined here is more then enough to get you into the world of *OOP* and open up a lot of possibilities for better code.

1.1.5 Tasty Source

When all is said and done writing code in a clean, readable, and reusable way is pointless unless the code can be saved for later use. In programming a source file (sometimes called a module) is a text file that contains our code. Separating code into smaller modular files that refer to code in one another module makes our code easier to read and edit because we can keep each concept stored in a single file. We do not have to, nor should we, keep all of our source code in the same file. This also makes redistributing or sharing the code very easy.

If you have not noticed, much of the goal in programming is to break down your thoughts into smaller concepts and then make them into even smaller pieces of code. If you have one massive function that does everything your program needs to do then it can often be hard to find and solve problems in this code. However, if you break your code into smaller pieces (wherever it makes sense to do so) then you are left with small, replaceable, and reusable pieces of the puzzle.

Also probably evident, in programming we want to create beautiful code while also being super lazy. This means that we tend to favor clean, small code that when read is clear about what it does. This is made even easier when the source code is not a big jumbled mess.

Pass the Source

Obviously if we split our code into different files we will need a method of telling one file that it uses something in another file. Today this is usually called "importing" (sometimes called include) and is often written at the top of a source file. Importing is used not just to reference your own code in other files but other code that is included in your programming language of choice, or even a library that someone else wrote that you wish to use.

In *Python* we can write the following code which will calculate `2` to the power of `10` using the `math` module that comes with *Python*.

```
>>> import math
>>> math.pow(2, 10)
1024.0
```

By using the `import` keyword we have told the language (*Python* in this case) that it should look for a module named `math` so that we can use the code defined in math.

When you ask to import something the language will go looking in some predefined locations that it thinks the source code is likely to be. Usually it will first look in the current directory that you or your root source file is located in. Then it might search other locations that it thinks its libraries will be installed to. This means that even though *Python* comes with a math module, if I wrote another *Python* file in the same directory that I started the `python` command line interpreter then that file would be imported instead. This is because it looks there before looking in the libraries included by *Python*.

Most languages work in a very similar way. Some languages have Imports (or includes) that work slightly differently or can be made more specific. In *C/C++* when another file is included then everything in that file is brought into the current file as if it where written here. Meaning if there is a function called `pow` in the file we are including and we also have one in the source file after the include then it will override the previous `pow`.

A similar thing can be done in *Python* but it is not recommended for exactly the reason I mentioned above. It pollutes your source code by importing everything into the current *Namespace*.

```
>>> from math import *
>>> pow(2, 10)
1024.0
```

While this is generally considered a bad thing to do in many modern programming circles it does present something nice. We do not have to prefix any `math` usage with its *Namespace*, we just call the `pow` function directly. This may sound like a trade off that needs to be considered but many languages have already beaten you to a beautiful solution, selective imports!.

If all we are going to do is use the `pow` function, and it is not going to cause any confusion or conflicts with other code in our own module, why not just import `pow` and nothing else. Seems logical, lets do that.

```
>>> from math import pow
>>> pow(2, 10)
1024.0
```

There we have it. We are now importing only what we use. There are arguments for both this method and just importing only the `math` module, *Namespace* and all, however they are a little beyond this book. No one here is trying to tell you how anything has to be done. But, I do encourage you to experiment and decide what is best for your self.

As a final note on importing things in cool ways. What if we do only want to import a specific function, or even then whole module, but we already have something that has the same name in our current module that would cause conflicts. A few languages will allow you to rename what you import, *Python* included.

```
>>> from math import pow as mathpow
>>> mathpow(2, 10)
1024.0
>>> import math as realmath
>>> realmath.pow(2, 10)
1024.0
```

Using these two methods we can usually dance around any import conflicts and land on some beautiful code.

Although when importing modules do be careful about "Circular Dependencies". If module A imports module B which imports module A, this can be a real problem and some languages will just fail to handle this at all if you are lucky. If you are unlucky they may get stuck in a loop or not give you any hint as to why it is failing.

Fair Shake of the Source

Some languages do not have an interactive interpreter and must be used by compiling/interpreting source files. In *Python* we can do both.

Source files are plain text files that use a specific file extension (the last few letters of a file name after a dot, ie. `filename.txt` the extension is `.txt`) that can be edited using anything that can write a simple text file although some editors have better support for programming.

We could start an entire war over which editor is the best to use, we covered some of the common choices in the *Introduction* and left it to you to research and decide what you like the best. Experiment away, that's what we are here to do.

Another important thing about source files is that they usually have some kind of entry point. An entry point is usually a function called `main` that is the start of our programs execution.

Python is a little different but every language has its twists. We are going to write a simple program that will just say `Hello World!` to the user. This is a pretty program that is very often used as a first tutorial or introduction to a specific programming language.

Source files for *Python* are denoted by the `.py` file extension. So here is `greeting.py`

```
def greeter():
    print("Hello World!")

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

Now if we run `python greeting.py` from the command line we will get our greeting. Finally we touched on something where *Python* may not be the easiest language to demonstrate with. I will give a little explanation.

In most languages there is just some kind of `main` function or equivalent entry point. Due to the nature of *Python* we need to jump through a few hoops to do the same thing.

We could get the same results if we wrote the file like this.

```
def greeter():
    print("Hello World!")
greeter()
```

Or, for those of you who are a little sharper, you may have noticed we can just make this a one line file and do the same thing still.

```
print("Hello World!")
```

However we do not do this when we are writing a program. As a single file script this works just fine but what if some other source files in our program wanted to use `greeting.py`. Give it a go and see what happens. If we change the `greeting.py` source code to either of the previous two examples and then open a new *Python* interpreter with `python` on the command line in the same directory as the `greeting.py` source file.

If we entered `import greeting` it would print out `Hello World!` which in the second examples case means that just by importing the module the `greeter` function was called. However the first example will only call the function when that source is the file being executed by python directly as we did by calling `python greeting.py` making the `greeting` module have the special `__name__` variable equal `"__main__"`.

In this case it does not really matter if you understand why this happens just that you know that with python that is how it is done.

If you are still having problems figuring out when to use which method just take a moment to experiment. After all experimenting is the best way to learn programming (in my opinion) and is a core concept of this book.

Packages

After all this we now have a greeting module but what if we want to step up the complexity. This is where packages come in. When there are lots of source files it can be easier to sort things into a hierarchy of directories. For example in a game we might want to have a package for everything to do with drawing to the screen and a different package for all the networking code. So what we can do is have a directory structure that looks like this:

```
./
./game.py
./graphics/
./graphics/screen.py
./network/
./network/client.py
```

With this directory structure `graphics` and `network` are packages that contain the `screen` and `client` modules (respectively) and as such we can use these modules by saying `from graphics import screen` or with *Python* and some other languages we can even get something specific from our modules:

```
from graphics.screen import draw
```

It is worth noting that the example file structure given above will not work in *Python* because it wont just accept any directory as a package. In *Python* a directory that should be considered a package must have a file in it called `__init__.py` that can be empty. This file is what is executed when you import just the package, for example `import graphics` would run any code in `__init__.py` before continuing. This is specific to python but the general package concept holds true even for languages that don't support modern "packages". For example in *C/C++* instead of giving the package/module names you give the file path:

```
#include "graphics/screen.h"
```

Using packages can make code maintenance much easier and the entire project easier to understand.

1.1.6 RTFM!

This is where we go from programming something that works to something that is good and useful. In the programming world the difference between the scrawlings of a mad man and the poetry of a genius is documentation. Even if you are the only person who uses the code you are writing and it is not to be distributed publicly, documentation is one of the most powerful tools a programmer has at their disposal.

If you look around programming forums and discussions enough you will find people talking about old code that they found from years ago that has no documentation and they have no idea what it does. At the time they wrote the code they could figure out what it did by looking at it but nowadays it makes little or no sense. Documentation can help us just as much as it can help others.

When starting a new programming project often you will come across a library that seems like it would be useful but then you try and find out how to use it but there is little or no documentation. At this point the only option to figure it out is to read the code and learn it like that. Some people enjoy this but others do not, especially if you are doing a large project that uses many different libraries. If the documentation is lacking then it doesn't matter how brilliant, fast, clean or effective the code is.

Having documentation is a great way to get people interested in how it works and a very handy way of reminding yourself of the direction you are going. Documentation often helps make your code more maintainable and even makes the code itself more readable, as often documentation is included in the source code itself. Writing or enhancing documentation is also a great way to learn how someone elses project works. It is also a good introduction on how to contribute to open source projects.

Documentation generally comes in two forms; comments and docs. Comments are pieces of text that are inside the source code itself and is only viewed by people looking at that code. Docs are sometimes, but not always, also included in the source code and are turned into readable documentation for the public. This can be done in multiple ways, nowadays the most common way is to generate a website out of the docs and publicly publish that website... much the same way this book is written. Actually that's exactly how this book is written.

This Title Explains the Content of This Sub-Chapter

Not all programming languages agree on a common method of denoting a comment but usually it is just a special character and then the rest of the line is ignored by the programming language. This means anything after that special character can only be seen by viewing the source code, sounds like the perfect place to explain things.

Programmers are lazy folk, we do not want to remember everything about our code and usually we can't. So what we often do is use a comment to describe what some code does, especially when it isn't readily apparent what the code really does.

Lets write some comments for our old 3d space example.

```
class InSpace(object):

    def __init__(self, posx=0, posy=0, posz=0):
        self.posx = posx
        self.posy = posy
        self.posz = posz

    def move_x(self, distance):
        self.posx += distance #Add distance to position x

    def move_y(self, distance):
        self.posy += distance #Add distance to position y

    def move_z(self, distance):
        self.posz += distance #Add distance to position z

class Cube(InSpace):

    def __init__(self, size, posx=0, posy=0, posz=0):
        #Call the parent constructor.
        super(Cube, self).__init__(posx, posy, posz)
        self.size = size
```

Our code functions the same but we have now added some comments to explain what is going on in some of the less obvious areas. In *Python* comments are started with the pound (`#`) character. Some languages us a double forward slash (`//`) for single line comments and forward slash asterisk (`/*`) to denote the start of a multi-line comment and the opposite to end that comment (`*/`). While pound only does single line comments in *Python* that does not mean that we are missing out. There is wisdom in the way *Python* does things, instead of providing multi-line comments it provides what it calls "doc strings" which are actually multi-line comments however they are also documentation that can be accessed by the user of your code and they look like strings. That's probably why they are called "doc strings".

Unlike Humans, Chuck Norris Doesn't Need Documentation

When learning how to use a piece of software one of the most useful things that it can provide is clear, up to date, documentation. Lets go straight to writing some documentation.

In *Python* we can use a "doc string" to document a piece of code like a class, a function, or a method.

Lets add some documentation to our 3d space code.

```
class InSpace(object):
    """
    Describes an object in a 3d environment.
    """
    def __init__(self, posx=0, posy=0, posz=0):
        self.posx = posx
        self.posy = posy
        self.posz = posz

    def move_x(self, distance):
        """Move on the X axis."""
        self.posx += distance #Add distance to position x

    def move_y(self, distance):
        """Move on the Y axis."""
        self.posy += distance #Add distance to position y

    def move_z(self, distance):
        """Move on the Z axis."""
        self.posz += distance #Add distance to position z

class Cube(InSpace):
    """
    A Cube in 3d space.

    Stores a single size variable for the size of all edges.
    """
    def __init__(self, size, posx=0, posy=0, posz=0):
        #Call the parent constructor.
        super(Cube, self).__init__(posx, posy, posz)
        self.size = size
```

Python has a handy `help` function that can output doc strings for anything that is given to it. This is the basis of documentation in *Python* and can be used in more complex ways in the future. For example, tools can be used that get all of the doc strings in your code and turn them into a website, or file, that can be shared with the world.

For now give this example a go. Put the new documented 3d space classes into a file and try using the `help` function to view the documentation.

1.2 Extras:

1.2.1 Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

You too can contribute to this book. As a matter of fact almost the entire proccess is covered in **Code for Thought** itself.

The whole book is a relatively simple to use github repository at https://github.com/Nekroze/codeforthought where it can easily be forked, modified and "pull requested".

If you see a problem with this book; I said something ineloquently (probably many things), Something was hard to understand or something just doesn't work properly (it can happen! dont look at me like that).

You can contribute in many ways:

Types of Contributions

Report Bugs

Is something not working the way the book says it does?

Report bugs at https://github.com/Nekroze/codeforthought/issues.

If you are reporting a bug, please include:

- The [bug] tag in the issue.
- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with "bug" is open to whoever wants to implement it.

Implement Features

Look through the GitHub issues for features. Anything tagged with "feature" is open to whoever wants to implement it.

Write Documentation

We can always explain something badly or word something poorly, this paragraph is no exception. Anyone is welcome to help make **Code for Thought** more readable and understanable for everyone.

Submit Feedback

The best way to send feedback is to file an issue at https://github.com/Nekroze/codeforthought/issues.

If you are proposing a feature:

- Include the [feature] tag in the issue.
- Explain in detail how it would work and why this it is better for everyone.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

Get Started!

Ready to contribute? Here's how to set up *codeforthought* for local development.

1. Fork the *codeforthought* repo on GitHub.
2. Clone your fork locally:

```
$ git clone git@github.com:your_name_here/codeforthought.git
```

3. Create a branch for local development:

```
$ git checkout -b name-of-your-bugfix-or-feature
```

   Now you can make your changes locally.

4. When you're done making changes, check that your changes pass all the tests. You can use either the following command to do all tests at once:

```
$ make test
```

Or seperately:

(a) Test web links:

```
$ make linkcheck
```

(b) Test doctests and code examples:

```
$ make doctest
```

(c) Test html building:

```
$ make html
```

5. Commit your changes and push your branch to GitHub:

```
$ git add --all .
$ git commit -m "Your detailed description of your changes."
$ git push origin name-of-your-bugfix-or-feature
```

6. Submit a pull request through the GitHub website.

1.2.2 Glossary

POSIX an acronym for "Portable Operating System Interface", is a family of standards specified by the IEEE for maintaining compatibility between operating systems. POSIX defines the application programming interface (API), along with command line shells and utility interfaces, for software compatibility with variants of Unix and other operating systems.

Python A widely used general-purpose, high-level programming language. Its design philosophy emphasizes code readability, and its syntax allows programmers to express concepts in fewer lines of code than would be possible many other common languages. The language provides constructs intended to enable clear programs on both a small and large scale

For more information head over to http://www.python.org/ and once installed python can be run from the command line as such:

```
$ python
```

Or for executing a python file called, for example, `mycode.py`:

```
$ python mycode.py
```

Pip A tool for installing and managing *Python* packages. Simply allows you to install and manage any *Python* packages that are available at https://pypi.python.org/ using the following syntax on the command line:

```
$ pip install six
```

The above command will automatically download and install the package called `six`. On some systems you may need to insert the `sudo` command to the begging of the pip command in order to have permissions to install the package.

Interpreter A computer program that executes, i.e. performs, instructions written in a programming language. An interpreter generally uses one of the following strategies for program execution: 1. parse the source code and perform its behavior directly 2. translate source code into some efficient intermediate representation and immediately execute this 3. explicitly execute stored precompiled code made by a compiler which is part of the interpreter system

While generally these are used on text files containing source code many interpreters feature a *REPL*.

REPL A read–eval–print loop (*REPL*) is a simple, interactive computer programming environment. The user enters one or more expressions (rather than an entire compilation unit), which are then evaluated, and the results displayed. These provide a simple and easy way to learn a language and experiment with *Snippets*.

Snippets A programming term for a small region of re-usable source code, machine code, or text. Ordinarily, these are formally-defined operative units to incorporate into larger programming modules. *Snippets* are often used to clarify the meaning of an otherwise "cluttered" function, or to minimize the use of repeated code that is common to other functions. The *Snippets* themselves may be either literal text, or written in a simple template language to allow substitutions, such as variable names. *Snippets* are a small-scale form of copy and paste programming.

OOP Object-oriented programming is a programming paradigm that represents concepts as "objects" that have data fields (attributes that describe the object) and associated procedures known as methods. Objects, which are usually instances of classes, are used to interact with one another to design applications and computer programs.

Templating A programming feature (present in some statically typed languages) that allows generic code to be written that is designed to work with many data types. When a template is instantiated with a type then that type will take the role of a generic variable within the template code as if it where written to use that type. In languages that draw a distinction between the types, for example, of an array of strings versus an array of numbers, *Templating* can be used to write one function that can act on both

Namespace In computer programming, namespaces are typically employed for the purpose of grouping symbols and identifiers around a particular functionality.

C/C++ In computing, C is a general-purpose programming language initially developed by Dennis Ritchie between 1969 and 1973 at AT&T Bell Labs. C++, developed by Bjarne Stroustrup starting in 1979 at Bell Labs, was originally named C with Classes, adding object oriented features, such as classes, and other enhancements to the C programming language.

Index