Learning How to Code: The Basics

Foreword - Motivation

Are you creative and do you like problem solving? Do you enjoy learning? Do you like a challenge? If your answer to any of these questions is yes then I'd highly recommend that you learn how to code. Why would learning how to code give me an advantage (I hear you ask)? If you know how to code, especially if you're a very experienced and specialised coder then you can end up working for companies such as Microsoft, IBM or even **Google**. Of course, you're not going to be a godly coder when you start. Coding takes practice.

Practicing is an important part of the learning process for many skills not just coding. Coding will improve your analytical problem solving skills and give you a lot of invaluable experience that you can apply to other fields outside of IT.

Introduction – What this tutorial is about

This is intended as an introductory tutorial consisting of a collection of information to encourage you to start learning how to code. Coding has a lot of meanings and contexts, but in this tutorial coding is the term that will used to refer to "the symbolic arrangement of statements or instructions in a computer, or a set of instructions in a program: 'the program took 3000 lines of code'."[1] The starting point for learning any new skill is where most people get stuck at.

This tutorial will aim to give you a good starting point to pave the stepping stones for you to learn coding. This tutorial will not teach you a specific language, say C++ or JavaScript. Rather than teaching the semantics of a specific programming language this tutorial aims to give you a general **overview** of how to start learning coding with good starting habits. *"Most good programmers do programming not because they expect to get paid or get adulation by public, but because it is fun to program."*[2] – Linus Torvalds.

Where to start?

In order to determine what you want to accomplish through learning coding you must ask yourself some **questions**: (a) *"What* is it that I want to accomplish through coding?" (b) "Do I enjoy working on *projects* as a group or alone?" (c) "Would I rather learn how to code a *specific* program based on what I want to make or would I rather

learn the *basics* of a programming language before getting into the details?" There are no right or wrong answers to these questions, however asking the right kinds of questions will be essentially in determining the specifics of what you want to learn.

When starting to learn coding you should familiarise yourself with the formalities of coding, otherwise your code will become hard to read and you'll end up with what is referred to as "*spaghetti code*"[3]. The semantics of the language and the common programming practices should be established after you have chosen which language you want to start programming in.

Picking your starting language

Your starting language will always be **personal**. Personally I chose C as my starting language. Do you want to have a lot of power and control over low level aspects of the computer? Do you want to have less control but more 'human relation' in your code? Do you like dealing with details or do you like seeing the overall picture? If you'd like an 'easier' start I would highly recommend Python. Python's syntax is very easy to learn and you don't need any previous experience of coding to become good at Python.

Although C is equally as good of a starting language, because the syntax is strongly typed, so it will give you a solid foundation to build your understanding of coding, C has a very steep learning curve compared to Python. So if you want a challenge I would recommend C, if you'd like an 'easier' start then I would recommend Python. It is important to note that C and Python are not the only Programming languages, there are many other programming languages[4] available.

Starting Resources

Textbooks are often a very good starting point to learning a programming language. There are 2 main types of programming textbooks: reference textbooks and 'how to text books'. References textbooks are useful for **learning** the semantics, 'how to text books' are useful for getting a good understanding on areas of a language, wherever it be up to the Object Orientated Programming (OOP) level, or more specific 'Artificial Intelligence – Neural Networks'. If you like more interactive forms of learning then there are many online learning platforms available which offer very good courses (often for free!) on programming. Personally I prefer online courses,

because I like structured learning and I like having a 'qualification' that verifies the knowledge that I have on a specific language. This, however, does not mean that you should not use one resource if you're using the other. I would highly recommend using every resource that is available to you.

Good Programming Practices – Asking Questions

Before attempting to write a program you should ask yourself some questions: "What do I want my program to do?", "Why do I want my program to do X?", "How am I going to get to the final result?", "What is the algorithm for getting from A to Z?" and finally "How am I going to **implement** my algorithm?". Of course, if it not necessary to do this for all programs, but it is good practice to start getting into the habit of asking these types of questions, to build a solid foundation of the fundamentals to programming.

Good Programming Practices – Naming conventions

So now that you've decided the above, you should adopt a formal style of programming. Hungarian Notation, an example being *szinputText*, is the most well-known programming convention where the variable name begins with an abbreviation that represents the type in this case *sz* means **z**ero-terminated-**s**tring. Using delimiters _ (underscore) and – (hyphen) to separate words referred to as list-case or COBOL_CASE, an example would be *my_apple* where *my* indicates that the *apple* belongs to that specific class, this can be used in class methods. CamelCase is when each word is capitalized to prevent having to use delimiters, an example would be *coefficientOfFriction* where. There are many other naming conventions that are language specific, but varations of: Hungarian Notation, list-case and CamelCase are adopted in many programming languages so it's useful to know about them. There are many other programming conventions of which are dealt with by boards such as X3J11[5], but these won't be covered in this tutorial.

Good Programming Practices – Abstraction and Decomposition

Abstraction is when you supress the details of a **method** to compute something that uses that computation. Decomposition is when you break a problem down into individual pieces. Abstraction and decomposition are both powerful practices that should be utilized to create clean code. Abstraction and decomposition will become very important when you later learn about classes. You experience abstraction and decomposition in your daily life, for example you

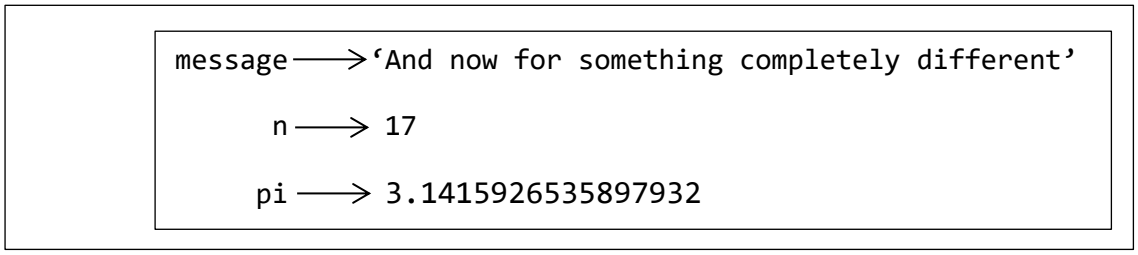
have probably used a bus to travel from A to B without having to know the internal workings of the bus you were able to get from your starting point A to your final destination B.

Good Programming Practices – State Diagram

A state diagram is a way of **representing** variables through graphical depictions, where the name of the variable has an arrow pointing the value of the variable. If I wanted to represent the assignment statements below[6]:

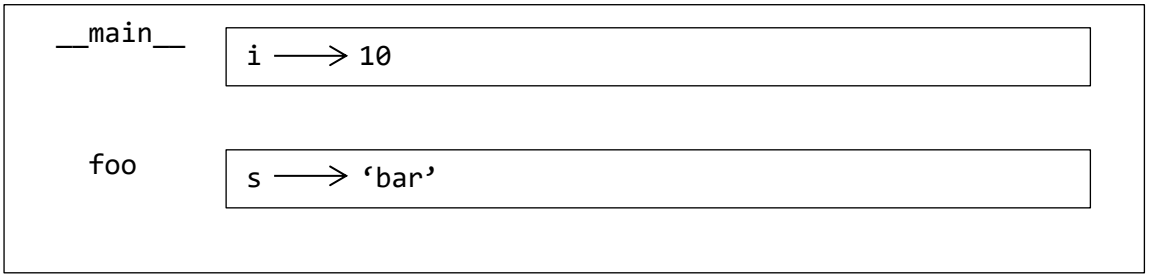
```
>>> message = 'And now for something completely different'

>>> n = 17

>>> pi = 3.1415926535897932
```

Then the state diagram would be as follows



Good Programming Practices – Stack Diagram

Stack Diagrams show the function to which variable values belong to. Each function is represented by a frame. A **frame** is a box with the name of the function beside it and the parameters and the variables of the function inside it.[7] For example I have a function called foo, and I want to print the string 'bar' 10 times then the stack diagram would look something like this:



Both the above examples are based on programs written in Python. If you want to learn more about Python I would recommend Think Python. For more detailed information refer to recommended resources.

Good Programming Practices – Refactoring

When you have written some code you can often rewrite the code to make it more **efficient**, without altering its external behaviour, which can save a lot of computational resources. As an example the following code is used feet to inches[8]:

```
using System;

class Test

{

  static void Main()

  {

    int x = 12 * 30;

    Console.WriteLine (x);

  }

}
```

The problem with this code is that the result of the program will always be the same. If you wanted to calculate the number of inches in an **arbitrary** number of feet then this refactored code would work:

```
using System;

class Test

{

  static void Main()

  {

    Console.WriteLine (FeetToInches (30));

    Console.WriteLine (FeetToInches (100));

  }

}

static int FeetToInches (int feet)

{

  int inches = feet * 12;

  return inches;}
```

Good Programming Practices – Debugging

More often than you'd like you will experience a problem or something will go wrong with your program. That's where debugging comes in… debugging lets you quickly **diagnose** the problem and hence lets you solve the problem faster. Most programming languages will have their own debuggers and conventions. For example the .NET Framework allows you to enforce code contracts[9].

There are three kinds of errors which can occur in you program: syntax errors, runtime errors and semantic errors. A syntax error refers to the structure of a program and the rules about that structure. Runtime errors appear when the program has started running and they indicate that something **exceptionally** bad has happened. Semantic errors refer to the meaning. When there is a semantic error the program will run without generating error messages but it will not do the right thing[10].

Probably the most common debugging tool is Visual Studio Debugger. The VSD allows you to setup **breakpoints** in your code by clicking in the margin to the left of a line of code[11].

One of the skills that every programmer develops over his or her career is how to detect **bugs** and debug programs. One small bug can reverberate through a program with incredible effects. In fact, in a language such as C, a single missing quote can wreak havoc on a programmer's weekend[12].

The most important parts of an error message are usually what kind of error was thrown and where it occurred. Of the three kinds of **errors** run time errors are usually the trickiest to fix, because they can occur anytime during the execution of a program. Although you could argue that semantic errors are the trickiest to solve because the programmer may not know what the right thing is or why the program is doing the wrong thing, perhaps even both! Syntax errors are easy to fix because the compiler will tell you exactly where the syntax error is occurring, so you only have to figure out what to change that specific part of the program to fix the error.

Breaking a large program down into smaller pieces will also help to reduce the amount of errors and problems that you'll experience and therefore reduce the time that you will have to spend debugging. So make sure to make your code as simple as possible and use **separate** functions to deal with individual parts of the problem.

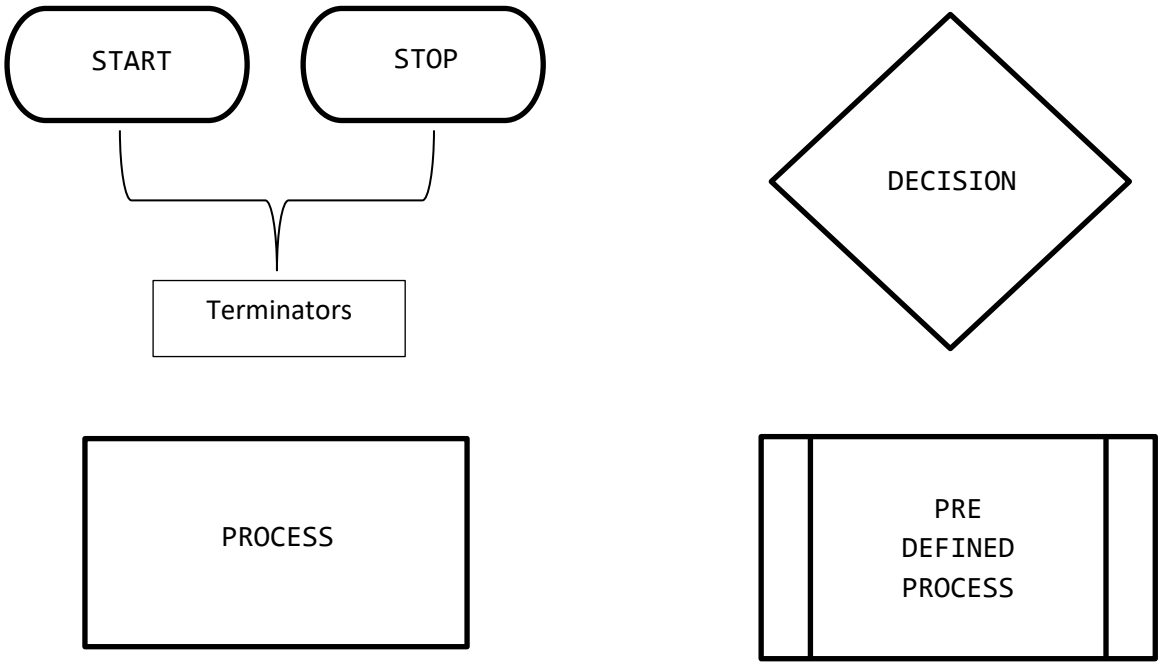
Good Programming Practices – Pseudocode

Pseudocode is an informal high-level **text-based** language that enables a programmer to develop an algorithm to solve a given problem. Details are often emitted and problems are simplified in order to give a better overall view of the problem. “The rules of Pseudocode are reasonably straightforward. All statements showing "dependency" are to be indented. These include while, do, for, if, switch. ... For looping and selection, The keywords that are to be used include Do While...Enddo; Do Until...Enddo; Case...Endcase; If...Endif; Call ... with (parameters); Call; Return ....; Return; When; Always use scope terminators for loops and iteration. As verbs, use the words Generate, Compute, Process, etc. Words such as set, reset, increment, compute, calculate, add, sum, multiply, ... print, display, input, output, edit, test , etc. with careful indentation tend to foster desirable pseudocode. Do not include data declarations in your pseudocode.[13]”

Good Programming Practices – Flow Charts

A flow chart is used as a **graphical** representation of a program, where the rules are depicted using symbols and design constructs which make them easy to develop and interpret.

The main symbols used are:



Arrows are used to indicate the flow of execution of the program.

Recommended Resources

I have compiled a small (*non-exhaustive*) list of resources for learning how to program. Here's the list:

1. "Think Python How To Think Like A Computer Scientist" – Allen B. Downey (O'REILLY)
2. "Sam's Teach Yourself C++ in 21 Days" – Jesse Liberty, David B. Horvath, CCP (SAMS)
3. "C# 6.0 in a Nutshell The Definitive Reference" – Joseph Albahari & Ben Albahari (O'REILLY)
4. Edx.org: https://www.edx.org/
5. Python Challenge: http://www.pythonchallenge.com/
6. CPP if you don't have your textbook on you: http://www.cplusplus.com/reference/
7. Python3 reference if you don't you're your textbook on you: https://docs.python.org/3/
8. "Exploring Artificial Intelligence On Your Spectrum+ and Spectrum" – Tim Hartnell (Interface Publications)
9. "C++ How To Program" – Deitel & Deitel (Prentice Hall)
10. http://csharp.net-tutorials.com/debugging/introduction/

"Live as if you were to die tomorrow. Learn as if you were to live forever." – **Mahatma Gandhi**

References:

1. http://www.dictionary.com/browse/coding
2. https://www.brainyquote.com/quotes/linus_torvalds_367382
3. https://en.wikipedia.org/wiki/Spaghetti_code
4. https://en.wikipedia.org/wiki/List_of_programming_languages
5. https://www.doc.ic.ac.uk/~nd/surprise_96/journal/vol1/mjbn/article1.html
6. Allen B. Downey 2015, Think Python: How to Think Like a Computer Scientist. Sebastopol, CA 95472: O'REILLEY, Page 11
7. Allen B. Downey 2015, Think Python: How to Think Like a Computer Scientist. Sebastopol, CA 95472: O'REILLEY, Page 28
8. Joseph A. & Ben A. 2015, C# 6.0 in a Nutshell: The Definitive Reference. Sebastopol, CA 95472: O'REILLEY, Pages 11-12

-832195389883164595046194491556360369181335980224887169400850845605187229213 13

[9.] Joseph A. & Ben A. 2015, C# 6.0 in a Nutshell: The Definitive Reference. Sebastopol, CA 95472: O'REILLEY, Page 525

[10.] Allen B. Downey 2015, Think Python: How to Think Like a Computer Scientist. Sebastopol, CA 95472: O'REILLEY, Page 17

[11.] https://docs.microsoft.com/en-gb/visualstudio/debugger/debugger-feature-tour

[12.] https://web.cs.dal.ca/~jamie/UWO/C/Debug/UGCS_CS2_debugging_notes.html

[13.] http://www.unf.edu/~broggio/cop2221/2221pseu.htm

Acknowledgements

I'd like to thank the many people that reviewed my work and allowed me to make appropriate changes to it. Due to privacy reasons all the people who reviewed my work will not have their names disclosed. The information that I gathered from the resources listed in "Recommended Resources" was also invaluable so I'd like to thank the Authors of the books and the founders and teachers of the courses.

Disclaimer