Introduction to Computers and Programming

Evan Halloran

2025

These are a series of notes aimed at introducing the basic concepts of programming and computer science. In many ways, it is language-independent. The choice of a "first language" is immaterial—what matters more are the principles that are inherent across the domains of computer science. That said, we will choose to use the Python programming language for a couple of reasons. Firstly, it is relatively light on syntax and allows the user to write the same code in fewer lines compared to most other languages. It is weakly-typed and forgiving on type mismatches. Secondly, Python is excellent at manipulating large amounts of data, and its usage has skyrocketed in fields such as machine learning, artificial intelligence, and quantitative finance. Facility in Python is an excellent skill to have.

1 The Theory of Computation

Computer science, contrary to popular belief, is not the study of coding and programming (though these *are* subfields of computer science). Computer science is the study of the theory of computation—how logical truths can be deduced (computed) from prior truths. Viewed this way, it becomes clear why computer science existed decades before the first modern computer. It began as a subfield of mathematics and picked up considerable traction during the turn of the 20th century, when mathematicians such as David Hilbert wanted to formalize the foundation of mathematics. If mathematics can be reduced to a small list of axioms, then all of known math can be proven from only a couple ground truths. For a theorem to rely on these axioms means that someone can sit down and apply the rules of inference (modus ponens, modus tollens, etc.) to the axioms repeatedly to generate (compute) the theorem. Therefore, we often say that a true statement which can be proven is computable.

With this realization comes interesting questions. What is beyond the scope of computation, i.e. do there exist true statements which cannot be proven? Can a logical system of axioms prove its own consistency, i.e. can we rigorously prove using our axioms that they are not in contradiction with each other? These questions were answered by Kurt Godel in the 1930s. To the horror of Hilbert, there exist mathematical truths within any formalized system of axioms that can not be proven (known as the Incompleteness of mathematics). Even worse, a system of axioms cannot prove its own consistency. These theorems of Godel expose a severe limitation in what can be computed.

A third issue, the question of Decidability, was explored by Alan Turing. Given a statement (either true of false), can we determine if its computable, i.e. that its truth or falsity follows from the axioms? The answer, Turing discovered, is that we cannot, and he had to invent the concept of a computer to arrive at this conclusion.

Turing's computer is simple: a computer is a system consisting of a strip of grid paper and a pen. The pen can either write on the paper at the current grid, shift right to the next grid, shift left to the previous grid, read what is written at the current grid, or do nothing (halt). The question of Decidability reduces to the famous Halting problem on this machine. Turing leveraged the idea of this theoretical machine to answer the problems of both Halting and Decidability.

This theoretical machine is aptly known as a Turing machine, and any computational system (be it real or theoretical) that can perform the same actions as a Turing machine is said to be Turing complete. Before the invention of the electronic computer, this is what a "computer" meant in computer science. Turing was able to prove that **there is no system of computation more powerful than a Turing machine, and all Turing machines are equivalent in what they can compute**. This is astounding—every system that is Turing complete is just as powerful as the pen and paper system he thought of. Toasters, TI84s, airplanes, supercomputers, and the pen-paper system are all computationally-equivalent Turing machines.

Here is what matters to us: learning a new programming language is simple once one realizes that all languages are Turing equivalent. There is no "better" programming language in the sense that they can all do exactly the same things. Anything that is programmed in Java can be programmed in Python can be programmed in C can be programmed in Haskell can be "programmed" on the pen-paper Turing machine. So there are just three requirements a programming language must meet to be Turing complete:

1. State $\rightarrow$ store and retrieve truth or values

2. Choice $\rightarrow$ do something whether a condition is true (conditional execution)

3. Loop → repeat something a number of times

These correspond to the actions of reading/writing, moving, and halting of the Turing machine. We will examine these in order, and show how they are achieved in Python.

2 What is Programming?

Programming is the art of taking what's in your mind and giving it a physical reality. Our first attempts as humans were artistic, probably mystical (cave art such as Grotte de Lascaux). In its modern incarnation, programming is writing that has symbols, rules for arranging these symbols, and a unique interpretation of these symbols. In the theoretical Turing machine, the symbols are 0 and 1. In fact, these binary symbols are still used today on modern computers. The rules for combining the symbols come from logic (think rules of inference). The freedom comes from how we decide to interpret the symbols—that's what makes each programming language different.

We will examine this more closely by showing how programs can model truth deductions in the real world. As rational beings, we are able to take prior truths and combine them using rules of logic to produce new truths:

$$\text{beliefs and truths} \models \text{new beliefs and truths}$$

We would then hope that this can be successfully modeled by the syntax of our programming language:

$$\text{symbols} \vdash \text{symbols}$$

where the resulting symbols are produced by the machine but logically interpreted by us. The "$\models$" and "$\vdash$" symbols denote semantic and syntactic consequence, respectively. The former details how truths produce new truths, and the later details how bits and symbols produce new bits and symbols. We formalize this with the concepts of completeness and soundness.

Definition. A programmed system of computation is **complete** if a logical deduction in the real world corresponds to a syntactic deduction in the system.

Definition. A programmed system of computation is **sound** if a syntactic deduction in the system corresponds to a logical deduction in the real world.

A complete programming language is one that could compute (prove) all truths. A sound programming language is one that could only compute (prove) truths; no false deductions could be made.

Every programming language is neither complete nor sound (an extension of Godel's Incompleteness Theorems). Let's look at an example of a real-world truth that does not hold in Python.

$$\textbf{Real world: } \sqrt{2} \times \sqrt{2} = 2^{\frac{1}{2}+\frac{1}{2}} \models 2^1 = 2$$

Python:

```
>>> ((2)**(0.5))**2
```

$\vdash$

```
2.0000000000000004
```

The real-world deduction that $\sqrt{2}^2 = 2$ does not hold in the structured world of Python. Further, Python was able to compute a falsity, that $\sqrt{2}^2 = 2.0000000000000004$. We must always understand that Python (and any programming language) is simply a model of computation that comes with its own logical limitations. In fact, our math friends from a hundred years ago could have told you that. This should not be a huge cause of concern however. There is still in incredible amount that can be achieved from programming even if it is neither complete nor sound. These same math friends would have their heads spinning if they could see the world of technology we live in today.

The art of programming is taking these logical deductions that we know to be true and modeling them to the best of our ability in the structured world. We then let the structured world make syntactic deductions; bits and symbols produce new bits and symbols based on the formal logic rules programmed into the system. Finally, we make use of these resulting bits and symbols by assigning valuable meaning to them. This is how computers aid us.

$$\textbf{Real world: } \text{truth \& behavior} \models \text{truth \& behavior} \models \text{truth \& behavior}$$

$$\textbf{Structured world: } \text{symbols} \vdash \text{symbols} \vdash \text{symbols}$$

You might be wondering how a computer works at all in the first place. The formal logic rules used by a computer couldn't be hard-coded via a programming language since they are the motor of all programming to begin with. It would be circular to suggest that the logic of a computer is programmed. How, then, are these rules known by a computer? The hardware of a computer contains extremely detailed circuit boards that control electronic impulses.

The direction, intensity, and flow of these impulses are entirely governed by the circuit board and the laws of physics. The ingenuity is the fact that every single circuit corresponds to a logical process we would like the computer to be able to make. For instance, `AND` is a logical operator used in mathematics; its corresponding circuit allows a computer to model the logic behind this operator. These circuits are contained in the Central Processing Unit (CPU) of a computer. When a program executes, the symbols are manipulated by the laws of physics via circuit boards. These manipulations themselves simulate the logical deductions that would be made in the real world. We will soon see examples of actual circuits modeled by the following logical mathematical operators:

- `AND` (multiplication, intersection)
- `OR` (addition, union)
- `NOT` (complement)
- `IF, THEN` (material implication, which is slightly different in programming)
- `NOR` (not or)
- `NAND` (not and)

The binary symbols 0 and 1 correspond to amounts of electricity. If the electricity flowing through a circuit gate is less than a given threshold, it represents a 0; if it is above the threshold, it represents a 1. Obviously these symbols must have a real-world interpretation—in fact, they correspond to falsity and truth themselves. The simple idea that binary can model logic is the reason for its ubiquitousness in computer science. A “bit" is simply a value of 0 or 1, that is, a statement that is false or true. When bits flow through circuits to produce new bits of information, prior truths are combined using the rules of logic to produce new truths. This concept is pertinent throughout all of computer science.

3 Introduction to Python

It is time to start applying these ideas to our first programs. Before that, let’s explain how one can actually code on their own computer. Installing a programming language is as simple as going to the language’s website and downloading the latest version. For Python, make sure you are using `ver.3.10` or higher. What you are downloading is a series of interpretations given to the different permutations of 0 and 1, as well as the rules used to combine these permutations. Together, these are known as the semantics and syntax of a programming language. When you execute a Python program, the Python interpreter will read through your code line-by-line and produce the corresponding output.

There are three ways to use Python. The first is to start an interactive session. Begin by opening a terminal (called a command prompt on Windows). Another name for the terminal is the shell, as the shell is the thinnest "layer" around the operating system and provides easy access to the computer's internal structure. In the terminal section, type the command "python" and hit enter. This runs Python itself as a program, allowing you to type instructions and have them directly executed:

```
PS C:\Users\ME> python
>>> happy = True
>>> know_it = True
>>> if happy and know_it:
...     print("Clap your hands")
...
Clap your hands
```

Don't worry about the Python yet, but take notice of the logical operators being used (if, and). Two variables are bound to the value of truth, and the computer's logical rules allow the conditional to execute, thus printing the message.

The second way is to run a completed program from the console. A program is simply a string of syntactically-valid text. Therefore, you can write your program in any text editor you want (Notepad, for example). Write and save the following program as "dino.py" (or any name) in an appropriate location on your computer:

```
happy = True
know_it = True
if happy and know_it:
    print("Clap your hands")
```

Use the file extension ".py" to signal to the computer that it is a Python program. Next, run the following command in a terminal. The completed program is sent to the Python interpreter in your computer, and the program is executed.

```
PS C:\Users\ME> python d:\Python\dino.py
Clap your hands
PS C:\Users\ME>
```

The output of the code is shown between the prompts. A terminal is continuously prompting, meaning it will always wait for more instructions after successfully executing a command.

The final way to run Python programs is to use an Integrated Development Environment (IDE). This is essentially a large-scale, fully-customizable platform to write, edit, and run code. Most IDEs will flag syntax errors and provide helpful debugging processes. This is the preferred way to write code for most people. A great IDE is Visual Studio Code (VSC), which is free and easy to set up. Go to the downloads page and follow the instructions to get started writing Python.

4 First Python Program

Let's get some practice with the different ways of using Python. We can start Python and write one statement at a time:

```
PS C:\Users\ME> python
>>> 1*2
2
>>> "Hello world"
'Hello world'
>>> print(1*2)
2
```

Notice that we use "*" for multiplication. We can also write a program (a series of instructions) and send it to Python all at once:

Listing 1: first.py

```
a = 1
b = 2
c = "Hello world"
print(a*b)
print(c)
```

```
PS C:\Users\ME> python d:\Python\first.py
2
Hello world
```

Going forward we will elect to show simpler code behavior in the terminal and more complex procedures as full programs.

Let's take a look at our first logical circuit, also called a truth table. The operation under consideration is `AND`. It is a two-place (or binary) operator, which means it takes two truth values as arguments. If p and q represent truth values (values that are either true or false), then $p \wedge q$ (read "p and q") represents the logical conjunction of these values. It is true only when both p and q are themselves true. We can represent this with the following table:

p	q	p $\wedge$ q
T	T	T
T	F	F
F	T	F
F	F	F

Notice that there are $2^2 = 4$ different rows of the truth table, corresponding to the 4 different permutations of {T, F}. Remember during the discussion about computer bits that 0 corresponds to False and 1 corresponds to True. This logical arithmetic (called Boolean algebra after the famous mathematician George Boole) can be performed in Python:

```
>>> True and False
False
>>> True and True
True
>>> 1 and 0
0
>>> 1 and 1
1
```

Let's now look at the truth table for the `OR` operator, also called logical disjunction. It is only false when both the inputs are false.

p	q	p $\vee$ q
T	T	T
T	F	T
F	T	T
F	F	F

Sit on the semantics of this operator for a minute. Clearly the sentence "the sky is blue or the moon landing was fake" is true because the sky is in fact blue. Is the sentence "the sky is blue or grass is green" also true? Many uses of "or" in human language are exclusive, meaning "one or the other is true, but not both." In mathematics and computer science, "or" is always an inclusive operator. So yes, the sky is blue or grass is green.

```
>>> True or False
True
>>> False or False
False
>>> 1 or 0
1
>>> 0 or 0
0
```

The next operator on our list is `NOT`, also called the complement. It is a unary operator whose truth value is opposite that of its argument.

p	$\neg p$
T	F
F	T

```
>>> not True
False
>>> not 1
0
```

```
def hello():
    print("Hello world")
```

Listing 2: Code

```
def hello():
    print("Hello world")
hello()
```

```
Hello world
```

```
def hello():
    print("Hello world")
```

```
$ python3 hello.py
Hello world
```

```
>>> def hello():
...     print("Hello world")
>>> hello()
Hello world
```