

Introduction to Python

1 Introduction

The purposes of this course are:

- Learn computational models of thinking.
- Master the art of computational problem solving.
- Make computers do what you want them to do.

1.1 What does a computer do?

Fundamentally, a computer does only two things:

1. It performs calculations.
2. It remember results.

One might be wondering how fast a computer performs calculations, so nowadays a computer can perform approximately a billion calculations per second. On the other hand, nowadays computers have a hard drive of nearly 1 TB which means that a computer can store approximately 1.5 million books of standard size.

1.2 What kind of operations does a computer perform?

Every computer comes with a set of built-in operations. These are typically primitive arithmetic operations such as addition, multiplication, division, and simple logic operations such as comparing true and false values in order make decisions.

Because this built-in tools are not enough, in this course one will learn how to define new calculations, new operations, and give them to the computer, so it can abstract them, encapsulate them, and treat them as they are primitives.

1.3 Simple calculations and storage enough?

It is very important to design good algorithms since simple tasks could take a lot of time, and results could require a lot of space to be stored. For example, searching on the world wide web using simple operations could take 5.2 days, or deciding a chess move without an optimal algorithm could take 30 minutes.

Considering the same chess problem, experts suggest that there are approximately 10^{123} different possible games. However, there are only 10^{80} atoms in the observable universe. Hence, it is simply not possible to store all different possible games in chess.

1.4 Are there limits?

Despite its speed and size, a computer has limitations. For instance, some problems are still too complex such as accurate weather prediction and cracking encryption schemes, or some problems are fundamentally impossible to compute as the famous Halting problem which basically asks to predict whether a piece of code will always halt with an answer for any input.

2 Types of Knowledge

It is possible to divide knowledge into two types:

1. *Declarative knowledge*: it is statements of fact, i.e., statements of truth.
2. *Imperative knowledge*: it is a recipe, i.e., how-to knowledge or how-to information. It gives a sequence of steps to find a solution.

Both kind of knowledge are important, but only the second one allows getting the computer to do something.

2.1 A numerical example

Declarative knowledge: the square root of a number x is y such that $y * y = x$. The previous sentence is a statement of fact. Though it is not telling how to find a square root, it is telling how to check if a number is the square root of another one.

Imperative knowledge: Heron of Alexandria's algorithm to find the square root of a number x (e.g. 16):

1. Start with a guess, g .
2. If $g * g$ is close enough to x , then stop and say g is the answer.

3. Otherwise, make a new guess by averaging g and x/g .
4. Using the new guess, repeat process until close enough.

g	$g * g$	x/g	$(g + x/g)/2$
3	9	5.333	4.1667
4.1667	17.36	3.837	4.0035
4.0035	16.0277	3.997	4.000002

Table 1: Heron's algorithm to find the square root of 16.

2.2 What is a recipe?

A recipe has:

1. A sequence of simple steps.
2. A flow of control, a process that specifies when each step is executed.
3. A way to decide when to stop.

The previous three pieces constitute what is known as an algorithm.

3 Machines

One might be wondering how to capture a recipe or algorithm in a mechanical process. Historically, there have been two options. The former is to use what is called as a *fixed program computer*, i.e., a computer specifically designed to calculate a particular computation. For example, handheld calculators have been fixed to do addition, subtraction, multiplication this is simple set of arithmetic operations, and that's all they can do.

The alternative to a fixed program computer is a *stored program computer*. One can load into this computer an algorithm, and inside of it, a set of parts are going to execute those instructions whenever one wants. Inside a stored program computer there is a special program known as the interpreter which is going to walk through each of those sequence of instructions doing the required computation. The advantage is that now one can load any algorithm without the need of having to build another computer to run the new program.

3.1 Basic Machine Architecture

Basically, a machine has memory which a place where to store information. This information could be data but it also can be a sequence of instructions

that constitute an algorithm. A machine will also have a way to load things into it and to print data out of it, i.e., input and output.

Inside the heart of the machine there are two elements. The former is called A.L.U. (Arithmetic Logic Unit) which takes information from memory, reads it in to do perform a primitive operation (as addition or multiplication), and then it will store stuff back up into memory. The latter is called control unit which keeps track of the specific operation that has to be computed in the A.L.U. at each point in time.

Inside the control unit there is an important element called program counter. When loading a program into a machine, the program counter points to the location of the first instruction. When one asks the machine to execute the program, the program counter reads that first instruction, so it will cause an operation to take place, and it will also add one to the program counter which is going to take it to the next instruction in the sequence and repeat the process. At some point, a test might be reached, and the value of this test is going to change the program counter. Eventually, something is going to indicate to stop the program.

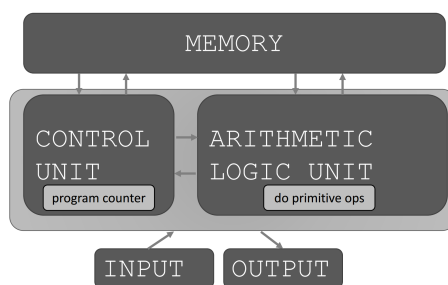


Figure 1: Basic Machine Architecture

3.2 Stored Program

A stored program is a sequence of instructions which is built out of simple arithmetic instructions, logic instructions and simple tests, and it also allows moving data around. This program has associated with it a special program called the interpreter which is going to execute each of those sequences of instructions in order according to the flow of control, and it will also terminate the program when indicated.

3.3 Basic Primitives

Most machines come with simple arithmetic and logic operations, but it is possible to go simpler than this. Alan Turing showed that one can compute anything just with six primitives this principle is known as Turing completeness. In fact, there is something called a Turing machine that is an infinite tape with a set of squares on it. In each square, there is a symbol that could be either 0 or 1.

What Turing showed is that if you have six operations and those are: move left, move right, print, scan, erase, and do nothing, one can compute anything that is computable. Fortunately, programming languages come with a more convenient set of primitives, but the fundamental idea is that *with a simple set of primitives, one is able to compute anything*.

The power of computational thinking relies on being able to write descriptions, i.e., to abstract methods in order to create new primitives. There is a fundamental property that says that anything that is computable in one programming language is computable in another programming language. What is different is that some tasks could be easily done in some programming languages than in others. This is also a proof of Turing completeness.

4 Languages

4.1 Creating Recipes

A programming language provides a set of primitive operations, and the next step is to put them together. In order to do that, one uses expressions which are complex but legal combinations of primitives. Any legal expression in a programming language, any computation, has associated with it a value.

4.2 Aspects of Languages

While in natural languages such as English the primitive constructs are words, in a programming language the primitive constructs are *numbers*, sequences of characters a.k.a. *strings*, and *simple operations* such as addition, subtraction, and comparison.

Once again, when having those primitives, one has to put them together considering both syntax and semantics. *Syntax* tells whether or not an expression or combination of expressions is legal. Associated with every expression that is syntactically valid is the meaning a.k.a semantics. On one hand, *static semantics* tells which syntactically valid expressions have meaning. On the other hand, *semantics* is the meaning associated with that syntactically correct string of symbols with no semantic errors. In order to clarify these three concepts *syntax*, *static semantics* and *semantics* see table 2.

Sentence	Syntax	Static Sem.	Semantics
I have a dog.	✓	✓	✓
Tim saw Jay with a telescope.	✓	✓	✗
I are hungry	✓	✗	no eval
cat dog boy	✗	no eval	no eval

Table 2: Syntax vs. static semantics vs semantics

Though in English it is possible to have multiple meaning for a sentence, as in the sentence *Tim saw Jay with a telescope*, in programming languages an expression has only one meaning. However, the meaning could be not the one that the programmer intended.

4.3 Where Things Go Wrong

Even when a programmer keeps in mind syntax, static semantics and semantics, things can go wrong. For example, he can encounter *syntactic errors* which are common but easy to find since most programming languages provide a way to catch them. In addition, a programmer might encounter *static semantic errors*, i.e., things that they are in the right order, but they don't make sense. Some programming languages check for them before running a program, but other languages, as Python, do it on the fly.

The bigger problem is when there are no semantic errors, but the programmer gets a different meaning than what he expected. There are different possible consequences such as the program can crash or stop running, the program can run forever, or even worse, the program gives an answer that is different than the expected.