Programming In General

Authors:
Brett Langdon & Alexander Ambrose

July 12, 2012

Contents

1 Introduction **7**

1.1 Who Is This Resource For . . . 8

1.2 Code Examples . . . 9

2 Getting Started **11**

2.1 Choosing A Programming Language . . . 12

2.1.1 Paradigm . . . 12

2.1.2 Syntax . . . 12

2.1.3 Platform . . . 12

2.1.4 Coolness . . . 12

2.1.5 Conclusion . . . 13

2.2 How To Read This Resource . . . 14

2.2.1 Keywords . . . 14

2.3 Sudo Language . . . 15

2.3.1 Example 1 . . . 15

2.3.2 Example 2 . . . 16

3 Functional Programming **21**

3.1 Variables . . . 22

3.1.1 Declaration . . . 22

3.1.2 Data Types . . . 22

3.1.3 Operators . . . 22

3.1.4 Increment/Decrement Operators . . . 24

3.1.5 Conclusion . . . 25

3.2 Control Statements . . . 26

3.2.1 If Statements . . . 26
3.2.2 If-Else Statements . . . 26
3.2.3 Else If Statements . . . 27
3.2.4 Switch Statements . . . 27
3.2.5 For Loops . . . 29
3.2.6 While Loops . . . 30
3.2.7 Do-While Loops . . . 31
3.2.8 Break Statements . . . 31
3.2.9 Continue Statements . . . 32
3.2.10 Conclusion . . . 32
3.3 Functions . . . 34
3.3.1 Declaration . . . 34
3.3.2 Scope . . . 34
3.3.3 Parameters . . . 35
3.3.4 Returns . . . 36
3.3.5 Recursion . . . 37

4 Object Oriented Programming **39**
4.1 Classes and Objects . . . 40
4.1.1 Classes . . . 40
4.1.2 Objects . . . 40
4.1.3 Properties . . . 41
4.1.4 Methods . . . 41
4.1.5 Special Methods . . . 41
4.2 Inheritence . . . 42
4.3 Polymorphism . . . 43

5 Design Patterns **45**
5.1 Singleton . . . 46
5.2 Factory . . . 47
5.3 Observer . . . 48
5.4 State . . . 49

6 Data Structures **51**

6.1 Big O Notation 52

6.2 Linked Lists 53

6.3 Doubly Linked Lists 54

6.4 Stacks 55

6.5 Queues 56

6.6 Hash Maps 57

6.7 Binary Trees 58

6.8 B-Trees 59

6.9 B+ Trees 60

7 Algorithms **61**

Chapter 1

Introduction

Programming In General is meant to be used as a resource to learn the concepts of computer programming that they can be applied to any language or platform.

This resource is and always will be provided for free.

This resource is currently a work in progress so please bear with me if any particular section or chapter is incomplete or if a section contains less than correct information. If you have any comments, questions, suggestions or corrections please feel free to contact me by e-mail at: *brett@blangdon.com*

Thank you and enjoy.

1.1 Who Is This Resource For

This resource is intended for everyone.

This resource is meant to be useful to programmers of all levels, those who have never programmed before, those who are just getting started and even those who have been programming for years. Since this resource will always be growing and changing as the industry changes the information should always be in some way applicable for all developers.

Those who are familiar with programming are most likely going to be able to skip the chapter *Getting Started* and in some cases might not read the resource in sequential order, but rather skip around picking and choosing which sections are applicable to them.

1.2 Code Examples

All code examples in this resource use a sudo language that is not meant to be run or compiled directly. I have chosen to use this approach so that the concepts can be extracted and implemented in any language on any platform. By focusing on the concepts at hand rather than specific implementations I can focus on trying to present the material in a clear and easy to understand manner.

I will cover how to use the sudo language and how to translate it to a useable programming language in the chapter *Getting Started* in the section titled *Sudo Language*.

Chapter 2

Getting Started

This chapter will cover how to get started with programming, how to choose which language or platform to start with and how to go about using this resource.

2.1 Choosing A Programming Language

It is wonderful that you have decided to undertake the hobby of computer programming, but which language should you choose: Python, PHP, Java, C#, C/C++, VB, Ruby, Scala, Groovy, Javascript, or one of the thousands of others languages available to programmers. There are many factors to consider when choosing a programming language especially when getting into programming for the first time, some of which are the languages paradigm, syntax, platform and even the coolness factor of the language.

2.1.1 Paradigm

A languages paradigm refers to the languages overall style of development. For example the three mainly adobted paradigms are functional, object-oriented and multi-paradigm. A functional languages are based around around the concept of completing tasks using Mathematical functionals; C is an example of a functional language because rather than using classes or objects to complete it's tasks it used constructed functions. Object-oriented languages on the other hand use classes and objects to complete programming tasks; Java is an example of an object-oriented programming language because regardless of the type of program you develop you must use classes and objects. Multi-paradigm languages are usually a mix of more than one paradigm. For example Python is a multi-paradigm language because you can choose whether or not to use classes and objects when programming.

There are many more types of paradigms that languages can follow but most languages you will come across today are either strictly functional, strictly object-oriented or they offer the best of both worlds by supporting both.

2.1.2 Syntax

The syntax of a language is very important when choosing a language. This is mainly going to be a personal preference. Personally, I like C style syntax languages like C, C++,Java, PHP, Javascript, etc. Other people might prefer other languages because their use of other syntax styles, like the almost sudo code style of Python. Your personal preference will come with time as you move from one language to another and develop your own personal styles and preferences.

2.1.3 Platform

This is a very important factor when choosing which programming language to use. What platforms do you have available to use? Do you only have a Windows computer at your disposal? That might remove some of your options as some languages might not support developing on your specific OS such as Windows or Mac OS.

When starting out, try and choose a language that works on a platform that is readily available to you. Do not try and move to a new or different operating system in order to learn programming. Keep things simple.

2.1.4 Coolness

What seems cool to you? What is everyone else raving about right now? What is new and different?

Some may think that coolness is a silly factor to introduce when trying to pick a programming language to use, but I can honestly say that it has effected my choices in the past. When I was learning programming in college we were being taught Java, but I taught myself PHP on the side mainly because my friend was using it and I wanted to impress him. This is not a bad thing. Let others help influence your decisions when programming. That is how you will grow and learn things you might not of experienced without the influence.

2.1.5 Conclusion

So, we have taken a quick look at how to go about picking a programming language. Some of you might say, "That was not really helpful. You did not tell me which language to use.", and your right, I didn't.It should not be my choice which language you learn first. I want to try and keep some bias out so that this resource is as lanuage agnostic as possible.

Advice:
If after doing some research you are still unsure which lanauge you want to use, especially for going through this resource try out Python. Python is available for every platform, or at least all of the ones I can think of. It is interpreted so you will not have to wait for the program to compile. Lastly, its syntax is one of the closest to the sudo langauge which this resource uses extensively.

2.2 How To Read This Resource

This resource is going to be laid out a little weird, more so for those who have already had some programming background.

For those who are new to programming I strongly suggest reading through Chapters 3 and 4 thoroughly before continuing with the rest of the resource. Those two chapters contain all of the core concepts needed in order to understand some of the higher level concepts presented with Data Structures and Algorithms. Once you have completed chapters 3 and 4 please feel free to jump around a little between sections presented in chapters 5 and 6 as some data structures or algorithms might interest you more than others.

2.2.1 Keywords

Throughout this resource some words will be highlighted, colored differently or emphasized in order to stand out. These words will generally be referring to code examples presented in the chapters:

Type	Example
Variable	***variableName***
Functions	***functionName***
Class Properties	***propertyName***
Values	**"Sample String Value"**
Program Output	**Console String Output**

Example:
We assign the value of **"Sample String"** to the variable ***sample*** then pass in ***sample*** as a parameter to the function ***printValue*** which will print: **The String Is: Sample String**.

2.3 Sudo Language

For the code examples presented in this resource I am going to be using a sudo language. The concept behind a sudo language is to be able to present programming concepts in a language agnostic form so that the concepts can be translated to your language of choice.

So it is great that you have chosen lanauge X to use throughout this resource, but how is the sudo lanauge going to help you out? Well, lets look at two examples and I will show their implementation in a few different languages. Hopefully this will help you be able to understand how the language should be tanslates (especially if your language of choice is one that I use).

2.3.1 Example 1

Listing 2.1: Example 1 - Sudo Code

```
name = "Brett"
if name == "Brett"
    print "Name Is Brett"
else
    print "Name Is Not Brett"
```

For this example lets break it down line by line to make sure we know exactly what is going on.

1. Store the value **"Brett"** into the variable ***name***
2. Check if the variable ***name*** is equal to the value **"Brett"**
    3. Print **"Name Is Brett"** to the console
4. Otherwise
    5. Print **"Name Is Not Brett"** to the console

As far as programming goes this is a fairly simple process but lets try and translate this example to a few different languages to see how it is done.

Listing 2.2: Example 1 - PHP

```
<?php
$name = 'Brett';
if( $name === 'Brett' ){
    echo 'Name Is Brett';
} else{
    echo 'Name Is Not Brett';
}
```

Listing 2.3: Example 1 - C

```
int main{
    char* name = "Brett";
    if( name == "Brett" ){
        printf("Name Is Brett");
    } else{
        printf("Name Is Not Brett");
    }
    return 0;
}
```

Listing 2.4: Example 1 - Python

```
name = "Brett"
if name is "Brett":
    print "Name Is Brett"
else:
    print "Name Is Not Brett"
```

Listing 2.5: Example 1 - Node.JS

```
var name = "Brett";
if( name == "Brett" ){
    console.log("Name Is Brett");
} else{
    console.log("Name Is Not Brett");
}
```

Listing 2.6: Example 1 - Java

```
class Example1{
    public static void main( String[] args ){
        String name = "Brett";
        if( name.equals("Brett") ){
            System.out.println("Name Is Brett");
        } else{
            System.out.println("Name Is Not Brett");
        }
    }
}
```

Notice that all of the actual examples end up looking the same? That is the point of using the sudo language, so that we can discuss the core concepts of the lesson at hand and then those concepts can be directly applied to any language of choice.

Also, notice the Python implementation, it is almost line for line, word for word identical to the sudo language example.

2.3.2 Example 2

Since we have seen a fairly simple example above, lets take a look at a more complicated example. Do not be afraid if it does not make too much sense right now, but try and notice the similarities between the sudo language and the actual code examples.

Listing 2.7: Example 2 - Sudo Code

```
class Person
    private name

    function getName()
        return this.name

    function setName( newName )
        this.name = newName


p = new Person()
p.setName("Brett")
```

```
13
14
15 if p.getName() == "Brett"
16     print "Name Is Brett"
17 else
18     print "Name Is Not Brett"
```

Just like the last one, lets break down this example line by line to fine out whats going on.

1. Create a new class called ***Person***
   2. Create a private property ***name***
   4. Create a method called ***getName*** that requires no parameters
      5. When the function is called return the class property ***name***
   7. Create a method called ***setName*** that takes a single parameter ***newName***
      8. When called set the class property ***name*** equal to the parameter ***newName***
11. Create a new ***Person*** object and store it in the variable ***p***
12. Call ***p***'s ***setName*** method passing in the value **"Brett"**
14. Call ***p***'s ***getName*** method and check if the returned value is equal to **"Brett"**
    15. Print **"Name Is Brett"** to the console
16. Otherwsie
    17. Print **"Name Is Not Brett"** to the console

Do not worry if this example does not make sense to you, you will be able to understand it well before the end of this resource.

Just like with Example 1, here are some translations of the example.

Listing 2.8: Example 2 - PHP

```
1  <?php
2  class Person{
3      private $name;
4
5      public function getName(){
6          return $this->name;
7      }
8
9      public function setname( $newName ){
10         $this->name = $newName;
11     }
12 }
13
14 $p = new Person();
15 $p->setName('Brett');
16
17 if( $p->getName() === 'Brett' ){
18     echo 'Name Is Brett';
19 } else{
20     echo 'Name Is Not Brett';
21 }
```

Listing 2.9: Eample 2 - Java

```java
class Person{
    private String name;

    public String getName(){
        return this.name;
    }

    public void setName( String newName ){
        this.name = newName;
    }

    public static void main(String[] args){
        Person p = new Person();
        p.setName("Brett");

        if( p.getName() == "Brett" ){
            System.out.println("Name Is Brett");
        } else{
            System.out.println("Name Is Not Brett");
        }
    }

}
```

Listing 2.10: Example 2 - Node.JS

```javascript
var Person = function(){}
Person.prototype.getName = function(){
    return this.name;
}
Person.prototype.setName = function( newName ){
    this.name = newName;
}

var p = new Person();
p.setName("Brett");

if( p.getName() == "Brett" ){
    console.log("Name Is Brett");
} else{
    console.log("Name Is Not Brett");
}
```

Listing 2.11: Example 2 - Python

```python
class Person:
    def getname( self ):
        return self.name
    def setName( self, newName ):
        self.name = newName

p = Person()
p.setName("Brett");

if p.getName() is "Brett":
    print "Name Is Brett"
```

```
12 else:
13     print "Name Is Not Brett"
```

This example does a better job of showing how each language can tackle the concepts in a different manner but the core concepts laid out by the sudo language can still be extrapolated and translated to each individual programming language. As long as the language supports the concepts. As you may notice that I left out the implementation of C in this example. It is because C does not support the use of classes and objects, yes there are ways of completing this example in C using structs but that is something that you should learn on your own.

So now you have seen a few examples, hopefully enough to give you an idea of how the examples in this resource will be presented.

Chapter 3

Functional Programming

In this chapter we are going to cover the basic concepts of functional programming. This could mean a few things to different people, but in regard to this resource we are going to refer to functional programming as programming without the use of classes and objects. Yes, some people are cringing a little in their seats as that is not the best definition of functional programming but to try and keep things simple and organized that is what we are going to refer to it as.

I am going to use this chapter to introduce topics other than just functions. Topics including control statements, loops and some input output (io).

Functional Programming:
Functional programming is a programming paradigm that treats computation as the evaluation of mathematical functions and avoids state and mutable data.
Wikipedia (2012)

3.1 Variables

Variables act as aliases to the values that we want them to represent and they allow us to access and manipulate the values that we assign to them. For example we could use the variable ***name*** to represent the value **"Brett Langdon"**. We do this with programming so that we can then access the value **"Brett Langdon"** with a shorter representation, ***name***.

3.1.1 Declaration

To start with variables we need to declare their existence. By declaring a variable we are saying to the program, here is our alias and here is the value that we want it to represent.

Listing 3.1: Variable Declaration

```
1 a = 10
2 print a
```

In the above example we are saying that we want to store the integer value **10** into the variable ***a***. We can then use the variable ***a*** to access the value **10**. This program will output **10** rather than **a**.

When we declare variables we are telling the programming language to allocate some space in your computers memory in order to store the value that you need it to. The amount of space needed to store each variable depends based on your specific language being used and which data type is being used to store the value.

3.1.2 Data Types

Programming languages support different types of data types or different types of values that they can represent in variables. Some programming languages use multiple different types of values but most of them support the basic types: string, integer (multiple kinds) and boolean (true or false).

Listing 3.2: Data Types

```
1 string = "Brett"
2 integer = 10
3 boolean = false
```

Please keep in mind that each programming language supports different data types and you should research those types to better understand variables in that language. As well some programming languages that are strickly typed which requires us to define the data type of the variable on declaration (unlike our sudo language).

3.1.3 Operators

Operators are symbols we can use to manipulate variables. There are a few different types of operators, Arithmetic, Comparison and Logical operators. A quick list of operators include:

1. Addition +
2. Subtraction -
3. Multiplication *
4. Division /
5. Equals =

6. Equal To ==
7. Not Equal To !=
8. Less Than <
9. Greater Than >
10. Logical AND &&

And there are more!

Arithmetic operators include the mathematical symbols for addition (+), subtractions (-), division(/), multiplication(*) and equals(=). These operators are used to directly manipulate and change variables. For example we can add two variables together and then store that sum into a third variable.

Listing 3.3: Addition Operator

```
1 a = 10
2 b = 5
3 c = a + b
4
5 print c
```

We store the value of **10** into the variable ***a*** and the value **5** into the variable ***b***. Lastly, we store the sum of the variables ***a*** and ***b*** into the variable ***c***. The output of this program will be **15**.

Comparison operators are used to compare variables against other variables or values. The output of a comparison operator will be a boolean value (**true** or **false**), which means they can be used in a conditional statement or even stored into a variable. Some of the popular comparison operators are the equals to (==), not equals to (!=), less than (<), less than equal to (<=), greater than (>) and greater than equal to (>=).

Listing 3.4: Comparison Operators

```
1 name = ''Brett''
2 a = 10
3 b = 5
4 c = a + b
5
6 print name == ''Brett''
7
8 print a > 10
9
10 print b >= 5
11
12 isNotSix = c != 6
13 print isNotSix
```

In this example we are showing a few of the comparison operators. The output of this code will be:
False
True
True.

Logical operators are used to evaluate multiple variables, comparisons or conditionals. The two most popularly used Logical operators are the Logical AND (&&) and the Logical OR (||). An example of using Logical operators would be to construct a conditional statement based on multiple inputs.

Listing 3.5: Logical AND Example

```
1 name = ''Brett''
```

```
2 age = 22
3
4 print name == ``Brett'' && age == 22
```

The output of this code will be **True**. With the Logical AND operator we are saying that our conditional statement is only true if both ***name*** is equal to **"Brett"** AND ***age*** is equal to **22**. If either side of the operator (&&) is false then the entire conditional is false.

The other Logical operator is the OR operator and tells our conditional that one side or the other must be true in order for the entire conditional to be true. This means that either the left or right side can be false as long as the other side is true.

Listing 3.6: Logical OR Example

```
1 name = ``Brett''
2 age = 23
3
4 print name == ``Brett'' || age == 22
```

This program will output **True** because eventhough ***age*** is not **22**, ***name*** is **"Brett"**. If we switched around the conditionals, put ***age == 22*** on the left hand side of ——, then the program will have the same output; this is becuase with the Logical OR only one side needs to be true.

One thing to remember when dealing with either of the Logical operators is that programming languages will evaluate the conditionals from left to right. For example, in the above "Logical AND" example, if ***name*** were not equal to **"Brett"** then the right hand side ***age == 22*** would never even get to evaluate. For the second example using the Logical OR, if ***name == "Brett"*** were true (which in this case it was) then the right hand side would never get to evaluate. This is becuase with Logical OR's only one side needs to be true, if the left hand side is true then there is no need to even try the right hand side.

3.1.4 Increment/Decrement Operators

Most programming languages give us a few operators to use to increment or decrement the value of number variables. There exists four operators for this purpose, increment by (1) (++), decrement by (1) (−−), increment by (+=) and decrement by (−=).

Listing 3.7: Increment Operators

```
1 num = 5
2 ++num
3 num += 10
4
5 print num
```

The output of this code will be **16**. The first line sets ***num*** to **5**. The second then increments ***num*** to **6**. The third line then increments ***num*** by **10**. Lastly we print the value of ***num*** which at this point is **16**.

Listing 3.8: Decrement Operators

```
1 num = 16
2 num--
3 num -= 10
4
5 print num
```

This program does the complete opposite of the one above. It starts with ***num*** at **16** and then decrements it to **15** and finally to **5**. There is one main difference in this program; the decrement operator (−−) is placed to the right of

num where with the increment operator (++) above is appears to the left of ***num***. These operators can be placed on either side of the variables you wish to effect.

When the operator is in front of the variable it is called pre-increment or pre-decrement and the latter is post-increment and post-decrement. This pre/post choice does not apply to the increment by (+=) and decrement by (−=) operators; they must be placed to the right of the variables you wish to effect. It is worth while to investigate how your language of choice handles pre/post operators as you may not get the desired result. For the remainder of this resource only the pre operators will be used.

3.1.5 Conclusion

In this section we have covered the basics of variables, how they are declared, used and evaluated. Variables will be the building blocks from which we will continue through this book. It is very important that you are able to use variables in your prefered language as their use is exhausted in every following section and chapter.

The next section will cover the use of Control Statements.

3.2 Control Statements

Control statements are almost exactly as they sound, statements that control our programs. Well, they control the flow of our code. With control statements we can change the course of our programs based on various conditions.

3.2.1 If Statements

If statement allow us to execute a given block of code based on a given condition. There are three main parts to an if statement **if**, the **conditional** and a **code block**.

Listing 3.9: If Statement

```
1 name = ''brett''
2
3 if( name == ''brett'' )
4     print ''Name is brett''
```

In this simple example the code block **print "Name is brett"** will only execute if the conditional **name == "brett"** is true. So the output of this code will be **Name is brett**.

Listing 3.10: False If Statement

```
1 name = ''brett''
2
3 if( name == ''john'' )
4     print ''Name is john''
```

In this example there will be no output, this is because the conditional **name == "john"** equates to false so the code block **print "Name is john"** will never get executed.

3.2.2 If-Else Statements

If statements are great and help us execute portions of our code based on the values of other variables, including based on input from users. But what if the condition of the if statement equates to false? With if statements we can append an else statement and a block of code to the end of an if statement that will get called if the if statement is false.

Listing 3.11: If-Else Statement

```
1 name = ''brett''
2
3 if name == ''john'':
4     print ''Name is john''
5 else:
6     print ''Name is not john''
```

The output of this program will be **Name is not john**. When the program hits the if statement it evaluates the conditional ***name == "john"*** which evaluates to **false**. Normally the program will continue on its way but since we provided an else statement that gets executed instead. An If-Else statement allows us to program "if this then do this, otherwise do this."

3.2.3 Else If Statements

Ok... wait, we just did If-Else statements not we are doing Else if statements? Yes, but they are different I swear! An If-Else statement allows us to execute code regardless of whether a conditional is true or false but with an else if statement we can provide multiple conditionals to an if statements.

Listing 3.12: Else If Statement

```
1 name = ''brett''
2
3 if name == ''john'':
4     print ''Name is john''
5 else if name == ''brett''
6     print ''Name is brett''
```

See? I told you it was different. So the output of this program is **Name is brett** and this is because when the program gets to the if statement and evaluates it as false, it then continues down the list of conditionals. This works similar to how the else statement before worked, but this time we are giving the if statement multiple conditionals to check. We can expand this example by adding more else if statements.

Listing 3.13: Else If Statement 2

```
1 name = ''brett''
2
3 if name == ''john'':
4     print ''Name is john''
5 else if name == ''brett''
6     print ''Name is brett''
7 else if name == ''barbara'':
8     print ''Name is barbara''
```

Just like the first example this program will output **Name is brett**. This is because when the program gets to *name == "john"* it evaluates to false causing the program to skip to the next conditional *name =="brett"*, which then evaluates to true causing the code block given to execute. The last conditional *name == "barbara"* will then be skipped and the program will continue past the if statement.

Now... what is we add an else statement to the end of this? With an if statement we could append an else statement to the end telling it what to do if the conditional failed. With an else if statement we can also append an else statement to the end telling it what to do if all of the conditionals fail.

Listing 3.14: Else If Else Statement

```
1 name = ''brett''
2
3 if name == ''john'':
4     print ''Name is john''
5 else if name == ''barbara'':
6     print ''Name is barbara''
7 else:
8    print ''Well, I'm not sure what your name is''
```

This program will output **Well, I'm not sure what your name is** because both conditionals, *name == "john"* and *name == "barbara"*, evaluate to false causing the if statement to continue on its merry way.

3.2.4 Switch Statements

A Switch statement is similar to a grouping of If, Else If and Else statements but where the conditional is always a direct comparison to a value. Switch statements are useful when you have a set number of values to compare a

variable against. For example, the following If statements are a perfect candidate for a switch statement.

Listing 3.15: Switch Statement Candidate

```
name = ''brett''

if name == ''john'':
    print ''name is john''
else if name == ''barbara'':
    print ''name is barbara''
else if name == ''eugene'':
    print ''name is eugene''
else if name == ''brett'':
    print ''name is brett''
else:
    print ''not sure what your name is''
```

With a Switch statement it can be rewritten as.

Listing 3.16: Switch Statement Example

```
name = ''brett''

switch name:
    case ''john'':
        print ''name is john''
        break
    case ''barbara'':
        print ''name is barbara''
        break
    case ''eugene'':
        print ''name is eugene''
        break
    case ''brett'':
        print ''name is brett''
        break
    default:
        print ''not sure what your name is''
        break
```

Both of these programs work in a similar manner, take a variable and do a direct comparison to a set of values until a match is made or else use a default action. As well they will both output the same **name is brett**. Think of a Switch statement as a set of If, Else If, Else statements where the conditionals are always a single ==.

A switch statement introduces a few new keywords, the switch followed by the variable name we wish to compare against. Then we can have as many case statements following, each with the value that we wish to compare our variable against. The only other weird part is that we are also introducing the break statement, which is required to terminate each case statement code block. What the break statement says to do is "break" away from the entire switch statement. As an excersise, try removing all of the break statements from the above example and run it again, what changed?

We have mainly been comparing string variables against string values but you can also use Switch statements to compare numbers as well.

Listing 3.17: Switch Statement Numbers Example

```
age = 22

switch age:
```

```
4       case 20:
5           print ``not old enough to drink''
6           break
7       case 21:
8           print ``congratulations, do not over do it''
9           break
10      case 22:
11          print ``you've been doing this awhile''
12          break
```

As you can see, we can also compare our number variable against number values. In this example we also have left out the default case, this case is optional, similar to the else statement.

3.2.5 For Loops

We have seen some statements that will help the direction of our code, but what about repeating code? Lets say that we need to manually determine what the square of a number is using multiplication (rather than the exponential operator ^). This can be expressed fairly easily.

Listing 3.18: Square Without Loop

```
1 num = 5
2
3 newNum = num * num
4
5 print newNum
```

Fairly easy enough and we know the output of this code will be **25**. Now lets say we need to do this same thing but to the power of 5.

Listing 3.19: Power of 5 Without Loop

```
1 num = 5
2
3 newNum = num * num * num * num * num
4
5 print newNum
```

Ok, now this is starting to get obnoxious. Now what if we need it to the power of 100... I'm not programming that. This is where loops come in, in particular the For loop. The For loop is the perfect candidate when you need an action performed a set number of times.

Listing 3.20: For Loop

```
1 num = 5
2
3 newNum = num
4
5 for i = 0; i < 99; ++i:
6     newNum = newNum * num
7
8 print newNum
```

So the output of this code should be, **7.888609052210123e+69**. For loops can be odd to look at the first time so lets break it down part by part. A For loop is broken into 4 parts, the Initializer, the Condition, the Update and the Code Block. The Initializer, Condition and Update are all separated by semicolons.

The Initializer, *i = 0*, initializes some value that is going to be used throughout the loop, usually a counter; in this case *i*. Why is *i* set to **0**? In computer programming we use a zero based counting system mainly out of tradition, but because of implementation choices made by language developers to base counting off of memory addressing offsets. So... we just do, get in the habit now of counting from 0, everyone else does it.

The Condition gets checked for every iteration of the loop and if the condition evaluates to true then the loop continues and once again executes the Code Block. In this example *i < 99* is out Condition. Why 99, I thought we were going to 100? True, we are going to 100, but remember that we initially set *newNum* to *num* which is the same as *num* to the first power. Ok, so why do we use *i < 99*? won't that take us to 98? Remember, we are using zero based counting, so 0 counts as "1".

The Update is a statement that get executed after the Code Block and is used to update any variables we need before continuing. In this case we are using the "pre-increment" operator to increase the value of *i*, our counter, by 1. We could have also used *i += 1*, but *++i* is just so elegant.

Lastly, the Code Block get executed on every iteration of the loop. The general flow of a For loop is, Initialize any variables, check the Condition if it is true then execute the Code Block, execute the Update statement, re-check the Condition, if it is true then execute the Code Block again or else leave the For loop and continue with the program.

For loops are great, they save not only time, but they save a lot of typing and a lot of code duplication. Lets say in out example above we wanted to raise 5 to the power of 50 rather than 100? It is simple enough to change 99 to 49 and call our job done, but if we had written out *num * num* 50 times, then it would be a pain to try and update this code.

One thing to look out for with For loops, or any loops, are infinite loops, meaning a loop whose Condition will always evaluate to true. Take the example above, if we were to change the Condition to *i >= 0* then we would have an infinite loop because *i* starts at 0 and is always increasing. The same would be true if we changed the Condition to *true* or *1==1* or any other conditional statement that will always be true.

3.2.6 While Loops

So we have just seen how For loops are used to loop based on a condition and a counter for a set interval, but what if we wanted to just loop forever until a condition was met? Well, we have While loops! While loops are great for things such as iterating over a file or a result set from a database query or when the duration of loop is unknown.

While loops contain two parts, the conditional and the code block. The conditional is checked for each iteration of the loop, if it evaluates to **true** then the code block is executed. The main difference between a While loop and a For loop is that a For loop is usually designed so that it runs at least once or for a set number of times, but a While loop has the potential to run the code block 0 times. Let us jump into an example.

Listing 3.21: While Loop

```
1 num = 0
2
3 while num < 25:
4     print ''Loop''
5     num += 5
```

This program will print **Loop** 5 times. When the program gets to the While loop it first evaluates the conditional to see if the code block should be run once. In this case *num* is less than **25** so the code block is executed, which prints **Loop** and then increments *num* by **5**. This loop continues until *num* is incremented to **25**.

This example is fairly simple and even in some resembles how a For loop works. I contains an initialization of a counter variable, *num* to **0**. The conditional ensures that *num* is below **25**. Finally the update is when we increment *num* by **5**. So let us take a look at an example that does not use numbers to see how the While loop can be useful.

Listing 3.22: While Loop Over File

```
1 file = OpenFile(``example.txt'')
2 line = ReadLineFromFile(file)
3
4 while line != EndOfFile:
5     print line
6     line = ReadLineFromFile(file)
7
8 CloseFile(file)
```

As you can guess from this program, a file **example.txt** is opened and the first line is read into the variable ***line***. When the While loop is reached the conditional checks to see if the end of the file has been reached. It is possible for this conditional to evaluate to **false** the first check (if the file is empty). For each iteration of the loop the line read is printed out. Lastly another line is read from ***file*** into ***line***; without line 6 pigVarline = ReadLineFromFile(file) then the loop would continue forever as ***line*** would not update and the conditional will always evaluate to the **true**.

Although the above example uses some concepts you might not be familiar with (functions and file input/output), it should illustrate the usefulness of the While loop and how it can differ from a For loop.

3.2.7 Do-While Loops

A Do-While loop is very similar to a While loop except in a single regard; the code block is guaranteed to run at least once. So as we are familiar with While loops lets jump right into an example.

Listing 3.23: Do-While Loop

```
1 num = 0
2
3 do:
4     print ``Loop''
5     num += 5
6 while num < 25
```

This example is just like the first While loop example we looked at and will run exactly the same number of times. The only difference is that the **Loop** is printed and ***num*** is incremented by **5** before the conditional is checked for the first time. Now let us take a look at an example where the Do-While loop is useful.

Listing 3.24: Another Do-While Loop

```
1 num = 0
2
3 do:
4     print ``Loop''
5     num += 5
6 while num > 10
```

This program will output **Loop** only once. The code block is executed before the conditional is checked for **Loop** is printed then ***num*** is incremented to **5**. Finally the conditional is checked but since ***num*** is less than **10** it evaluates to **false** and Do-While loop stops.

3.2.8 Break Statements

We have seen a sample of Break statements already in Switch statements. They are used to break away from control statements and are used mainly in loops. If you are looping through and come across condition where the loop needs to stop then a break statement can be used to end the loop and continue the program after the loop. Lets look at an example.

Listing 3.25: Break Statement

```
print ``Loop Started''

for i = 0; i < 1000; ++i:
    if i > 20:
        break;
    print i

print ``Loop Finished''
```

What will the output of this code be? It will first output **Loop Started** followed by a listing of 0 through 20 followed by **Loop Finished**.

As we have seen before Break statements are also used in Switch statements. If you take a look at their purpose in loops you can see how they are useful in Switch statements as well. When a Switch statement reached a matching case a block of code is executed followed by a Break statement telling the Switch statement to stop matching cases and continue with the program.

3.2.9 Continue Statements

Continue statements are used in a similar fashion to Break statements but with a different consequence. Where Break statements stop a loops execution a Continue statement tells it to skip the remaining of the code block. Lets jump into it.

Listing 3.26: Continue Statement

```
print ``Loop Started''
for i = 0; i < 1000; ++i:
    if i > 20 && i < 980:
        continue
    print i
print ``Loop Ended''
```

This program will output **Loop Started** followed by a listing of 0 through 20 then 980 through 999 and lastly **Loop Ended**. So as you can see unlike a Break statement which stops a loop in its tracks a Continue statement just says stop processing this iteration of the loop and continue to the next.

3.2.10 Conclusion

We have finished a substantial chapter so lets re-cap. We have covered how to go about controlling the flow of our programs. To make decisions based on the value of variables as well as to continue the execution of code until a given condition is met and even how to control those features.

If, If-Else and Else-If statements allow us to use conditionals to perform given blocks of code given the output of the conditional.

Switch statements are used to perform a given block of code given the specific value of a variable.

For loops allow us to use a counter to execute a given block of code based on the size of the counter.

While and Do-While loops allow us to execute a given block of code while a conditional evaluates to true. A While loop checks the conditional before executing the code block while a Do-While will execute the code block first before checking the conditional.

Lastly, we use Break and Continue statements to control our control statements. A Break statement is used to break free from a loop and stop its execution. A Continue statement allows a loop to skip to the next iteration.

3.3 Functions

3.3.1 Declaration

What is a Function? We have been learning Functional programming, how come we haven't come across Functions until now? Because I said so.

A Function allows use to assign a block of code to a name which allows us to eaily repeat the execution of that code without having to retype the code. Let us take a look at a quick example.

Listing 3.27: Simply Function Example

```
1 def PrintHello():
2     print ''Hello''
3
4 PrintHello()
5 print ''Goodbye''
6 PrintHello()
7 Print ''I said Goodbye!''
```

This code will output the following:
Hello
Goodbye
Hello
I said Goodbye!
As you can see we are declaring a Function called ***PrintHello*** with the code block ***print "Hello"***. As may have noticed when the code gets to the declaration of the Function it does not execute the code block, instead it saves the code block for execution later when the Function name is called. To call the function we use the Function name followed by opening and closing parentheses, ***PrintHello()***.

Be careful, some languages require that Function definitions occur before they are used, while others are more lenient.

3.3.2 Scope

Before getting much further it is important to talk about Scope. Scope refers to the portions of a programs where variables are accessible. So far all of the variables we have been using in past examples belong to the same Scope.

To best explain variable Scope lets look at some examples that explain how Scope works.

Listing 3.28: Scope Example 1

```
1 num = 5
2
3 if num == 5:
4     print num
```

Because we are declaring ***num*** outside of the If statement then that means we can have access to that variable from within the If statement because they belong to the same Scope.

Listing 3.29: Scope Example 2

```
1 num = 5
2
3 if num == 5:
4     another = num + 5
5
6 print another
```

This program will not run properly, because we are declaring ***another*** from within the Scope of the If statement then it is not accessible outside of the If statement. This also goes for any other Conditional or Loop statement (While,For,If-Else,Do-While,Switch,etc). When it comes to Conditional and Loop statements the best way to think about it is that variables can go down into code blocks but cannot come back out.

Listing 3.30: Scope Example 3

```
1 name = ''Brett''
2
3 def PrintHello():
4     print ''Hello '' + name
5
6 PrintHello()
```

This program will not run properly, unlike Conditional and Loop statements, variables cannot go into the Scope of Functions. As well, they cannot leave the Scope of Functions either, so any variables declared inside of the Function are stuck there.

Please remember, as with every other section, please refer to your languages specific rules regarding Scope as some languages break this "normal" paradigm. Scope in Javascript is odd to new comers.

3.3.3 Parameters

We can use parameters to pass variables from outside the Scope of the Function into the Scope of the Function. Lets rewrite the example (Scope Example 3) from above but this time taking into account variable Scope and using Parameters.

Listing 3.31: PrintHello Parameter Example

```
1 def PrintHello( name ):
2     print ''Hello '' + name
3
4 MyName = ''Brett''
5
6 PrintHello( MyName )
```

The output of this program will be **Hello Brett**. By adding the ***name*** Parameter to the ***PrintHello*** Function definition we are able to then pass in the variable ***MyName*** from outside of the Function Scope into the Function Scope.

To define Parameters of a Function, you simple give a list of Parameter names between the two parenthesis separated by commas. When you define a Parameter for a Function you are requiring whoever uses that Function in the future of the program that they must supply that many Parameters. Let us take a look at an example Function that uses multiple Parameters.

Listing 3.32: Multiple Parameters Example

```
1 def Sum( x, y ):
2     sum = x + y
3     print ''The Sum Is: '' + sum
4
5 Sum( 5, 6 )
6 Sum( 12 )
```

We are simply defining a Function that is used to sum the values of two numbers together and then printing **The Sum Is:** followed by the sum of the numbers. By defining two Parameters ***x*** and ***y*** we are telling any users of the Function that they must supply two values into the Function. The first example usage of ***Sum***, ***Sum(5, 6)***

correctly calls the Function and will cause the program to output **The Sum Is: 11**. The Second example ***Sum(12)*** will cause the program to fail because only one Parameter is given when two were defined.

When a Function is called and Parameters are given, the values passed into the Function are assigned to variables, inside of the Functions Scope, with the same name given in the Function definition. In our example above the value **5** is passed into the Function and assigned to the variable ***x*** because both are the first value and first Parameter. **6** is then assigned to ***y***.

3.3.4 Returns

We have just seen how we can use Parameters to pass values into Functions, but what if we want to pass values back out of Functions? We would use a Return statement. When using a Return statement you use ***return*** followed by the values or variable you wish to return out of the Function.

Listing 3.33: Return Example

```
1 def Sum( x, y )
2     sum = x + y
3     return sum
4
5 MySum = Sum( 5, 6 )
6 print ``The Sum Is: `` + MySum
```

This program will output **The Sum Is: 11**. The Function is the same as before but this time we are using ***return*** to push the value assigned to ***sum*** back out of the Function. We can then assigned the output of the Function ***Sum*** to a variable ***MySum*** which will be equal to whatever the value that is returned is, in this case **11**. When using Return statements in Functions you can then think about using Functions in the same way as you would a raw value (like a number).

When the Return statement is reached the Function stops executing and the value of the Return statement is returned. So we can use Return statements similar in fashion to how Break statements are used to quickly stop the execution of a Function.

Listing 3.34: Return to Stop Function

```
1 def PrintHello( name ):
2     if name == ``Brett'':
3         return
4     print ``Hello `` + name
5
6 PrintHello( ``Brett'' )
7 PrintHello( ``James'' )
```

This example program will only output **Hello James**, because when the value **Brett** is passed into the Function the If statement evaluates to **True** and the Return statement causes the Function to exit and the ***print*** statement is never reached.

A Function can also contain multiple Return statements.

Listing 3.35: Multiple Return Statements

```
1 def SayHello( time ):
2     if time < 12:
3         return ``Good Morning''
4     else:
5         return ``Good Afternoon''
6
7 print SayHello( 10 )
```

```
8 print SayHello( 3 )
```

The output of this program will be **Good Morning** followed by **Good Afternoon**. If the value passed into ***time*** is less than **12** then **Good Morning** is returned, otherwise **Good Afternoon** is returned.

Side Note: I do have to mention, some people will argue with the example above and say Functions should only have a single Return statement. For those that agree or who wish to follow this mentality I have provided the example below to show the same example above with a single Return statement.

Listing 3.36: Single Return Statement

```
1  def SayHello( time ):
2      ReturnString = null
3      if time < 12:
4          ReturnString = ''Good Morning''
5      else:
6          ReturnString = ''Good Afternoon''
7      return ReturnString
8
9  print SayHello( 10 )
10 print SayHello( 3 )
```

This example will output the exact same as above but it only uses a single Return statement.

3.3.5 Recursion

Recursion, now this section is going to be FUN! Please take your time reading through this chapter as it is very easy to get confused by Recursive Functions and can be difficult to wrap your head around the first time through (I know I had issues when I first learned it). Basically what it means for a Function to be recursive is when it makes a call to itself from within it's own code block. The example we are going to be using is a recursive Function used calculate the factorial of a number. The factorial of the number 6, denoted 6!, is 6 x 5 x 4 x 3 x 2 x 1 = 720. Lets look at how this would look as a recursive Function.

Listing 3.37: Recursive Factorial

```
1  def Factorial( num ):
2      if num < 1:
3          return 1
4      else:
5          return num * Factorial( num - 1 )
6
7  print ''6! = '' + Factorial( 6 )
```

There are a few parts of a Recursive Function that we need to mention before diving deeping. First off we have the actual Recursive call which is when the Function calls it's, in this Function it is the code ***Factorial (num - 1)***. Secondly, and probably the most important part, is the Base Case:
if num ¡ 1: return 1
The base case tells the Function when to stop calling itself recursively, otherwise the Function will call itself infinitely and cause an infinite loop and could crash the program.

It might not be clear to some exactly how this Function is working, how is it getting the right value back out into the program? Well, lets do what we can to break down the Function call ***Factorial(6)*** down to see exactly how the program interprets this Function.

Chapter 4

Object Oriented Programming

4.1 Classes and Objects

Classes *AND* Objects? What is the difference?

Well I am glad you asked. A class is the definition or blueprint of an object. A class tells a program what to expect when coming across an object of the given class. What methods and properties to expect and even how to create and destroy objects.

An object refers to a single instance of a class
Objects are refered to as being instances of a class. When you deine a class you are not creating a usable object that you can then call methods on or access properties of. You must then create an instance of that class (object) to be able to use it throughout your program.

4.1.1 Classes

Ok, so as I mentioned before we need to first define a class before we can start creating objects and using them in our program. How do we do this?

Listing 4.1: Class Definition

```
1 class Person
```

Ok...? That seems too easy?
Yes creating classes is usually fairly easy, just make sure to check how to create a class in your language of choice.

4.1.2 Objects

Ok, so we have our class definition from above, but how do we create an instance of this class so we can use it in our program?

Listing 4.2: Object Declaration

```
1 class Person
2
3 p = new Person()
```

That is it. We can create an instance of our ***Person*** class by using the ***new*** keyword and calling ***Person()***. We can assign this instance to a variable, ***p***, and then use ***p*** as an alias for our object throughout our program.

Can we only have one object? No, you can have as many instances as your would like.

Listing 4.3: Multiple Object Instances

```
1 class Person
2
3 p1 = new Person()
4 p2 = new Person()
5 p3 = new Person()
```

This then allows us to act on each of these instances as though they are separate. What does that mean? It means that if we were to modify a property of ***p1*** then it would not have any effect on the same properties in ***p2*** and ***p3***.

4.1.3 Properties

We are able to store variables inside of a class, these are called properties. To define a property we must define its name, access modifier and default value (if any).

An access modifier can either be *public*, *private* or *protected* (some languages do not support access modifiers). The *public* modifier means that anyone who has access to the object can read and modify that property. The *private* modifier means that no one outside of the object can read and modify the property, meaning that only the object itself has acess to the given property. The *protected* modifier means that the given object and its children (we will get to this later in the chapter) will have access to read and modify the property. Lets look at an example.

Listing 4.4: Class Properties

```
1 class Person
2       public name
3       private age = 22
4
5 p = new Person()
6 p.name = ``Brett Langdon''
7
8 p.age = 23 //this will cause an error
```

In this example we are creating a class with two properties, one is public (***name***) and the other is private (***age***). We then create a new instance of our class assigning it to the variable ***p***. Then we set the public property ***name*** to **"Brett Langdon"**. In line 8 there is the comment "this will cause an error" this is because the property ***age*** is private and cannot be accessed from outside of the class.

4.1.4 Methods

So what is a Method? A method, simply put, is a function that belongs to a class. We use methods for the same reasons that we use functions for, to provide code reuse within our applications. Ok, so we know how to use functions, but how do we use them from within a class?

Listing 4.5: Class Methods

```
1  class Person
2        public name
3        private age
4
5        def printName()
6                print this.name
7
8  p = new Person()
9  p.name = ``brett''
10 p.printName()
```

The output of this code would be **brett**. You access methods the same way as your would class properties except you include the parenthesis. You may also notice a reference to a variable ***this*** in the method definition. This special variable is used within methods to refer to the object that the method belongs to. So using ***this*** within the method ***printName*** is similar to using the variable ***p*** to access the specific instance that that method belongs to.

4.1.5 Special Methods

4.2 Inheritence

4.3 Polymorphism

Chapter 5

Design Patterns

5.1 Singleton

5.2 Factory

5.3 Observer

5.4 State

Chapter 6

Data Structures

6.1 Big O Notation

6.2 Linked Lists

6.3 Doubly Linked Lists

6.4 Stacks

6.5 Queues

6.6 Hash Maps

6.7 Binary Trees

6.8 B-Trees

6.9 B+ Trees

Chapter 7

Algorithms