Programming with Python for Engineers

Release 0.0.1

Sinan Kalkan, Onur Tolga Sehitoglu, Gokturk Ucoluk

Oct 10, 2020

Contents

Preface

i. About this book

a. Target audience of the book

This book is intended to be an accompanying textbook for teaching programming to science and engineering students with no prior programming expertise. This endeavour requires a delicate balance between providing details on computers & programming in a complete manner and the programming needs of science and engineering disciplines. With the hopes of providing a suitable balance, the book uses Python as the programming language, since it is easy to learn and program. Moreover, for keeping the balance, the book is formed of three parts:

- **Part I: The Basics of Computers and Computing**: The book starts with what computation is, introduces both the present-day hardware and software infrastructure on which programming is performed and introduces the spectrum of programming languages.
- **Part II: Programming with Python**: The second part starts with the basic building blocks of Python programming and continues with providing the ground formation for solving a problem in to Python. Since almost all science and engineering libraries in Python are written with an object-oriented approach, a gentle introduction to this concept is also provided in this part.
- **Part III: Using Python for Science and Engineering Problems**: The last part of the book is dedicated to practical and powerful tools that are widely used by various science and engineering disciplines. These tools provide functionalities for reading and writing data from/to files, working with data (using e.g. algebraic, numerical or statistical computations) and plotting data. These tools are then utilized in example problems and applications at the end of the book.

b. How to use the book

This is an 'interactive' book with a rather 'minimalist' approach: Some details or specialized subjects are not emphasized and instead, direct interaction with examples and problems are encouraged. Therefore, rather than being a 'complete reference manual', this book is a 'first things first' and 'hands on' book. The pointers to skipped details will be provided by links in the book. Bearing this in mind, the reader is strongly encouraged to read and interact all contents of the book thoroughly.

The book's interactivity is thanks to Jupyter notebook[1]. Therefore, the book differs from a conventional book by providing some dynamic content. This content can appear in audio-visual form as well as some applets (small applications) embedded in the book. It is also possible that the book asks the the reader to complete/write a piece of Python program, run it, and inspect the result, from time to time. The reader is

[1] https://jupyter.org

encouraged to complete these minor tasks. Such tasks and interactions are of great assistance in gaining acquaintance with Python and building up a self-confidence in solving problems with Python.

Thanks to Jupyter notebook running solutions on the Internet (e.g. Google Colab[2], Jupyter Notebook Viewer[3]), there is absolutely no need to install any application on the computer. You can directly download and run the notebook on Colab or Notebook Viewer. Though, since it is faster and it provides better virtual machines, the links to all Jupyter notebooks will be served on Colab.

ii. What is computing?

Computing is the process of inferring data from data. What is going to be inferred is defined as the *task*. The original data is called the *input (data)* and the inferred one is the *output (data)*.

Let us look at some examples:

- Multiplying two numbers, X and Y, and subtracting 1 from the multiplication is a *task*. The two numbers X and Y are the *input* and the result of $X \times Y - 1$ is the *output*
- Recognizing the faces in a digital picture is a *task*. Here the *input* is the color values (3 integers) for each point (pixel) of the picture. The *output* is, as you would expect, the pixel positions that belong to faces. In other words, the output can be a set of numbers.
- The board instance of a chess game, as *input*, where black has made the last move. The task is to predict the best move for white. The best move is the *output*.
- The *input* is a set of three-tuples which look like **<Age_of_death, Height, Gender>**. The *task*, an optimization problem in essence, is to find out the curve (i.e. the function) that goes through these tuples in a 3D dimensional space spanned by Age, Height and Gender. As you have guessed already, the *output* is the parameters defining the function and an error describing how well the curve goes through the tuples.
- The *input* is a sentence to a chatbot. The *task* is to determine the sentence (the *output*) that best follows the input sentence in a conversation.

These examples suggest that computing can involve different types of data, either as input or output: Numbers, images, sets, or sentences. Although this variety might appear intimidating at first, we will see that, by using some 'solution building blocks', we can do computations and solve various problems with such a wide spectrum of data.

iii. Are all 'computing machinery' alike?

Certainly not! This is a common mistake a layman does. There are diverse architectures based on totally different physical phenomena that can compute. A good example is the *brain* of living beings, which rely on completely different mechanisms compared to the *micro processors* sitting in our laptops, desktops, mobile phones and calculators.

The building blocks of a *brain* is the *neuron*, a cell that has several input channels, called *dendrites* and a single output channel, the *axon*, which can branch like a tree (see Fig. 1).

[2] https://colab.research.google.com/

[3] https://nbviewer.jupyter.org/

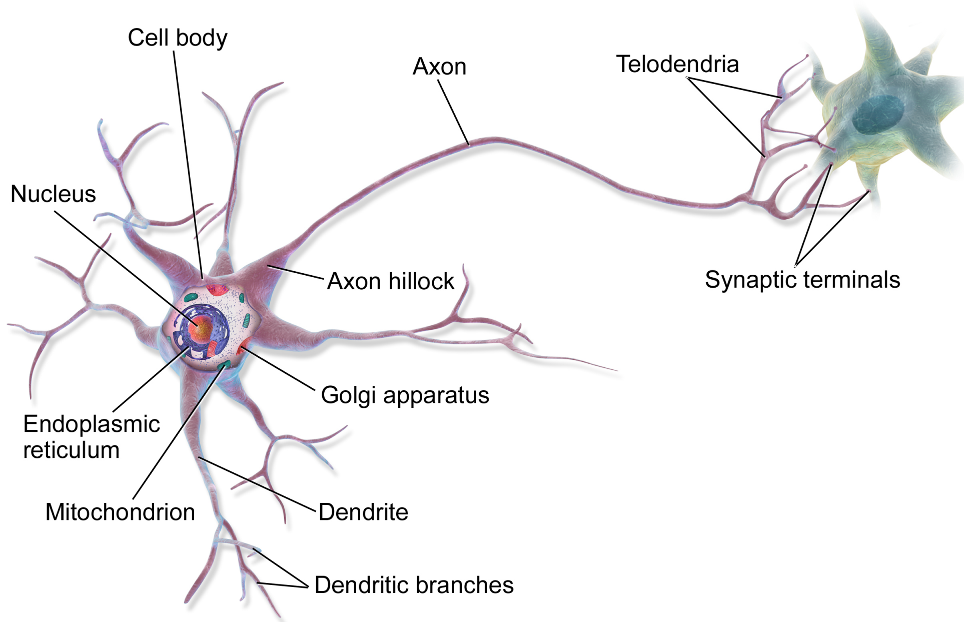

Fig. 1: Our brains are composed of simple processing units, called neurons. Neurons receive signals (information) from other neurons, process those signals and produce an output signal to other neurons. [Drawing by BruceBlaus - Own work, CC BY 3.0, https://commons.wikimedia.org/w/index.php?curid=28761830]

The branches of an axon, each carrying the same information, connect to other neurons' dendrites (Fig. 2). The connection with another neuron is called the *synapse*. What travels through the synapse are called *neurotransmitters*. Without going into details, one can simplify the action of neurotransmitters as messengers that cause an excitation or inhibition on the receiving end. In other words, the neurotransmitters, through a chemical process along the axon, are released into the synapse as the 'output' of the neuron, they 'interact' with the dendrite (i.e. the 'input') of another neuron and potentially lead to an excitation or an inhibition.

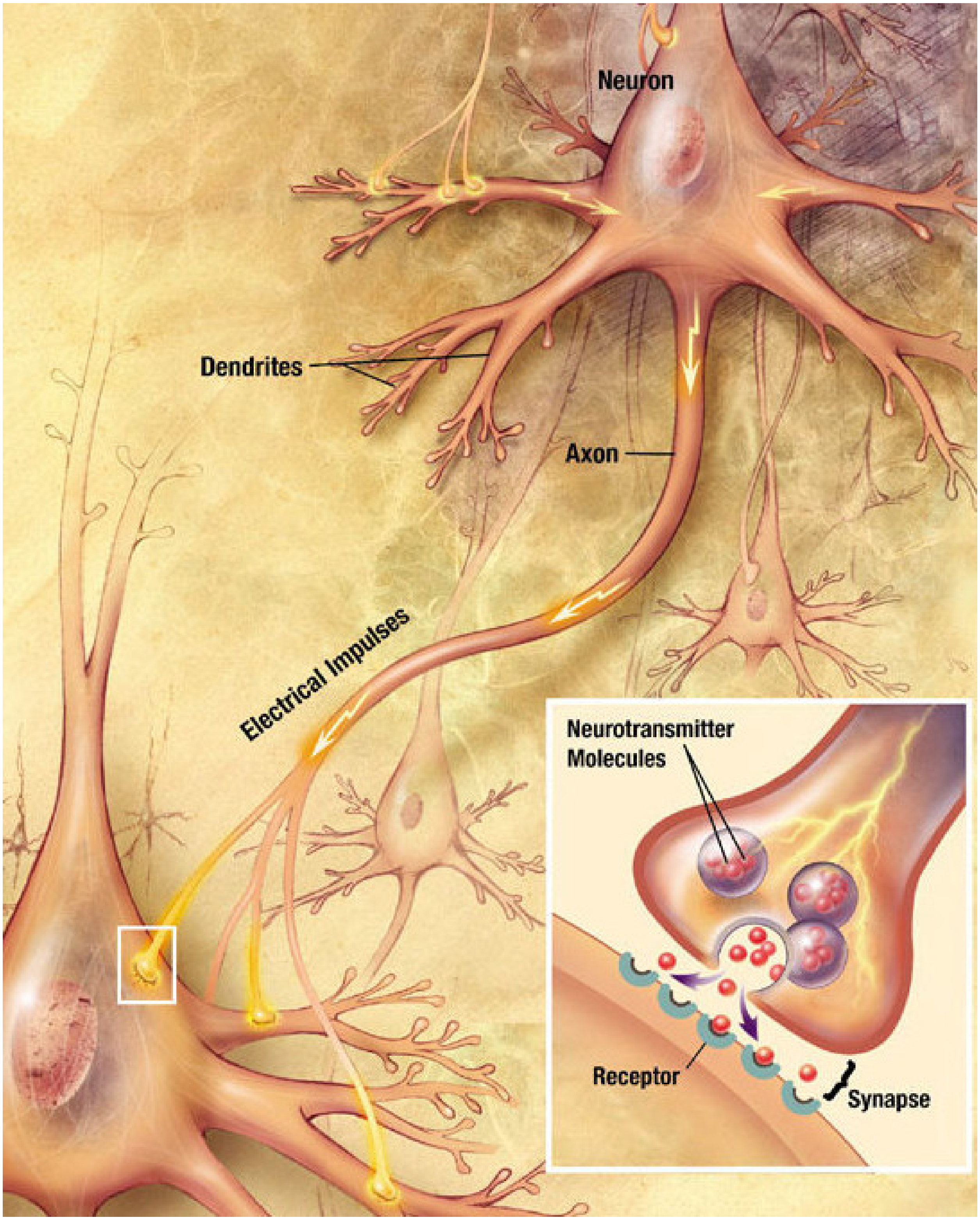

Fig. 2: Neurons 'communicate' with each other by transmitting neurotransmitters via synapses. [Drawing by user:Looie496 created file, US National Institutes of Health, National Institute on Aging created original - http://www.nia.nih.gov/alzheimers/publication/alzheimers-disease-unraveling-mystery/preface, Public Domain, https://commons.wikimedia.org/w/index.php?curid=8882110]

Interestingly enough, the synapse is like a valve, which reduces the neurotransmitters' flow. We will come to this in a second. Now, all the neurotransmitters flown in through the input channels (dendrites) have an accumulative effect on the (receiving) neuron. The neuron emits a neurotransmitter burst through its axon. This emission is not a 'what-comes-in-goes-out' type. It is more like the curve in Fig. 3.

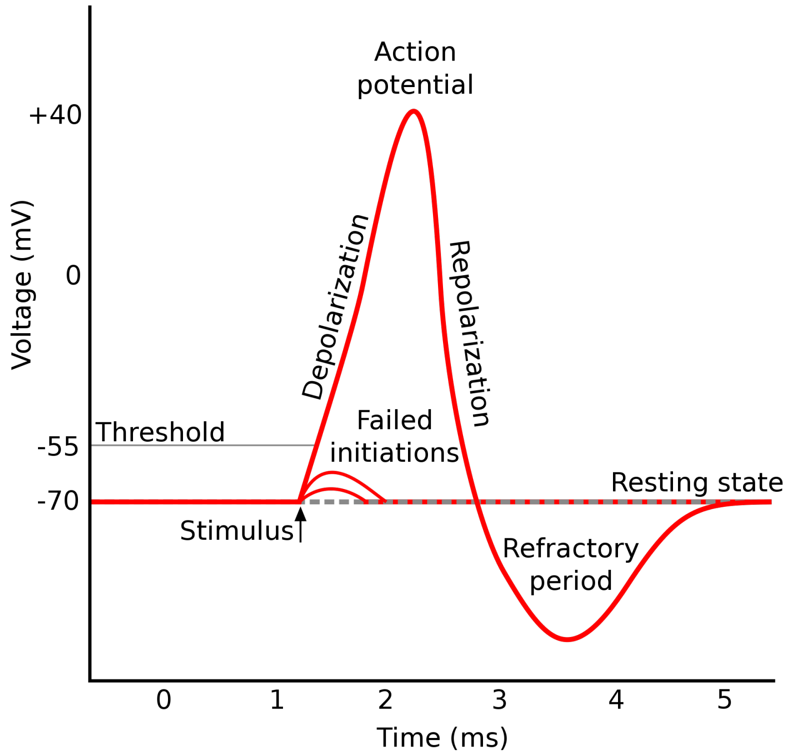

Fig. 3: When a neuron receives 'sufficient' amount of signals, i.e. stimulated, it emits neurotransmitters on its axon, i.e. it fires. [Plot by Original by en:User:Chris 73, updated by en:User:Diberri, converted to SVG by tiZom - Own work, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=2241513]

The throughput of the synapse is something that may vary with time. Most synapses have the ability to ease the flow over time if the neurotransmitter amount that entered the synapse was constantly high. High activity widens the synaptic connection. The reverse also happens: Less activity over time narrows the synaptic connection.

Some neurons are specialized in creating neurotransmitter emission under certain physical effects. Retina neurons, for example, create neurotransmitters if light falls on them. Some, on the other hand, create physical effects, like creating an electric potential that will activate a muscle cell. These specialized neurons feed the huge neural net, the brain, with inputs and receive outputs from it.

The human brain, containing about 10^{11} such neurons with each neuron being connected to 1000-5000 other neurons by the mechanism explained above, is a very unique computing 'machine' that inspires computational sciences.

A short video on synaptic communication between neurons

The brain never stops processing information and the functioning of each neuron is only based on signals (the neurotransmitters) it receives through its connections (dendrites). There is no common synchronization timing device for the computation: i.e. each neuron behaves on its own and functions in parallel.

An interesting phenomenon of the brain is that the information and the processing are distributed. Thanks to this feature, when a couple of neurons die (which actually happens each day) no information is lost completely.

On the contrary to the brain, which uses varying amounts of chemicals (neurotransmitters), the microprocessor based computational machinery uses the existence and absence of an electric potential. The information is stored very locally. The microprocessor consists of subunits but they are extremely specialized in function and far less in number compared to 10^{11} all alike neurons. In the brain, changes take place at a pace of 50 Hz maximum, whereas this pace is 10^9 Hz in a microprocessor.

In Chapter 1, we will take a closer look at the microprocessor machinery which is used by today's computers. Just to make a note, there are man-made computing architectures other than the microprocessor. A few to mention would be the 'analog computer', the 'quantum computer' and the 'connection machine'.

iv. What is a 'computer'?

As you have already noticed, the word 'computer' is used in more than one context.

1. **The broader context:** Any physical entity that can do 'computation'.
2. **The most common context:** An electronic device that has a 'microprocessor' in it.

From now on, 'computer' will refer to the second meaning, namely a device that has a 'microprocessor'.

A computer…

- is based on binary (0/1) representations such that all inputs are converted to 0s and 1s and all outputs are converted from 0/1 representations to a desired form, mostly a human-readable one. The processing takes places on 0s and 1s, where 0 has the meaning of 'no electric potential' (no voltage, no signal) and 1 has the meaning of 'some fixed electric potential (usually 5 Volts, a signal).
- consists of two clearly distinct entities: The Central Processing Unit (CPU), also known as the microprocessor (□P), and a *Memory*. In addition to these, the computer is connected to or incorporates other electronic units, mainly for input-output, known as 'peripherals'.
- performs a 'task' by executing a sequence of instructions, called a 'program'.
- is deterministic. That means if a 'task' is performed under the same conditions, it will produce always the same result. It is possible to include randomization in this process only by making use of a peripheral that provides electronically random inputs.

v. What is programming?

The CPU (the microprocessor - □P) is able to perform several types of actions:

- Arithmetic operations on binary numbers that represent (encode) integers or decimal numbers with fractional part.
- Operations on binary representations (like shifting of digits to the left or right; inverting 0s and 1s).
- Transferring to/from memory.
- Comparing numbers (e.g. whether a number n_1 larger than n_2) and performing alternative actions based on such comparisons.
- Communicating with the peripherals.

- Alternating the course of the actions.

Each such unit action is recognized by the CPU as an *instruction*. In more technically terms, tasks are solved by a CPU by executing a sequence of instructions. Such sequences of instructions are called machine codes. Constructing machine codes for a CPU is called ‘machine code programming’.

But, programming has a broader meaning:

> *a series of steps to be carried out or goals to be accomplished.*

And, as far as computer programming is concerned, we would certainly like these steps to be expressed in a more natural (more human readable) manner, compared to binary machine codes. Thankfully, there exist ‘machine code programs’ that read-in such ‘more natural’ programs and convert them into ‘machine code programs’ or immediately carry out those ‘naturally expressed’ steps.

Python is such a ‘more natural way’ of expressing programming steps.

Basic Computer Organization

In this chapter, we will provide an overview of the internals of a modern computer. To do so, we will first describe a general architecture on which modern computers are based. Then, we will study the main components and the principles that allow such machines to function as general purpose "calculators".

The von Neumann Architecture

Basic Computer Organization. In this chapter, we will provide an overview of the internals of a modern computer. To do so, we will first describe a general architecture on which modern computers are based. Then, we will study the main components and the principles that allow such machines to function as general purpose "calculators".

John von Neumann

From: Oxford Reference[4]

"Hungarian-born US mathematician, creator of the theory of games and pioneer in the development of the modern computer. Born in Budapest, the son of a wealthy banker, von Neumann was educated at the universities of Berlin, Zürich, and Budapest, where he obtained his PhD in 1926. After teaching briefly at the universities of Berlin and Hamburg, von Neumann moved to the USA in 1930 to a chair in mathematical physics at Princeton. In 1933, he joined the newly formed Institute of Advanced Studies at Princeton as one of its youngest professors. By this time he had already established a formidable reputation as one of the most powerful and creative mathematicians of his day. In 1925 he had offered alternative foundations for set theory, while in his Mathematischen Grundlagen der Quantenmechanik (1931) he removed many of the basic doubts that had been raised against the coherence and consistency of quantum theory. In 1944, in collaboration with Oskar Morgenstern (1902–77), von Neumann published The Theory of Games and Economic Behaviour. A work of great originality, it is reputed to have had its origins at the poker tables of Princeton and Harvard. The basic problem was to show whether it was possible to speak of rational behaviour in situations of conflict and uncertainty as in, for example, a game of poker or wage negotiations. In 1927 von Neumann proved the important theorem that even in games that are not fully determined, safe and rational strategies exist. With entry of the USA into World War II in 1941 von Neumann, who had become an American citizen in 1937, joined the Manhattan project (for the manufacture of the atom bomb) as a consultant. In 1943 he became involved at Los Alamos on the crucial problem of how to detonate an atom bomb. Because of the enormous quantity of computations involved, von Neumann was forced to seek mechanical aid. Although the computers he had in mind could not be made in 1945, von Neumann and his colleagues began to design Maniac I (Mathematical analyser, numerical integrator, and computer). Von Neumann was one of the first to see the value of a flexible stored program: a program that could be changed quite easily without altering the computer's basic circuits. He went on to consider deeper problems in the theory of logical automata and

[4] https://www.oxfordreference.com/view/10.1093/oi/authority.20110803120234729

finally managed to show that self-reproducing machines were theoretically possible. Such a machine would need 200 000 cells and 29 distinct states. Having once been caught up in affairs of state von Neumann found it difficult to return to a purely academic life. Thereafter much of his time was therefore spent, to the regret of his colleagues, advising a large number of governmental and private institutions. In 1954 he was appointed to the Atomic Energy Commission. Shortly after this, cancer was diagnosed and he was forced to struggle to complete his last work, the posthumously published The Computer and the Brain (1958)."

Fig. 4: John von Neumann (1903 – 1957)

Components of the von Neumann Architecture

The von Neumann architecture (Fig. 5) defines the basic structure, or outline, used in most computers today. Proposed in 1945 by Von Neumann, it consists of two distinct units: An *addressable memory* and a *Central Processing Unit* (CPU). All the encoded actions and data are stored together in the memory unit. The CPU, querying these actions, the so-called *instructions*, executes them one by one, sequentially (though, certain instructions may alter the course of execution order).

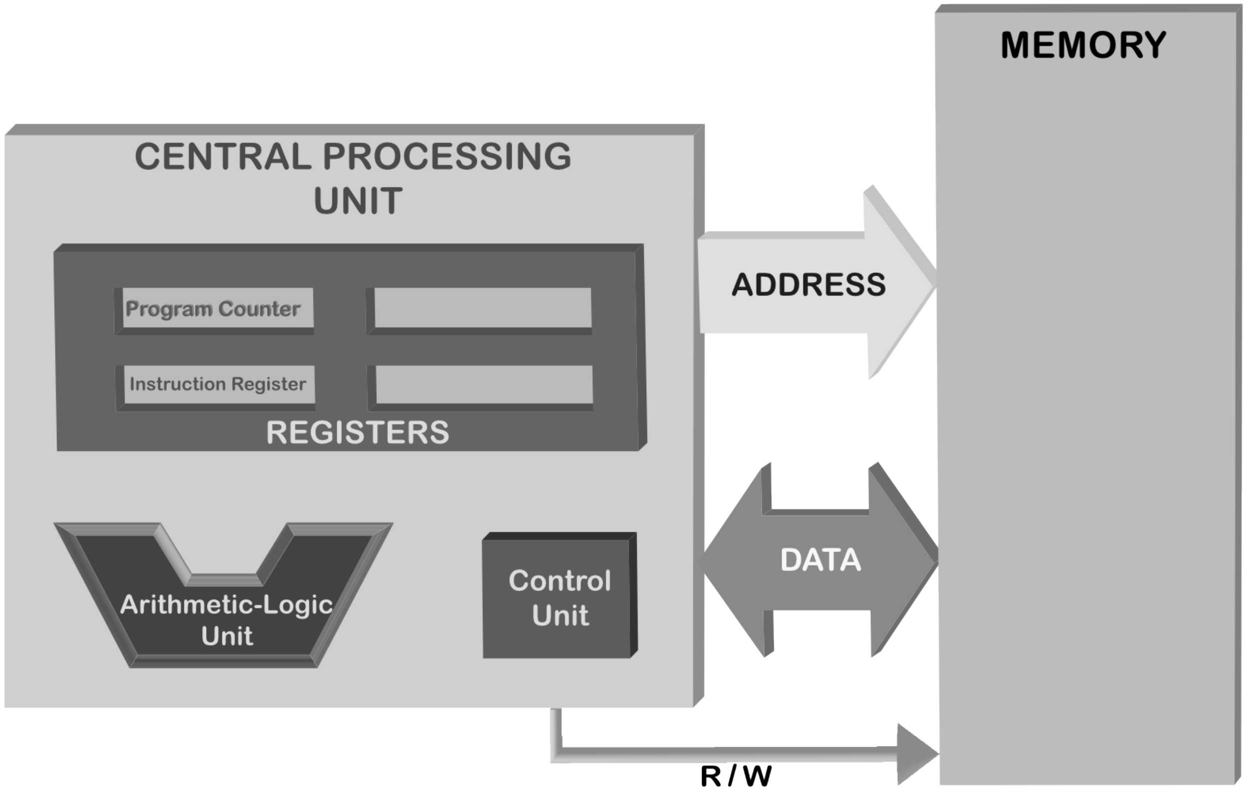

Fig. 5: A block structure view of the von Neumann Architecture.

The CPU communicates with the memory by two sets of wires, namely the *address bus* and the *data bus*, plus a single *R/W* wire (Fig. 5). These busses consist of several wires and carry a binary information to/from the memory. Each wire in a bus carries one bit of the information (either a zero (0) or a one(0)). Todays von Neumann architecures are working on electricity, therefore, these zeros and ones correspond to Voltages. A one in usally the presences of a 5V and a zero is the absense of it.

The Memory

The memory can be imagined as pigeon holes organized as rows (Fig. 6). Each row has eight pigeon holes, each being able to hold a zero (0) or one (1) – in electronic terms, each of them is capable of storing a Voltage (can you guess what type of an electronical component a pigeon hole is?). Each such row is named to be of the size *byte*. In computer terms, a byte means 8 bits.

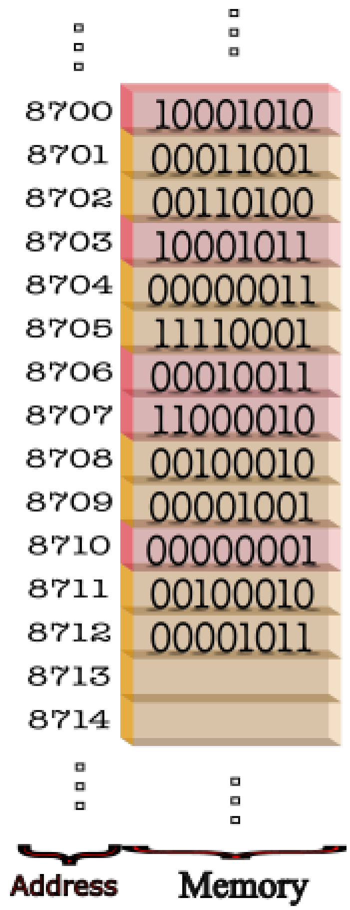

Fig. 6: The memory is organized as a stack of rows such that each row has an associated address.

Each byte of the memory has a unique address. When the address input (also called address bus – Fig. 5) of the memory is provided a binary number, the memory byte that has this number as the address becomes accessible through the data output (also called output data bus). Based on W/R wire being set to Write (1) or Read (0), the action that is carried out on the memory byte differs:

- **W/R wire is set to WRITE (1)** :

  The binary content on the input data bus is copied into the 8-bit location whose address is provided on the address bus, the former content is *overwritten*.

- **W/R wire is set to READ (0)** :

  The data bus is set to a copy of the content of 8-bit location whose address is provided on the address bus. The content of the accessed byte is left intact.

The information stored in this way at several addresses live in the memory happily, until the power is turned off.

The memory is also referred as Random Access Memory (RAM). Some important aspects of this type of memory has to be noted:

- Access any content in RAM, whether for reading or writing purposes, is *only* possible when the content's address is provided to the RAM through the address bus.
- Accessing any content takes exactly the same amount of time, irrespective of the address of the content. In todays RAMs, this access time is around 50 nanoseconds.
- When a content is overwritten, it is gone forever and it is not possible to undo this action.

An important question is who sets the address bus and who communicates through the data bus (sends and receives bytes of data). As depicted in Fig. 5, the CPU does. How this is done on the CPU side will become clear in the section below.

The CPU

The Central Processing Unit, which can be considered as the 'brain' of a computer, consists of the following units:

- **Control Unit (CU)**, which is responsible for fetching instructions from the memory, interpreting ('decoding') them and executing them. After executing an instruction finishes, the control unit continues with the next instruction in the memory. This "fetch-decode-execute" cycle is constantly executed by the control unit.
- **Arithmetic Logic Unit (ALU)**, which is responsible for performing arithmetic (addition, subtraction, multiplication, division) and logic (less-than, greater-than, equal-to etc.) operations. CU provides the necessary data to ALU and the type of operation that needs to be performed, and ALU executes the operation.
- **Registers**, which are mainly storage units on the CPU for storing the instruction being executed, the affected data, the outputs and temporary values.

The size and the quantity of the registers differ from CPU model to model. They generally have size in a range of [2-64] bytes and today's most popular CPUs most registers have size 64 bits (i.e. 8 bytes). Their quantity is not high and in the range of [10-20]. The registers are broadly categorized into two: *Special Purpose Registers* and *General Purpose Registers*.

Two special purpose registers are worth mentioning to understand how a CPU's Fetch-Decode-Execute cycle runs. The first is the so-called *Program Counter* (PC) and the second is the *Instruction Register* (IR).

- **Inputs/Outputs connections**, which connect the CPU to the other components in the computer.

The Fetch-Decode-Execute Cycle

The CPU is in fact a *state machine*, a machine that has a representation of its current *state*, reads the next instruction and executes the instruction according to its current state. The state consists what is stored in the registers. Until it is powered off, it follows the Fetch-Decode-Execute cycle (Fig. 7) where each step of the cycle is based on its state. The *control unit* is responsible for delicate cycle.

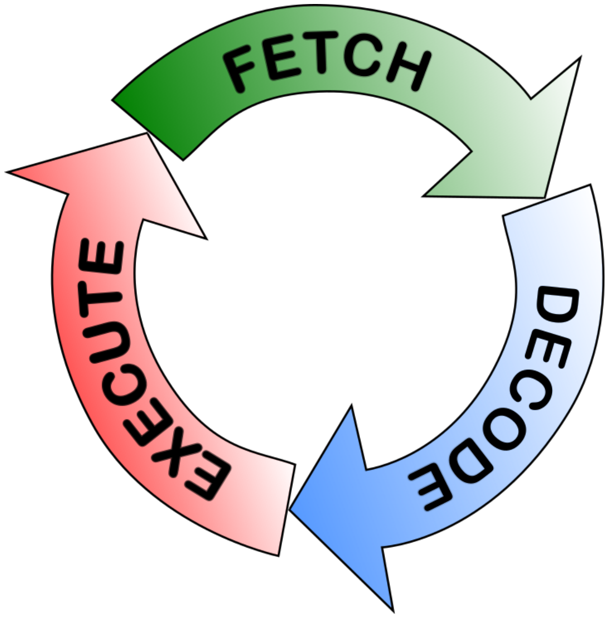

Fig. 7: The CPU constantly follows the fetch-decode-execute cycle while the computer is running a program.

- **The Fetch Phase**

  The cycle starts with the Fetch Phase. The CPU has the address (the position in the memory) of the next instruction in the PC (Program Counter) Register. During this phase, the address bus is set to this value in the PC register and the R/W wire is set to Read (0). The memory responds to this by providing the memory content at the given address on the data bus.

  How many bytes are sent through the data bus? That is architecture dependent. Usually it is 4-8 bytes. These bytes are received into the IR (Instruction Register).

- **The Decode Phase**

  The content of the first part of IR electronically triggers some action. Every CPU has an electronically built-in hard-wired instruction table in which every possible atomic instruction that the CPU can carry out has an associated binary code, called *operation code* (opcode in short). This table differs from CPU brand to brand.

  An example for such an atomic instruction (simply called as *instruction*) is integer addition on two registers. Prior to the instruction, the two registers contain integers, and after the instruction is executed, one of the registers will be incremented by the amount of the other (by means of integer addition).

  In the decode phase, the electronic activation of the specific electronic circuitry, a subpart of the ALU, that will carry out the instruction is performed. In contrary to the 'addition' example we just introduced, some instructions may require additional data. For example, loading a register with an certain integer or floating point value; or transferring a result obtained in a register to a certain place in the memory. Where does this additional data come from? (the integer or floating point in the first case; the address of the place that will receive the resulting content of the register, in the second case). This type of data adjunct to an instruction follows right after the instruction byte.

  This is illustrated below in an 8-byte instruction. In this example, the designer of the CPU allocated the first four bits for representing the type of operation the instruction is supposed to execute (e.g. the designer could design the CPU such that 0001 means that this is an instruction for reading data from memory) and the remaining four bits contain the affected data (in this case, the address of the memory content we are going to read).

Opcode	Affected data or address
0001	0100

There are three types of instructions:

- *Data manupulation*: Arithmetic/Logic operations on/among registers,
- *Data transfer*: Memory-to-Register, Register-to-Memory, Register-to-Register transfers,
- *Control flow of execution*: Instructions that stop execution, jump to a different part of the memory for next instruction, instead of the next one in the memory.
- **The Execute Phase**

  As the name implies, the electronically activated and initialized circuitry carries out the instruction. Depending on the instruction, the registers, the memory or other components are affected. When the instruction completes, the PC is updated by one unless it was a control flow changing instruction in which case PC is updated to point to the to-be-jumped address in the memory. Not all instructions take the same amount of time to be carried out. Floating point division, for example, takes much more time compared to others.

A CPU goes through the *Fetch-Decode-Execute* cycle until it is powered off. What happens at the very beginning? The electronics of the CPU is manufactured such that, when powered up, the PC register has a very fixed content. Therefore, the first instruction is always fetched from that position.

An intelligent question would be "when does the CPU jump from one state to another?". One possible answer is: whenever the previous state is completed electronically, a transition to the next state is performed. Interestingly, this is *not* true. The reality is that there is an external input to the CPU from which electronic pulses are fed. This input is called the *system clock* and each period of it is named as a *clock cycle*. The best performance would be that each phase of the fetch-decode-execute cycle is completed in one-and-only-one clock cycle. On modern CPUs, this is true for addition instruction, for example. But there are instructions (like floating point division) which take about 40 clock cycles.

What is length of a clock cycle? CPUs are marked with their *clock frequency*. For example, Intel's latest processor, i9, has a maximal clock frequency of 5GHz (that is 5×10^9 pulses per second). So, since (period) $=$ $1/$(frequency), for this processor a clock cycle is 200 pico seconds. This is such a short time that light would travel only 6 cm.

A modern CPU has many more features and functional components: *interrupts, ports, various levels of caches* are a few of them. To cover them is certainly out of the scope of this course material.

The Stored Program Concept

In order for the CPU compute something, the corresponding instructions to do the computation have to be placed into the memory (how this is achieved will become clear in the next chapter). These instructions and data that perform a certain task are called a *Computer Program*. The idea of storing a computer program into the memory to be executed is coined as the *Stored Program* Concept.

What does a stored program look like? Below you see a real extract from the memory, a program that multiplies two integer numbers sitting in two different locations in the memory and stores the result in another memory location (to save space consecutive 8 bytes in the memory are displayed in a row, the next row displays the next 8 bytes):

```
01010101 01001000 10001001 11100101 10001011 00010101 10110010 00000011
00100000 00000000 10001011 00000101 10110000 00000011 00100000 00000000
00001111 10101111 11000010 10001001 00000101 10111011 00000011 00100000
00000000 10111000 00000000 00000000 00000000 00000000 11001001 11000011
...
11001000 00000001 00000000 00000000  00000000 00000000
```

Unless you have a magical talent, this should not be understandable to you. It is difficult because it is just a sequence of bytes. Yes the first byte is presumably some instruction, but what is it? Furthermore, since we do not know what it is, we do not know whether it is followed by some data or not, so we cannot say where the second instruction starts. However, the CPU for which these instructions were written for would know this, hard-wired in its electronics.

When a programmer wants to write a program at this level, i.e. in terms of binary CPU instructions and binary data, s/he has to understand and know each instruction the CPU can perform, should be able to convert data to some internal format, make a detailed memory layout on paper and then start writing down each bit of the memory. This is an extremely painful job; though it is possible, it is impractical.

Alternatively, consider the text below:

```
main:
    pushq   %rbp
    movq    %rsp, %rbp
    movl    alice(%rip), %edx
    movl    bob(%rip), %eax
    imull   %edx, %eax
    movl    %eax, carol(%rip)
    movl    $0, %eax
    leave
    ret
alice:
    .long   123
bob:
    .long   456
```

Though pushq and moveq are not immediately understandable, the rest of the text provides some hints. alice and bob must be some programmer's name invention, something like variables with values 123 and 456 respectively; imull must have something to do with 'multiplication', since only registers can be subject to arithmetic operations %edx and %eax must be some denotation used for registers; having uncovered this, movls start to make some sense: they are some commands to move around data… and so on. Even without knowing the instruction set, with a small mind gaming we can uncover the action sequence.

This text is an example *assembly* program. A human invented denotation for instructions and data. An important piece of knowledge is that each line of the assembler text corresponds to a single instruction. This assembly text is so clear that even manual conversion to the cryptic binary code above is feasible. Form now on, we will call the binary code program as a *Machine Code Program* (or simply the *machine code*).

How do we automatically obtain machine codes from assembly text? We have machine code programs that convert the assembly text into machine code. They are called *Assemblers*.

Despite making programming easier for programmers, compared to machine codes, even assemblers are insufficient for efficient and fast programming. They lack some high-level constructs and tools that are necessary for solving problems easier and more practical. Therefore higher level languages that are much easier to read and write compared to assembly are invented.

We will cover the spectrum of programming languages in more detail in the next chapter.

Pros and Cons of the von Neuman Architecture

Advantages

- CPU retrieves data and instruction in the same manner from a single memory device. This simplifies the design of the CPU.
- Data from input/output (I/O) devices and from memory are retrieved in the same manner. This is achieved by mapping the device communication electronically to some address in the memory.
- The programmer has a considerable control of the memory organization. So, s/he can optimize the memory usage to its full extend.

Disadvantages

- Sequential instruction processing nature makes parallel implementations difficult. Any parallelization is actually a quick sequential change in tasks.
- The famous *"Von Neumann bottleneck"* : Instructions can only be carried out one at a time and sequentially.
- Risk of an instruction being unintentionally overwritten due to an error in the program.

Peripherals of a computer

Though it is somewhat contrary to your expectation any device outside of the Von Neumann structure, namely the CPU and the Memory is a *peripheral*. In this aspect even the keyboard, the mouse and the display are peripherals. So are the USB and ethernet connection and the internal hard disk. To study the technical details of how those devices are connected to the Von Neumann architecture is out of the scope of this book. Though we can summarize it in a few words.

All devices are electronically listening to the busses (the address and data bus) and to a wire running out of the CPU (which is *not* pictured above) is 1 or 0. This wire is called the *port_io* line and tells the memory devices as well as to any other device that listens to the busses whether the CPU is talking to the (real) memory or not. If it is talking to the memory all the other listeners keep extreme silent. But if the port_io line is 1, meaning the CPU doesn't want to talk to the memory but to the device which is electronically sensitive to that specific address that was put on the address bus (by the CPU), then that device jumps up and responds (through the data bus). The CPU can send as well as receive data from that particular device. A computer has some precautions to prevent address clashes, i.e. two devices responding to the same address information in port_io. Another mechanism aids communication requests initiated from the peripherals. Of course it would be possible for the CPU from time to time stop and do a port_io on all possible devices, asking them for any data they want to send in. This technique is called *polling* and is extremely inefficient for devices that send asynchronous data (data that is send in irregular intervals). You cannot know when there will be a keyboard entry so, in polling, you have to ask very frequently the keyboard device for the existence of any data. Instead of the dealing with the inefficiency of polling another mechanism is built into the CPU. The interrupt mechanism is an electronic circuitry of the CPU which has inlets (wires) connected to the peripheral devices. When a device wants to communicate with (send or receive some data to/from) the CPU then send a signal (1) from that specific wire. This gets the attention of the CPU, the CPU stops what it is doing at a convenient point in time, and ask the device for a port_io. So the device gets a chance to send/receive data to/from the CPU.

The running of a computer

When you power on a computer, it first goes through a start-up process (also called booting), which, after performing some routine checks, loads a program from your disk called Operating System.

Start up Process

At the core of a computer sits a Von Neumann architecture. But how a machine code finds its way into the memory, gets settled there, so that the CPU starts executing it, is still unclear. It is obvious that even when you buy a brand new computer and turn it on for the first time it does some actions which is traceable on its display. So there is a machine code in the memory which, even when the power is off, remains. Or, of course, there is a portion of the memory which is supplied with electricity even when the computer is turned off. It is the first option. There is a portion of the memory which does not lose its content, very much like a flash drive. It is electronically located exactly at the address where the CPU looks for its first instruction. This portion of the memory, with its content, is called Basic Input Output System, or in short BIOS. In the former days the BIOS was manufactured as write-only-once. To change the program a chip had to be replaced with a new one. The size of the BIOS of the first PCs was 16KB, nowadays it is about 1000 times larger, 16MB. When you power up a PC the BIOS program will do the following in sequence:

- Power-On Self Test, abbreviated as POST, determines whether the CPU and memory is intact, identifies and if necessary initializes devices like the video display card, keyboard, hard disk drive, optical disc drive and other basic hardware.
- In a predefined order locate boot loader software on "boot devices". It is possible to set some parameters of the BIOS, one of which is the boot devices (e.g. a hard disk, a floppy disk, a USB flash stick, CD, or DVD) and in which order they will be queried. BIOS will get the Master Boot Record (MBR) from the first available device in the given order. MBR is supposed to contain a short machine code program that will load the *Operating System* (OS). So, BIOS loads the content of the MBR into the memory and start to execute it. This program loads the actual operating system and then starts the execution of it.

The Operating System

The operating system is a program that after being loaded into the memory manages resources and services like the use of memory, CPU and devices. The OS has components that manage these resources which are explained below:

- **Memory Management**: Refers to the management of the Memory connected to the CPU. In modern computers there are more than one machine code programs loaded into the memory. Each device for some task. Some of them are initiated by the user (like a browser, document editor, excel, music player, etc.) and some are initiated by the operating system at the boot up of the computer. The CPU switches very fast from one program in the memory (this is called as *process*) to another. The user (usually) does not feel the switching. The memory manager, keeps track of the space allocated by processes in the memory. When a new program (process) is being started it has to be placed into the memory. The memory manager decides where it is going to be placed. Of course when the process ends the place in the memory occupied by it has to be reclaimed, that is the memory managers job. It is also possible, while a process is running, it demands some additional space in the memory (e.g. a photoshop-like program needs space for a newly opened jpg image) then the process makes this demand to the memory manager, which grands it or denies it.

- **Processor (Time) Management**: As said above, in modern computers more than one machine code programs are loaded into different locations of the memory. An electronic mechanism forces the CPU to switch to the *Time Manager* component of the OS. At least 20 times a second the time manager is invoked and makes a decision on behalf of the CPU. This of the processes that sit in the memory will be run for the next period? When a process gets the turn, the current state of the CPU (content of all its registers) is saved to some secure position in the memory, in association to the last executing process. From that secure position the last saved state information which going to take the turn is found and the CPU is set to that state. Then the CPU, for a period of time executed that process. At the and of that period, CPU switches over to the time manager and the time manger makes a decision for the next period. Which process will get the turn? And so on. This decision making is a complex task. Still there are Ph.D. level research going on on this subject. The time manager takes some statistics about each individual process about it system resource utilization. Also there is the possibility that a process has a high priority associated due to several reasons. The time manager has to solve a kind of optimization problem under some constraints. As mentioned this is a complex task and a hidden quality factor of an OS.

- **Device Management**: All external devices of a computer have a software plug-in to the operating system. An operating system has some standardized demands from devices and these software plug-ins implement these standardized functionality. This software is usually provided by the device manufacturer and is loaded into the operating system as a part of the device installing process. These plug-ins are named as *device drivers*.

  An Operating System performs device communication by means of these drivers. It does the following activities for device management:

  - Keeps tracks of all devices' status.
  - Decides which process gets access to the device when and for how long.
  - Implements some intelligent caching, if possible, for the data communication with the device.
  - De-allocates devices.

- **File Management**: A computer is basically a data processing machine. Various data are produced or used for very diverse purposes. Textual, numerical, audio-visual data is handled. One kind of handling of data is *storing* and *retrieving* it on some external recording device. Examples to such recording devices are hard disks, flash drives, CDs and DVDs. Data is stored on devices under files. A *file* is a persistent piece of information that has a name, some meta data (data about the data like owner, creation data, size, content type, etc) and the data. The organizational way how files are stored on devices is called the *file system*. There are various alternatives to do this. FAT16, FAT32, NTFS, EXT2, EXT3, ExFAT, HFS+ are a few of about a hundred (actually the most common ones). Each of them have their pros and cons as far as *max allowed file size, security, robustness (repairability), extensibility, metadata, lay out policies* aand some more are concerned. Files are most often managed in a hierarchy. This is achieved by a concept of directories. On the surface (as far as the user sees them) a file system usually consist of files separated into groups, so called *directories*, where directories can contain files or other directories.

  The file manager is responsible of creation, initialization of a file system, inclusion and removal of devices from this system and management of all sort of alternation in the file system: Creation, removal, copying of files and directories, dynamically assigning access rights for files and directories to processes are among them.

- **Security**: This is basically the maintenance of system integrity, availability, and confidentiality. The security of a computer exists at various layers such as

  - maintaining the physical security of the computer system,

– the security of the information the system is in hold of,

– and the security of the network to which the computer is connected.

In all of these areas, the operating system plays a vital role in keeping the security. Especially the second item is where the operating system is involved at most. Information, as you know by now, is placed in the computer in two locations. The internal memory and the external storing devices. The internal memory is in hold of the processes and the processes shall not interfere with each other (unless specifically intended). Actually in a modern day computer there can be more than one user working at the same time on the computer. Their processes running in the memory as well as their files on the file system must remain extremely private. Even their existence have to be hidden from every other user.

Computers are connected and more and more integrated in a global network. This integration is done on a spectrum of interactions. In the extreme case a computer can be solely controlled over the network. Of course this is done by supplying some credentials but, as you would guess, such is a system is prone to malicious attacks. An operating system has to take measures to protect the computer system from such harms. Sometimes it is the case that bugs of the OS is discovered and exploited in order to breach security.

- **User Interface**: As the computer's user, when you want to do anything you do this by ordering the operating system to do it. Starting/terminating a program, exploring or modifying the file system, installing/uninstalling a new device are all done by talking to the operating system. For this purpose an OS provides an interface, coined as the *user interface*. In the older days this was done by typing some cryptic commands into a typewriter of a console device. Time changed and the first computer with a *Graphical User Interface (GUI)* emerged. A GUI is a visual interface to the user where the screen can host several windows each dedicated to a different tasks. Elements of the operating system (processes, files, directories, devices, network communicates) and their status are symbolized by icons and the interactions is mostly by moving and clicking a pointing device which is another icon (usually a movable arrow) on the screen. The Xerox Alto[5] introduced on March 1973, was the first computer that had a GUI. The first personal computer with a GUI was Apple Lisa[6] , introduced in 1983 with a price of US$10000. Almost three years later, by the end of 1985 Microsoft released its first OS with GUI: Windows 1.0. The archaic console typing still exists, in the form a type-able window, which is still very much favored among programming professionals.

Important Concepts

We would like our readers to have grasped the following crucial concepts and keywords from this chapter:

- The von Neumann Architecture
- The interaction between the CPU and the memory via address, R/W and data bus lines.
- The crucial components on the CPU: The control unit, the arithmetic logic unit and the registers
- The fetch-decode-execute cycle
- The stored program concept
- Operating system and its responsibilities

[5] https://github.com/sinankalkan/CENG240/blob/master/figures/XeroxAlto.jpg?raw=true
[6] https://github.com/sinankalkan/CENG240/blob/master/figures/AppleLisa.jpg?raw=true

Further Reading

- Computer Architectures:
  - Von Neumann Architecture: http://en.wikipedia.org/wiki/Von_Neumann_architecture
  - Harvard Architecture: http://en.wikipedia.org/wiki/Harvard_architecture
  - Harvard vs. Von Neumann Architecture: http://www.pic24micro.com/harvard_vs_von_neumann.html
  - Quantum Computer: http://en.wikipedia.org/wiki/Quantum_computer
  - Chemical Computer: http://en.wikipedia.org/wiki/Chemical_computer
  - Non-Uniform Memory Access Computer: http://en.wikipedia.org/wiki/Non-Uniform_Memory_Access

Exercises

- To gain more insight, play around with the Von Neuman machine simulator at http://vnsimulator.altervista.org
- Using Google and the manufacturer's web site, find the following information for your desktop/laptop:
  - Memory (RAM) size
  - CPU type and Clock frequency
  - Data bus size
  - Address bus size
  - Size of the general purpose registers of the CPU
  - Harddisk or SSD size and random access time

Opcode	Affected address
0001	0100

- Assume that we have a CPU that can execute instructions with the following format and size given above.
  - What is the number of different instructions that this CPU can decode?
  - What is the maximum number of rows in the memory that can be addressed by this CPU?

A Broad Look at Programming and Programming Languages

The previous chapter provided a closer look at how a modern computer works. In this chapter, we will first look at how we generally solve problems with such computers. Then, we will see that a programmer does not have to control a computer using the binary machine code instructions we introduced in the previous chapter: We can use human-readable instructions and languages to make things easy for programming.

How do we solve problems with programs?

The von Neumann machine, on which computers' design is based, makes a clear distinction between instruction and data (do not get confused by the machine code holding both data and instructions: The data field in such instructions are generally addresses of the data to be manipulated and therefore, data and instructions exist as different entities in memory). Due to this clear distinction between data and instruction, the solutions to world problems were approached and handled with this distinction in mind (Fig. 8):

> *"For solving world problems, the first task of the programmer is to identify the information to be processed to solve the problem. This information is called data. Then, the programmer has to find an action schema that will act upon this data, carry out those actions according to the plan, and produce a solution to the problem. This well-defined action schema is called an algorithm."*
>
> [From: G. Üçoluk, S. Kalkan, Introduction to Programming Concepts with Case Studies in Python, Springer, 2012[7]]

[7] https://link.springer.com/book/10.1007/978-3-7091-1343-1

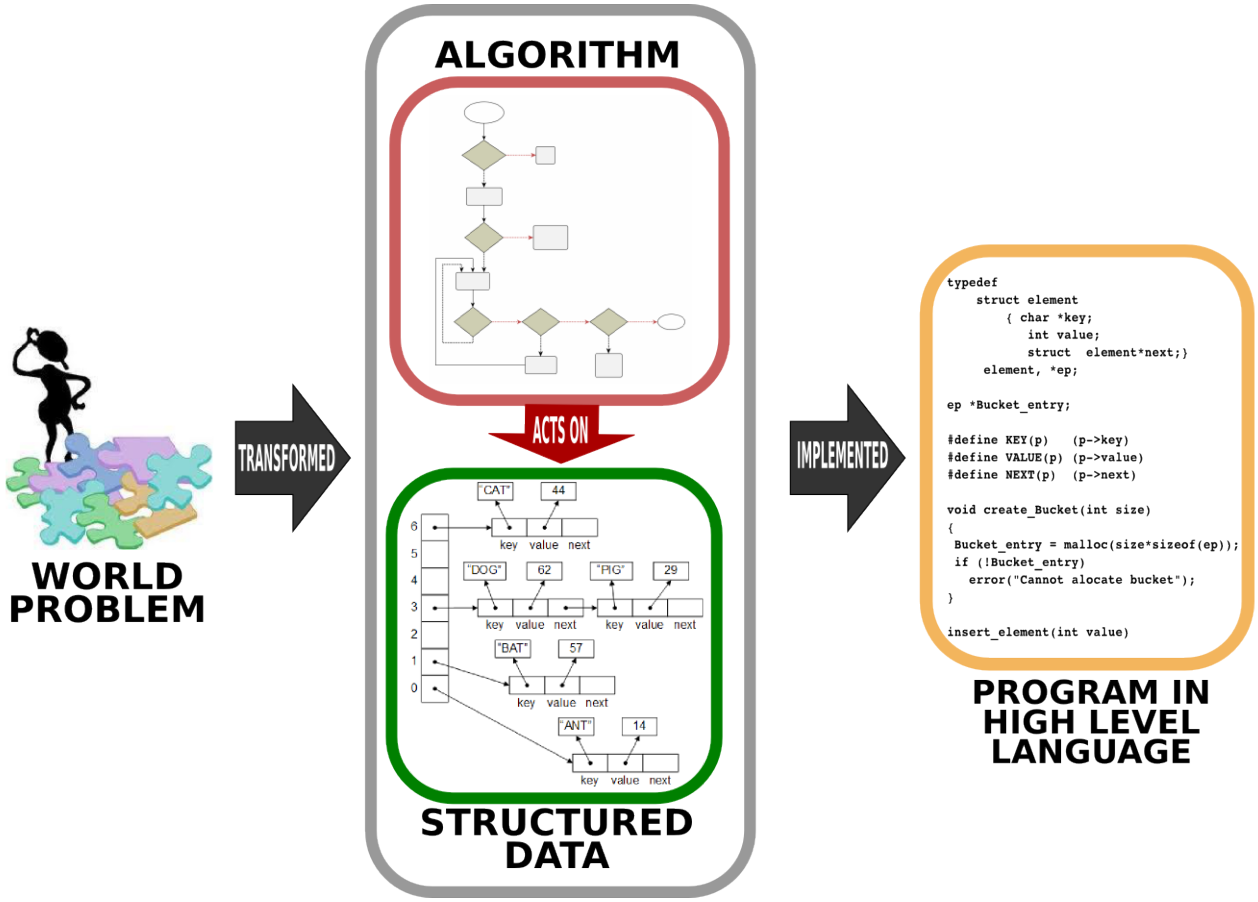

Fig. 8: Solving a world problem with a computer requires first designing how the data is going to be represented and specifying the steps which yield the solution when executed on the data. This design of the solution is then written (implemented) in a programming language to be executed as a program such that, when executed, the program outputs the solution for the world problem. [From: G. Üçoluk, S. Kalkan, Introduction to Programming Concepts with Case Studies in Python, Springer, 2012[8]]

Algorithm

An algorithm is a step-by-step procedure that, when executed, leads to an output for the input we provided. If the procedure was correct, we expect the output to be the desired output, i.e. the solution we wanted for the algorithm to compute.

Algorithms can be thought of as recepies for cooking. This analogy makes sense since we would define a recipe as a step-by-step procedure for cooking something: Each step performs a little action (cutting, slicing, stirring etc.) that brings us closer to the outcome, the meal.

This is exactly the case in algorithms as well: At each step, we make a small progress towards the solution by performing a small computation (e.g. adding numbers, finding the minimum of a set of real numbers etc.). The only difference with cooking is that each step needs to be *understandable* by the computer; otherwise, it is not an algorithm.

The origins of the world ‘algorithm’

[8] https://link.springer.com/book/10.1007/978-3-7091-1343-1

The word 'algorithm' comes from the Latin word *algorithmi*, which is the Latinized name of Al-Khwarizmi. Al-Khwarizmi was a Persian Scientist who has written a book on algebra titled "The Compendious Book on Calculation by Completion and Balancing" during 813–833 which presented the first systematic solution of linear and quadratic equations. His contributions established algebra, which stems from his method of "al-jabr" (meaning "completion" or "rejoining"). The reason the world algorithm is attributed to Al-Khwarizmi is because he proposed systematic methods for solving equations using sequences of well-defined instructions (e.g. "take all variables to the right. divide the coefficients by the coefficient of x…") – i.e. using what we call today as algorithms.

Fig. 9: Muhammad ibn Musa al-Khwarizmi (c. 780 – c. 850)

** Are algorithms the same thing as programs? **

It is very natural to confuse algorithms with programs as they are both step-by-step procedures. However, algorithms can be studied and they were invented long before there were computers or programming languages. We can design and study algorithms without using computers with just a pen and paper. A program, on the other hand, is just an implementation of an algorithm in a programming language. In other words, algorithms are designs and programs are the written forms of these designs in programming languages.

How to write algorithms

As we have discussed above, before programing our solution, we first need to design it. While designing an algorithm, we generally use two mechanisms:

1. **Pseudo-codes**. Pseudo-codes are natural language descriptions of the steps that need to be followed in the algorithm. It is not as specific or restricted as a programming language but it is not as free as the language we use for communicating with other humans: A pseudo-code should be composed of precise and feasible steps and avoid ambigous descriptions.

Here is an example pseudo-code:

Algorithm 1. Calculate the average of numbers provided by the user.

```
Step 1: Get how many numbers will be provided and store that in N
Step 2: Create a variable named Result with initial value 0
```

(continues on next page)

(continued from previous page)

```
Step 3: Execute the following step N times:
Step 4:    Get the next number and add it to Result
Step 5: Divide Result by N to obtain the result
```

2. **Flowcharts**. As an alternative to pseudocodes, we can use flowcharts while designing algorithms. Flowcharts are diagrams composed of small computational elements that describe the steps of the algorithm. An example in Fig. 10 illustrates what kind of elements are used and how they are brought together to describe an algorithm.

Flowcharts can be more intuitive to work with. However, for complex algorithms, flowcharts can get very large and prohibitive to work with.

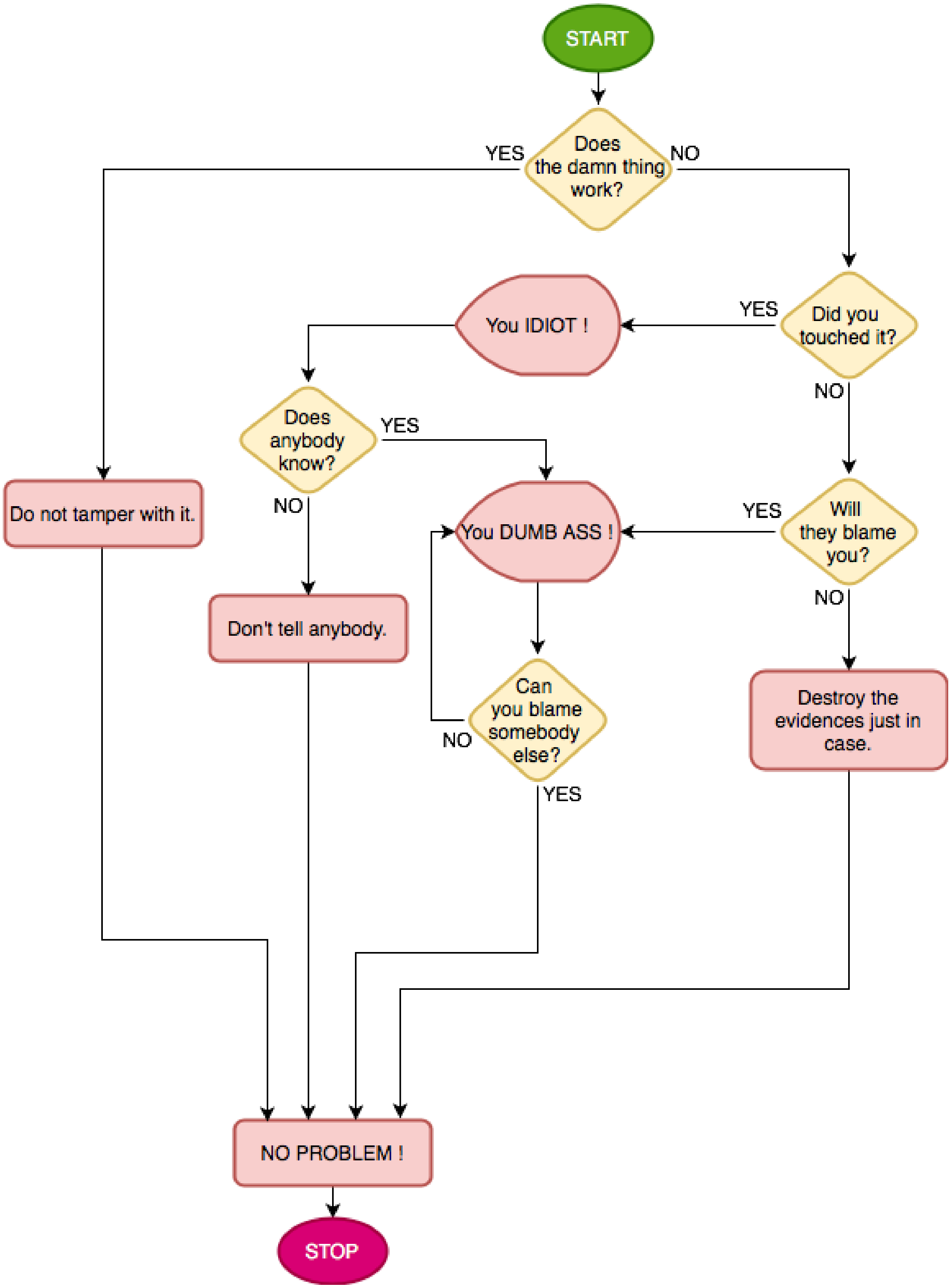

Fig. 10: Flowcharts describe relationships by using basic geometric symbols and arrows. The program start or end is depicted with an oval. A rectangular box denotes a simple action or status. Decision making is represented by a diamond and a parallelogram of the Input/Output process. A silhouette of a TV tube means displaying a message. The Internet is portrayed as a cloud.

How to compare algorithms

If two algorithms find the same solution are they of the same quality For a second, recall a game we used to play when we were in primary school: “Guess My Number”.

The rule is as follows: There is a setter and a guesser. The setter sets a number from 1 to 1000 which s/he does not tell. The guesser has to find this number. At each turn the guesser can propose any number from 1 to 1000. The setter answers by one of following:

- **HIT:** The guesser found the number.
- **LESSER:** The hidden number is less than the proposed one.
- **GREATER:** The hidden number is greater than the proposed one.

In how many turns the number is found is recorded. The guesser and the setter switch. This goes on for some agreed count of rounds. Whoever has a lesser total turn count wins.

Many of you have played this game and certainly have observed that there are three categories of children:

1. *Random guessers:* Worst category. Usually they cannot keep track of the answers and just based on the last answer they rondomly utter a number that cames to their mind. Quite possibly they repeat themselves.
2. Sweepers: They start at either 1 or 1000, and then systematically increase or decrease their proposal, e.g.:

   - *-is it 1000?* Answer: LESSER
   - *-is it 999?* Answer: LESSER
   - *-is it 998?* Answer: LESSER

   … and so on. Certainly at some point such players do get a HIT. There is a group which decrease the number by two or three as well. With a first GREATER reply, they will start to increment by one.
3. *Middle seakers:* Keeping a possible lower and a possible upper value based on the reply they got, at every stage propose the number just in the middle of lower and upper values, e.g.:

   - *-is it 500?* Answer: LESSER
   - *-is it 250?* Answer: LESSER
   - *-is it 125?* Answer: GREATER
   - *-is it 187?* (which was (125+250)/2)? Answer: GREATER
   - … and so on.

All three categories actually adopt different algorithms, which will find the answer in the end. However, as you may have realised even as you were a child, the first group peform the worst, then comes the second group. The third group, if they do not make mistakes, are unbeatable.

In other words, algorithms that aim to solve a certain problem may not be of the same “quality”: Some perform better. This is so for all algorithms and one of the challenges in Computer Science is to find “better” algorithms. But, what is “better”? Is there a quantitative measure for “better”ness? The answer is yes.

Let us look at this in the child game described above. First consider the last group’s algorithm (the middle seekers). At every turn this kind of seeker narrows down the search space by a factor of 1/2. So, at the worst case, it will take m turns till $1000/2^m$ gets down to 1 (the one remaining number, which has to be the hidden number). So, obviously $m = \log_2(1000)$. For 1000 this means approximately 10 turns ($2^{10} = 1024$). If we double the range, m would change only by 1 (yes, think about it, only 11 turns).

We call such an algorithm of "order log(n)" or more technically, $\mathcal{O}(\log(\backslash))$. In our case 1000 determines the 'size' of the problem. This is symbolized with n. $\mathcal{O}(\log(\backslash))$ is the quantitative information about the algorithm which tells that the solution time is proportional to $\log(n)$. This information about an algorithm is named as *complexity*.

What about the sweepers algorithm for the problem above? In the worst case, the sweeper would ask a question 1000 times (the correct number is at the other end of the sequence). If the size (1000 in our case) is symbolized with n, then it will take a time proportional to n to reach the solution. In other words this algorithm's complexity is $\mathcal{O}(\backslash)$.

Certainly the algorithm that has $\mathcal{O}(\log(\backslash))$ is better than the one with $\mathcal{O}(\backslash)$, which is illustrated in Fig. 11. In other words, an $\mathcal{O}(\log(\backslash))$ algorithm requires less number of steps and is likely to run faster than the one with $\mathcal{O}(\backslash)$ complexity.

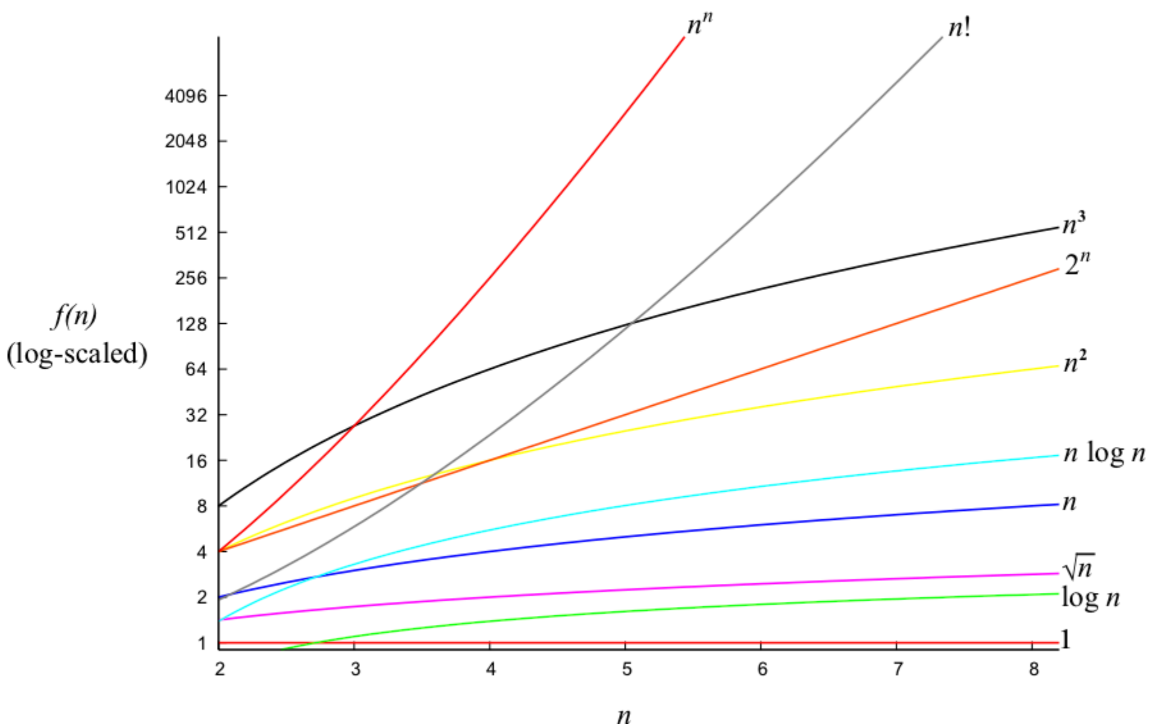

Fig. 11: A plot of various complexities.

Data Representation

The other crucial component of our solutions to world problems is the data representation, which is the representation of the information regarding the problem in a form that is most suitable for our algorithm.

If our problem is the calculation of the average of grades in a class, then before implementing our solution, we need to determine how we are going to represent the grades of students. This is what we are going to determine in the 'data representation' part of our solution and to discuss in :numref:chapter_3.

The World of Programming Languages

Since the advent of computers, many programming languages have been developed with different designs and levels of complexity. In fact, there are about 700 programming languages - see, e.g. the list of programming languages[9] that offer different abstraction levels (hiding the low-level details from the programmer) and computational benefits (e.g. providing built-in rule-search engine).

In this section, we will give a flavor of programming languages in terms of abstraction levels (low-level vs. high-level – see Fig. 12) as well as the computational benefits they provide.

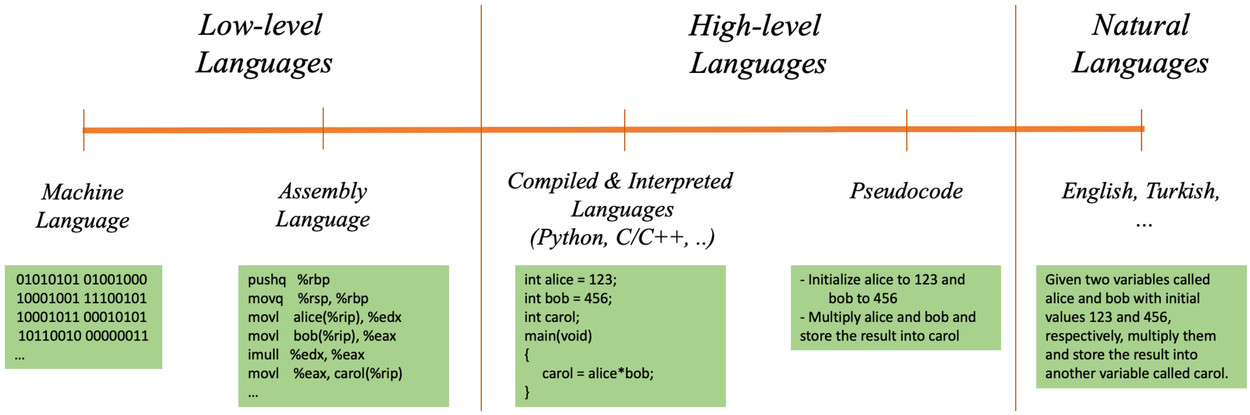

Fig. 12: The spectrum of programming languages, ranging from low-level languages to high-level languages and natural languages.

Low-level Languages

In the previous chapter we have introduced the concept of machine code program. A machine code program is an aggregate of instructions and data, all being represented in terms of zeros (0) and ones (1). A machine code is practically unreadable and very burdensome to create, as we have seen before and illustrated below:

```
01010101 01001000 10001001 11100101 10001011 00010101 10110010 00000011
00100000 00000000 10001011 00000101 10110000 00000011 00100000 00000000
00001111 10101111 11000010 10001001 00000101 10111011 00000011 00100000
00000000 10111000 00000000 00000000 00000000 00000000 11001001 11000011
...
11001000 00000001 00000000 00000000  00000000 00000000
```

To overcome this, *assembly* language and assemblers were invented. An assembler is a machine code program that serves as a translator from some relatively more readable text, the assembly program, into machine code. The key feature of an assembler is that each line of a assembly program corresponds exactly to a single machine code instruction. As an example, the binary machine code above can be written in an assembly language as follows:

```
main:
    pushq   %rbp
    movq    %rsp, %rbp
    movl    alice(%rip), %edx
    movl    bob(%rip), %eax
```

(continues on next page)

[9] https://en.wikipedia.org/wiki/List_of_programming_languages_by_type

(continued from previous page)

```
        imull   %edx, %eax
        movl    %eax, carol(%rip)
        movl    $0, %eax
        leave
        ret
alice:
        .long   123
bob:
        .long   456
```

Pros of assembly:

- Instructions and registers have human recognizable mnemonic words associated. Like integer addition instruction being ADDI, for example.
- Numerical constants can be written down in human readable, base-10 format, the assembler does the conversion to internal format.
- Implements naming of memory positions that hold data. In other words has a primitive implementation of the variable concept.

Cons of assembly:

- No arithmetic or logical expressions
- No concept of functions
- No concept of statement grouping
- No concept of data containers

High-level Languages

To overcome the limitations of binary machine codes and the assembly language, more capable *Programming Languages* were developed. We call these languages *High-level languages*. These languages hide the low-level details of the computer (and the CPU) and allow a programmer to write code in a human-readable form.

A high-level programming language (or an assembly language) is defined, similar to a natural language, by syntax (a set of grammar rules governing how to bring together words) and semantics (the meaning – i.e. what is meant by sequences of words in the syntax) associated for the syntax. The syntax is based on keywords from a human language (due to historical reasons, English). Using human-readable keywords eased comprehension.

The following example is a program expressed in Python that asks for a Fahrenheit value and prints its conversion into Celsius:

```
Fahrenheit =  input("Please Enter Fahrenheit value:")
print("Celsius equivalent is:", (Fahreheit − 32) * 5/9)
```

Here input and print are keywords of the language. The semantics of both of them are self explanatory. Fahrenheit is a naming we have chosen for a variable that will hold the input value.

High-level languages (*HL-languages* from now on) implement many concepts which are not present at the machine code programming level. Most outstanding features are:

- human readable form of numbers and strings *(like decimal, octal, hexadecimal representations for numbers)*,
- containers *(automatic allocation for places in the memory to hold data and naming them)*,
- expressions *(calculation formulas based on operators which have precedences the way we are used to from mathematics)*,
- constructs for repetitive execution *(conditional re-execution of code parts)*,
- functions,
- facilities for data organization *(ability to define new data types based on the primitive ones, organizing them in the memory in certain layouts)*.

Implementing with a High-level Language: Interpreter vs. Compiler

We can implement our solution in a programming language in two manners:

1. **Compilative Approach**. In this approach, a translator, named as *compiler*, takes a high-level programming language program as input and converts all the actions into a machine code program (Fig. 13). The outcome is a machine code program that can be run any time (by asking the OS to do so) and does the job described in the high-level language program. Conceptually this is correct, but actually, this schema has another step in-the-loop. The compiler produces an almost complete machine code with some holes in it. These holes are about the parts of the code which is not actually coded by the programmer, but filled in from a pre-created machine code library (it is actually named as *library*). A program, named *linker* fills those holes. Linker knows about the library and patches in the parts of the code that are referenced by the programmer.

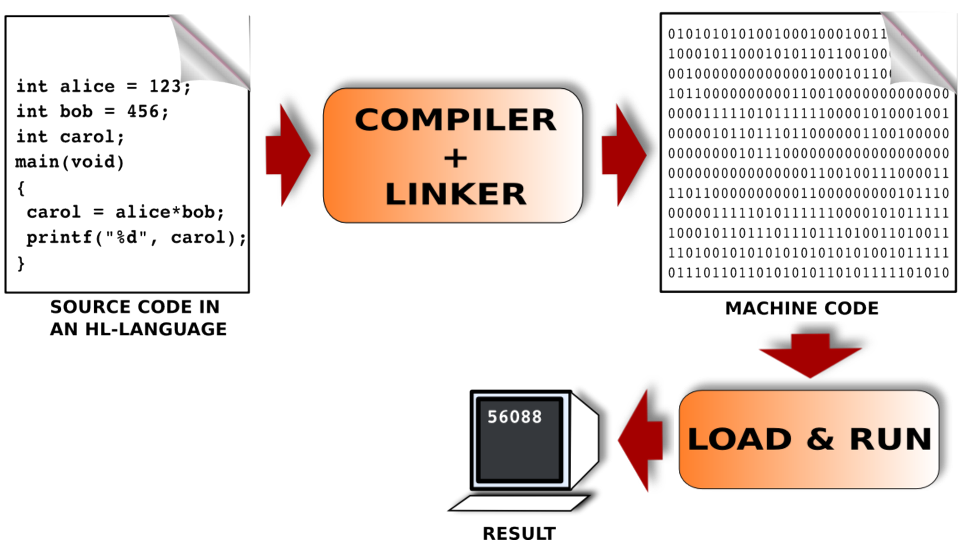

Fig. 13: A program code in a high-level language is first translated into machine understandable binary code (machine code) which is then loaded and executed on the machine to obtain the result. [From: G. Üçoluk, S. Kalkan, Introduction to Programming Concepts with Case Studies in Python, Springer, 2012[10]]

2. **Interpretive Approach**. In this approach, a machine code program, named as *interpreter*, when run, inputs and processes the high-level program line by line (Fig. 14). After taking a line as input, the actions described in the line are *immediately* executed; if the action is printing some value, the output

[10] https://link.springer.com/book/10.1007/978-3-7091-1343-1

is printed right away; if it is an evaluation of a mathematical expression, all values are substituted and at that very point-in-time, the expression is evaluated to calculate the result. In other words, any action is carried out immediately when the interpreter comes to its line in the program. In practice, it is always possible to write down the program lines into a file, and make the interpreter read the program lines one by one from that file as well.

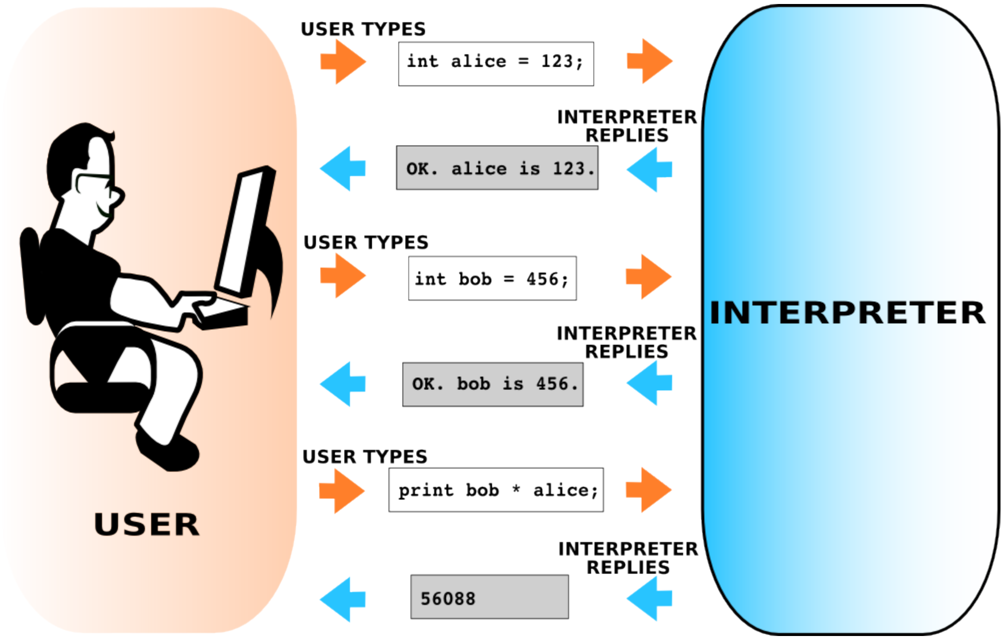

Fig. 14: Interpreted languages (e.g. Python) come with interpreters that process and evaluate each action (statement) from the user on the run and returns an answer. [From: G. Üçoluk, S. Kalkan, Introduction to Programming Concepts with Case Studies in Python, Springer, 2012[11]]

Which approach is better?

Both approaches have their benefits. When a specific task is considered, compilers generate fast executing machine codes compared to the same task being carried out by an interpreter. On the other hand compilers are unpleasant when trial-and-errors are possible during the task realization. Interpreters, on the other hand, allow making small changes and the programmer receives immediate responses, which makes it easier to observe intermediate results and adjust the algorithm accordingly. However, interpreters are slower since they involve an interpretation component while running the code. Sometimes this slowness is by a factor of 20. Therefore, the interpretive approach is good for quick implementations whereas using a compiler is good for computation intense big projects or time-tight tasks.

Programming-language Paradigms

As we mentioned before, there are more than 700 programming languages. Certainly some are for academic purposes and some did not gain any popularity. But there are about 20 programming languages which are commonly used for writing programs. How do we choose one?

Picking a particular programming language is not just a matter of taste. During the course of the evolution of the programing languages, different strategies or world views about programming have also developed. These world views are reflected in the programming languages. For example, one world view regards the programming task as transforming some initial data (the initial information that defines the problem) into

[11] https://link.springer.com/book/10.1007/978-3-7091-1343-1

a final form (the data that is the answer to that problem) by applying a sequence of functions. From this perspective, writing a program consists of defining some functions which are then used in a functional composition; a composition which, when applied to some initial data, yields the answer to the problem.

This concept of world views are coined as *programming paradigms*. The Oxford dictionary defines the word paradigm as follows:

> **paradigm** |’parə,dïm|
>
> *noun*
>
> A world view underlying the theories and methodology of a particular scientific subject.

Below is a list of some major paradigms:

- **Imperative**: Is a paradigm where programming statements and their composition directly map to the machine code segments, so that the whole machine code is covered.
- **Functional**: In this paradigm, solving a programming task is to construct a group of functions so that their ‘functional composition’ acting on the initial data produces the solution.
- **Object oriented**: In this paradigm the compulsory separation (due to the von Neumann architecture) of algorithm from data is lifted, and algorithm and data are reunited under an artificial computational entity: *the object*. An object has algorithmic properties as well as data properties.
- **Logical-declarative**: This is the most contrasting view compared to the imperative paradigm. The idea is to represent logical and mathematical relations among entities (as rules) and then ask an inference engine for a solution that satisfies all rules. The inference engine is a kind of ‘prover’, i.e. a program, that is constructed by the inventor of the logical-declarative programming language.
- **Concurrent**: A paradigm where independent computational entities work towards the solution of a problem. For problems that can be solved by a divide-and-concure strategy this paradigm is very suitable.
- **Event driven**: This paradigm considers a world of events into programming. Events are assumed to be asynchronous and they have ‘handlers’, i.e. programs that carry out the actions associated with that particular event. Programming graphical user interfaces (GUIs) is usually performed using event-driven languages: An event in a GUI is generated e.g. when the user clicks the “Close” button, which triggers the execution of a handler function that performs the associated closing action.

In contrary to the layman programmers’ assumption, these paradigms are not mutually exclusive. Many paradigms can very well co-exist in a programming language together. At a meta level, we can call them ‘orthogonal’ to each other. This is why we have so many programming languages around. A language can provide imperative as well as functional and object-oriented constructs. Then it is up to the programmer to blend them in his or her particular program. As it is with many ‘world views’ among humans, in the field of programming, fanaticism exists too. You can meet individuals that do only functional programming or object-oriented programming. We better consider them outliers.

Python, the subject language of this book, supports strongly the imperative, functional and object-orient paradigms. It also provides some functionality in other paradigms by some modules.

Important Concepts

We would like our readers to have grasped the following crucial concepts and keywords from this chapter:

- How we solve problems using computers
- Algorithms: What they are, how we write them and how we compare them
- The spectrum of programming languages
- Pros and cons of low-level and high-level languages
- Interpretive vs. compilative approach to programming
- Programming paradigms

Further Reading

- The World of Programming chapter available at: https://link.springer.com/chapter/10.1007/978-3-7091-1343-1_1
- Programming Languages:
  - For a list of programming languages: http://en.wikipedia.org/wiki/Comparison_of_programming_languages
  - For a comparison of programming languages: http://en.wikipedia.org/wiki/Comparison_of_programming_languages
  - For more details: Daniel P. Friedman, Mitchell Wand, Christopher Thomas Haynes: Essentials of Programming Languages, The MIT Press 2001.
- Programming Paradigms:
  - Introduction: http://en.wikipedia.org/wiki/Programming_paradigm
  - For a detailed discussion and taxonomy of the paradigms: P. Van Roy, Programming Paradigm for Dummies: What Every Programmer Should Know, New Computational Paradigms for Computer Music, G. Assayag and A. Gerzso (eds.), IRCAM/Delatour France, 2009 http://www.info.ucl.ac.be/~pvr/VanRoyChapter.pdf
  - Comparison between Paradigms: http://en.wikipedia.org/wiki/Comparison_of_programming_paradigms

Exercise

- Draw the flowchart for the following algorithm:

```
Step 1: Get a list of N numbers in a variable named Numbers
Step 2: Create a variable named Sum with initial value 0
Step 3: For each number i in Numbers, execute the following line:
Step 4:     if i > 0: Add i to Sum
Step 5:     if i < 0: Add the square of i to Sum
Step 5: Divide Sum by N
```

- What is the complexity of Algorithm 1 (in Section 2.2)?
- What is the complexity of the following algorithm?

```
Step 1: Get a list of N numbers in a variable named Numbers
Step 2: Create a variable named Mean with initial value 0
Step 3: For each number i in Numbers, execute the following line:
Step 4:     Add i to Mean
Step 5: Divide Mean by N
Step 6: Initialize a variable named Std with value 0
Step 7: For each number i in Numbers, execute the following line:
Step 8:    Add the square of (i-Mean) to Std
Step 9: Divide Std by N and take its square root
```

- Assuming that a step of an algorithm takes 1 second, fill in the following table for different algorithms for different input sizes (n):

Input Size

$\mathcal{O}(\log \backslash)$

$\mathcal{O}(\backslash)$

$\mathcal{O}(\backslash \log \backslash)$

$\mathcal{O}(\backslash^{\in})$

$\mathcal{O}(\backslash^{\ni})$

$\mathcal{O}(\in^{\backslash})$

$\mathcal{O}(\backslash^{\backslash})$

$n = 10^2$

$n = 10^3$

$n = 10^4$

$n = 10^5$

- Assume that we have a parser than can process and parse natural language descriptions (without any syntactic restrictions) for programming a computer. Given such a parser, do you think we would use natural language to program computers? If no, why not?
- For each situation below, try to identify which paradigm is more suitable compared to the others:
- Writing a program which should take an image as input, an RGB color value and find all pixels in the image that match the given color.
- Writing a theorem proving program.
- Writing the auto pilot program flying an airplane.
- Writing a document editing program as an alternative to Microsoft Word.