# Programming Mindset

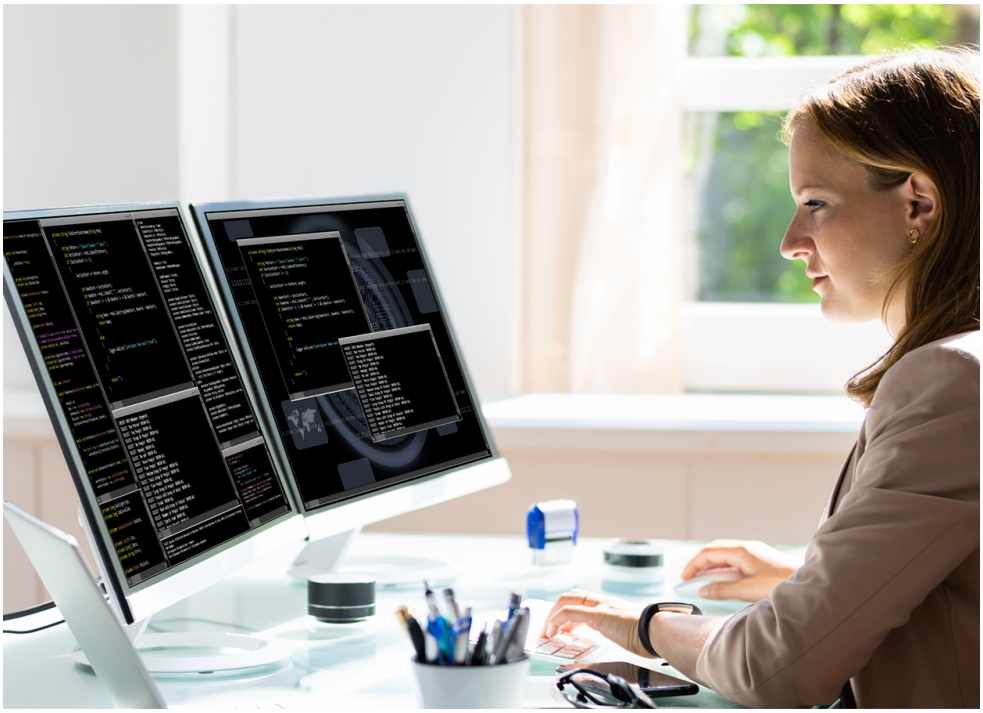
*by Sophia*

WHAT'S COVERED

In this lesson, you will learn about how to think like a developer and what's involved in the development process. Specifically, this lesson covers:

1. Writing Programs
2. Input, Processing, and Output
3. Understanding the Development Life Cycle

## 1. Writing Programs



Writing programs (or programming) is a very creative and rewarding activity. You can write programs for many reasons, ranging from making your living or solving a difficult data analysis problem, to having fun or helping

someone else solve a problem. This course assumes that everyone needs to know how to program, and that once you know how to program you will figure out what you want to do with your newfound skills.

We are surrounded in our daily lives with computers ranging from laptops to cell phones. We can think of these computers as our “personal assistants” who can take care of many things on our behalf. The hardware in our current-day computers is essentially built to continuously ask us the question, “What would you like me to do next?”

Our computers are fast and have vast amounts of memory and can be very helpful to us if we know the language to speak to explain to the computer what we would like it to “do next.” If we knew this language, we could tell the computer to do tasks on our behalf that were repetitive. Interestingly, the kinds of things computers can do best are often the kinds of things that we humans find boring and mind-numbing.

### EXAMPLE

For instance, look at the first three paragraphs of this lesson and tell me the most commonly used word and how many times the word is used. While you were able to read and understand the words in a few seconds, counting them is almost painful because it is not the kind of problem that human minds are designed to solve. For a computer, the opposite is true. Reading and understanding text from a piece of paper is hard for a computer to do, but counting the words and telling you how many times the most used word was used is very easy for the computer.

This very fact that computers are good at things that humans are not is why you should become skilled at talking “computer language.” Once you learn this new language, you can delegate mundane tasks to your partner (the computer), leaving more time for you to do the things that you are uniquely suited for. You bring creativity, intuition, and inventiveness to this partnership.

In the rest of this course, we will turn you into a person who is skilled in the art of programming. In the end you will be a programmer—perhaps not a professional programmer, but at least you will have the skills to look at a data/information analysis problem and develop a program to solve the problem.

In a sense, you need two skills to be a programmer:

- First, you need to know the programming language (Python)—its vocabulary and grammar. You need to be able to spell the words in this new language properly and know how to construct well-formed “sentences” in this new language.
- Second, you need to “tell a story.” When writing a story, you combine words and sentences to convey an idea to the reader. There is a skill and art in constructing the story, and skill in story-writing is improved by doing some writing and getting some feedback. In programming, our program is the “story” and the problem you are trying to solve is the “idea.”

Once you learn one programming language such as Python, you will find it much easier to learn a second programming language such as JavaScript or C++. This new programming language may have a very different vocabulary and grammar, but the problem-solving skills will be the same across all programming languages. You will learn the “vocabulary” and “sentences” of Python pretty quickly. It will take longer for you to be able to write a coherent program to solve a brand-new problem. We teach programming much like we teach writing. We start reading and explaining programs, then we write simple programs, and then we write increasingly complex programs over time. At some point you “get your muse." You begin to see the patterns on your own and can see more naturally how to take a problem and write a program that solves that problem. And once you get to that point, programming becomes a very pleasant and creative process.

The definition of a **program** at its most basic is a sequence of Python statements that have been crafted to do something. It might be easiest to understand what a program is by thinking about a problem that a program might be built to solve, and then looking at a program that would solve that problem.

Let's say you're doing social computing research on Facebook posts and you're interested in the most frequently used word in a series of posts. You could print out the stream of Facebook posts and pore over the text looking for the most common word, but that would take a long time and be very mistake-prone. You would

be smarter to write a Python program to handle the task quickly and accurately, so you can spend the weekend doing something fun.


Python is a way for us to exchange useful instruction sequences (i.e., programs) in a common language that can be used by anyone who installs Python on their computer. So, neither of us are talking *to* Python; instead, we are communicating with each other *through* Python.

TERM TO KNOW

**Program**
A sequence of computer language statements that have been crafted to do something.

# 2. Input, Processing, and Output

THINK ABOUT IT

Imagine having to cook a recipe for the first time. You'd need to have the specific ingredients and their amounts, as well as the detailed instructions of how to prepare the recipe. Similarly, a computer program consists of lines of code that the computer runs and uses variables to perform them. Things must run in a specific order. For example, if you were baking a cake, you wouldn't put the cake in the oven prior to mixing in the ingredients. You must think about all of the steps in logical order.

Typically there are three main types of operations that are executed in any program.

- The first is the input to the program. The **input** to the program allows data to be entered in through a variety of ways, including by you through the keyboard and mouse. It could also be from files on the computer, other hardware devices, images, sounds or many other potential options. The input from these sources is stored and placed into the memory of the computer and can include all types of data, including text and numeric values.
- The next type of operation is **processing**. When it comes to processing data, there can be many different steps that are performed on the data. It could be storing them, using them for calculations, validating the data or sorting the data.
- The last type of operation is the **output**. After the data has been processed, output is sent to the monitor, printer, email, file, report, or other means where individuals can view the resulting data. The goal of most programs is to take the raw data or input, process it into information that can be useful, and then output it to the user.

In order to write these programs, we need to use a programming language that the computer understands, such as Python, Java, C++ or C#. In this course, we will be using Python. Each programming language has its strengths and weaknesses for the task at hand. Most of these languages have similar logic behind them so you can carry that between the different languages. Where they differ are the rules called **syntax**. Each programming language has a very specific set of syntax rules that needs to be followed. Computers are not smart enough to understand a program unless the syntax is correct.


Python was created as a programming language for those who aren't doing software development around the clock, but instead for those who are interested in code to handle common tasks. When we write a program using programming languages like C or C++, we have to compile it. In the process of compiling it, the **compiler** takes the human-readable code that we write and translates it to **machine code** (also known as machine language) that can be executed by the computer. If a program is successfully compiled, the compiler creates a file that can be run. A compiled program has limitations as it can only be run on specific platforms that it is written for.

Python, on the other hand, is an interpreted language similar to Java. It is not a compiled language (although we still go through and compile our code). With Python, it is written as a .py file and when it is compiled into bytecode, it is saved as a .pyc or .pyo format. The **bytecode** is different from the machine code, as the bytecode is a low-level set of instructions that can be run through an **interpreter**. This interpreter has to be installed on the computer. Instead of running the program directly on the computer, the bytecode is run through a **virtual machine**. One of the key reasons why interpreted languages are preferred is because they are platform independent. This means that as long as the bytecode and the virtual machine have the same version, a Python program can be run on any platform regardless of if it is a Windows, MacOS, or Linux system. Although there are differences in how compilers and interpreters function, the core purpose is the same, with interpreters having that added step.

TERMS TO KNOW

**Input**
Ways a program gets its data; user input through the keyboard, mouse, or other device.

**Processing**
Takes the data inputs and prompts and calculates the results (output).

**Output**
The results at the end of a program based on user input and system processing.

**Syntax**
The syntax is the “grammar” rules of the programming language. Each programming language (like Python) has its own syntax.

**Compiler**
A compiler scans an entire program and attempts to convert the whole program at once to machine code.

**Machine Code**
Also known as machine language. The computer uses binary (0s and 1s) to perform tasks. At a high level, the programming language code that you write gets translated or compiled to the machine code that the computer understands.

**Bytecode**
An intermediary step for code conversion between the programming language that you write and the machine code that a computer uses.

**Interpreter**
An interpreter takes the bytecode one line at a time and converts it to machine code.

**Virtual Machine**
A software program that behaves like a completely separate computer within an application.

# 3. Understanding the Development Life Cycle

A development life cycle is a high-level process for planning, creating, testing and deploying an application. There are many different approaches to the development life cycle, but at a high level, they are all relatively the same. There are certain steps that are part of the development life cycle that should be performed when tackling a problem. Developers generally should not just sit down and start writing code. Rather, they should break things down into individual steps. Some of those steps can be combined, and some may never be needed for every program. The steps are:


1. Understand what the problem being asked is.

2. Plan out the logic for the problem by breaking it down into a sequence of steps or algorithm.
3. Write the code to perform the algorithm.
4. Compile/translate the code to a format the computer can read.
5. Test the program to ensure there are no syntax or logical errors.
6. Deploy the program to be used.
7. Support the maintenance of the program.

If we don't understand what the problem is that we're trying to solve, we'll have errors right from the start. This step may require some planning to gather the requirements from those that are affected or what we consider as stakeholders. As developers, we may not have the knowledge about the industry that we're building the program for and have to rely on those subject matter experts to provide their input on the business processes.


For example, if you are tasked with writing a program for a high school to email to the students how much the school fees are, we may have some questions. For example, are the fees the same for all students, or are they different by year, or perhaps by the courses that they take? When should the students be notified? Where do we find the student list? Are there exemptions to the fees? These are not questions that as a developer you can answer, and in order to understand the problem, you have to go to someone that can provide that guidance.

Once we understand what the problem is, we can then start to plan the logic to break it down further. We'll explore this in upcoming lessons, but the idea is that we want to break it down in detail so that the computer can understand what needs to be done. Imagine if you had to direct someone that was blindfolded through an obstacle course. You would need to be very specific with the steps that they would need to take and when. If you accidentally told them to jump when they should duck under a pole, they may hurt themselves. In the same way, programs need to be written in the correct logical order for them to work correctly. This is the process where we'll form an algorithm as the sequence of steps to solve the problem.

With the algorithms in place, the program can then be developed in a programming language. In our course, we will focus on Python, but there are hundreds of languages that could be selected. Each has their advantages and disadvantages, so selecting one can be based on several criteria. One advantage may be familiarity; there are some programming languages like C++, C#, Java, and Python that are fairly widely used in various methods. If you are already familiar with the language, you may choose to use that method.

Another criterion, if you are developing a program for an organization, is what languages they use within the organization. As a program could be developed by many individuals and teams, having consistency across the developers is important. Another important criterion is the support for specific libraries and programs that you want to use. Different programs will have different levels of support for programming languages. For example, if we are developing code for a game engine called Unity, we would have to use C#, Boo (similar to Python), or JavaScript. We don't have a choice to use other options.

Although we do have many different programming languages, at the computer level, there is only the machine code that's coded in 0s and 1s. These 0s and 1s are set up as electrical switches that can be turned off and on, which are represented by the 0s and 1s respectively. At a high level, the programming language code that you write gets translated or compiled to the machine code that the computer understands.

If there are errors in the syntax, this is normally caught during the prior step. Without the program written correctly, the compiler would throw an error to the user to fix. It would be similar to having an English program that said, "The ct and mouse were friends." This would result in an error, because "ct" isn't a word in the English language. We meant to have it as "cat."

Logical errors are issues that the computer wouldn't catch. For example, if the program meant to give everyone a 10% discount on their orders, but it was accidentally coded to give everyone a 100% discount, that wouldn't be caught by the compiler and would be a logical error. In order to find these errors, we have to test the code and compare the results with the expected values.

Once the code has been tested, it can then be deployed for the program to be used for the purpose that it was originally developed.

Once a program is deployed, there may be bugs or errors that come up that have to be addressed and patched. There may be improvements to performance that have to be addressed. All of these factors may require maintenance. In some cases, it may require some additional development, which in turn would start right back at understanding the problem again.

## SUMMARY

In this lesson, we learned what programming is and about the purpose of **writing programs** to solve problems. We also learned the three main types of operations of a program: **input, processing, and**

**output**. Finally, we talked about how the **development life cycle** requires a high-level process for planning, creating, testing, and deploying an application. Developers generally do not just sit down and start writing code; rather, they break the process down into individual steps.

Best of luck in your learning!

Source: THIS CONTENT AND SUPPLEMENTAL MATERIAL HAS BEEN ADAPTED FROM “PYTHON FOR EVERYBODY” BY DR. CHARLES R. SEVERANCE ACCESS FOR FREE AT www.py4e.com/html3/ LICENSE: CREATIVE COMMONS ATTRIBUTION 3.0 UNPORTED.

## TERMS TO KNOW

**Bytecode**
An intermediary step for code conversion between the programming language that you write and the machine code that a computer uses.

**Compiler**
A compiler scans an entire program and attempts to convert the whole program at once to machine code.

**Input**
Ways a program gets its data; user input through the keyboard, mouse, or other device.

**Interpreter**
An interpreter takes the bytecode one line at a time and converts it to machine code.

**Machine Code**
Also known as machine language. The computer uses binary (0s and 1s) to perform tasks. At a high level, the programming language code that you write gets translated or compiled to the machine code that the computer understands.

**Output**
The results at the end of a program based on user input and system processing.

**Processing**
Taking the data inputs and prompts and calculates the results (output).

**Program**
A sequence of computer language statements that have been crafted to do something.

**Syntax**
The syntax is the “grammar” rules of the programming language. Each programming language (like Python) has its own syntax.

**Virtual Machine**

A software program that behaves like a completely separate computer within an application.