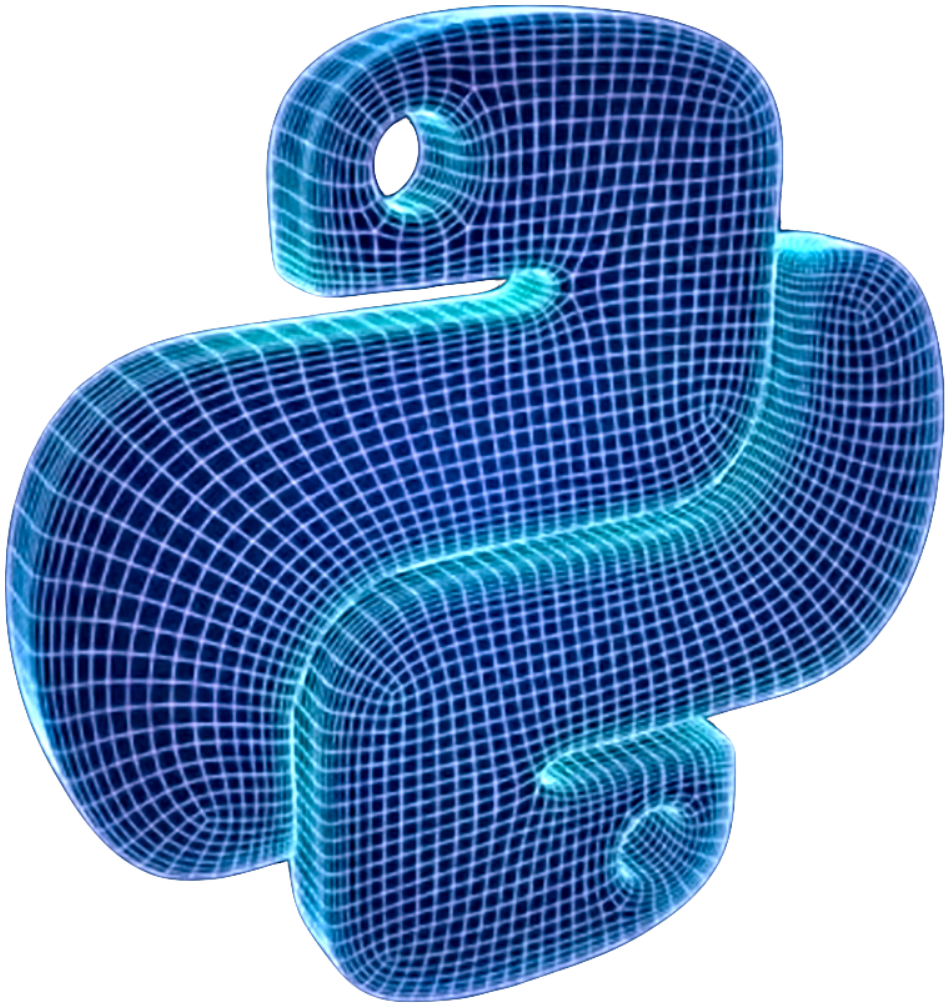


Programming with Python for Engineers

Release 0.0.1

Sinan Kalkan, Onur Tolga Sehitoglu, Gokturk Ucoluk

Nov 02, 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/

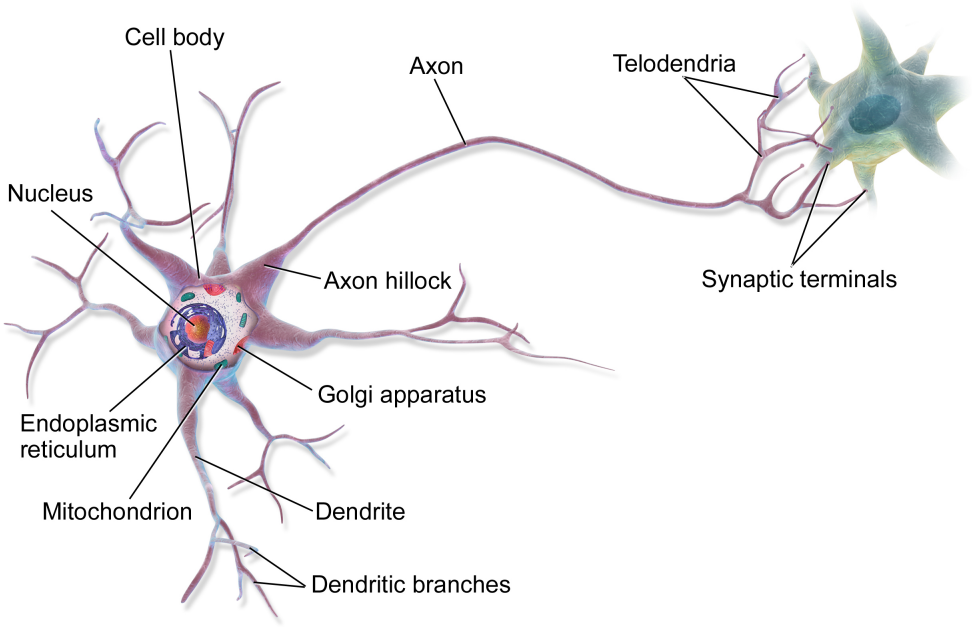


Fig. 1: Our brains are composed of simple processing units, called neurons. Neurons receive signals (information) from other neurons, process those signals and produce an output signal to other neurons. [Drawing by BruceBlaus - Own work, CC BY 3.0, https://commons.wikimedia.org/w/index.php?curid=28761830]

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.

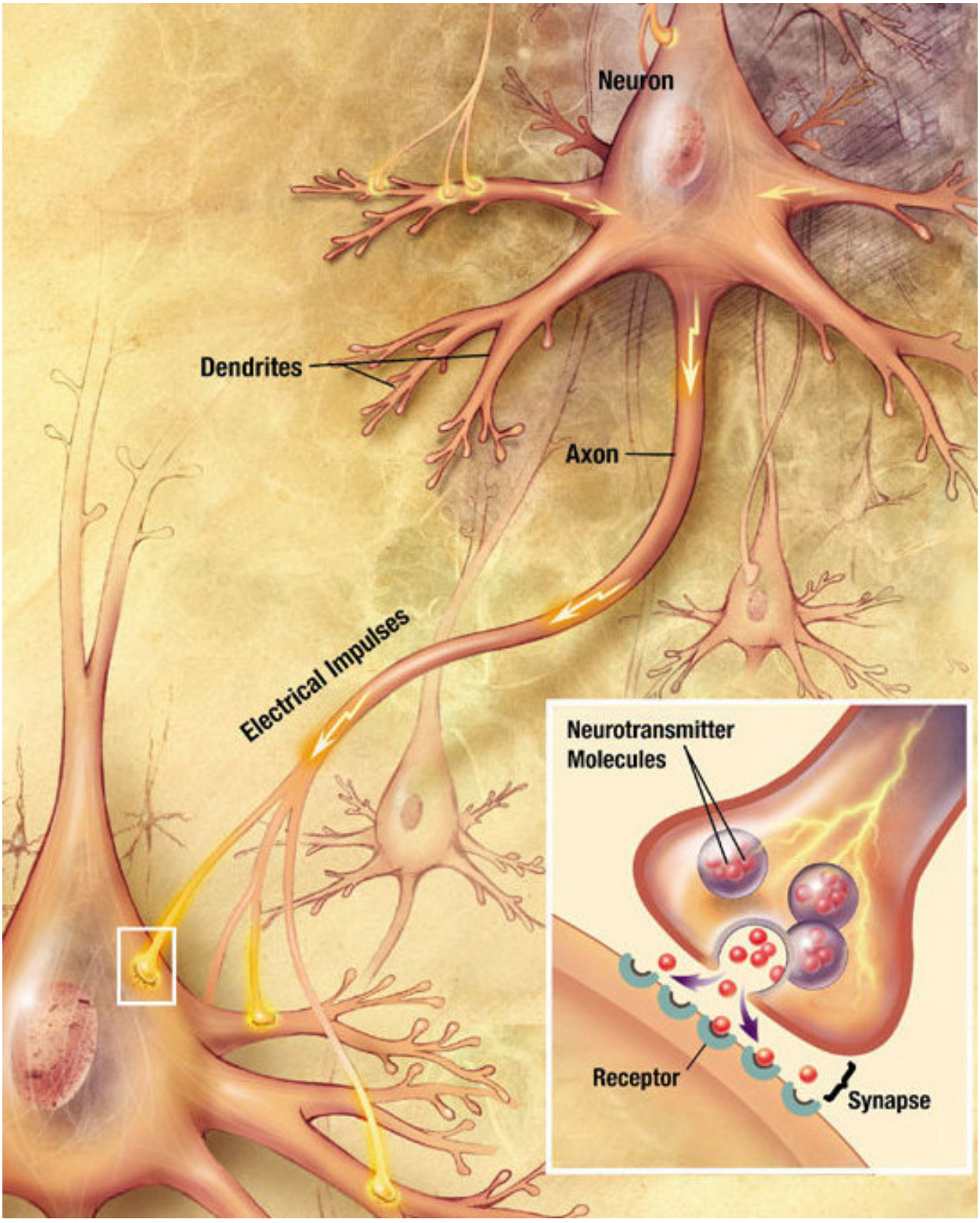


Fig. 2: Neurons 'communicate' with each other by transmitting neurotransmitters via synapses. [Drawing by user:Looie496 created file, US National Institutes of Health, National Institute on Aging created original - http://www.nia.nih.gov/alzheimers/publication/alzheimers-disease-unraveling-mystery/preface, 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.

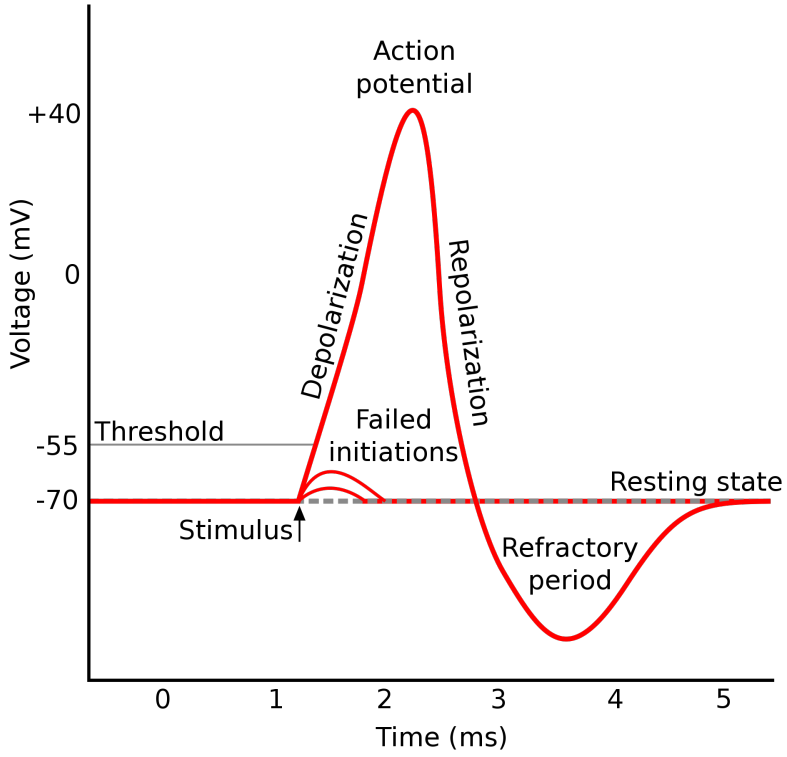


Fig. 3: When a neuron receives 'sufficient' amount of signals, i.e. stimulated, it emits neurotransmitters on its axon, i.e. it fires. [Plot by Original by en:User:Chris 73, updated by en:User:Diberri, converted to SVG by tiZom - Own work, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=2241513]

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 com-

pletely.

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

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 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

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

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).

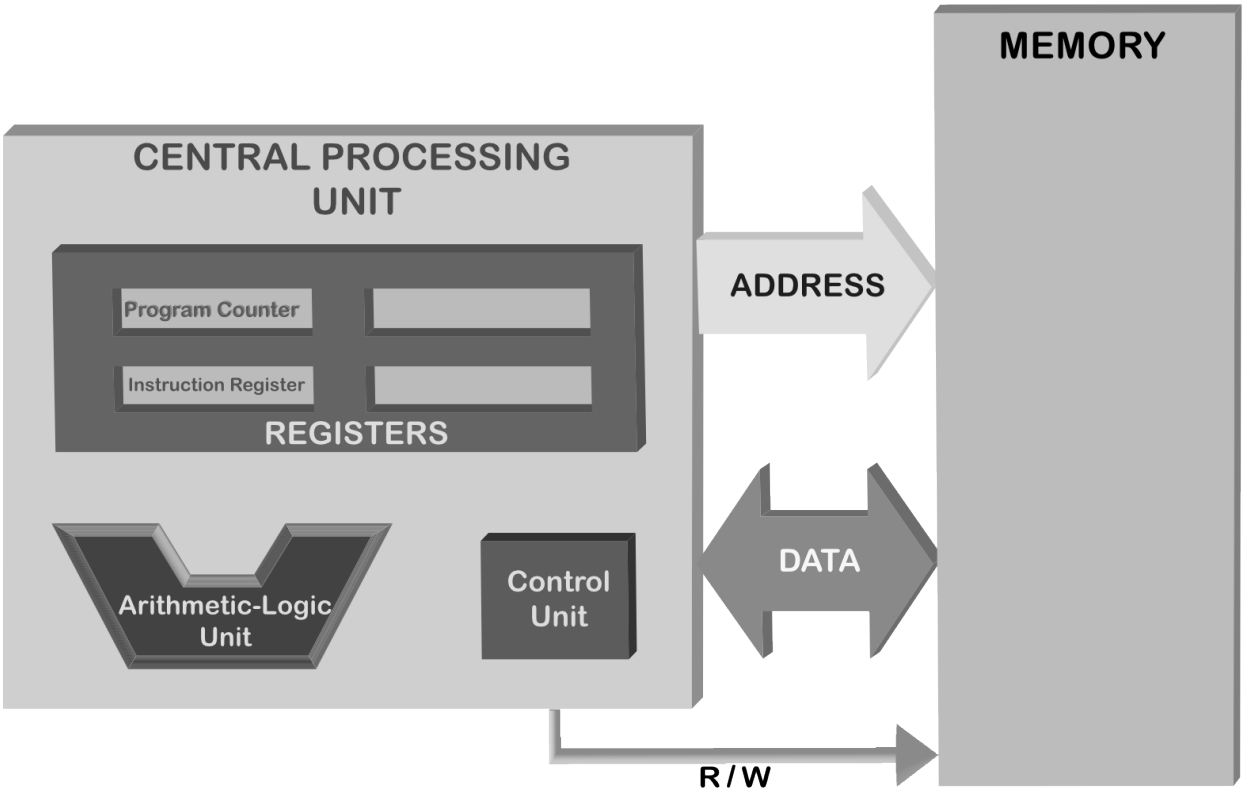


Fig. 5: A block structure view of the von Neumann Architecture.

The CPU communicates with the memory via 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 binary information to/from the memory. Each wire in a bus carries one bit of the information (either a zero (0) or a one (1)). Today's von Neumann architectures are working on electricity, and therefore, these zeros and ones correspond to voltages. A one indicates usually the presence of a 5V and a zero denotes the absence 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 pigeon hole 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*; i.e., a byte means 8 bits.

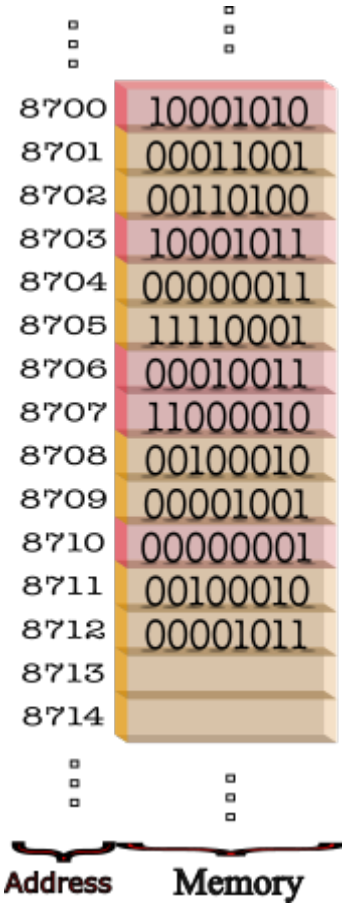


Fig. 6: The memory is organized as a stack of rows such that each row has an associated address.

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 have to be noted:

- Accessing 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 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 next section.

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 the range of [2-64] bytes and most registers on today's most popular CPUs have size 64 bits (i.e. 8 bytes). Their quantity is not high and in the range of [10-20]. The registers can be 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).

- **Input/Output 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*. The machine, being in a state, reads the next instruction and executes the instruction according to its current state. The state consists of what is stored in the registers. Until it is powered off, the CPU 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 the functioning of the cycle.

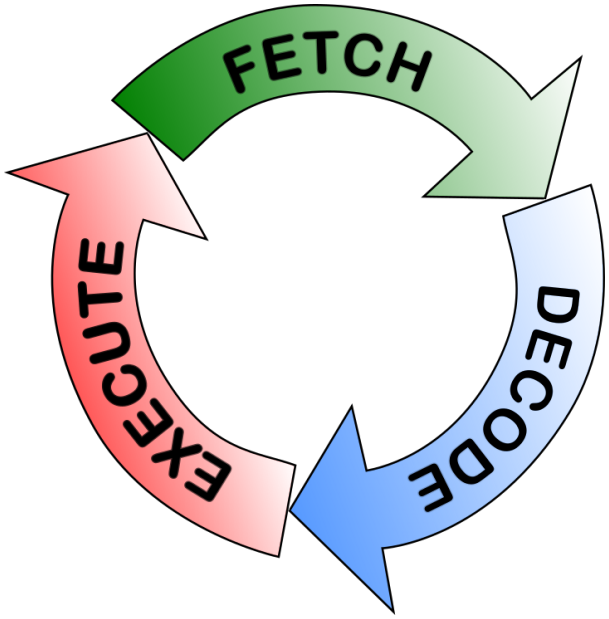


Fig. 7: The CPU constantly follows the fetch-decode-execute cycle while the computer is running a program.

1- **The Fetch Phase**

The cycle starts with the Fetch Phase. At the beginning of this 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 address 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 is architecture dependent. Usually it is 4-8 bytes. These bytes are received into the IR (Instruction Register).

2- **The Decode Phase**

At the beginning of this phase, the IR is assumed to be holding the current instruction. The content of the first part of the IR electronically triggers some action. Every CPU has an electronically built-in hard-wired instruction table in which every possible atomic operation 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.

There are three types of instructions:

- *Data manipulation*: 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.

Let us assume that our instruction looks like this:

Opcode	Effected data or address
0001	0110

This is an 8-byte instruction that has the first 4 bits as representing the opcode. The designer could have designed the CPU such that the opcode 0001 denotes an instruction for reading data from the memory, writing data to the memory or adding the contents of the two registers etc. The remaining four bits then

contain the parameters of the instruction, which are the data to be operated on, the address in the memory or the codes of the registers etc.

Let us assume that this 8-bit example instruction (i.e. the opcode 0001) denotes an addition on two registers and that the remaining 4 bits encode the registers in question, with 01 denoting one register and 10 the other register. Prior to the instruction, we can assume the two registers to 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).

Although this was a simple and hypothetical example, it illustrates how modern CPUs can decode an instruction and decipher its elements. Though, the length of an instruction and the variety of instructions are clearly different.

3- **The Execute Phase**

As the name implies, the electronically activated and initialized circuitry carries out the instruction in this phase. Depending on the instruction, the registers, the memory or other components are effected. When the instruction completes, the PC is updated by one unless it was a control flow changing instruction in which case the PC is updated to the to-be-jumped address in the memory (in some designs, the PC can be updated in the fetch phase, after fetching the instruction). 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 a certain 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 to 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 an 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, to make a detailed memory layout on paper and then to start writing down each bit of the memory. This way of programming 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, e.g. denoting 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 brainstorming 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

The von Neumann architecture has certain advantages and disadvantages:

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 extent.

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.

An alternative to the von Neumann Architecture is the Harvard Architecture[5] which could not resist the test of time due to crucial disadvantages compared to the von Neumann architecture.

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 connections and the internal hard disk. Explaining 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) which 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 quiet. But if the port_io line is 1, meaning the CPU doesn't 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

[5] https://en.wikipedia.org/wiki/Harvard_architecture

entry so, in polling, you have to ask very frequently the keyboard device for the existence of any data. Instead of 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 they 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 asks 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 is the 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.

When you buy a brand new computer and turn it on for the first time, it does some actions which are traceable on its display. Therefore, there must be a machine code in a memory which, even when the power is off, 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 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, which determines whether the CPU and the memory are intact, identifies and if necessary, initializes devices like the video display card, keyboard, hard disk drive, optical disc drive and other basic hardware.
- Looking for an *operating system* (OS): The BIOS program goes through storage devices (e.g. hard disk, floppy disk, USB disk, CD-DVD drive, etc.) connected to the computer in a predefined order (this order is generally changeable by the user) and looks for and settles for the first operating system that it can find. Each storage device has a small table at the beginning part of the device, called the Master Boot Record (MBR), which contains a short machine code program to load the operating system if there is one.
- When BIOS finds such a storage device with an operating system, it loads the content of the MBR into the memory and starts executing it. This program loads the actual operating system and then runs 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, the CPU and the devices. It essentially hides the internal details of the hardware and makes the ugly binary machine understandable and manageable to us.

An OS has the following responsibilities:

- **Memory Management**: Refers to the management of the memory connected to the CPU. In modern computers, there is more than one machine code program loaded into the memory. Some programs are initiated by the user (like a browser, document editor, Word, music player, etc.) and some are initiated

by the operating system. The CPU switches very fast from one program (this is called a *process*) in the memory 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 a process ends, the place in the memory occupied by the process has to be reclaimed; that is the memory manager's job. It is also possible that, while running, a process demands additional space in the memory (e.g. a photoshop-like program needs more space for a newly opened JPG image file) then the process makes this demand to the memory manager, which grants it or denies it.

- **Process (Time) Management**: As said above, a modern memory generally contains more than one machine code program. 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 to make a decision on behalf of the CPU: Which of the processes that sit in the memory will be run during the next period? When a process gets the turn, the current state of the CPU (the contents of all 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 executes that process. At the end of that period, the CPU switches over to the time manager and the time manager 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 collects some statistics about each individual process and its 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 are handled. Handling data also includes *storing* and *retrieving* it on some external recording device. Examples for such recording devices are hard disks, flash drives, CDs and DVDs. Data is stored on these devices as files. A *file* is a persistent piece of information that has a name, some meta data (e.g. information about the owner, the creation time, size, content type, etc.) and the data.

  The organizational mechanism for 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 has its own pros and cons as far as *max allowed file size, security, robustness (repairability), extensibility, metadata, lay out policies* and some other aspects are concerned. Files are most often managed in a hierarchy. This is achieved by a concept of *directories* or *folders*. On the surface (as far as the user sees them), a file system usually consist of

files separated into directories where directories can contain files or other directories.

The file manager is responsible for creation & initialization of a file system, inclusion and removal of devices from this system and management of all sorts of changes in the file system: Creation, removal, copying of files and directories, dynamically assigning access rights for files and directories to processes etc.

- **Security**: This is basically for the maintenance of the computer's 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, 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 storage devices. The internal memory is in hold of the processes and the processes should 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 being on the file system must remain extremely private and separate. Even their existence has 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 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 are 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 at a console device. Over the years, 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 task. Elements of the operating system (processes, files, directories, devices, network communications) and their status are symbolized by icons and the interactions are mostly via moving and clicking a pointing device which is another icon (usually a movable arrow) on the screen. The Xerox Alto[6] introduced on March 1973 was the first computer that had a GUI. The first personal computer with a GUI was Apple Lisa[7], introduced in 1983 with a price of $10,000. Almost three years later, by the end of 1985, Microsoft released its first OS with a GUI: Windows 1.0. The archaic console typing still exists, in the form a type-able window, called terminal, which is still very much favoured among programming professionals because it provides more control for the OS.

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

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.

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
- Running a computer:
  - Booting a computer: https://en.wikipedia.org/wiki/Booting
  - History of operating systems: https://en.wikipedia.org/wiki/History_of_operating_systems
  - Operating systems: https://en.wikipedia.org/wiki/Operating_system

Exercises

- To gain more insight, play around with the von Neumann 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 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[8]]

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

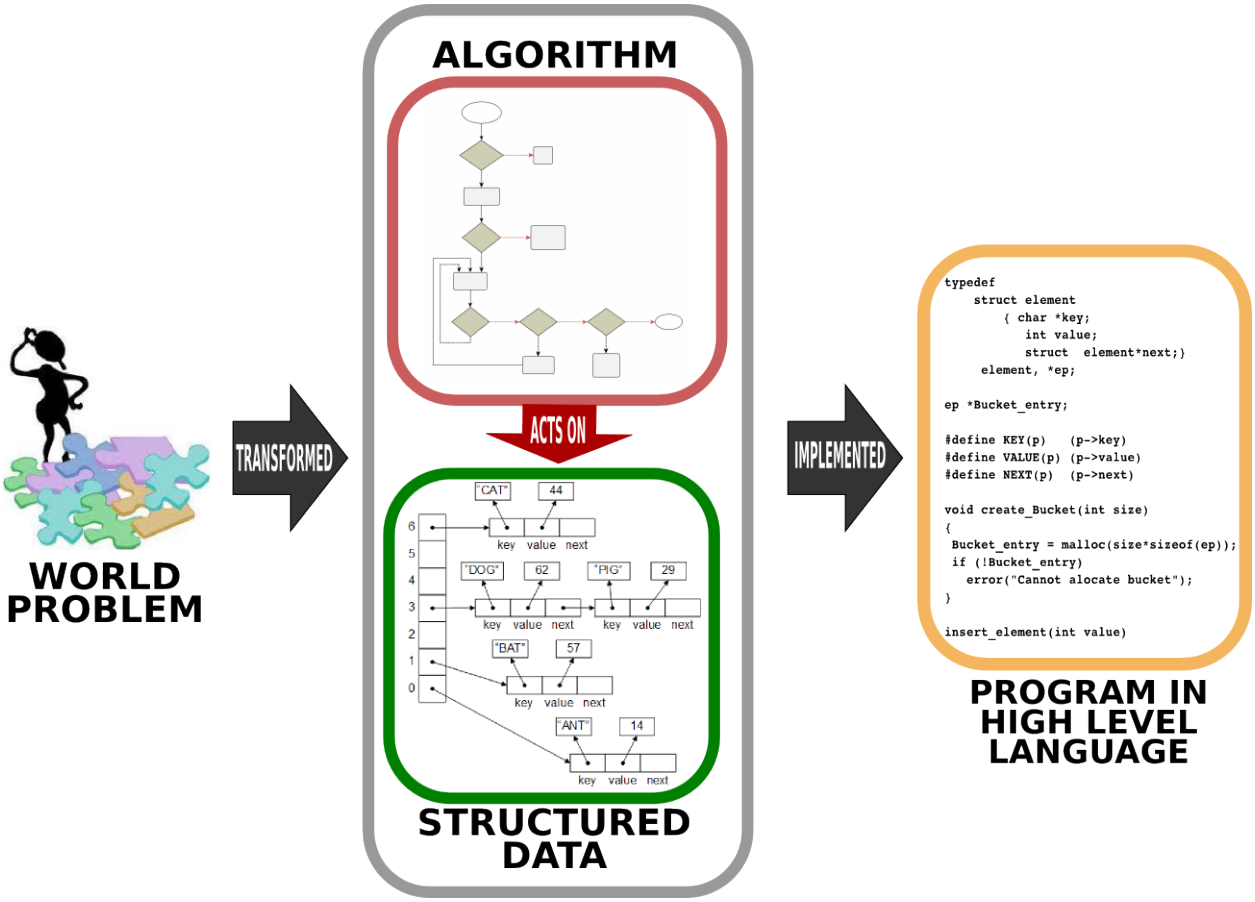


Fig. 8: Solving a world problem with a computer requires first designing how the data is going to be represented and specifying the steps which yield the solution when executed on the data. This design of the solution is then written (implemented) in a programming language to be executed as a program 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[9]]

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 recipes 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’ 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 al-

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

gebra 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 programming 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 ambiguous descriptions.

Here is an example pseudo-code:

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

```
Input: N -- the count of numbers
Output: The average of N numbers to be provided
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
Step 3: Execute the following step N times:
```

(continues on next page)

(continued from previous page)

```
Step 4:    Get the next number and add it to Result
Step 5: Divide Result by N to obtain the average
```

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.

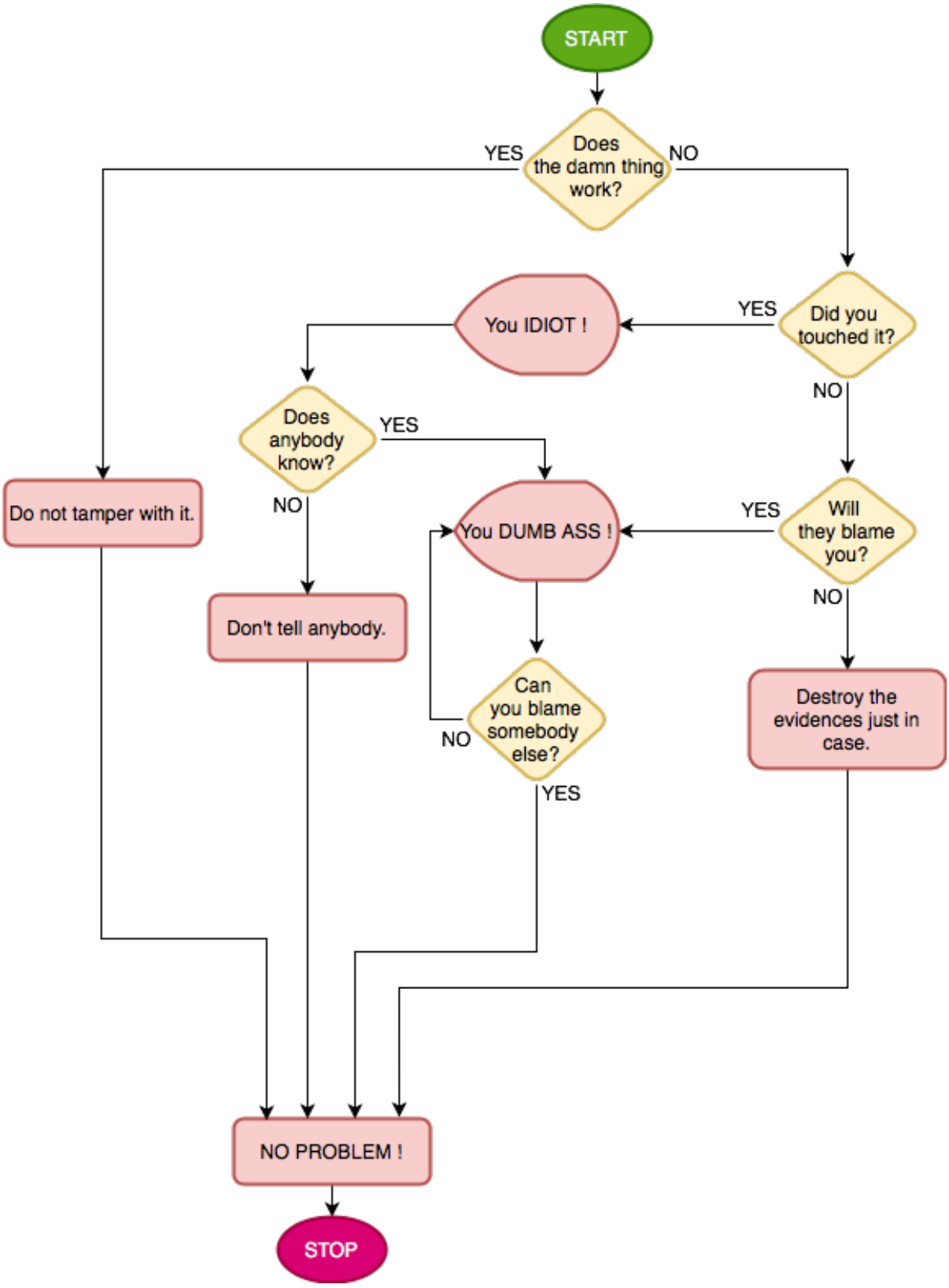


Fig. 10: Flowcharts describe relationships by using basic geometric symbols and arrows. The program start or end is depicted with an oval. A rectangular box denotes a simple action or status. Decision making is represented by a diamond and a parallelogram of the Input/Output process. A silhouette of 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 of the game, 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 lower total count of turns 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 randomly utter a number that comes 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 decreases the number by two or three as well. With a first GREATER reply, they start to increment by one.
3. *Middle seekers:* Keeping a possible lower and a possible upper value based on the reply they got, at every stage they 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 performs the worst, then comes the second group. The third group, if they do not make mistakes, is unbeatable.

In other words, algorithms that aim to solve the same problem may not be of the same "quality": Some perform better. This is the case 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. Starting with 1000 numbers, the search space is reduced as follows: 1000, $1000/2$, $1000/2^2$, … So, in the worst case,

it will take m turns until $1000/2^m$ gets down to 1 (the one remaining number, which has to be the hidden number). In other words, in the worst case, $1000/2^m = 1$ and from this we can derive $m = \log_2(1000)$. For 1000, this means approximately $m = 10$ turns. 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 signifies 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.

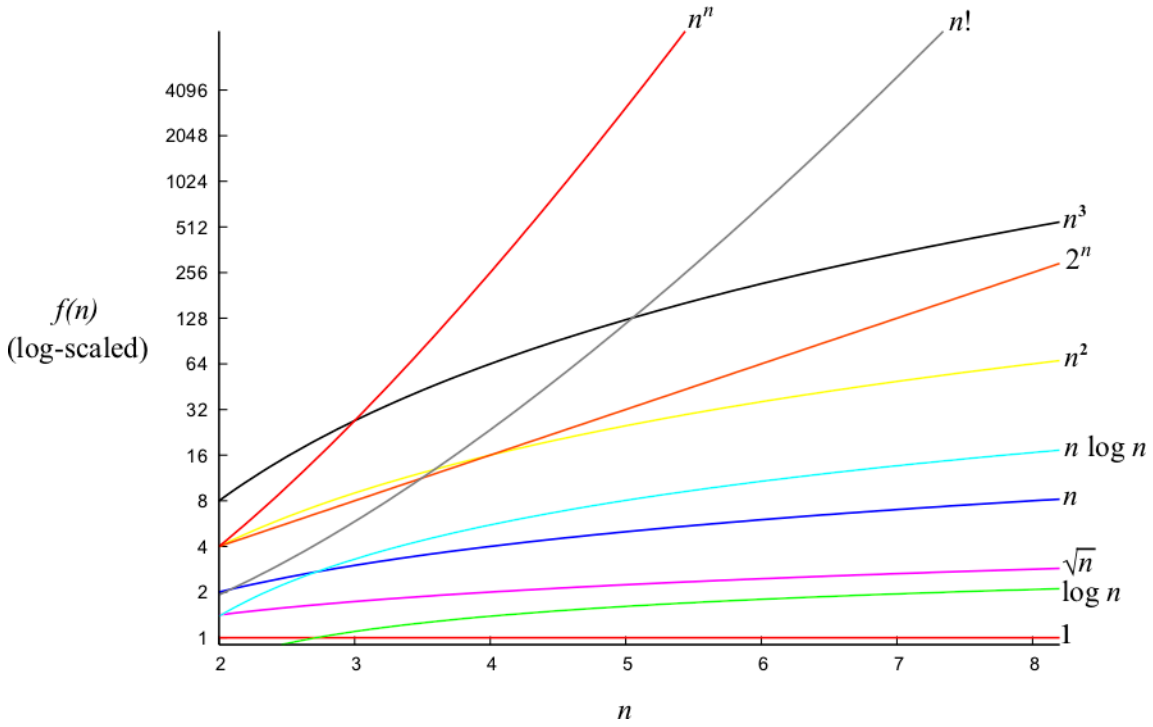


Fig. 11: A plot of various complexities.

Data Representation

The other crucial component of our solutions to world problems is the data representation, which deals with encoding 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 (encode) the grades of students. This is what we are going to determine in the 'data representation' part of our solution and to discuss in 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[10] - 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 flavour of programming languages in terms of abstraction levels (low-level vs. high-level – see Fig. 12) as well as the computational benefits they provide.

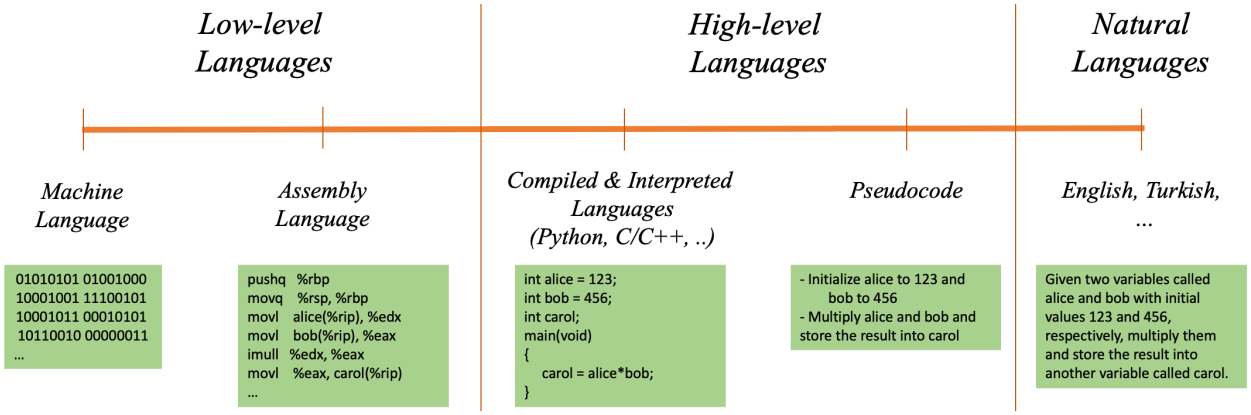


Fig. 12: The spectrum of programming languages, ranging from low-level languages to high-level languages and natural languages.

Low-level Languages

In the previous chapter, we 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 an 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)

[10] 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, assembly 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 more 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 the 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 ease 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:", (Fahrenheit − 32) * 5/9)
```

Here input and print are keywords of the language. Their semantics is 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, access and name data)*,
- 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 high-level programming language in two manners:

1. **Compilative Approach**. In this approach, a translator, called *compiler*, takes a high-level programming language program as input and converts all actions in the program 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. The linker knows about the library and patches in the parts of the code that are referenced by the programmer.

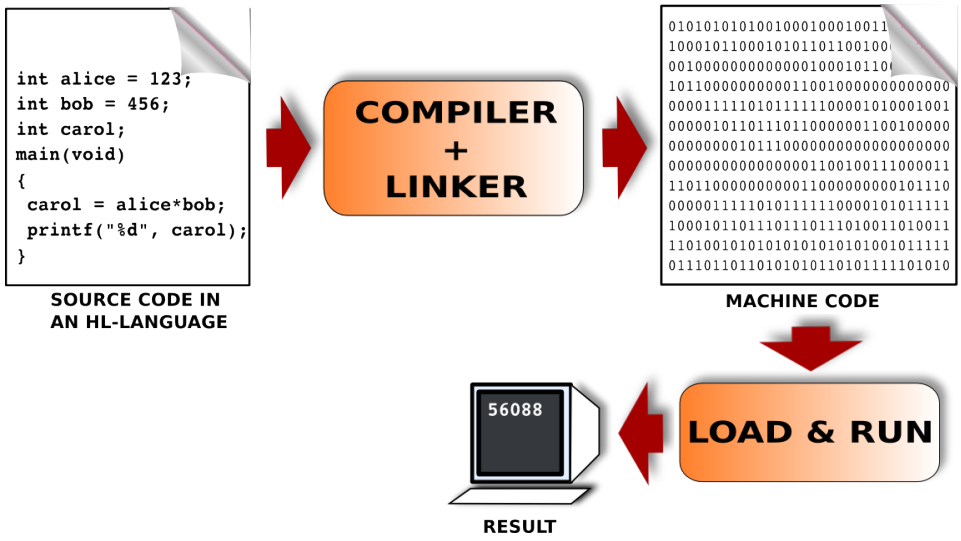


Fig. 13: A program code in a high-level language is first translated into machine understandable binary code (machine code) which is then loaded and executed on the machine to obtain the result. [From: G. Üçoluk, S. Kalkan, Introduction to Programming Concepts with Case Studies in Python, Springer, 2012[11]]

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

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

actions described in the line are *immediately* executed; if the action is printing some value, the output 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.

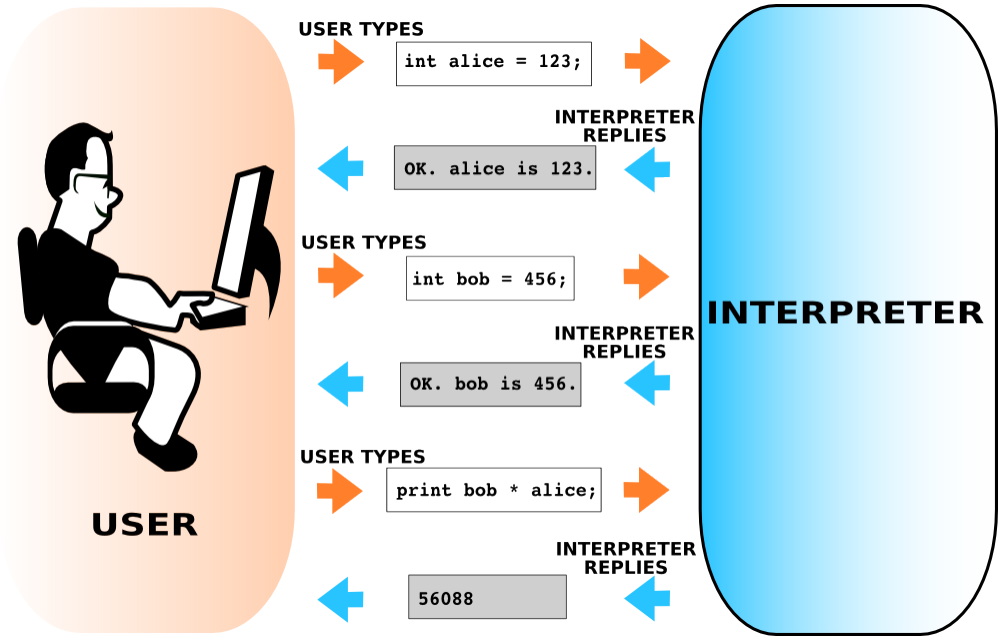


Fig. 14: Interpreted languages (e.g. Python) come with interpreters that process and evaluate each action (statement) from the user on the run and returns an answer. [From: G. Üçoluk, S. Kalkan, Introduction to Programming Concepts with Case Studies in Python, Springer, 2012[12]]

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 while developing the solution. 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.

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

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 when implementing our solution?

Picking a particular programming language is not just a matter of taste. During the course of the evolution of the programming 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 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-conquer strategy, this paradigm is very suitable.
- **Event driven**: This paradigm introduces the concept of events into programming. Events are assumed to be asynchronous and they have 'handlers', i.e. programs that carry out the actions associated with a 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.

Introducing Python

After having provided background on the world of programming, let us introduce Python: Although it is widely known to be a recent programming language, Python's design, by Guido van Rossum[13], dates back to 1980s, as a successor of the ABC programming language. The first version was released in 1991 and with the second version released in 2000, it started gaining a wider interest from the community. After it was chosen by some big IT companies as the main programming language, Python became one of the most popular programming languages.

An important reason for Python's wide acceptance and use is its design principles. By design, Python is a programming language that is easier to understand and to write but at the same time powerful, functional, practical and fun. This has deep roots in its design philosophy (a.k.a. The Zen of Python[14]):

> "Beautiful is better than ugly. Explicit is better than implicit. Simple is better than complex. Complex is better than complicated. Readability counts. […] *there are 14 more*]"

Python is multi-paradigm programming language, supporting imperative, functional and object-oriented paradigms, although the last one is not one of its strong suits, as we will see in Chapter 7. Thanks to its wide acceptance especially in the open-source communities, Python comes with or can be extended with an ocean of libraries for practically solving any kind of task.

The word 'python' was chosen as the name for the programming language not because of the snake species python but because of the comedy group Monty Python[15]. While van Possum was developing Python, he read the scripts of Monty Python's Flying Circus and thought 'python' was "short, unique and mysterious"[16] for the new language. To make Python more fun to learn, earlier releases heavily used phrases from Monty Python in example programming codes.

With version 3.9 being released in October 2020 as the latest version, Python is one of the most popular programming languages in a wide spectrum of disciplines and domains. With an active support from the open-source community and big IT companies, this is not likely to change in the near future. Therefore, it is in your best interest to get familiar with Python if not excel in it.

This is how the Python interpreter looks like at a Unix terminal:

```
$ python3
Python 3.8.5 (default, Jul 21 2020, 10:48:26)
[Clang 11.0.3 (clang-1103.0.32.62)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

The three symbols >>> indicate that the interpreter is ready to collect our computational demands, e.g.:

[13] https://en.wikipedia.org/wiki/Guido_van_Rossum
[14] https://en.wikipedia.org/wiki/Zen_of_Python
[15] https://en.wikipedia.org/wiki/Monty_Python
[16] https://docs.python.org/2/faq/general.html#why-is-it-called-python

```
>>> 21+21
42
```

where we asked what was 21+21 and Python responded with 42, which is one small step for a man but one giant leap for mankind.

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.

1 | Representation of Data

As we discussed in detail in Chapter 2, when we want to solve a world problem using a computer, we have to find what data is involved in this problem and how we can process this data (i.e. determine the algorithm) towards the solution of the problem – let us recall this with Fig. 1.1 from Chapter 2.

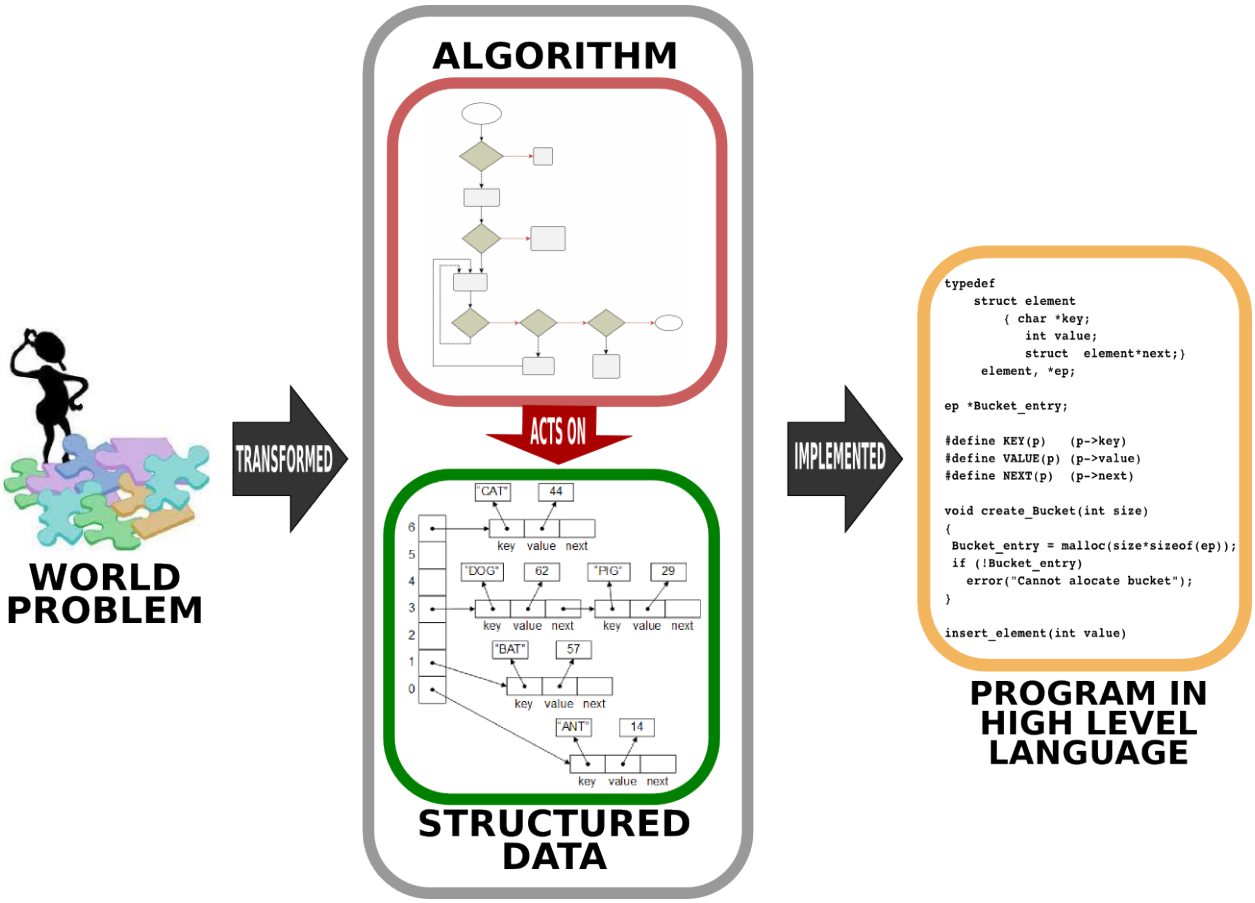


Fig. 1.1: Solving a world problem with a computer requires first designing how the data is going to be represented and specifying the steps which yield the solution when executed on the data. This design of the solution is then written (implemented) in a programming language to be executed as a program 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[17]]

At this stage, you may be wondering what 'structured data' is and how it differs from 'data'. As we already know, data is stored in the memory. Let us illustrate this with an example: Assume that we have a table full

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

of rows such that each row holds an angle value and its cosine value (both expressed as decimal numbers). How would you organize the storing of these rows in the memory? Two straightforward options are:

a) Row-by-row:

$$Anglevalue_1$$
$$Cosinevalue_1$$
$$Anglevalue_2$$
$$Cosinevalue_2$$
$$\vdots$$
$$Anglevalue_n$$
$$Cosinevalue_n$$

b) Column-by-Column:

$$Anglevalue_1$$
$$Anglevalue_2$$
$$\vdots$$
$$Anglevalue_n$$
$$Cosinevalue_1$$
$$Cosinevalue_2$$
$$\vdots$$
$$Cosinevalue_n$$

Apart from this ordering, there is the issue of whether we should sort the values. If yes by which one: the angle or the cosine values? In a descending order or ascending order? All these questions are what we try to determine in a structured data.

There are other ways to organize data even for the example data as simple as the above example table. Unfortunately, these are beyond the scope of this book.

Now, we will look into the atomic structure of data representation: i.e. how integers, floating points and characters are represented.

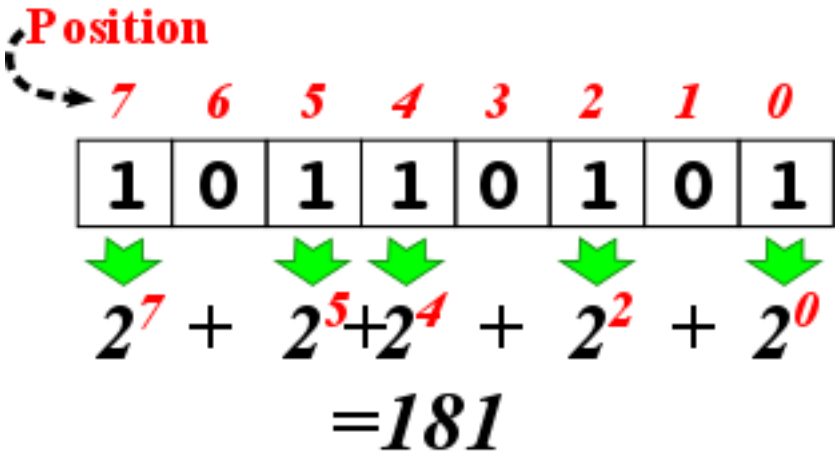
The electronic architecture used for the von Neumann machine, namely the CPU and the memory, is based on the presence and absence of a fixed voltage. We conceptualize this absence and presence by '0' and '1'. This means that any data that is going to be processed on this architecture has to be converted to a representation of '0's and '1's. We, though, keep in mind, that any '1' (or '0') in our representation is actually the presence (or absence) of a voltage in a specific point of the electronic circuitry.

1.1 Representing integers

Mathematics already has a solution for representing integers in binary via base-2 calculation or representation. The idea behind base-2 representation is the same as representing base-10 (decimal) integers. Namely, we have a sequence of digits, each of which can be either 0 or 1. Counting starts from the right-most digit. A '1' in a position k means "additively include a value of 2^k":

$$\begin{array}{lccccccc} Position: & n & n-1 & ... & 2 & 1 & 0 \\ Meaning: & 2^n & 2n-1 & ... & 2^2 & 2^1 & 2^0 \end{array}$$

The following is an example for expressing the integer **181** in base-2:



In other words, $(181)_{10} = (10110101)_2$.

Converting a decimal number into binary

A decimal number can be easily converted into binary by repeatedly dividing the number by 2, as shown in Fig. 1.1.1. At each step, the quotient of the previous step is divided by two. This division is repeated until the quotient is zero. At this point, the binary sequence of remainders is the representation of the decimal number.

Having doubts? You can easily cross-check your calculation by multiplying each bit by its value (2^i if the bit is at position i) and sum up the values (like we did above).

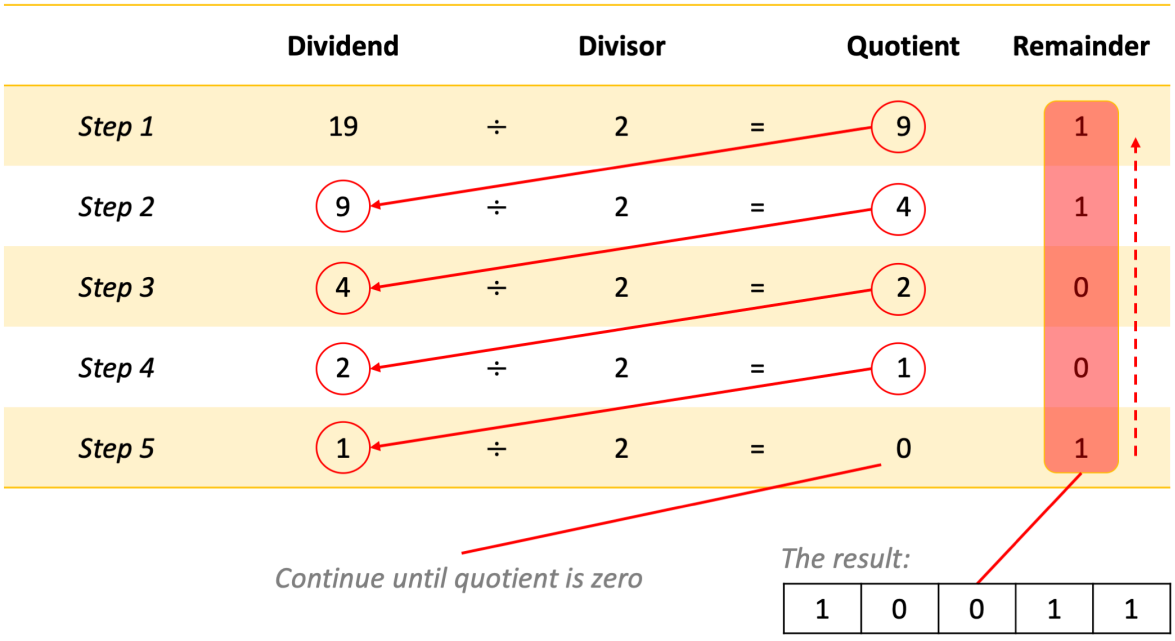


Fig. 1.1.1: The division method for converting a decimal number into binary.

This looks easy. However, this is just a partial solution to the 'binary representation of integers' problem: It does not answer how we can represent negative integers.

1.1.1 Sign-Magnitude Notation

Decimal arithmetics provide the minus sign (-) for representing negativity. However, electronics of the von Neumann machinery requires that the minus is also represented by either a '1' or '0'. We can reserve a bit, for example the left-most digit, for this purpose. If we do this, when the left-most digit is a '1', the rest of the digits are encoding the magnitude of the 'negative' integer. One point to be mentioned is that this requires a fixed size (a fixed number of digits) for the 'integer representation'. In this way, the electronics can recognize the sign bit and the magnitude part. This is called *the sign-magnitude notation*.

Sadly, this notation has certain disadvantages:

1. Addition and subtraction

Consider adding two integers. Based on their signs, we have the following possibilities:

- *positive* + *positive*
- *negative* + *positive*
- *positive* + *negative*
- *negative* + *negative*

Then, to be able to perform addition, the electronics must do the following:

- if both integers are *positive* then the result is *positive*, obtained by adding the magnitudes (and not setting the sign bit).
- if both integers are negative, then the result is negative, obtained by adding the magnitudes and setting the sign bit to *negative*.
- otherwise, if one is *negative* and the other is *positive* then:
    1. find the bigger magnitude,
    2. subtract the smaller magnitude from the bigger one, obtain the result,
    3. if the bigger magnitude integer was negative the result is negative (set the sign bit) else don't set the sign bit in result.

This was only for the case of addition, a similar electronic circuitry is needed for subtracting two integers.

This technique has the following drawbacks: * Requires separate electronics for subtraction. * Requires electronics for magnitude-based comparison and an algorithm implementation for setting the sign bit. * Electronically, it has to differentiate among addition and subtraction.

2. Representing number zero

Another limitation of the sign-magnitude notation is that number zero $(0)_{10}$ has two different representations, e.g. in a 4-bit representation:

- $(1\,000)_2 ==> (-0)_{10}$
- $(0\,000)_2 ==> (+0)_{10}$

In modern computers, we use a method that does not have these drawbacks.

1.1.2 Two's Complement Representation

Two's complement representation can be considered as an extension of the sign-magnitude notation:

- The positive integers are represented by their base-2 representation.
- For negative integers, we need a one-to-one mapping for all negative integers bounded by value to a binary representation (also bounded by value), so that :
    1. The sign bit is set to 1, and
    2. When the whole binary representation (including the sign bit) is treated as a single binary number it operates correctly under addition. When the result, obtained purely by addition, produces a sign bit, this means the result is the encoding of a negative integer.

There are two alternatives for this mapping: One's-complement and two's complement. In this section, we will introduce the more popular one, namely the two's complement representation:

If we are given n binary digits to be used (for a 32-bit computer, $n = 32$; for a 64-bit computer, it is 64), then we are able to represent integer values in the range $[-2^{n-1}, 2^{n-1}-1]$. Then, two's complement representation of an integer can be obtained as follows:

- If the integer is positive simply, convert it to base-2.
- If it is negative: Let the magnitude (its absolute value) be p, then
    1. convert p to base-2
    2. negate this base-2 representation by flipping all 1s to 0s and all 0s to 1s: $1 \leftrightarrow 0$.
    3. add 1 to the result of the negation.

Here are some examples for 8-bit numbers (note that the valid decimal range is $[-128, 127]$):

Integer	2′s complement	Pos./Neg.	Action:Binary Result
0	00000000	–	Direct base-2 encoding:
1	00000001	*Positive*	Direct base-2 encoding:
−1	11111111	*Negative*	*magnitude*:00000001 *inverse*:11111110 +1:11111111
−2	11111110	*Negative*	*magnitude*:00000010 *inverse*:11111101 +1:11111110
18	00010010	*Positive*	Direct base-2 encoding:
−18	11101101	*Negative*	*magnitude*:00010010 *inverse*:11101101 +1:11101101
−47	11010001	*Negative*	*magnitude*:00101111 *inverse*:11010000 +1:11010001
−111	10010001	*Negative*	*magnitude*:01101111 *inverse*:10010000 +1:10010001
111	01101111	*Positive*	Direct base-2 encoding:
112	01110000	*Positive*	Direct base-2 encoding:
−128	10000000	*Negative*	*magnitude*:10000000 *inverse*:01111111 +1:10000000
−129	Not Calculated	*Negative*	(Out of valid range):
128	Not Calculated	*Positive*	(Out of valid range):

(1.1.1)

1.1.3 Why does Two's Complement work?

The Two's Complement method may sound arbitrary at first. However, there are solid reasons for why it works. To be able to explain why it works, let us first revisit our basic computer organization from Chapter 1:

The CPUs can only understand and work with fixed-length representations: Assume that our computer is an 8-bit computer such that registers in the CPU holding data and the arithmetic-logic unit can only work with 8 bits. The fixed-length design of the CPU has a severe implication. Consider the following 8-bit number (which is $2^8 - 1$) in a register in the CPU:

1	1	1	1	1	1	1	1

If you add 1 to this number, the result would be:

0	0	0	0	0	0	0	0

In other words, we lose the 9th bit since the CPU cannot fit that into the 8-bit representation. Mathematically, this arithmetic corresponds to *modular arithmetic*: For our example, the modulo value is 2^8 and the addition that we just performed can be written mathematically as:

$$(2^8 - 1) + 1 \equiv 0. \qquad (\text{mod } 2^8) \tag{1.1.2}$$

What does this have to do with the Two's Complement method? Consider taking the negative of a number a in a 2^n-modulo system:

$$-(a) \equiv 0 - a \equiv 2^n - a. \qquad (\text{mod } 2^n) \tag{1.1.3}$$

Now let us rewrite this in a form that will seem familiar to us:

$$2^n - a \equiv 2^n - 1 + 1 - a \equiv \underbrace{(2^n - 1 - a)}_{\text{Invert the bits of } a} + \underbrace{1}_{\text{Add 1 to the inverted bits}} \qquad (\text{mod } 2^n) \tag{1.1.4}$$

In other words, Two's Complement representation uses the negative value of a number relying on the modular arithmetic, i.e. the fixed-length representation of the CPUs. This technique is not new and was used in mechanical calculators long before there were computers.

1.1.4 Benefits of the Two's Complement Representation

Let us revisit the limitations of the sign-magnitude representation:

- **Addition and subtraction**: With the two's complement method, we do not need to check the signs of the numbers and perform addition and subtraction using just an addition circuitry. E.g. $(+2)_{10} + (-3)_{10}$ is just equal to $(-1)_{10}$ without doing anything extra other than plain addition (for a 4-bit representation):

$$\begin{array}{rc|c} & (0010)_2 & (+2)_{10} \\ & (1101)_2 & (-3)_{10} \\ + & & \\ \hline & (1111)_2 & (-1)_{10} \end{array} \tag{1.1.5}$$

- **Representation of +0 and -0**: Another issue with the sign-magnitude representation was that +0 and -0 had different representations. In the Two's Complement representation, we see that this is resolved – for example (for a 4-bit representation):
- $(+0)_{10} = (0000)_2$
- $(-0)_{10} = -(0000)_2 = (1111)_2 + (1)_2 = (0000)_2$, where we used Two's Complement to convert $-(0000)_2$ into $(1111)_2 + (1)_2$ by flipping the bits and adding 1.

The fifth bit is lost because we have only four bits for representation.

1.1.5 PRACTICE TIME

Please follow the Colab link at the top of the page to have a practical session on Two's Complement representation.

1.2 Representing real numbers

Floating point is the data type used to represent non-integer real numbers. On today's computers, all non-integer real numbers are represented using the floating point data type. Since all integers are real numbers, we could represent integers also using the floating point data type. Although you could do so, this is usually not preferred, since floating point operations are more time consuming compared to integer operations. Also, there is the danger of precision loss, which we will discuss later.

Almost all processors have adopted the IEEE 754 binary floating point standard for binary representation of floating point numbers. The standard allocates 32 bits for the representation, although there is a recent 64-bit definition which is based on the same layout idea just with some more bits.

Let us see how we represent a floating point number with an example. Let us consider a decimal number with fraction: 12263.921875. This number can be represented in binary as two binary numbers: the whole part in binary and the fractional part in binary. Then, we can join them with a period.

To see this, let us first dissect decimal (base-10) 12263.921875:

10^4	10^3	10^2	10^1	10^0	.	10^{-1}	10^{-2}	10^{-3}	10^{-4}	10^{-5}	10^{-6}
1	2	2	6	3	.	9	2	1	8	7	5

(1.2.1)

Keeping the denotational similarity, but switching to (base-2), in other words to binary, we can express the same number as:

2^{13}	2^{12}	2^{11}	2^{10}	2^9	2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	.	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}
1	0	1	1	1	1	1	1	1	0	0	1	1	1	.	1	1	1	0	1	1

(1.2.2)

How did we obtain the fractional part? By multiplying the fractional by two until we obtain zero for the fraction part, as illustrated in Fig. 1.2.1. This is in certain ways the reverse of what we did for converting the whole part into binary, in Section 1.1.

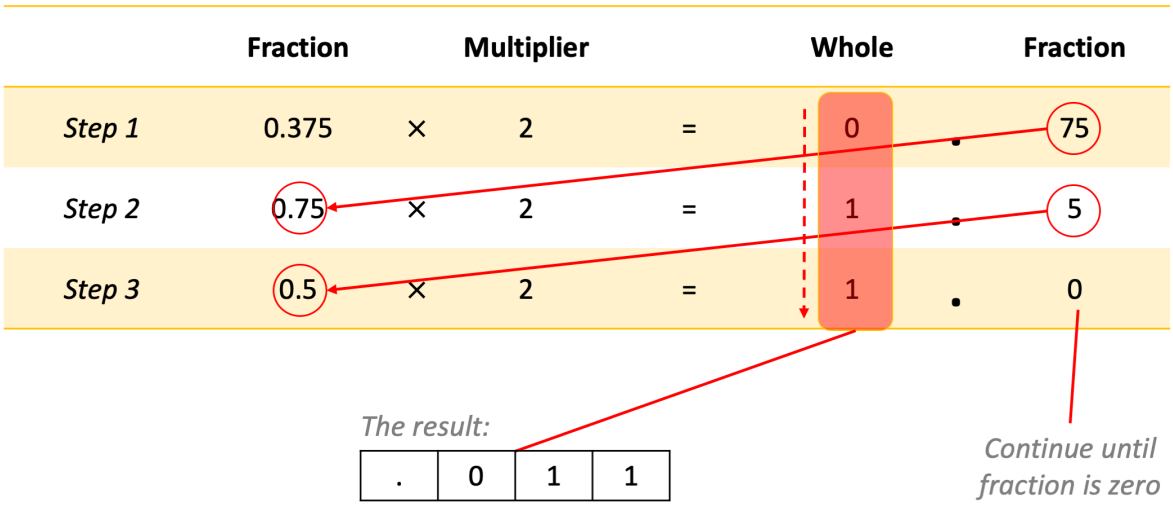


Fig. 1.2.1: The multiplication method for converting a fractional number into binary.

At this stage, it is worth mentioning that it is not always possible to convert the fractional part into binary with finite number of bits. In other words, it is quite possible to have a finite number of fractional digits in one base, and infinitely many in another base for the same value. We will come back to this point later.

Depending on the size of the whole part, the period could be anywhere. Therefore, the next step towards obtaining the IEEE 754 representation is to reposition the period that separates the whole part from the fraction, to be placed just after the leftmost '1'. To be able to do this without changing the value of the number, a multiplicative factor of 2^n has to be introduced, where n is how many bits the period is moved. If it is moved to the left, it has a positive value, otherwise it is negative. This factor is important, because it will contribute to the representation.

For our example, the period has to be moved left by 13 digits. Therefore, the multiplicative factor (to keep

the value the same) is $\times 2^{+13}$. At this stage our representation has become:

	2^0	.	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}	2^{-7}	2^{-8}	2^{-9}	2^{-10}	2^{-11}	2^{-12}	2^{-13}	2^{-14}	2^{-15}	2^{-}
$2^{+13}\times$	1	.	0	1	1	1	1	1	1	1	0	0	1	1	1	1	1	1	0

(1.2.3)

Since this is doable for all values (except 0.0, which will be dealt with with an exception), there is no need to keep a record of the whole part, i.e. the only remaining '1' value to the left of the period. Also, the period is always there, so we can simply drop it. The *mantissa* part of the representation is obtained by keeping exactly the first 23 bits of what is left. This leads to the following 23 bits for our example (note the extra zeros at the end, added to fill complete the representation to 23 bits):

0	1	1	1	1	1	1	1	0	0	1	1	1	1	1	1	0	1	1	0	0	0	0

1.2.1 The IEEE754 Representation

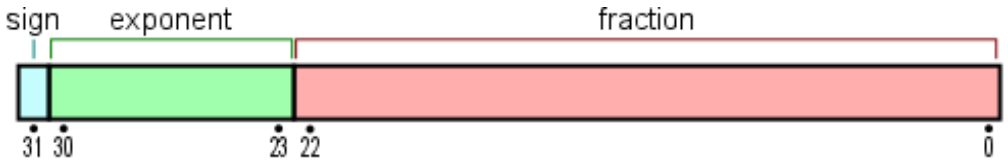


Fig. 1.2.2: The 32-bit IEEE754 floating point representation.

The exponent of the multiplicative factor, namely n (which can be negative) in 2^n, becomes a part of the representation – see Fig. 1.2.2. To get rid of the minus sign problem, a constant value, 127, is added to n. This value becomes the *exponent* part of the representation.

Adding a constant value to a number to be able to represent positive and negative numbers in binary is called the excess representation, or k-excess representation, with k being the constant number that is added, e.g. $k = 127$. Why we use k-excess representation is going to be clear when we compare it with two's complement representation in Table 1.2.1. You should see from the table that if the binary representation of a decimal number is larger, the decimal number is also larger with the k-excess representation; however, this is not the case for the two's complement representation. This is important for comparing two floating point numbers: Without decoding the whole floating point representation, which is expensive, we can just look at the k-excess representation of the exponents and the fractional parts to compare numbers.

Table 1.2.1: A comparison between the k-excess representation and the two's complement representation.

Decimal number	k-excess ($k = 8$)	Two's complement
7	1111	0111
6	1110	0110
5	1101	0101
4	1100	0100
3	1011	0011
2	1010	0010
1	1001	0001
0	1000	0000
−1	0111	1111
−2	0110	1110
−3	0101	1101
−4	0100	1100
−5	0011	1011
−6	0010	1010
−7	0001	1001
−8	0000	1000

In our example the factor was 2^{13}. Adding $k = 127$ yields $13 + 127 = 140$, which has an 8-bit representation of 10001100. This will become the exponent portion (the 8 green bits in Figure 3.4) of the IEEE754 representation.

It is possible that the value that is going to be represented is a negative value. This information is stored as a '1' in the first bit of the representation. For a positive value, a '0' bit is used.

Finally, our example gets a IEEE754 representation as:

0	1	0	0	0	1	1	0	0	0	1	1	1	1	1	1	1	0	0	1	1	1	1	1	0	1	1	0	0	0	0

How is real number '0.0' represented?

Floating point number zero is represented as all zero bits in the positional range [1-31]. The zeroth bit, namely the sign bit, can be either '0' or '1'.

1.2.2 Information loss in floating-point representations

The condition for a floating point number to be represented "exactly" by the IEEE754 standard is for the number to be equal to:

$$f = \sum_{k=n}^{m} digit_k \times 2^k, \tag{1.2.4}$$

where m and n are two integers so that $|m-n| \leq 24$ (it is fine if the equation is not very clear – we will show some examples below). In addition to this, there is the constraint that $|n| \leq 127$ (think about why). Nothing can be done about the second constraint, but as far as the first constraint is concerned, practically we can approximate the number leaving off (truncating) the less significant bits (those to the right, or mathematically those with a smaller k value in the sum above), so that $|m - n| \leq 24$ is satisfied.

Actually, many rational numbers, even, are not expressible in the form of the sum above: As an example, try deriving the representation of 4.1 and see whether you can represent it in binary with finite number of bits.

In addition, we have infinitely many irrational numbers which do not have finite fractions: e.g. $\sqrt{2}, \sqrt{3}, \pi, e$. To be able to use all these numbers in computers, we approximate them. Do not worry, it is something quite common in applied sciences.

Approximation comes with some dangers: When you subtract two truncated numbers which are close to each other, then the result you obtain has a high imprecision. This also is the case with multiplication and division of relatively big numbers.

Though we have not started working with Python yet, we will display some self explanatory examples and comment on them:

- ```
  >>> 0.9375 - 0.9
  0.03749999999999998
  ```

  The first line is some input that we typed in, to be carried out, and the following line is what the Python interpreter returned as the result. Surprise! (Actually a bad one!) In contrary to our expectation, the result is not 0.0375. The reason is that 0.9375 is one of the rare numbers that could be represented as a finite sum $\sum_{k=1}^{N}(digit_k \cdot 2^{-k}), \quad digit_k \in$
  $0, 1$ such that N stays in the IEEE representation limit. Actually for 0.9375, $digit = [1, 1, 1, 1]$ and $N = 4$.

  On the other hand, quite unexpectedly, 0.9 is not so. It cannot be expressed as a similar sum where N stays in the IEEE representation limit. Actually, N extents to $+\infty$ for 0.9. Hence the $digit_n$ sequence has to be truncated before it can stored in the IEEE representation. The bits that do not fit into the representation are simply ignored: In other words, the number loses its precision.

  This is combined with a similar representation problem of 0.0375, which leads to another loss and we arrive at 0.03749999999999998. Bottom line: many numbers cannot exactly be represented in the IEEE754 format, and this causes precision loss.

- Things can get even worse. Consider:

  ```
  >>> 2000.0041 - 2000.0871
  -0.0829999999998563

  >>> 2.0041 - 2.0871
  -0.08299999999999974
  ```

  Actually both results should have been -0.0830. Despite having an imprecision, the imprecision is not consistent. This is because the loss in the first example (the one where the whole part is 2000) is bigger than the loss in the second one since 2000 needs more bits to be represented compared to 2.

- Pi (π) is a transcendental number. The fractional part never stops, in any base. Let us give it a shot on the Python interpreter:

  ```
  >>> PI = 3.14159265358979323846264338327950288419716939937510582097494459230781640628 6
  >>> print(PI)
  3.141592653589793
  ```

  Ok, from this we understand that the IEEE representation could only accommodate so many bits for the 15 places in the fractional part. That looks quite precise, but let us take a look at the sin and cos values:
  ```

```
>>> sin(PI)
1.2246467991473532e-16
>>> cos(PI)
-1.0
```

Interestingly, we received a slight error for the sine value, which is different from 0.0. But when it comes to the cosine value, we were lucky that imprecision somewhat cancelled out and gave us the correct result.

- We are thought since primary school that addition is associative. Hence $A + (B + C)$ is the same as $(A + B) + C$. In floating point arithmetic, this may not be so:

```
>>> A = 1234.567
>>> B = 45.67834
>>> C = 0.0004
>>> AB = A + B
>>> BC = B + C
>>> print (AB+C)
1280.2457399999998

>>> print (A+BC)
1280.2457400000001
```

This is again a combination of the precision loss phenomena introduced above. Most of the intermediate steps of a calculation have precision losses of their own.

As a final word about using floating point numbers, it is worth stressing a common mistake commonly made but has nothing to do with the precision loss mentioned. Let us assume you provide your program the Pi number to be 3.1415. You do your calculations and obtain floating point numbers with 14-15 digit fractional parts. Knowing about the precision loss, you assume that maybe a couple of the last digits are wrong but at least 10 digits after the decimal point are correct. However, this is a mistake: You made an approximation in the 5th digit after the decimal point in number Pi which will propagate through your calculation. It can get worse (for example if you subtract two very close numbers and use it in the denominator), but can never get better. Your best chance is to get a correct result in the 4th digit after the decimal point. The following digits, as far as precision is concerned, are bogus.

So, what can we do? Here are some rules of thumb about using floating point numbers:

- It is in your best interest to refrain from using floating points. If it is possible transform the problem to the integer domain.
- Use the most precise type of floating point provided by your high-level language, some languages provide you with 64 bit or even 128 bit floats, use them.
- Use less precision floating points only when you are in short of memory.
- Subtracting two floating points close in value has a potential danger.
- If you add or subtract two numbers which are magnitude-wise not comparable (one very big the other very small), it is likely that you will lose the proper contribution of the smaller one. Especially when you iterate the operation (repeat it many times), the error will accumulate.
- You are strongly advised to use well-known, commonly used scientific computing libraries instead of coding floating point algorithms by yourself.

1.2.3 PRACTICE TIME

Please follow the Colab link at the top of the page to have a practical session on the IEEE754 floating point representation.

1.3 Numbers in Python

Python provides the following representations for numbers:

- **Integers:** You can use integers as you are used to from your math classes. Interestingly Python adopts a seamless internal representation so that integers can effectively have any number of digits. The internal mechanism of Python switches from the CPU-imposed fixed-size integers to some elaborated big-integer representation silently when needed. You do not have to worry about it. Furthermore, bear in mind that "73." is **not** an integer in Python. It is a floating point number (73.0). An integer cannot have a decimal point as part of it.
- **Floating point numbers (float in short):** In Python, numbers which have decimal point are taken and represented as floating point numbers. For example, 1.45, 0.26, and -99.0 are float but 102 and -8 are not. We can also use the scientific notation ($a \times 10^b$) to write floating point numbers. For example, float 0.0000000436 can be written in scientific notation as 4.36×10^{-8} and in Python as 4.36E-8 or 4.36e-8.
- **Complex numbers:** In Python, complex numbers can be created by using j after a floating point number (or integer) to denote the imaginary part: e.g. 1.5-2.6j for the complex number $(1.5 + 2.6i)$. The j symbol (or i) represents $\sqrt{-1}$. There are other ways to create complex numbers, but this is the most natural way, considering your previous knowledge from high school.

1.4 Representing text

As we said in the first lines of this chapter, programming is mostly about a world problem which generally includes human related or interpretable data to be processed. These data do not consist of numbers only but can include more sophisticated data such as text, sound signals and pictures. We leave the processing of sound and pictures out of this book's scope. Text, though, is something we have to study.

1.4.1 Characters

Written natural languages consist of basic units called graphemes. Alphabetic letters, Chinese-Japanese-Korean characters, punctuation marks, numeric digits are all graphemes. There are also some basic actions that go commonly hand in hand with textual data entry. "Make newline", "Make a beep sound", "Tab", "Enter" are some examples. These are called "unprintables".

How can we represent graphemes and unprintables in binary? Graphemes are heavily culture dependent. The shapes do not have a numerical foundation. As far as computer science is concerned, the only way to represent such information in numbers is to make a table and build this table into electronic input/output devices. Such a table will have two columns: The graphemes and unprintables in one column and the assigned binary code in the other, e.g.:

Grapheme or Unprintable	Binary Code
⋮	⋮

Throughout the history of computers, there has been several such tables, mainly constructed by computer manufacturers. In time, most of them vanished and only one survived: The ASCII (American Standard Code for Information Interchange) table which was developed by the American National Standards Institute (ANSI). This American code, developed by Americans, is naturally quite 'American'. It incorporates all characters of the American-English alphabet, including, for example, the dollar sign, but stops there. The table does not contain a single character from another culture (for example, even the pound sign '£' is not in the table).

The ASCII table has 128 lines. It maps 128 American graphemes and unprintables to 7-bit long codes. Since the 7-bit long code can be interpreted also as a number, for convenience, this number is also displayed in the ASCII table – see Fig. 1.4.1.

Dec	Bin	Char	Dec	Bin	Char	Dec	Bin	Char	Dec	Bin	Char
0	0000 0000	[NUL]	32	0010 0000	space	64	0100 0000	@	96	0110 0000	`
1	0000 0001	[SOH]	33	0010 0001	!	65	0100 0001	A	97	0110 0001	a
2	0000 0010	[STX]	34	0010 0010	"	66	0100 0010	B	98	0110 0010	b
3	0000 0011	[ETX]	35	0010 0011	#	67	0100 0011	C	99	0110 0011	c
4	0000 0100	[EOT]	36	0010 0100	$	68	0100 0100	D	100	0110 0100	d
5	0000 0101	[ENQ]	37	0010 0101	%	69	0100 0101	E	101	0110 0101	e
6	0000 0110	[ACK]	38	0010 0110	&	70	0100 0110	F	102	0110 0110	f
7	0000 0111	[BEL]	39	0010 0111	'	71	0100 0111	G	103	0110 0111	g
8	0000 1000	[BS]	40	0010 1000	(	72	0100 1000	H	104	0110 1000	h
9	0000 1001	[TAB]	41	0010 1001	)	73	0100 1001	I	105	0110 1001	i
10	0000 1010	[LF]	42	0010 1010	*	74	0100 1010	J	106	0110 1010	j
11	0000 1011	[VT]	43	0010 1011	+	75	0100 1011	K	107	0110 1011	k
12	0000 1100	[FF]	44	0010 1100	,	76	0100 1100	L	108	0110 1100	l
13	0000 1101	[CR]	45	0010 1101	-	77	0100 1101	M	109	0110 1101	m
14	0000 1110	[SO]	46	0010 1110	.	78	0100 1110	N	110	0110 1110	n
15	0000 1111	[SI]	47	0010 1111	/	79	0100 1111	O	111	0110 1111	o
16	0001 0000	[DLE]	48	0011 0000	0	80	0101 0000	P	112	0111 0000	p
17	0001 0001	[DC1]	49	0011 0001	1	81	0101 0001	Q	113	0111 0001	q
18	0001 0010	[DC2]	50	0011 0010	2	82	0101 0010	R	114	0111 0010	r
19	0001 0011	[DC3]	51	0011 0011	3	83	0101 0011	S	115	0111 0011	s
20	0001 0100	[DC4]	52	0011 0100	4	84	0101 0100	T	116	0111 0100	t
21	0001 0101	[NAK]	53	0011 0101	5	85	0101 0101	U	117	0111 0101	u
22	0001 0110	[SYN]	54	0011 0110	6	86	0101 0110	V	118	0111 0110	v
23	0001 0111	[ETB]	55	0011 0111	7	87	0101 0111	W	119	0111 0111	w
24	0001 1000	[CAN]	56	0011 1000	8	88	0101 1000	X	120	0111 1000	x
25	0001 1001	[EM]	57	0011 1001	9	89	0101 1001	Y	121	0111 1001	y
26	0001 1010	[SUB]	58	0011 1010	:	90	0101 1010	Z	122	0111 1010	z
27	0001 1011	[ESC]	59	0011 1011	;	91	0101 1011	[	123	0111 1011	{
28	0001 1100	[FS]	60	0011 1100	<	92	0101 1100	\	124	0111 1100	|
29	0001 1101	[GS]	61	0011 1101	=	93	0101 1101	]	125	0111 1101	}
30	0001 1110	[RS]	62	0011 1110	>	94	0101 1110	^	126	0111 1110	~
31	0001 1111	[US]	63	0011 1111	?	95	0101 1111	_	127	0111 1111	[DEL]

Fig. 1.4.1: The ASCII table.

Do not worry, you do not have to memorize it, even professional computer programmers do not. However, some properties of this table has to be understood and kept in mind:

- The general layout of the ASCII table:

Dec. Range	Property
0-31	Unprintables
32	Space char.
33-47	Punctuations
48-57	Digits 0-9
58-64	Punctuations
65-90	Upper case letters
91-96	Punctuations
97-122	Lower case letters
123-127	Punctuations

- There is no logic in the distribution of the punctuations.
- It is based on the English alphabet, characters of other languages are simply not there. Moreover, there is no mechanism for diacritics.
- Letters are ordered and uppercase letters come first in the table (have a lower decimal value)
- Digits are also ordered but are not represented by their numerical values. To obtain the numerical value for a digit, you have to subtract 48 form its ASCII value.
- The table is only and only about 128 characters, neither more nor less. There is nothing like Turkish-ASCII, French-ASCII. The extensions, where the 8th bit is set has nothing to do with the ASCII table.
- Python makes use of ASCII character representation.

The frustrating discrepancies and shortcomings of the ASCII table have led the programming society to seek a solution. A non-profit group, the Unicode Consortium, was founded in the late 80s with the goal of providing a substitute for the current character tables, which is also compliant (backward compatible) with them. The Unicode Transformation Format (UTF) is their suggested representation scheme.

This UTF representation scheme has variable length and may include components of 1-to-4 8-bit wide (in the case of UTF-8) or 16-bit wide components of 1-to-2 (in the case of UTF-16). UTF is now becoming part of many recent high-level language implementations, including Python, Java, Perl, TCL, Ada95 and C#, gaining wide popularity.

1.4.2 Strings

Text is as vital a data as numbers are. Text is expressed as character sequences. These sequences are named as strings. But we have here a problem. As introduced, numbers (integers and floating points) have a niche in the CPU. There are instructions designed for them: we can store and retrieve them to/from the memory; we can perform arithmetical operations among them. A character data can be represented and processed as well because they are mapped to one byte integers. But when it comes to strings, the CPU does not have any facility for them.

The only reasonable way is to store the codes of each character that make up a string into the memory in consecutive bytes. Does this solve the problem of ‘representation’? Unfortunately no. The trouble is determining how to know where the string ends. Two methods come to mind:

1. Prior to the string characters, store the length (the number of characters in the string) as an integer of fixed number of bytes.
2. Store a special byte value, which is not used to represent any other character, at the end of the string characters.

1.5 Containers

The representation of a string is maybe the most simple representation of a data that is not directly supported by the CPU. However, it gives an idea for what can be done in similar cases. The key concept is to find a layout that is a mapping from the space of the data to an organization in the memory.

Also notice that whatever is placed in the memory has an 'address' and the value of the address can also become a part the organization. As far as the string is concerned, the usage of the 'address concept' was simple: A single address marks the start of the string. When we want to process the string, we go to this address and then start to process the character codes sequentially.

It is possible to have data organizations which include addresses of other data organizations. In other words, it is possible to jump from one group of data to some other in the memory. This type of organizations are name *Data Structures* in Computer Science. So, string is a data structure. As far as Python is concerned, there are several other data structures. They are coined in Python as *containers*. In addition to strings Python provide lists, tuples, sets and dictionaries as containers. Except strings, which is introduced above, the others have more complex data structures which will not be covered, because it falls outside of the scope of this book.

1.6 Representing truth values (Booleans)

Boolean is another data type that has its roots in the very structure of the CPU. Answers to all questions asked to the CPU are either *true* or *false*. The logic of a CPU is strictly based on binary evaluation system. This logic system is coined as *Boolean logic*. It was introduced by George Boole in his book "The Mathematical Analysis of Logic" (1847).

It is tightly connected to the binary 0 and 1 concepts. In all CPUs, falsity is represented with a 0 whereas truth is represented with a 1 and on some with any value which is not 0.

1.7 Important Concepts

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

- Sign-magnitude notation and two's complement representation for representing integers.
- The IEEE754 standard for representing real numbers.
- Precision loss in representing floating point numbers.
- Representing characters with the ASCII table.
- Representing truth values.

1.8 Further Reading

- Two's Complement: http://en.wikipedia.org/wiki/Two%27s_complement
- The Method of Complements: https://en.wikipedia.org/wiki/Method_of_complements
- Excess-k Representation: https://en.wikipedia.org/wiki/Offset_binary
- IEEE 754 Floating Point Standard: http://en.wikipedia.org/wiki/IEEE_754-2008
- ASCII: http://en.wikipedia.org/wiki/ASCII
- UTF—UCS Transformation Format: http://en.wikipedia.org/wiki/UTF-8

Exercises

- By hand, find the 5-bit Two's Complement representation of the following numbers: 4, −5, 1,−0, 11.
- Represent 6 and (-7) in a 5-bit sign-magnitude notation and add them up in binary, without looking at their signs.
- Find the IEEE 754 32-bit representation of the following floating point numbers: 3.3, 3.37, 3.375.

2 | Dive into Python

After having covered some background on how a computer works, how we solve problems with computers and how we can represent data in binary, we are ready to interact with Python and to learn representing data and performing actions in Python.

The Python interpreter that we we introduced in Chapter 2 waits for our computational demands if not executing something already. Those demands that describe an algorithm have to be expressed in Python as a sequence of 'actions' where each action can serve two broad purposes:

- **Creating or modifying data**: These actions take in data, perform sequential, conditional or repetitive execution on the data, and produce other data. Computations that form the bases for our solutions are going to be these kinds of actions.
- **Interacting with the environment**: Our solutions will usually involve interacting with the user or the peripherals of the computer to take in (input) data or take out (output) data.

Irrespective of its purpose, an action can be of two types:

- **Expression**: An expression (e.g. 3 + 4 * 5) specifies a calculation, which, when evaluated (performed), yields some data as a result. An expression can consist of:
  - basic data (integer, floating point, boolean etc.) or container data (e.g. string, list, set etc.).
  - expressions involving operations among data and other expressions.
  - functions acting on expressions.
- **Statement**: Unlike an expression, a statement does *not* return data as a result and can be either basic or compound:
  - **Basic statement**: A basic statement can be e.g. for storing the result of an expression in a memory location (an assignment statement for further use (in following actions), deleting an item from a collection of data etc. Each statement has its special syntax that generally involves a special keyword.
  - **Compound statement**: Compound statements are composed of other statements and executing the compound statement means executing the statements in the compound.

Naming and Printing Data

For better illustrations and examples, we will use two concepts in the first part of the chapter before they are introduced. Let us briefly describe them and leave the coverage of the details to their individual sections:

- Variables: In programming languages, we can give a name to data and use that name to access it, e.g.:

```
>>> a = 3
>>> 10 + a
13
```

We call such names variables and the action a = 3 is called an assignment. We defer a more detailed coverage until Section 4.4.

- Printing data: Python provides the print() function to display data items on screen:

```
print(item1, item2, ..., itemN)
```

For example:

```
>>> print('Python', 'is', 'so', 'fun')
Python is so fun
```

2.1 Basic Data

Let us remember what basic data types we had:

- Numbers
  - Integers
  - Floating points
  - Complex numbers
- Booleans

Python is a language in which all arithmetic operations among the same type of numbers are provided very much as expected from our math knowledge. Furthermore, mixed type operations (e.g. subtracting a floating point number form an integer) are also defined.

In programming, being able to ask questions about data is of vital importance. The atomic structures for asking questions are the *comparison operations* (e.g. "is a value equal to another value", or "is a value greater than another value", etc.). Operators that serve these purposes do exist in Python and provide resulting values that are True or False (Booleans). It is also possible to combine such questions under *logical operations*. The and, or and not operators stand for conjunction, disjunction and negation, correspondingly. Needless to say, these operators also return Boolean values.

2.1.1 Numbers in Python

Python provides the following representations for numbers (the following is an essential reminder from the previous chapter):

- **Integers:** You can use integers as you are used to from your math classes. Interestingly Python adopts a seamless internal representation so that integers can effectively have any number of digits. The internal mechanism of Python switches from the CPU-imposed fixed-size integers to some elaborated big-integer representation silently when needed. You do not have to worry about it. Furthermore, bear in mind that "73." is **not** an integer in Python. It is a floating point number (73.0). An integer cannot have a decimal point as part of it.
- **Floating point numbers (float in short):** In Python, numbers which have decimal point are taken and represented as floating point numbers. For example, 1.45, 0.26, and -99.0 are float but 102 and -8 are not. We can also use the scientific notation ($a \times 10^b$) to write floating point numbers. For example, float 0.0000000436 can be written in scientific notation as 4.36×10^{-8} and in Python as 4.36E-8 or 4.36e-8.

- **Complex numbers:** In Python, complex numbers can be created by using j after a floating point number (or integer) to denote the imaginary part: e.g. 1.5-2.6j for the complex number $(1.5 + 2.6i)$. The j symbol (or i) represents $\sqrt{-1}$. There are other ways to create complex numbers, but this is the most natural way, considering your previous knowledge from highschool.

More on Integers and Floating Point Numbers

Python provides int data type for integers and float data type for floating point numbers. You can easily play around with int and float numbers and check their type as follows:

```
>>> 3+4
7
>>> type(3+4)
<class 'int'>
>>> type(4.1+3.4)
<class 'float'>
```

where <class 'int'> indicates type int.

In Python version 3, integers do not have fixed-size representation and their size is only limited by your available memory. In Python version 2, there were two integer types: int, which used the fixed length representation supported by the CPU, and long type, which was unbounded. Since this book is based on Python version 3, we will assume that int refers to an unbounded representation.

As for the float type, Python uses the 64-bit IEEE754 standard, which allows representing numbers in the range [2.2250738585072014E-308, 1.7976931348623157E+308].

Useful Operations:

The following operations can be useful while working with numbers:

- abs(<Number>): Takes the absolute value of the number.
- pow(<Number1>, <Number2>): Takes the power of <Number1>, i.e. <Number1>$^{\text{<Number2>}}$
- round(<FloatNumber>): Rounds the floating point number to the *closest* integer.
- Functions from the math library: sqrt(), sin(), cos(), log(), etc (see the Python documentation[18] for a full list). This requires *importing* from the built-in math library first as follows:

```
>>> from math import *
>>> sqrt(10)
3.1622776601683795
>>> log10(3.1622776601683795)
0.5
```

[18] https://docs.python.org/3/library/math.html

2.1.2 Boolean Values

Python provides the bool data type which allows only two values: True and False. For example:

```
>>> type(True)
<class 'bool'>
>>> 3 < 4
True
>>> type(3 < 4)
<class 'bool'>
```

Also due to decades of programming experience, Python converts several instances of other data types to some certain boolean values, if used in place of a boolean value. For example,

- 0 (the integer zero)
- 0.0 (the floating point zero)
- "" (the empty string)
- [] (the empty list)
- {} (the empty dictionary or set)

are interpreted as False. All other values of similar kinds are interpreted as True.

Useful Operations:

With boolean values, we can use not (negation or inverse), and and or operations:

```
>>> True and False
False
>>> 3 > 4 or 4 < 3
False
>>> not(3 > 4)
True
```

and returns a True value only if both of its operands are True, otherwise it returns False. or returns True if any or both of its operands are True. The following table gives result of the boolean operations for the given operand pair:

a	b	a and b	a or b
True	True	True	True
True	False	False	True
False	True	False	True
False	False	False	False

2.2 Container data (str, tuple, list, dict, set)

Up to this point we have seen the basic data types. They are certainly needed for computation but many world problems for which we seek computerized solutions need more elaborate data. Just to mention a few:

- Vectors
- Matrices
- Ordered and unordered sets
- Graphs
- Trees

Vectors and matrices are used in almost all simulations/problems of the physical world; sets are used to keep any property information as well as orders of items, equivalences; graphs are necessary for many spatial problems; trees are vital to representing hierarchical relational structures, action logics, organizing data for a quick search.

Python provides five container types for these:

1. **String (``str``):** A string can hold a sequence of characters or only a single character. A string cannot be modified after creation.
2. **List (``list``):** A list can hold ordered sets of all data types in Python (including another list). A list's elements can be modified after creation.
3. **Tuple (``tuple``):** The tuple type is very similar to the list type but the elements cannot be modified after creation (similar to strings).
4. **Dictionary (``dict``):** A very very efficient method to form a mapping from a set of numbers, booleans, strings and tuples to any set of data in Python. Dictionaries are easily modifiable and extendable. Querying the mapping of an element to the 'target' data is performed in almost constant time (regardless of how many elements the dictionary has). In Computer Science terms, it is a *hash table*.
5. **Set (``set``):** The set type is equivalent to sets in mathematics. The element order is undefined. *(We deemphasize the use of* **set***)*.

The first three, namely String, List and Tuple are called *sequential containers*. They consist of consecutive elements indexed by integer values starting at 0. Dictionary is not sequential, element indexes are arbitrary. For simplicity we will abbreviate sequential containers as *s-containers*.

All these containers have external representations which are used in inputting and outputting them (with all their content). Below we will walk through some examples to explain them.

Mutability vs. Immutability

Some container types are created 'frozen'. After creating them you can wholly destroy them but you cannot change or delete their individual elements. This is called *immutability*. Strings and tuples are immutable whereas lists, dictionaries and sets are *mutable*. With a mutable container, adding new elements and changing or deleting existing ones is possible.

2.2.1 Accessing elements in sequential containers

All containers except set reveal their individual elements by an indexing mechanism with brackets:

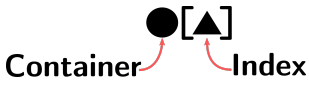


Fig. 2.2.1: How elements of a container are accessed

For the s-containers, the index is an ordinal number where counting starts with zero. For dictionaries, the index is a Python data item from the source (domain) set. A *negative index* (usable only on s-containers) has a meaning that the (counting) value is relative to the end. A negative index can be converted to a positive index by adding to the negative value the length of the container. It is nothing but index obtained by adding the length of the container.

Below we have an s-container that has a length of $(n+1)$
(careful: indexing started at 0!):

$$\begin{array}{cccccc} \square_0 & \square_1 & & \cdots & \square_{n-1} & \square_n \\ [0] & [1] & \cdots & & [n-1] & [n] \\ [-(n+1)] & [-n] & \cdots & & [-2] & [-1] \end{array} \tag{2.2.1}$$

As you surely have observed, when you add $(n+1)$ to the negative index you obtain the positive one.

Slicing

s-containers provide a rich mechanism, called slicing, that allows accessing multiple elements at once. Thanks to this mechanism, you can define a start and end index and obtain the portion that lies in between (Fig. 2.2.2):

- The element at the start index is the first to be accessed.
- The end index is where accessing stops (the element at the end index is **not** accessed – i.e. the end index is not inclusive).
- It is also possible to optionally define an increment (jump amount) between indexes. After the element at $[start]$ is accessed first, $[start + increment]$ is accessed next. This goes on until the accessed position is equal or greater than the end index. For negative indexing, a negative increment has to work from the bigger index towards the lesser, so $(start\ index > end\ index)$ is expected.

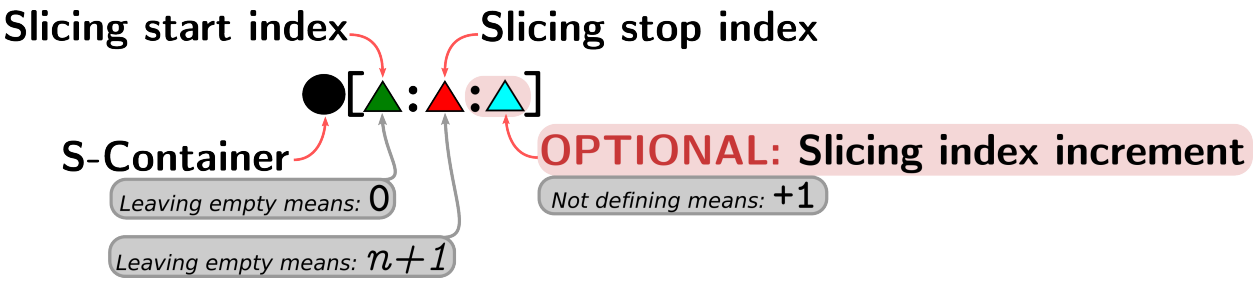


Fig. 2.2.2: Accessing multiple elements of an s-container is possible via the slicing mechanism, which specifies a starting index, an ending index and an index increment between elements.

Below, with the introduction of strings, we will have extensive examples on slicing.

If the s-container is immutable (e.g. string and tuple containers) then slicing creates a copy of the sliced data. Otherwise, i.e. if the s-container is mutable (i.e. the 'list' container) then slicing provides direct access to the original elements and therefore, they can be updated, which updates the original s-container.

2.2.2 Useful operations common to containers

The following operations are common to all or a subset of containers:

1- *Number of elements*:

For all containers len() is a built-in function that returns the count of elements in the container that is given as argument to it, e.g.:

```
>>> len("Five")
4
```

2- *Concatenation*:

String, tuple and list data types can be combined using '+' operation:

```
<Container1> + <Container2>
```

where the containers need to be of the same type. For example:

```
>>> "Hell" + "o"
'Hello'
```

3- *Repetition*: String, tuple and list data types can be repeated using "*" operation:

```
<Container1> * <Number>
```

where the container is copied '' many times. For example:

```
>>> "Yes No " * 3
'Yes No Yes No Yes No '
```

4- *Membership*: All containers can be checked for whether they contain a certain item as an element using in and not in operations:

```
<item> in <Container>
```

or

```
<item> not in <Container>
```

Of course, the result is either True or False. For dictionaries in tests if the domain set contains the element, for others it simply tests if element is a member.

2.2.3 String

As was explained in the previous chapter, a string is used to hold a sequence of characters. Actually, it is a container where each element is a character. However, Python does not have a special representation for a single character. Characters are represented externally as strings containing a single character only.

Writing strings in Python

In Python a string is denoted by enclosing the character sequence between a pair of quotes (') or double quotes ("). A string surrounded with triple double quotes (""" . . . """) allows you to have any combination of quotes and line breaks within a sequence, and Python will still view it as a single entity.

Here are some examples:

- "Hello World!"
- 'Hello World!'
- 'He said: "Hello World!" and walked towards the house.'
- "A"
- """ Andrew said: "Come here, doggy". The dog barked in reply: 'woof' """

The backslash (\) is a special character in Python strings, also known as the *escape character*. It is used in representing certain, the so called, unprintable characters: (\t) is a tab, (\n) is a newline, and (\r) is a carriage return. Table 2.2.1 provides the full list.

Table 2.2.1: The list of escape characters in Python.

Escape Sequence	Meaning
\\	Backslash (\)
\'	Single quote (')
\"	Double quote (")
\a	ASCII Bell (BEL)
\b	ASCII Backspace (BS)
\f	ASCII Formfeed (FF)
\n	ASCII Linefeed (LF)
\r	ASCII Carriage Return (CR)
\t	ASCII Horizontal Tab (TAB)
\v	ASCII Vertical Tab (VT)
ooo	ASCII character with octal value *ooo*
\x*hh*	ASCII character with hex value *hh*
\u*hhhh*	UNICODE character with hex value *hhhh*

Conversely, prefixing a special character with (\) turns it into an ordinary character. This is called *escaping*. For example, (\') is the single quote character. 'It\'s raining' therefore is a valid string and equivalent to "It's raining". Likewise, (") can be escaped: "\"hello\"" is a string that begins and ends with the literal double quote character. Finally, (\) can be used to escape itself: (\\) is the literal backslash character. For '\onnn', *nnn* is a number in base 8, for '\xnn' *nn* is a number in base 16 (including letters from A to F as digits for values 10 to 15).

In Python v3, all strings use the Unicode representation where all international symbols are possible, e.g.:

```
>>> a = "Fıstıkçı şahap"
>>> a
'Fıstıkçı şahap'
```

Examples with strings

Let us look at some examples to see what strings are in Python and what we can do with them:

```
>>> "This is a string"
"This is a string"
>>> "This is a string"[0]
'T'
>>> s = "This is a string"
>>> print(s[0])
T
>>> print(s[0],s[1],s[8],s[14],s[15])
T h a n g
```

Since strings are immutable an attempt to change a character in a string will badly fail:

```
>>> s = "This is a string"
>>> print(s)
This is a string
>>> s[2] = "u"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

Bottom line: You cannot change a character in a created string.

Let us go through a sequence of examples. We encourage you to run the following examples in the Colab version of this chapter. Feel free to change the text and rerun the examples.

```
my_beautiful_string  = "The quick brown fox jumps over the lazy dog"
print("THE STRING:", my_beautiful_string, len(my_beautiful_string), "CHARACTERS")
```

```
THE STRING: The quick brown fox jumps over the lazy dog 43 CHARACTERS
```

```
my_beautiful_string[0]
```

```
'T'
```

```
my_beautiful_string[4]
```

```
'q'
```

```
my_beautiful_string[0:4]
```

```
'The '
```

```
my_beautiful_string[:4]
```

```
'The '
```

```
my_beautiful_string[4:]
```

```
'quick brown fox jumps over the lazy dog'
```

```
my_beautiful_string[10:15]
```

```
'brown'
```

```
my_beautiful_string[:-5]
```

```
'The quick brown fox jumps over the laz'
```

```
my_beautiful_string[-8:-5]
```

```
'laz'
```

```
my_beautiful_string[:]
```

```
'The quick brown fox jumps over the lazy dog'
```

```
my_beautiful_string[::-1]
```

```
'god yzal eht revo spmuj xof nworb kciuq ehT'
```

```
my_beautiful_string[-6:-9:-1]
```

```
'zal'
```

```
my_beautiful_string[0:15:2]
```

```
'Teqikbon'
```

Strings are used to represent non-mathematical textual information. Common places where strings are used are:

- Textual communation in natural language with the user of the program.
- Understandable labeling of parts of data: City names, names of individuals, addresses, tags, labels.
- Denotation needs of human-to-human interactions.

Useful operations with strings:

- String creation: In addition to using quotes for string creation, the str() function can be used to create a string from its argument, e.g.:

```
>>> str(4)
'4'
>>> str(4.578)
'4.578'
```

- Concatenation, repetition and membership:

```
>>> 'Programming' + ' ' + 'with ' + 'Python is' + ' fun!'
'Programming with Python is fun!'
>>> 'really fun ' * 10
'really fun really fun really fun really fun really fun really fun really fun really fun really fun '
>>> 'fun' in 'Python'
False
>>> 'on' in 'Python'
True
```

- Evaluate a string: If you have a string that is an expression describing a computation, you can use the eval() function to evaluate the computation and get the result, e.g.:

```
>>> s = '3 + 4'
>>> eval(s)
7
```

Deletion and Insertion from/to strings

Since strings are immutable, this is not possible. The only way is to create a new string, making use of slicing and concatenation operation (using +), then replacing the new created string in same place of the former one. For example:

```
>>> a = 'Python'
>>> a[0] = 'S'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
>>> b = 'S' + a[1:]
>>> b
'Sython'
```

2.2.4 List and tuple

Both list and tuple data types have a very similar structure on the surface: They are sequential containers that can contain any other data type (including other tuples or lists) as elements. The only difference concerning the programmer is that tuples are immutable whereas lists are mutable. As previously said, being immutable means that, after being created, it is not possible to change, delete or insert any element in a tuple.

Lists are created by enclosing elements into a pair of brackets and separating them with commas, e.g. ["this", "is", "a", "list"]. Tuples are created by enclosing elements into a pair of parentheses, e.g. ("this", "is", "a", "tuple"). There is no restriction on the elements: They can be any data (basic or container).

Let us look at some examples for lists:

- [9,3,1,-1,6]
- []
- [2020]
- [3.1415, 2.718281828]
- [["pi,3.1415"], ["e",2.718281828], 1.41421356]
- ['the' 'quick','brown','fox','jumped','over','the','lazy','dog']
- [10, [5, [3, [[30, [[30, [], []]], []]], []], [8, [], []]], [30, [], []]]
- [[1,-1,0],[2,-3.5,1.1]]

and some examples for tuples:

- ('north','east','south','west')
- ()
- ('only',)
- ('A',65,"1000001","0x41")
- ("abx",[1.32,-5.12],-0.11)

Of course, tuples can become list members as well, or vice versa:

- [("ahmet","akhunlar",("deceased", 1991)), ("huri","huriyegil","not born")]
- (["ahmet","akhunlar",("deceased", 1991)], ["huri","huriyegil","not born"])

Programmers generally prefer using lists over tuples since they allow changing elements, which is often a useful facility in many problems.

Lists (and tuples) are used whenever there is a need for an ordered set. Here are a few usecases for lists and tuples:

- Vectors.
- Matrices.
- Graphs.
- Board game states.
- Student records, address book, any inventory.

Useful operations with lists and tuples

1- *Deletion from lists*

As far as deletion is concerned, you can use two methods:

- Assigning an empty list to the slice that is going to be removed:

```
>>> L = [111,222,333,444,555,666]
>>> L[2:5] = []
>>> print(L)
[111, 666]
```

- Using the del statement on the slice that is going to be removed:

```
>>> L = [111,222,333,444,555,666]
>>> del L[2:5]
>>> print(L)
[111, 666]
```

2- *Insertion into lists*

For insertion, you can use three methods:

- Using assignment with a degenerate use of slicing:

```
>>> L = [111,222,333,444,555,666]
>>> L[2:2] = [888,999]
>>> print(L)
[111, 222, 888, 999, 333, 444, 555, 666]
```

- The second method can insert only one element at a time, and requires object-oriented features, which will be covered in Chapter 7.

```
>>> L = [111,222,333,444,555,666]
>>> L.insert(2, 999)
>>> print(L)
[111, 222, 999, 333, 444, 555, 666]
```

where the insert function takes two parameters: The first parameter is the index where the item will be inserted and the second parameter is the item to be inserted.

- The third methods uses the append() method to insert an element only to the end or extend() to append more than one element to the end:

```
>>> L = [111,222,333,444,555]
>>> L.append(666)
>>> print(L)
[111, 222, 333, 444, 555, 666]
>>> L.extend([777, 888])
[111, 222, 333, 444, 555, 666, 777, 888]
```

3- *Data creation with ``tuple()`` and ``str()`` functions*: Similar to other data types, tuple and list data types provide two functions for creating data from other data types.

4- *Concatenation and repetition with lists and tuples*: Similar to strings, + and * can be used respectively to concatenate two tuples/lists and to repeat a tuple/list many times.

5- *Membership*: Similar to strings, in and not in operations can be used to check whether a tuple/list contains an element.

Here is an example that illustrates the last three items:

```
>>> a = ([3] + [5])*4
>>> a.append([3, 5])
>>> print(a)
[3, 5, 3, 5, 3, 5, 3, 5, [3, 5]]
>>> a.extend([3, 5])
```

(continues on next page)

(continued from previous page)

```
>>> print(a)
[3, 5, 3, 5, 3, 5, 3, 5, [3, 5], 3, 5]
>>> b = tuple(a)
>>> print(b)
(3, 5, 3, 5, 3, 5, 3, 5, [3, 5], 3, 5)
>>> [3, 5] not in b
False
>>> [5, 3] not in b
True          #  test for single element, not subsequence
```

As you have recognized, the examples with some containers included two types of constructs which were not covered yet: One is the use of 'functions' on containers, e.g. the use of len(). Due to your high school background, this is certainly easy to understand. The second construct is something new. It appears that we can use some functions suffixed by a dot to a container (e.g. the append and insert usages in the examples above). This construct is an internal function call of data structure called *object*. Containers are actually objects and in addition to their data containment property, they also have some defined action associations. These actions are named as *member functions*, and called (applied) on that object (in our case -the container-) by means of this dot notation:

$$\bullet.f(\square) \quad \text{has the conceptual meaning of} \quad f(\bullet, \square) \tag{2.2.2}$$

(Don't try this explicitly, it will not work. The equivalence is just 'conceptual': meaning the function receives the object internally, as a 'hidden' argument). All these will be covered in details in Chapter 7. Till then, for sake of completeness, from time to time we will referring to this notation. Till then simply interpret it based on the equivalence depicted above.

Example: Matrices as Nested Lists

In many real-world problems, we often end up with a set of values that share certain semantics or functionality, and benefit from representing them in a regular grid structure that we call matrix:

$$A = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \ddots & & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{pmatrix}, \tag{2.2.3}$$

which has n columns and m rows, and we generally shorten this as $m \times n$ and say that matrix A has size $m \times n$. The following set of equations describe relations among a set of variables and called system of equations:

$$\begin{aligned} 3x + 4y + z &= 4, \\ -3x + 3y + 5 &= 3, \\ x + y + z &= 0, \end{aligned} \tag{2.2.4}$$

which can be represented with matrices as:

$$\begin{pmatrix} 3 & 4 & 1 \\ -3 & 3 & 5 \\ 1 & 1 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} 4 \\ 3 \\ 0 \end{pmatrix}, \tag{2.2.5}$$

which defines the same problems in terms of matrices and matrix multiplication. It is okay if you are not familiar with matrix multiplication – we will briefly explain it and use it as an example in the next chapter.

Writing a problem in a matrix form like we did above allows us to group the related aspects of the problem together and focus on these groups for solving a problem. In our example, we grouped coefficients in a

matrix and this allows us to analyze and manipulate the coefficients matrix to check whether for this system of equations there is a solution, whether the solution is unique or what the solution is. All these are questions that are studied in Linear Algebra and beyond the scope of our book.

Now let us see how we can represent matrices in Python. A very straightforward approach that would also allow changing elements of the matrix later on is to use lists in a nested form, as follows:

```
>>> A = [[3, 4, 1],
... [-3, 3, 5],
... [1, 1, 1]]
>>> A
[[3, 4, 1], [-3, 3, 5], [1, 1, 1]]
```

where each row is represented as a list and a member of the outer list. This would allow us to access entries like this:

```
>>> A[1]
[-3, 3, 5]
>>> A[1][0]
-3
```

Although we can represent matrices like this, there are very advanced libraries that make representing and working with matrices more practical, as we will see in Chapter 10.

2.2.5 Dictionary

Dictionary is a container data type where accessing items can be performed with indexes that are not numerical; in fact, in dictionaries, indexes are called keys. A list, tuple, or string data type stores a certain element at each numerical index (key). Similarly, a dictionary stores an element (value) for each key. In other words, a dictionary is just a mapping from keys to values (Fig. 2.2.3).

The keys can be only immutable data types, i.e., numbers, strings, or tuples (with only immutable elements) – some other immutable types that we do not cover in the book are also possible. Since lists and dictionaries are mutable, they cannot be used for keys for indexing. As for values, there is no limitation on the data type.

A dictionary is a discrete mapping from a set of Python elements to another set of Python elements. Itself, as well as its individual elements, are mutable (you can replace them). Moreover, it is possible to add and remove items from the mapping.

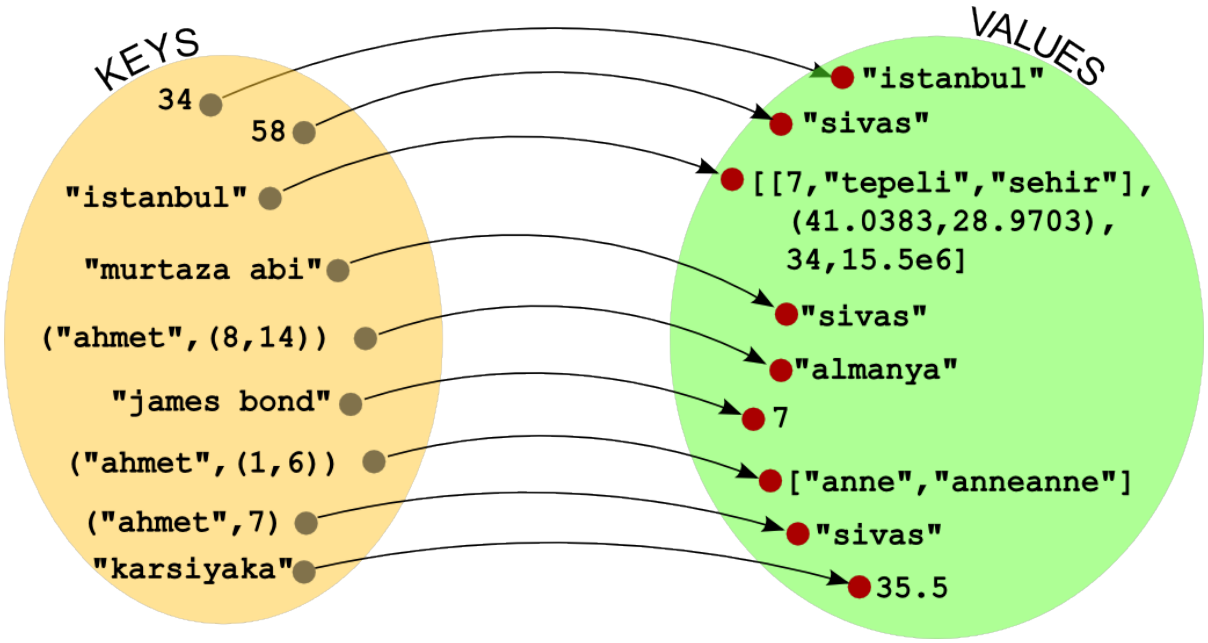


Fig. 2.2.3: A dictionary provides a mapping from keys to values.

A dictionary is represented as key-value pairs, each separated by a column sign (:) and all enclosed in a pair of curly braces. The dictionary in Fig. 2.2.3 would be denoted as:

```
{34:"istanbul",
 58:"sivas",
 "istanbul":[[7,"tepeli","sehir"],(41.0383,28.9703),34,15.5e6],
 "murtaza abi":"sivas",
 ("ahmet",(8,14)):"almanya","james bond":7,
 ("ahmet",(1,6)):["anne","anneanne"],
 ("ahmet",7):"sivas",
 "karsiyaka":35.5}
```

Similar to other containers, we usually store containers under a variable. Let us assume the dictionary above was assigned to a variable with the name conno and look at some examples:

```
conno = {34:"istanbul", 58:"sivas", "istanbul":[[7,"tepeli","sehir"],(41.0383,28.9703),34,15.5e6], "murtaza abi":
↪"sivas", ("ahmet",(8,14)):"almanya","james bond":7,  ("ahmet",(1,6)):["anne","anneanne"], ("ahmet",7):"sivas",
↪"karsiyaka":35.5}
print(conno["murtaza abi"])
print(conno["istanbul"])
```

```
sivas
[[7, 'tepeli', 'sehir'], (41.0383, 28.9703), 34, 15500000.0]
```

Let us ask for something that does not exist in the dictionary:

```
>>> print(conno["ankara"])
KeyError: 'ankara'
```

Ups, that was bad. Though we have a decent method to test for the existence of a key in a dictionary, namely the use of in:

```
print("ankara" in conno)
```

```
False
```

It is also possible to remove/insert mappings from/to a dictionary:

```
print("conno has this many keys:", len(conno))
conno["ankara"] = ["baskent", "anitkabir", 6]
print("conno has this many keys now:", len(conno))
print(conno["ankara"])
print(conno["murtaza abi"])
del conno["murtaza abi"]
print("murtaza abi" in conno)
print("After murtaza abi is deleted we have this many keys:", len(conno))
```

```
conno has this many keys: 9
conno has this many keys now: 10
['baskent', 'anitkabir', 6]
sivas
False
After murtaza abi is deleted we have this many keys: 9
```

The benefit of using a dictionary is 'timewise'. The functionality of a dictionary could be attained by using (key, value) tuples inserted into a list. Then, when you need the value of a certain key, you can search one-by-one each (key, value)-tuple element of the list until you find your key in the first position of a tuple element. However, this will consume a time proportional to the length of the list (worst case). On the contrary, in a dictionary, this time is almost constant.

Moreover, a dictionary is more practical to use since it already provides accessing elements in a key-based fashion.

Useful Operations with Dictionaries

Dictionaries support len() and membership (in and not in) operations that we have seen above. You can also use <dictionary>.values() and <dictionary>.keys() to obtain lists of values and keys respectively.

2.2.6 Set

Sets are created by enclosing elements into a pair of curly-braces and separating them with commas. Any immutable data type, namely a number, a string or a tuple, can be an element of a set. Mutable data types (lists, dictionaries) cannot be elements of a set. Being mutable, sets **cannot** be elements of a sets.

Here is a small example with sets:

```
a = {1,2,3,4}
b = {4,3,4,1,2,1,1,1}
print (a == b)
a.add(9)
a.remove(1)
print(a)
```

```
True
{2, 3, 4, 9}
```

The most functionalities of sets can be undertaken by lists. Furthermore, lists do not possess the restrictions sets do. On the other hand, especially membership tests are much faster with sets, since member repetition is avoided.

Frozenset

Python provides an immutable version of the set type, called frozenset. A frozenset can be constructed using the frozenset() function as follows:

```
>>> s = frozenset({1, 2, 3})
>>> print(s)
frozenset({1, 2, 3})
```

Being immutable, frozensets can be a member of a set or a frozenset.

Useful Operations with Sets

Apart from the common container operations (len(), in and not in), sets and frozensets support the following operators:

- S1 <= S2: True if S1 is a subset of S2.
- S1 >= S2: True if S1 is a superset of S2.
- S1 | S2: Union of the sets (equivalent to S1.union(S2)).
- S1 & S2: Intersection of the sets (equivalent to S1.intersection(S2)).
- S1 - S2: Set difference (equivalent to S1.difference(S2)).

The following are only applicable with sets (and not with forezensets) as they require a mutable container:

- S.add(element): Add a new element to the set.
- S.remove(element): Remove element from the set.
- S.pop(): Remove an arbitrary element from the set.

2.3 Expressions

Expressions such as 3 + 4 describe calculation of an operation among data. When an expression is *evaluated*, the operations in the expression are applied on the data specified in the expression and a resulting value is provided.

Operations can be graphically illustrated as follows:

$$\square_1 \odot \square_2 \tag{2.3.1}$$

where $\odot$ is called the operator, and $\square_1$ and $\square_2$ are called operands. This was a binary operator; i.e. it acted on two operands.

We can also have unary operators:

$$\odot \square \tag{2.3.2}$$

or operators that have more than two operands (see chained comparison operators below).

Before we can cover how such operations are evaluated, let us look at commonly used operations (operators) in Python.

2.3.1 Arithmetic, Logic, Container and Comparison Operations

Python provides the operators in Table 2.3.1 for arithmetic (addition, subtraction, multiplication, division, exponentiation), logic (and, or, not), container (indexing, membership) and comparison (less, less-than, equality, not-equality, greater, greater-than) operations.

Note that operators such as + and * have different meanings on different data types. For numerical data, they mean addition and multiplication whereas for container data, they mean concatenation and repetition.

Table 2.3.1: Arithmetic, Logic, Container and Comparison operators in Python.

Operator	Operation	Result Type
[]	Indexing	Any data type
**	Exponentiation	Numeric
*	Multiplication or Repetition	Numeric or container
/	Division	Numeric (floating point)
//	Integer Division	Numeric (integer)
+	Addition or concatenation	Numeric or container
-	Subtraction	Numeric
<	Less than	Boolean
<=	Less than or equal to	Boolean
>	Greater than	Boolean
>=	Greater than or equal to	Boolean
==	is equal to	Boolean
!=	is not equal to	Boolean
in	is a member	Boolean
not in	is not a member	Boolean
not	logical negation	Boolean
and	logical and	Boolean
or	logical or	Boolean

Below are some illustrations:

```
S1 = "Four"
S2 = "Five"
B1 = len(S1) < len(S2)
print("B1 is: ", B1)
B2 = S1 != S2
print("B2 is: ", B2)
B3 = B1 or B2
print("B3 is: ", B3)
```

```
B1 is:  False
B2 is:  True
B3 is:  True
```

Exercise

Give one example for each row in Table 2.3.1. Please complete this exercise in the Colab version of the chapter.

2.3.2 Evaluating Expressions

In the previous section, we have seen simple use of operators in an expression. In many cases, we combine several operators for brevity and readability, e.g. 2.3 + 3.4 * 4.5. This expression can be evaluated in two different ways:

- (2.3 + 3.4) * 4.5, which would yield 25.65.
- 2.3 + (3.4 * 4.5), which would yield 17.599999999999998 in Python.

Since the results are very different, it is very important for a programmer to know in which order operators are evaluated when they are combined. There are two rules that govern this:

1. Precedence: Each operator has an associated precedence (priority) based on which we can determine which operator is going to be evaluated first. E.g. multiplication has higher precedence than addition, and therefore, 2.3 + 3.4 * 4.5 would be evaluated as 2.3 + (3.4 * 4.5) in Python.
2. Associativity: If two operators have the same precedence, evaluation order is determined based on associativity. Associativity can be from left to right or from right to left.

For the operators, the complete associativity and precedence information are listed in Table 2.3.2.

Table 2.3.2: Precedence and associativity for the operators in Table 2.3.1.

Operator	Precedence	Associativity
[]	1.	Left-to-right
**	2.	Right-to-left
*, /, //, %	3.	Left-to-right
+, -	4.	Left-to-right
<, <=, >, >=, ==, !=, in, not in	5.	Special
not	6.	Unary
and	7.	Left-to-right (with short-cut)
or	8.	Left-to-right (with short-cut)

Therefore, according to Table 2.3.2, a sophisticated expression such as 2**3**4*5-2//2-1 is equivalent to 2**81*5-1-1 which is equivalent to 120892581961462917470617 60-2 which is 120892581961462917470617 58.

Below are some notes and explanations regarding expression evaluation:

(i) The *Special* keyword in Table 2.3.2 means some treatment which is common to mathematics but not programming. In that sense Python is unique among commonly used programming languages. If $\odot_i$ is any boolean comparison operator and $\square_j$ is any numerical expression, the sequence of

$$\square_1 \ \odot_1 \ \square_2 \ \odot_2 \ \square_3 \ \odot_3 \ \square_4 \ \cdots \ \square_{n-1} \ \odot_{n-1} \ \square_n \tag{2.3.3}$$

is interpreted as:

$$\square_1 \ \odot_1 \ \square_2 \quad \text{and} \quad \square_2 \ \odot_2 \ \square_3 \quad \text{and} \quad \square_3 \ \odot_3 \ \square_4 \quad \cdots \quad \square_{n-1} \odot_{n-1} \ \square_n \tag{2.3.4}$$

(ii) It is **always** possible to override the precedence by making use of parenthesis. We are familiar with this since our primary school days.

(iii) If two numeric operands are of the same type, then the result is of that type unless the operator is / (for which the result is always a floating point number). Also comparison operators return bool typed values.

(iv) If two numeric operands that enter an operation are of different types, then a computation occurs according to the following rules:

- **if one operand is integer and the other is floating point:** The integer is converted to floating point.
- **if one operand is complex:** The complex arithmetic is carried out according to the rules of mathematics among the real/imaginary part coefficients (which are either integers or floating points). Each of the two resulting coefficients are separately checked for having zero (.0) fractional part. If so, that one is converted to integer.

(v) Except for the two logical operators and and or, all operators have an evaluation scheme which is coined as *eager evaluation*. Eager evaluation is the strategy where all operands are evaluated first and then the semantics of the operators kicks in, providing the result. Here is an example: Consider mathematical expression of: 0*(2**150-3**95) As a human being, our immediate reaction would be: Anything multiplied with 0 (zero) is 0, therefore we do not have to compute the two huge exponentiations (the second operand). This is taking a *short-cut* in evaluation and is certainly **far from** eager evaluation. Eager evaluation would evaluate the internals of the parenthesis, obtain -693647454339354238433323618063349607247325483 and then multiply this with 0 to obtain 0. Yes, Python would go this 'less intelligent' way and do eager evaluation. The logical operators and and or, though, **do not** adopt eager evaluation. On the contrary, they use *short-cuts* (this is known as *based-on-need evaluation* in computer science).

In a conjunctive expression like:

$$\square_1 \quad \text{and} \quad \square_2 \quad \text{and} \quad \square_3 \quad \text{and} \quad \cdots \quad \text{and} \quad \square_n \tag{2.3.5}$$

The evaluation proceeds as follows:

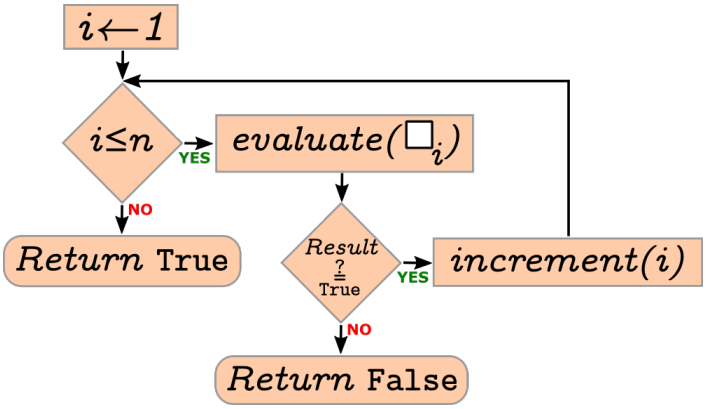


Fig. 2.3.1: Logical AND evaluation scheme

Similarly, a disjunctive expression:

$$\square_1 \quad \text{or} \quad \square_2 \quad \text{or} \quad \square_3 \quad \text{or} \quad \cdots \quad \text{or} \quad \square_n \tag{2.3.6}$$

has the following evaluation scheme:

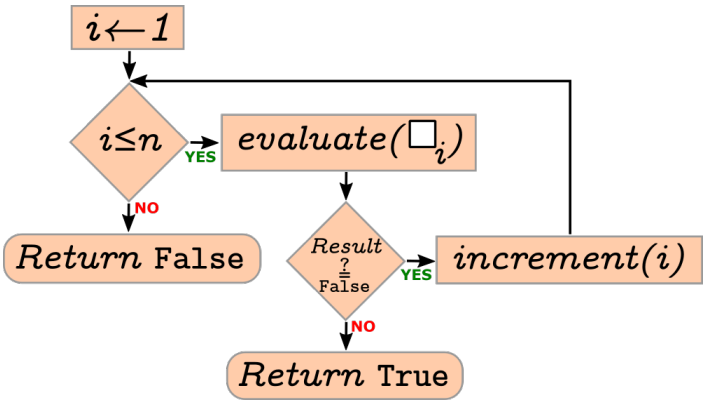


Fig. 2.3.2: Logical OR evaluation scheme

Although high-level languages provide mechanisms for evaluating expressions involving multiple operators, it is not a good programming practice to leave multiple operators without parentheses. A programmer should do his/her best to write code that is readable and understandable by other programmers and that does not include any ambiguity whatsoever. This includes expressions.

2.3.3 Implicit and Explicit Type Conversion

In Python, when you apply a binary operator on items of two different data types, it tries to convert one data to another one if possible. This is called *implicit type conversion*. For example,

```
>>> 3+4.5
7.5
>>> 3 + True
4
>>> 3 + False
3
```

which illustrates that an integer is converted to a float, True is converted to integer 1, and False is converted to integer zero.

Although Python can do such conversions, it is a good programming practice to make these conversions explicit and make the intention clear to the reader. Explicit type conversion, also called as *type casting*, can be performed using the keyword for the target type as a function. For example:

```
>>>> 1.1*(7.1+int(2.5*5))
21.01
```

Not all conversions are possible, of course. Implicit conversions are allowed only for the basic data types as illustrated below:

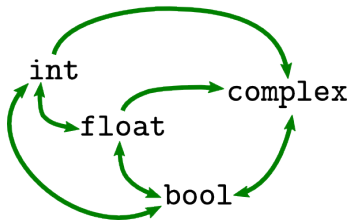


Fig. 2.3.3: Type casting among numeric and Boolean

Type casting can accommodate conversion between a wider spectrum of data types, including containers:

```
>>> str(34)
'34'
>>> list('34')
['3', '4']
```

2.4 Basic Statements

Now, let us continue with actions that do not provide (return) us data as a result.

2.4.1 Assignment Statement and Variables

When we perform computation in Python, we either print the result and/or keep the result for further computations that we are going to perform. *Variables* help us here as named memory positions in which we can store data. You can imagine them as pigeonholes that are able to hold a **single** data item.

There are two methods to create and store some data into a variable. Here we will mention the overwhelmingly used one, the second is more implicit and will be introduced when we deal with functions.

A variable receives data for storage by the use of the *assignment statement*. It has the following form:

$$\boxed{Variable} = \boxed{Expression}$$

Although the equal sign resembles an operator that is placed between two operands, it is not an operator: Operators return a value, and in Python, the assignment is not an operator and it does not return a value (this can be different in other programming languages).

The action semantics of the assignment statement is simple:

1. The *Expression* is evaluated.
2. If the *Variable* does not exist, it is created.
3. The evaluation result is stored into the *Variable* (by doing so, any former value, if existing, is purged).

After assignment, the value can be repetitively used. To use the value in a computation, we can use the name of the variable. Here is an example:

```
>>> a = 3
>>> b = a + 1
>>> (a-b+b**2)/2
7.5
```

It is quite common to use a variable on both sides of the assignment statement. For example:

```
>>> a = 3
>>> b = a + 1
>>> a = a + 1
>>> (a-b+b**2)/2
8.0
```

The expression on the second line uses the 3 value for a. In the next line, namely the third line, again 3 is used for a in the expression (a+1). Then, the result of the evaluation (3+1) which is a 4 is stored in variable a. The variable a had a previous value of 3; that value is purged and replaced by 4. The former value is not kept anywhere and cannot be recovered.

Multiple assignments

It is possible to have multiple variables assigned to the same value, e.g.:

```
>>> a = b = 4
```

assigns an integer value of 4 to both a and b. After the multi-assignment above, if you change b to some other value, the value of a will still remain to be 4.

Multiple assignment with different values Python provides a powerful mechanism for providing different values to different variables in assignment:

```
>>> a, b = 3, 4
>>> a
3
>>> b
4
```

This is internally handled by Python with tuples and equivalent to:

```
>>> (a,b) = (3, 4)
>>> a
3
>>> b
4
```

This is called *tuple matching* and would also work with lists (i.e. [a, b] = [3, 4]).

Swapping values of variables Tuple matching has a very practical benefit: Let us say you want to swap the values in variables. Normally, this requires the use of a temporary variable:

```
>>> print(a,b)
3 4
>>> temp = a
>>> a = b
>>> b = temp
>>> print(a,b)
4 3
```

With tuple matching, we can do this in one line:

```
>>> print(a,b)
3 4
>>> a,b = b,a
>>> print(a,b)
4 3
```

Frequently-asked questions about assignments

- **QUESTION:** Considering the example below, one may have doubts about the value in b: Is it updated? On the fourth line, there is an expression using b: Which value is it refering to? 4 or 5? When we use b in a following expression, will the 'definition' be retrived and recalculated?

```
>>> a = 3
>>> b = a + 1
>>> a = a + 1
>>> (a-b+b**2)/2
8.0
```

- **ANSWER:** *No. Statements are executed only once: the moment they are entered (it is possible to repetitively execution a statement but here it is not used). Each assignment in the example above is executed only once. No reevaluation is performed, the use of a variable in an expression, the ``a`` and those ``b``s, refer solely to the last calculated and stored values: In the evaluation of ``(a-b+b**2)/2``, ``a`` is 4 and ``b`` is 4.*
- **QUESTION:** We had variables in math, especially in middle school and high school. So, this is very similar to that right? But I am confused having seen a line like a = a + 1. What's happening? The as cancel out and we are left with 0 = 1?
- **ANSWER:** *The use of the equality sign (=) is confusing to some extent. It does not stand for a balance among the left-hand-side and the right-hand-side. Do not interpret it as an 'equality of mathematics'. It has a absolutely different semantics. As said, it only means:*
  1. **First** *calculate the right-hand side*,
  2. **then** *store the result into an (electronic) pigeon hole which has the name label given to the left-hand-side of the equal sign.*
- **QUESTION:** I typed in the following lines:

```
>>> x = 5
>>> y = 3
>>> x = y
>>> y = x
>>> print x,y
3 3
```

However, it should have been 3 5, right? Or am I doing something wrong?

- **ANSWER:** *You are somehow missing the time flow. Statements are executed in order:*
  1. **First:** *``x`` is set to 5.*
  2. **Second:** *``y`` is set to 3.*
  3. **Third:** *``x`` is set to the value stored in ``y`` which is 3. ``x`` now holds 3.*
  4. **Fourth:** *``y`` is set to the value stored in ``x`` which is 3 (just look to the 3. item which right above this one). ``y`` now holds 3.*
  5. **Fifth:** *print both the values ``x`` and ``y``. Both are ``3`` and they got printed.*
- **QUESTION:** So, how do I perform a swap of content of two variables x and y for example?
- **ANSWER:** *We leave this as an exercise! (Hint: Assume you have two soup bowls. Each containing a different soup. For some peculiar reason you have to switch the contents of the bowls. How would you proceed?)*

2.4.2 Variables & Aliasing

There is something peculiar about lists that we need to be careful while assigning them to variables. First consider the following code:

"The first example"

```
print("address of 5: ", id(5))
a = 5
print("address of a: ", id(a))
b = a
print("address of b: ", id(b))
a = 3
print("address of a: ", id(a))
print("b is: ", b)
```

```
address of 5:  4415314608
address of a:  4415314608
address of b:  4415314608
address of a:  4415314544
b is:  5
```

Here, we used the id() function to display the address of variables in the memory to help us understand what happens in those assignments: * The second line creates 5 in memory and links that with a. * The fourth line links the content of a with variable b. So, they are two different names for the same content. * The sixth line creates a new content, 3, and assigns it to a. Now, a points to a different memory location than b. * b still points to 5, which is printed.

Now keep the task the same but change the data from integer to a list:

"The second example"

```
a = [5,1,7]
b = a
print("addresses of a and b: ", id(a), id(b))
```

(continues on next page)

(continued from previous page)

```
print("b is: ", b)
a = [3,-1]
print("addresses of a and b: ", id(a), id(b))
print("b is: ", b)
```

```
addresses of a and b:  4449234304 4449234304
b is:  [5, 1, 7]
addresses of a and b:  4448539200 4449234304
b is:  [5, 1, 7]
```

In other words, this works similar to the first example, as expected. But now, consider the following slightly different example:

"The third example"

```
a = [5,1,7]
b = a
print("addresses of a and b: ", id(a), id(b))
print("b is: ", b)
a[0] = 3
print("addresses of a and b: ", id(a), id(b))
print("b is: ", b)
```

```
addresses of a and b:  4449200960 4449200960
b is:  [5, 1, 7]
addresses of a and b:  4449200960 4449200960
b is:  [3, 1, 7]
```

To our surprise, the first element's replacement of a got also reflected in b. This should not be surprising since both a and b point to the same memory location and since a list is mutable, when we change an element in that memory location, it affects both a and b that are just names for that memory location.

Although the examples above used lists for illustration purposes, aliasing pertains to all mutable data types.

Aliasing is a powerful concept that can be hazardous and beneficial depending on the content:

- If you carelessly assign mutable variable to another variable, changes on one variable is going to reflect the other one. If this was not intended and the location in the code where the aliasing was initiated could not be identified, you may lose hours or days trying to identify what the problem is with your code.
- Aliasing can be beneficial especially when we want our changes on one variable to be reflected on another. This will be useful for passing data to functions and getting the results from functions.

2.4.3 Naming variables

Programmers usually select names for variables so that it indicates what the content will be. Variable names may be arbitrarily long. They may contain letters (from the English alphabet) as well as numbers and underscores, but they must start with a letter or an underscore. While using upper case letters is allowed, bear in mind that programmers reserve starting with an upper case to differentiate a property (scope) of the variable which you will learn later (when we introduce functions).

Here are a few examples for variable names (all are different):

y	y1	Y	Y_1	_1
te mperature	temperat ure_today	Tempera tureToday	C umulative	coo rdinate_x
a1b145c	a 1_b1_45_c	s_s_	_	___

As you might have recognized, though they are perfectly valid, the five examples in the last line do not make much sense. Here is a short list that you should prefer to follow in variable naming:

- Name variables in the context of the value they are going to store. For example

```
>>> a = b * c
```

is syntactically correct, but its purpose is not evident. Contrast this with:

```
>>> salary = hours_worked * hourly_pay_rate
```

- Use different variables for different data, a.k.a. the 'Single Responsibility Principle'. For example, even if you could use the same variable in a statement for counting and in another statement, for holding the largest grade, this is not recommended: Do not economise with variables. Devise two different variables where each of which will reflect the semantics and the context, uniquely.
- Variable names should be pronounceable, which makes them easier to remember.
- Do not use variable names which could misguide you or whoever look into your code.
- Use i,j,k,m,n for counting only: Programmers implicitly recognize them as integer holding (the historical reason for this dates to 60 years ago).
- Use x,y,z for coordinate values or multi-variate function arguments.
- Do not use single character variable l as it can be easily confused with 1 (one).
- If you are going to use multiple words for a variable, choose one of these:
  - Put all words into lowercase, affix them using _ as separator (e.g. highest_midterm_grade, shortest_path_distance_up_to_now).
  - Put all words but the first into first-capitalized-case, put the first one into lowercase, affix them without using any separator (e.g.highestMidtermGrade, shortestPathDistanceUpToNow).

Reserved names

The following keywords are being used by Python already and therefore, you cannot use them to name your variables.

and	def	exec	if	not	return
assert	del	finally	import	or	try
break	elif	for	in	pass	while
class	else	from	is	print	yield
continue	except	global	lambda	raise	

2.4.4 Other Basic Statements

Python has other basic statements listed below:

pass, del, return, yield, raise, break, continue, import, future, global, nonlocal.

Among these, we have seen del and we will see some others in the rest of the book.

print used to be a statement in version 2, which however changed in version 3: print is a function in version 3 and therefore, it has a value (which we will see later).

2.5 Compound Statements

Like other high-level languages, Python provides statements combining many statements as their parts. Two examples are:

- Conditional statements where different statements are executed based on the truth value of a condition, e.g.:

```
if <boolean-expression>:
   statement-true-1
   statement-true-2
   ...
else:
  statement-false-1
  statement-false-2
  ...
```

- Repetitive statements where some statements are executed more than once depending on a condition or for a fixed number of times, e.g.:

```
while <boolean-condition>:
   statement-true-1
   statement-true-2
   ...
```

which executes the statements while the condition is true.

We will see these and other forms of compound statements in the rest of the book.

2.6 Basic actions for interacting with the environment

In our programs, we frequently require obtaining some input from the user or displaying some data to the user. We can use the following for these purposes.

2.6.1 Actions for input

In Python, you can use the input() function for obtaining input from the user, e.g.:

```
>>> s = input("Now enter your text: ")
Now enter your text: This is the text I entered
>>> print(s)
This is the text I entered
```

If you expect the user to enter an expression, you can evaluate it using the eval() function as we explained in Section 4.2.3.

2.6.2 Actions for output

For displaying data to the screen, Python provides the print() function:

```
print(item1, item2, ..., itemN)
```

For example:

```
>>> print('Python', 'is', 'so', 'fun')
Python is so fun
```

In many cases, we end up with strings that have placeholders for data items which we can fill in using a formatting function as follows:

```
>>> print("I am {0} tall, {1} years old and have {2} eyes".format(1.86, 20, "brown"))
I am 1.86 tall, 20 years old and have brown eyes
```

Alternatively, instead of using integers for the placeholders, we can give names to the placeholders:

```
>>> print("I am {height} tall, {age} years old and have {eyes} eyes. Did I tell you that I was {age}?".format(age=20,
↪ eyes="brown", height=1.86))
I am 1.86 tall, 20 years old and have brown eyes. Did I tell you that I was 20?
```

The format() function provides much more functionalities than the ones we have illustrated. However, this extent should be sufficient for general uses and this book. The reader interested in a complete coverage is referred to the Python's documentation on string formatting[19].

[19] https://docs.python.org/3/library/string.html

2.7 Actions that are ignored

In Python, we have two actions that are ignored by the interpreter:

2.7.1 Comments

Like other high-level languages, Python provides programmers mechanisms for writing comments on their programs:

- Comments with #: When Python encounters # in your code, it ignores the rest of the line, assumes the current line is finished, evaluates & runs the current line and continues interpretation with the next line. For example:

```
>>> 3 + 4 # We are adding two numbers here
7
```

- Multi-line comments with triple-quotes: If you wish to provide comments longer than one line, you can use triple quotes:

```
"""
This is a multi-line comment.
We are flexible with the number of lines &
 characters,
   spacing. Python
     will ignore them.
"""
```

Triple-quote comments are generally used by programmers to write documentation-level explanations and descriptions for their codes. There are document-generation tools that process triple-quotes for automatically generating documents for codes.

As we have seen before, the triple-quote comments are actually strings in Python. Therefore, if you use triple-quotes for providing a comment, it should not overlap with an expression or statement line in your code.

2.7.2 Pass statements

Python provides the pass statement that is ignored by the interpreter. The pass statement is generally used in incomplete function implementations or compound statements to place a dummy holder (some instruction) such that the interpreter does not complain about a statement being missing. For example:

```
if <condition>:
    pass # @TODO fill this part
else:
    statement-1
    statement-2
    ...
```

2.8 Actions and data packaged in libraries

Many high-level programming languages like Python provide a wide spectrum of actions and data predefined and organized in 'packages' that we call libraries. For example, there is a library for mathematical functions and constant definitions which you can access using from math import * as follows:

```
>>> pi
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'pi' is not defined
>>> sin(pi)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'sin' is not defined
>>> from math import *
>>> pi
3.141592653589793
>>> sin(pi)
1.2246467991473532e-16
```

Below is a short list of libraries that might be useful:

Library	Description
math	Mathematical functions and definitions
cmath	Mathematical functions and definitions for complex numbers
fractions	Rational numbers and arithmetic
random	Random number generation
statistics	Statistical functions
os	Operating system functionalities
time	Time access and conversion functionalities

Of course, the list is too wide and it is not practical to list and explain all libraries here. The interested reader is direct to to check the comprehensive list at Python docs[20].

The import statement that we used above loads the library and makes its contents directly accessible to us by directly using their names (e.g. sin(pi)). Alternatively, we can do the following:

```
>>> import math
>>> math.sin(math.pi)
1.2246467991473532e-16
```

which requires us to specify the name math everytime we need to access something from it. We could also change the name:

```
>>> import math as m
>>> m.sin(m.pi)
1.2246467991473532e-16
```

However, we discourage this way of using libraries until Chapter 7 where we introduce the concept of objects and object-oriented programming.

However, to be able to learn what is available in a library, you can use this form (with dir() function):

[20] https://docs.python.org/3/library/

```
>>> import math
>>> dir(math)
['__doc__', '__file__', '__loader__', '__name__', '__package__', '__spec__', 'acos', 'acosh', 'asin', 'asinh', 'atan', 'atan2',
↪ 'atanh', 'ceil', 'comb', 'copysign', 'cos', 'cosh', 'degrees', 'dist', 'e', 'erf', 'erfc', 'exp', 'expm1', 'fabs', 'factorial', 'floor',
↪'fmod', 'frexp', 'fsum', 'gamma', 'gcd', 'hypot', 'inf', 'isclose', 'isfinite', 'isinf', 'isnan', 'isqrt', 'ldexp', 'lgamma', 'log',
↪'log10', 'log1p', 'log2', 'modf', 'nan', 'perm', 'pi', 'pow', 'prod', 'radians', 'remainder', 'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'tau
↪', 'trunc']
```

2.9 Providing your actions to the interpreter

Python provides different mechanisms for you to provide your actions and get them executed.

2.9.1 Directly interacting with the interpreter

As we have seen up to now, we can interact with the interpreter directly by typing our actions to the interpreter. When you are done with the interpreter, you can quit using quit() or exit() functions or by pressing CTRL-D. E.g.:

```
$ python3
Python 3.8.5 (default, Jul 21 2020, 10:48:26)
[Clang 11.0.3 (clang-1103.0.32.62)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> print("Python is fun")
Python is fun
>>> print("Now I am done")
Now I am done
>>> quit()
$
```

where we see that after quitting the interpreter, we are back at the terminal shell.

Although this way of coding is simple and very interactive, when your code gets longer and more complicated, it becomes difficult to manage your code. Moreover, when you exit the interpreter, all your actions are lost and to be able to run them again, you need to type them again, from scratch. This can be tedious, redundant, impractical and inefficient.

2.9.2 Writing actions in a file (script)

Alternatively, we can place our actions in a file with an .py extension in the filename as follows:

```
print("This is a Python program that reads two numbers from the user, adds the numbers and prints the result\n\n")
[a, b] = input("Enter two numbers: ")
print("You have provided: ", a, b)
result = a + b
print("The sum is: ", result)
```

Assuming that you save these lines into a file named test.py, you can execute those statements from a terminal window as follows:

```
$ cat test.py
print("This is a Python program that reads two numbers from the user, adds the numbers and prints the result\n\n")
s = input("Enter a list of two numbers [a, b]: ")
[a, b] = eval(s)
print("You have provided: ", a, b)
result = a + b
print("The sum is: ", result)
$ python3 test.py
This is a Python program that reads two numbers from the user, adds the numbers and prints the result


Enter a list of two numbers [a, b]: [3, 4]
You have provided:  3 4
The sum is:  7
```

It is possible to provide command-line arguments to your script and use the provided values in your script (named test.py):

```
from sys import argv

print("The arguments of this script are:\n", argv)

exec(argv[1]) # Get a
exec(argv[2]) # Get b

print("The sum of a and b is: ", a+b)
```

which can be run as follows:

```
$ python3 test.py a=10 b=20
The arguments of this script are:
 ['test.py', 'a=10', 'b=20']
The sum of a and b is:  30
```

Note that this example used the function exec(), which executes the statement provided to the function as a string argument. Compare this with the eval() function that we have introduced before: eval() takes an expression whereas exec() takes a statement.

2.9.3 Writing your actions as libraries (modules)

Another mechanism for executing your actions is to place them into libraries (modules) and provide them to Python using import statement, as we have illustrated in Section 4.7. For example, if you have a test.py file with the following content:

```
a = 10
b = 8
sum = a + b
print("a + b with a =", a, " and b =", b, " is: ", sum)
```

In another Python script or in the interpreter, you can directly type:

```
>>> from test import *
a + b with a = 10  and b = 8  is:  18
>>> a
10
>>> b
8
```

In other words, what you have defined in test.py becomes accessible after being imported.

2.10 Important Concepts

We would like our readers to have grasped the following crucial concepts and keywords from this chapter (all related to Python):

- Basic data types.
- Basic operations and expression evaluation.
- Precedence and associativity.
- Variables and how to name them.
- Aliasing problem.
- Container types.
- Accessing elements of a container type (indexing, negative indexing, slicing).
- Basic I/O.
- Commenting your codes.
- Using libraries.

2.11 Further Reading

- A detailed history of Python and its versions: https://en.wikipedia.org/wiki/History_of_Python
- The concept of aliasing: http://en.wikipedia.org/wiki/Aliasing_%28computing%29

Exercises

- Without using Python, determine the results of the following expressions and validate your answers with Python:
- 2 - 3 ** 4 / 8 + 2 * 4 ** 5 * 1 ** 8
- 4 + 2 - 10 / 2 * 4 ** 2
- 3 / 3 ** 3 * 3
- Assuming that a is True, b is True and c is False, what would be the values of the following expressions?

(a) not a == b + d < not a

(b) a == b <= c == True

(c) True <= False == b + c

(d) c / a / b

- The Euclidean distance between two points (a, b) and (c, d) is defined as: $\sqrt{(a-c)^2 + (b-d)^2}$. Write a Python code that reads a, b, c, d from the user, calculates the Euclidean distance and prints the result.

3 | Conditional and Repetitive Execution

Many circumstances require the execution to change its flow based on the truth value of a condition or to repeat a set of steps for a number of items or until a condition is not true any more. These two kinds of actions are very crucial components of programming solutions and Python provides several alternatives for them. As we are going to see in this chapter, some of these alternatives are compound statements, i.e. they include other statements as part of the statement, whereas others are expressions.

3.1 Conditional execution

Asking (mostly) mathematical questions that have Boolean type answers and taking some actions (or not) based on the Boolean answers are crucial tools in designing algorithms. This is so vital that there is no single programming language that does not provide conditional execution.

3.1.1 if statement

The syntax of the conditional (if) statement is

if *Boolean expression* : *Statement*

The semantics is obvious. If the *Boolean expression* evaluates to True, then the *Statement* is carried out; otherwise *Statement* is not executed.

```
>>> if 5 > 2: print("Hurray")
Hurray
>>> if 2 > 5: print("No I will not believe that !")
```

We observe that the Boolean expression in the last example evaluates to False and hence the statement with the print function is not carried out.

How to group statements?

It is quite often that we want to do more then one action instead of executing a single statement. As a general syntax rule of Python:

- switch to a new line,
- indent by one tab,
- write your first statement to be executed,
- switch to the next line,
- indent the same amount (usually your Python-aware editor will do it for you)

- write your second statement to be executed,
- and so on…
- To finish this writing task, and presumably enter a statement which is *not* part of the group action, just do not intend.

It is very very important that the statements that are part of the if part have the same amount and type of whitespace. E.g. if the first statement has 4 spaces, all following statements in the if statement need to start with 4 spaces. If it is a tab character, all following statements in the if statement need to start with a tab character. Since these whitespaces are not visible, it is very easy to make mistakes and get annoying errors from the interpreter. To avoid this, we strongly recommend you to choose a style of indentation (e.g. 4 whitespaces or a single tab) and stick to it as a programming style.

Here is an example:

```
>>> if 5 > 2:
...     print(7*6)
...     print("Hurray")
...     x = 6
42
Hurray
>>> print(x)
6
```

The >>> as well as ... prompts appear when you sit in front of the interpreter. When you type your programs into a file and then run them, of course they will not show up. If you type in these small programs (we call them *snippets*) just ignore the >>> and ... from the examples, as follows:

```
if 5 > 2:
  print(7*6)
  print("Hurray")
  x = 6
print(x)
```

```
42
Hurray
6
```

As you continue practicing programming, you will soon come across a need to be able to define actions that should be carried out in case a Boolean expression turns out to be False. If it were not provided, we would have to ask the same question again, but this time negated:

```
>>> if x == 6: print("still it is six")
>>> if not (x == 6): print("it is no more six")
```

One of them will certainly fire (be true). Although this is a feasible solution, there is a more convenient way: We can combine the two parts into one if-else statement with the following syntax:

if *Boolean expression* : *Statement*

else : *Statement*

Making use of the else part, the last example would be written as:

```
>>> if x == 6: print("still it is six")
... else: print("it is no more six")
```

Of course, using multiple statements in the if part or the else part is possible by intending them to the same level.

Exercise

In the following code cell, you are expected to insert a couple of lines so that the output is always the absolute value of N (try the program for various values of N by altering the right-hand side of the assignment):

```
N = -100   # Feel free to change this to any numerical value #

# @TODO type in your code that changes the content of N to its absolute value, below this line

# @TODO print the result
```

3.1.2 Nested if statements

It is quite possible to nest if-else statements: i.e. combine multiple if-else statements within each other. There is no practical limit to the depth of nesting. This enables us to code a structure, called a *decision tree*. A decision tree is a set of binary questions & answers where at every case of an answer either a new question is asked or an action is carried out. Fig. 3.1.1 displays such a decision tree for *forecasting rain* based on the temperature, wind and air pressure values.

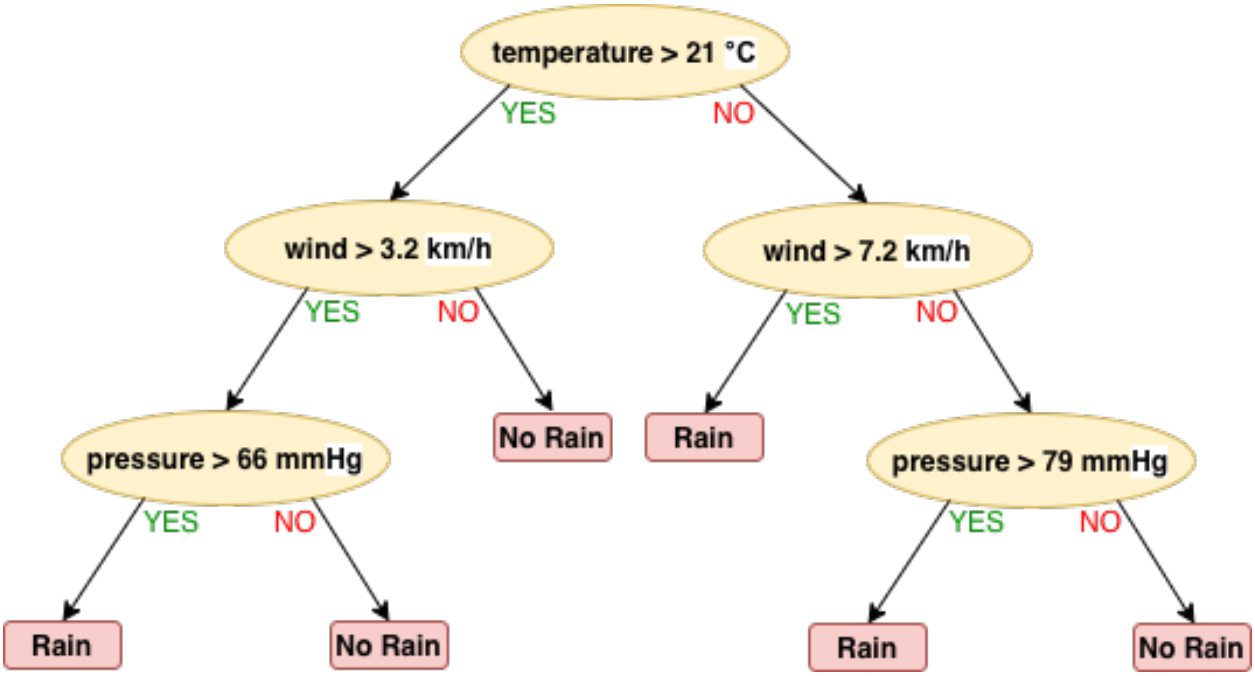


Fig. 3.1.1: An example decision tree.

The tree in Fig. 3.1.1 can be coded as nested if-else statements:

```
if temperature > 21:
    if wind > 3.2:
        if pressure > 66: rain = True
        else: rain = False
    else: rain = False
else:
    if wind > 7.2: rain = True
    else:
        if pressure > 79: rain = True
        else: rain = False
```

Sometimes the else case contains another if which has an else of its own. And this ladder goes on. For such a case, elif serves as a combined keyword for the else if action, as follows:

if $\boxed{Boolean\ expression_1}$: $\boxed{Statement_1}$

elif $\boxed{Boolean\ expression_2}$: $\boxed{Statement_2}$

⋮

elif $\boxed{Boolean\ expression_n}$: $\boxed{Statement_n}$

else : $\boxed{Statement_{otherwise}}$

The flowchart in Fig. 3.1.2 explains the semantics of the if-elif-else combination.

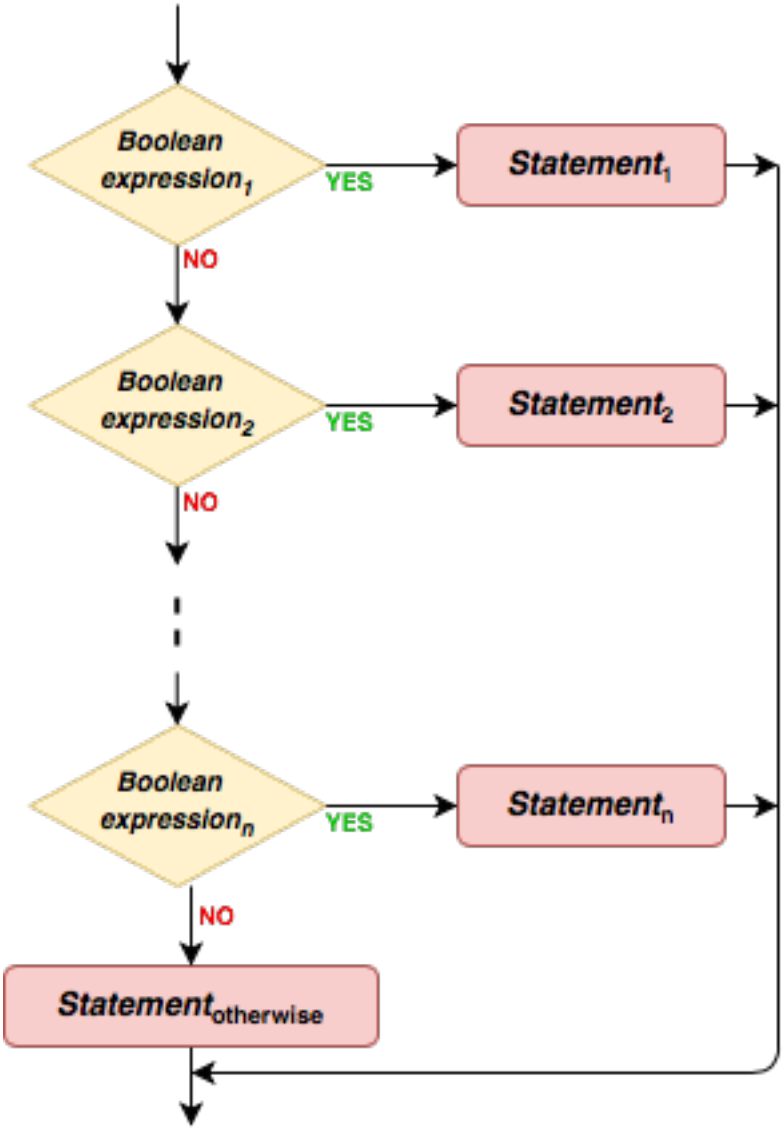


Fig. 3.1.2: The semantics of the if-elif-else combination.

There is no restriction on how many elif statements are used. Furthermore, the presence of the last statement (the else) is optional.

3.1.3 Practice

Let us assume that you have a value assigned to a variable x. Based on this value, a variable s is going to be set as:

$$s = \begin{cases} (x+1)^2, & x < 1 \\ x - 0.5, & 1 \leq x < 10 \\ \sqrt{x+0.5}, & 10 \leq x < 100 \\ 0, & otherwise \end{cases} \tag{3.1.1}$$

It is convenient to make use of an if-elif-else structure:

```
#@TODO Assign a value to x
x = 10

if x < 1: s = (x+1)**2
elif x < 10: s = x-0.5
elif x < 100: s = (x+0.5)**0.5
else: s = 0

print("s is: ", s)
```

```
s is:  3.24037034920393
```

3.1.4 Conditional expression

The if statement does not return a value. As said, that is so for all statements. They do not have a value of their own. A non-statement alternative that has a return value is the *conditional expression*.

It also uses the if and else keywords. This time if is not at the start of a statement, but following a value. Here is the syntax:

$\boxed{expression_{YES}}$ if $\boxed{Boolean\ expression}$ else $\boxed{expression_{NO}}$

This whole structure is an expression. It yields a value. So, it can be used in any place an expression is allowed to be used. It is evaluated as follows:

- First the $Boolean\ expression$ is evaluated.
- If the outcome is True then $expression_{YES}$ is evaluated and used as value ($expression_{NO}$ is left untouched).
- If the outcome is False then $expression_{NO}$ is evaluated and used as value ($expression_{YES}$ is left untouched).

Though it is not compulsory, it is wise to use conditional expression enclosed in parenthesis to prevent wrong sub-expression groupings.

Let us look at an example:

```
>>> x = -34.1905
>>> y = (x if x > 0 else -x)**0.5
>>> print(y)
5.84726431761
```

As you have observed, y is assigned the value $\sqrt{|x|}$. For code readability, it is strongly advisable not to nest conditional expressions. Instead, restructure your coding into nested if-else statements (by making use of intermediate variables).

3.2 Repetitive execution

Repeating a sequence of instructions over-and-over is a programming ingredient which is used frequently. This action is called *iteration* or *loop*.

We use iteration

to systematically deal with all possible cases or all elements of a collection, or

to jump from one case to another as a function of the previous case.

The repetition of a sequence of instructions is carried out either for a known number of times or until a criterion is met.

Iteration examples:

- Computing letter grades of a class (Type I, for all students).
- Finding the shortest path from city A to city B in a map (Type II).
- Root finding by Newton-Raphson method (Type II).
- Finding darkest and brightest point(s) in a image (Type I, for all pixels).
- Computing a function value using Taylor's expansion (Type I).
- Computing the next move in a chess game (Type II).
- Obtaining the letter frequency of a text (Type I, for all letters).

Python provides two statements for iteration. Namely, while and for. for is used mostly for (Type I) iterations whereas while is used for both types.

3.2.1 while statement

The syntax of while resembles the synax of the if statement:

while *Boolean expression* : *Statement*

It is certainly possible to have a statement group subject to the while, as it was with the if statement. The semantics can be expressed as the flowchart in Fig. 3.2.1:

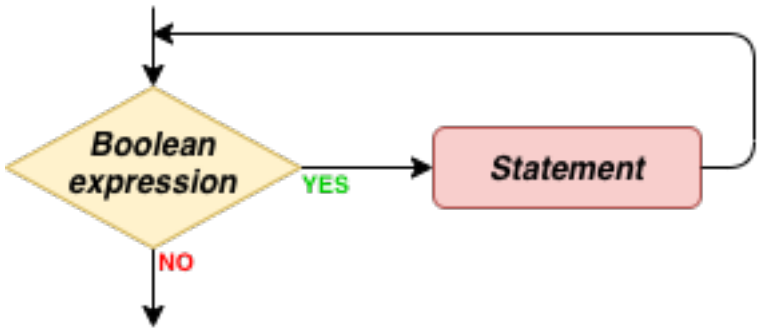


Fig. 3.2.1: The flowchart illustrating how a while statement is executed.

Later in this chapter, when we have introduced the for statement as well, we will consider some special statements (break and continue) that are allowed only under the scope of a while or for and are capable of altering the execution flow (to some limited extent).

Since a while statement is a statement, it can be part of another while statement (or any other compound statement):

```
while <condition-1>:
  statement-1
  statement-2
  ...
  while <condition-2>:
    statement-inner-1
    statement-inner-2
    ...
    statement-inner-M
  ... # statements after the second while
  statement-N
```

Of course, there is no practical limit on the nesting level.

Let us look at several examples below to illustrate these concepts.

3.2.2 Examples with while statement

Example 1: Finding Average of Numbers in a List

Let's say we have a list of numbers and we are asked to find their average. To make things easier to follow, let us start with the mathematical definition:

$$\text{avg}(L) = \frac{1}{N}\sum_{i=1}^{N} L_i, \tag{3.2.1}$$

where L_i is the ith number in the list which contains N numbers.

Let us see the algorithm before implementing a solution in Python (note that, in the equation, indexing started with 1 – compare this with the following):

```
Input: A list L that includes N numbers
Output: avg, which holds the average of numbers in L

Step 1: Initialize a sum variable with value 0
Step 2: Initialize an index variable, i, with value 0
Step 3: While i is less than N, Execute Steps 4-5
Step 4:     sum = sum + L[i]
Step 5:     i = i + 1
Step 6: avg = sum/N
```

Before we proceed with the Python implementation, make sure that you have understood the algorithm. The best way to make sure that is to take a pen & paper and go through each step while keeping a track of the variables, as if you are the computer.

Here is the implementation in Python:

```
# L: the list of numbers
# @TODO: Change the list and check if the code works
L = [10, -4, 4873, -18]
N = len(L)

sum = 0          # Step 1
i = 0            # Step 2

while i < N:     # Step 3
  sum = sum + L[i] # Step 4
  i = i + 1      # Step 5

avg = sum / N    # Step 6
print(avg)
```

```
1215.25
```

Have you noticed how close the Python code is to the algorithm? Python's principles for keeping things simple makes it so much easier to write that it can be very close to the pseudo-code.

Example 2: Standard Deviation

Now, let us look at a highly related problem, that of calculating the standard deviation. Standard deviation can be formally defined as follows:

$$\text{std}(L) = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(L_i - \text{avg}(L))^2}. \tag{3.2.2}$$

The extension of the previous example is rather straightforward therefore but we will write it down nonetheless since these are your first iterative examples.

```
L = [10, 20, 30, 40]
N = len(L)

# CALCULATE THE AVG FIRST (COPY-PASTE FROM THE FIRST EXAMPLE)
sum = 0
i = 0

while i < N:
  sum = sum + L[i]
  i = i + 1

avg = sum / N

# CALCULATE THE STD NOW
sum = 0
i = 0

while i < N:
  sum = sum + (L[i] - avg)**2
  i = i + 1

std = (sum / N)**0.5

print("Avg & Std of the list are: ", avg, std)
```

```
Avg & Std of the list are:  25.0 11.180339887498949
```

Exercise

Write down the algorithm that we used in this Python code as a pseudo-code.

Example 3: Factorial Factorial of a number, denoted by $n!$, is an important mathematical construct that we frequently use while formulating our solutions. $n!$ can be defined formally as:

$$n! = \begin{cases} n \times (n-1) \times ...2 \times 1, & \text{if } n > 0 \\ 1, & \text{if } n = 0. \end{cases} \tag{3.2.3}$$

Let us first write down the algorithm:

```
Input: n
Output: n_factorial

Step 1: If n is negative, n_factorial is 1. Return n_factorial and exit
Step 2: Initialize n_factorial with 1
Step 3: Initialize an index variable, i, with 1
Step 4: While i <= n, Execute Steps 4 an 5:
Step 5:     n_factorial = n_factorial * i
Step 6:     i = i + 1
Step 7: The result is n_factorial
```

which can be implemented in Python as follows:

```
# Let us choose an n value:
n = 5

if n < 0: n_factorial = 1  # Step 1
elif n > 0:                # Steps 2-6
  n_factorial = 1          # Step 2
  i = 1                    # Step 3
  while i <= n:            # Step 4
    n_factorial *= i       # Step 5
    i += 1                 # Step 6

print("n! is ", n_factorial)
```

```
n! is  120
```

Exercise

Modify our factorial implementation such that variable i starts from n.

Example 3: Computing Sine with Taylor Expansion

Let us assume we want to compute $\sin(x)$, for a given value of x, up to a given precision. This is just for the sake of having an understandable example. Actually today's CPUs have embedded math coprocessors that do this at microcode level. We will not use this convenience and do the computation ourselves.

The computation of many analytic functions are performed using the Taylor series expansion. The Taylor series expansion of $\sin(x)$ around zero is:

$$\sin(x) \approx x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} \cdots, \tag{3.2.4}$$

which could also be written as:

$$\sum_{k=0}^{\infty} \frac{(-1)^k}{(2k+1)!} x^{2k+1}. \tag{3.2.5}$$

Let us implement this. We will use the factorial example as a nested iteration. The outer iteration runs over the terms in the summation, and continues until a term becomes too small ($|term| < \epsilon$, where ϵ is a small value, e.g. 10^{-12}).

```
epsilon = 1.0E-12
x = 3.14159265359/4.0   # i.e. pi/4 (45 degrees in radians)
result = 0.0
k = 0
term = 2*epsilon    #just a trick to bypass the first test
while abs(term) > epsilon:
   # Calculate the denominator - i.e. (2k+1)!
   factorial = i = 1
   while i <= 2*k+1:
     factorial *= i
     i += 1

   # Now calculate the term
   term = (((-1)**k) / factorial) * (x**(2*k+1))

   result += term
   k += 1
```

This is a readable but at the same time quite an inefficient code:

* 2*k+1 is computed several times. Actually, do we need 2*k+1 at all? * It uses factorial computation which does not make use of its preceeding computations. For example 9! is actually $9 \times 8 \times 7!$. But it is computed as $2 \times 3 \times 4 \times 5 \times 6 \times 7 \times 8 \times 9$. Therefore, there is significant inefficiency regarding the use of factorial in the nested iteration. * Similar to the inefficiency with the factorial computation, the computation of $x^{(n+2)}$ does not make use of the already computed x^n value.

* Do we need to compute $(-1)^k$ to get an alternating sign? Can there be a simpler way to program this?

Let us investigate the proportionality of two consecutive terms $(k = n)$ and $(k = n + 1)$:

$$\frac{term_{n+1}}{term_n} = \frac{(-1)^{(n+1)} x^{2(n+1)+1}}{(2(n+1)+1)!} \times \frac{(2n+1)!}{(-1)^n x^{2n+1}} = \frac{-x^2}{(2n+2)(2n+3)}. \tag{3.2.6}$$

This is interesting because it suggests us a relatively faster way to compute the next term in the series. We do not have to compute x^n for each new term that we are going to add. We also discover that there is nothing magical about $(2n+1)$. It can simply be called as $(d \equiv 2n+1)$ which increments by 2 for each consecutive term. Then we have

$$\frac{term_{n+1}}{term_n} = \frac{-x^2}{(d+1)(d+2)} \tag{3.2.7}$$

Now, let us see the more efficient implementation in action:

```
epsilon = 1.0E-12
x = 3.14159265359/4.0   # i.e. pi/4 (45 degrees in radians)
x_square = x*x
term = x
result = term
d = 1    # 2*n+1
while abs(term) > epsilon:
    term *= -x_square/((d+1)*(d+2))
    result += term
    d += 2

print("sin(x) [Ours] is: ", result)

# Compare this with the CPU-implemented version
from math import *
print("sin(x) [CPU ] is: ", sin(x))
```

```
sin(x) [Ours] is:  0.707106781186584
sin(x) [CPU ] is:  0.7071067811865841
```

3.2.3 for statement

The syntax of the for statement is:

for $\boxed{Variable}$ in $\boxed{Iterator}$: $\boxed{Statement}$

An *iterator* is an object that provides a countable number of values on demand. It has an internal mechanism that responds to three requests:

1. Reset (initialize) it to the start.
2. Respond to the "Are we at the end?" question.
3. Provide the next value in line.

The for statement is a mechanism that allows traversing all values provided by an $Iterator$, one-by-one assigning them to a $Variable$ and then executiong a given $Statement$.

The semantics of the for statement is displayed in Fig. 3.2.2:

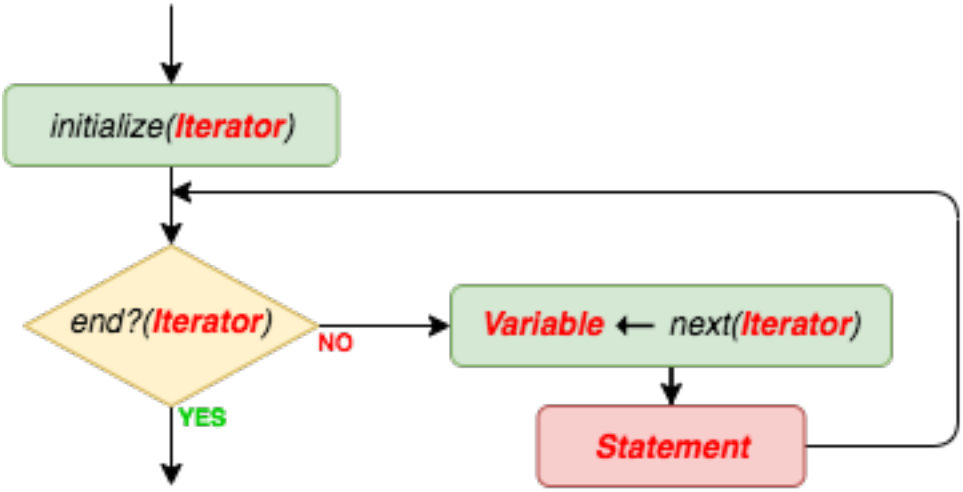


Fig. 3.2.2: The flowchart illustrating how a for statement is executed.

What iterators do we have?

1. All containers are also iterators (strings, lists, dictionaries, sets).
2. The built-in function range returns an iterator that generates a sequence of integers, starting from 0 (default), and increments by 1 (default), and stops before a given number:

range(*start*, *stop*, *step*)

Parameter	Opt./Req.	Default	Description	
start	Optional	0	An integer specifying the start value.	
stop	Required		An integer specifying the stop value (stop value will not be taken)	
step	Optional	1	An integer specifying the in crementation.	

3. A user-defined iterator. Since this is an introductory book, we will not cover how this is done.

3.2.4 Examples with for statement

Example 1: Words Starting with Vowels and Consonants

Let us split a list of words into two lists: those that start with a vowel and those that start with a consonant.

```
mixed = ["lorem","ipsum","dolor","sit","amet,","consectetur","adipiscing","elit","sed","do","eiusmod","tempor",
↪"incididunt","ut","labore","et","dolore","magna","aliqua"]
vowels = []
consonants = []
for word in mixed:
   if word[0] in ['a','e','i','o','u']: vowels += [word]
   else: consonants += [word]

print("Starting with consonant:", consonants)
print("Starting with vowel:", vowels)
```

```
Starting with consonant: ['lorem', 'dolor', 'sit', 'consectetur', 'sed', 'do', 'tempor', 'labore', 'dolore', 'magna']
Starting with vowel: ['ipsum', 'amet,', 'adipiscing', 'elit', 'eiusmod', 'incididunt', 'ut', 'et', 'aliqua']
```

Exercise

Write this solution in pseudo-code.

Example 2: Run-length Encoding

Run-length encoding is a compression technique for data that repeats itself contiguously. Images bear such data: For example, a clear sky portion is actually rows of 'blue'. In the example below, we investigate a string for consequent character occurrences: If found, repetitions are coded as a list of [character, repetition count].

So, a text of "aaaaaaxxxxmyyyaaaasssssssssttttuivvvv" gets encoded as:
[['a',6],['x',4],'m',['y',3],['a',4],['s',9],['t',3],'u','i',['v',4]].

```
text = "aaaaaaxxxxmyyyaaaasssssssssttttuivvvv"
code_list = []
last_character = text[0]
count = 1

# Go over each character except for the first
for curr_character in text[1:]:
    # If curr_character is equal to last_character, we found a duplicate
    if last_character == curr_character:
        count += 1
    else:
        # We have finished a sequence of same characters: Save the count and
        # reinitialize last_character and count accordingly
        code_list += [last_character if count==1 else [last_character, count]]
        count = 1
        last_character = curr_character

# handle the last_character here:
code_list += [last_character if count==1 else [last_character,count]]

print(code_list)
```

```
[['a', 6], ['x', 4], 'm', ['y', 3], ['a', 4], ['s', 9], ['t', 3], 'u', 'i', ['v', 4]]
```

Exercise

Modify this code such that it removes consecutive duplicate characters. In other words, "aaaabbbcdd" should be reduced to "abcd".

Example 3: Permutations

Permutations are keys to shuffling a sequence of data. Permutations can be denoted in various forms. One way is to give an order list: The i^{th} element of the list stores the old position where the i^{th} element of the new sequence will come from. In our implementation counting starts from 0 (zero).

```
word_list = ["he","came","home","late","yesterday"]
permutation = [4,3,2,0,1]
length = len(permutation)
new_list = [None]*length
```

(continues on next page)

```
for i in range(length):
    new_list[i] = word_list[permutation[i]]

print(new_list)
```

```
['yesterday', 'late', 'home', 'he', 'came']
```

Example 4: List Split

In the example below the first element of the list is taken and the rest is split into two lists: Those that are smaller than the first element, and those that are not.

```
list_to_split = [42, 59, 53, 84, 43, 8, 75, 34, 40, 89, 29, 15, 51, 6, 90, 32, 58, 77, 4, 24]
list_smaller = []
list_not_smaller = []

for x in list_to_split[1:]:
    if x < list_to_split[0]: list_smaller += [x]
    else: list_not_smaller += [x]

print("list head was.       :",list_to_split[0])
print("smaller than head    :", list_smaller)
print("not smaller than head :", list_not_smaller)
```

```
list head was.       : 42
smaller than head    : [8, 34, 40, 29, 15, 6, 32, 4, 24]
not smaller than head : [59, 53, 84, 43, 75, 89, 51, 90, 58, 77]
```

Example 5: Dot Product

A frequently used operation with vectors is the dot product between two vectors, e.g. **u** and **v**:

$$\mathbf{u} \cdot \mathbf{v} = \sum_{i=1}^{n} u_i v_i, \tag{3.2.8}$$

where the vectors have n elements each. Let us implement this in Python:

```
# Define the vectors (as lists)
u = [1, 2, 4, 10]
v = [10, 4, 2, 1]
n = len(u)
dot_prod = 0

if n != len(v): print("Sizes don't match!")
else:
  for i in range(n):
    dot_prod += u[i] * v[i]
  print("u . v is: ", dot_prod)
```

```
u . v is:  36
```

Exercise

Implement dot product between two vectors using a while statement.

Example 6: Angle between Two Vectors

The angle between two vectors **u** and **v** is closely related to their dot product:

$$\mathbf{u} \cdot \mathbf{v} = ||\mathbf{u}||\ ||\mathbf{v}||\ \cos(\theta), \tag{3.2.9}$$

where θ is the angle between the vectors, and $||\cdot||$ denotes the norm of a vector:

$$||\mathbf{u}|| = \sqrt{\mathbf{u} \cdot \mathbf{u}} = \sqrt{\sum_{i=1}^{n} u_i^2}. \tag{3.2.10}$$

The angle can be derived from these as follows:

$$\theta = \arccos\left(\frac{\mathbf{u} \cdot \mathbf{v}}{||\mathbf{u}||\ ||\mathbf{v}||}\right). \tag{3.2.11}$$

Let us implement this:

```
from math import sqrt, acos

# Define two vectors that have 90deg (pi/2) between them
u = [1, 0, 0]
v = [0, 1, 0]
n = len(u)

if n != len(v):
  print("Sizes don't match!")

else:
  # Calculate dot product & norms
  dot_prod = u_norm_sum = v_norm_sum = 0
  for i in range(n):
    dot_prod   += u[i] * v[i]
    u_norm_sum += u[i] ** 2
    v_norm_sum += v[i] ** 2

  u_norm = sqrt(u_norm_sum)
  v_norm = sqrt(v_norm_sum)

  theta = acos(dot_prod / u_norm * v_norm)

  print("angle between u and v is: ", theta)
```

```
angle between u and v is:  1.5707963267948966
```

Example 7: Matrix Multiplication

Our last example is matrix multiplication. As you know, a matrix A can be multipled with the matrix B if the column-count (m) of A is equal to the row-count of B. The product matrix is of a size where row-count equals that of A and the column-count equals that of B.

$$(A \cdot B)_{ij} = \sum_{k=0}^{m-1} A_{ik} \cdot B_{kj}. \tag{3.2.12}$$

As we discussed in the previous chapter, since Python provides no two dimensional containers, but a very fexible list container, matrices are mostly represented as lists of lists each of which represent a row. For example, a 3×4 matrix:

$$\begin{pmatrix} -2 & 3 & 5 & -1 \\ 0 & 3 & 10 & -7 \\ 11 & 0 & 0 & -8 \end{pmatrix} \tag{3.2.13}$$

can be represented as:

```
A = [[-2,3,5,-1], [0,3,10,-7], [11,0,0,-8]]
```

This representation is also coherent with the indexing of matrices: A_{ij} is accessible in Python as A[i][j]. The code below multiples two matrices, given in this representation, and prints the result.

```
A = [[-2,3,5,-1], [0,3,10,-7], [11,0,0,-8]]
B = [[2,1], [-1,1], [0,4], [8,0]]

# C = A*B

# Create a result matrix with entries filled with 0 (zeros)
C = []
for i in range(len(A)):
  C += [[0] * len(B[0])]  # This is done to overcome the aliasing problem

for i in range(len(C)):
   for j in range(len(C[0])):
      for k in range(len(B)):
         C[i][j] += A[i][k]*B[k][j]

print(C)
```

```
[[-15, 21], [-59, 43], [-42, 11]]
```

3.2.5 continue and break statements

It is possible to alter the flow of execution in a while or for repetition. This is almost always used in combination with statement grouping, where a group of statements is subject to the while or for.

Let's assume you are executing a sequence of statements by means of statement grouping. Somewhere in the process, without waiting for the terminating condition to become False in a while statement or exhausting all items in the iterator in a for statement, you decide to terminate looping. The break statement does exactly serve this purpose: The while (or for) is stopped immediately and the execution continues with the statement after the while (or for).

Similarly, somewhere in the process you decide not to execute the rest of the statements in the grouping and straight away continue with the test. This is achieved by a continue statement usage. See Fig. 3.2.3 and Fig. 3.2.4 for how the continue statement changes the flow of execution.

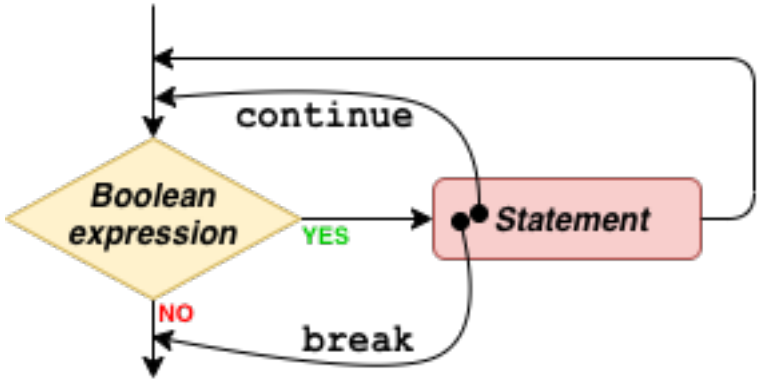


Fig. 3.2.3: How continue and break statements change execution in a while statement.

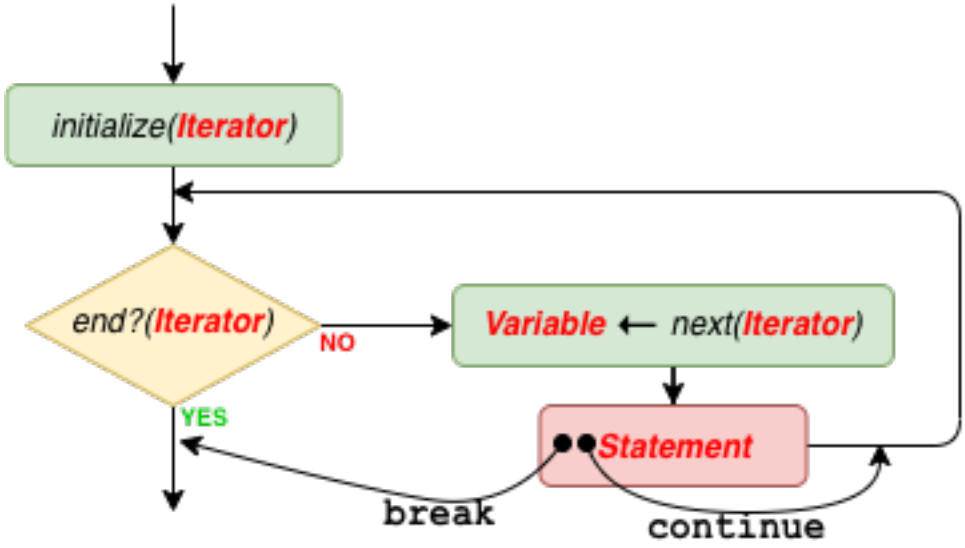


Fig. 3.2.4: How continue and break statements change execution in a while statement.

3.2.6 Set and list comprehension

A well-known and extensively-used notation to describe a set in mathematics is the so-called 'set-builder notation'. This is also known as *set comprehension*. This notation has three parts (from left-to-right):

1. an expression in terms of a variable,
2. a colon or vertical bar separator, and
3. a logical predicate.

Something like this:

$$\{x^3 \mid x \in \{0, 1, \ldots, 7\}\} \tag{3.2.14}$$

which defines the set:

$$\{0, 1, 8, 27, 64, 125, 216, 343\} \tag{3.2.15}$$

or as something more elaborate:

$$\{x^3 \mid x \in \{0, 1, \ldots, 7\} \wedge (x \bmod 2) = 1\} \tag{3.2.16}$$

defining

$$\{1, 27, 125, 343\} \tag{3.2.17}$$

There exists a Python notation with the same semantics both for sets and lists:

```
>>> [x**3 for x in range(8)]
[0, 1, 8, 27, 64, 125, 216, 343]
>>> {x**3 for x in range(8)}
{0, 1, 64, 8, 343, 216, 27, 125}
```

The second example, which imposes a constraint on x to be an odd number, would be coded as:

```
>>> [x**3 for x in range(8) if x%2 == 1]
[1, 27, 125, 343]
>>> {x**3 for x in range(8) if x%2 == 1}
{1, 27, 125, 343}
```

Certainly the condition (following the if keyword) could have been constructed as a more complex boolean expression.

The expression that can appear to the left of the for keyword can be any Python expression and container. Here are a couple of more examples on matrices.

```
print( [[0 for i in range(3)] for j in range(4)] )
print( [[1 if i==j else 0 for i in range(3)] for j in range(3)] )
print( [[(i,j) for i in range(3)] for j in range(4)] )
print( [[(i+j)%2 for i in range(3)] for j in range(4)] )
print( [[i+3*j for i in range(3)] for j in range(4)] )
```

```
[[0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0]]
[[1, 0, 0], [0, 1, 0], [0, 0, 1]]
[[(0, 0), (1, 0), (2, 0)], [(0, 1), (1, 1), (2, 1)], [(0, 2), (1, 2), (2, 2)], [(0, 3), (1, 3), (2, 3)]]
[[0, 1, 0], [1, 0, 1], [0, 1, 0], [1, 0, 1]]
[[0, 1, 2], [3, 4, 5], [6, 7, 8], [9, 10, 11]]
```

3.3 Important Concepts

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

- Conditional execution with if, if-else, if-elif-else statements.
- Nested if statements.
- Conditional expression as a form of conditional computation in an expression.
- Repetitive execution with while and for statements.
- Changing the flow of iterations with break and continue statements.
- Set and list comprehension as a form of iterative computation in an expression.

3.4 Further Reading

- "Managing the size of a problem", Chapter 4 of "Introduction to Programming Concepts with Case Studies in Python": https://link.springer.com/chapter/10.1007/978-3-7091-1343-1_4

Exercises

We have provided many exercises throughout the chapter, please study and answer them as well.

- Implement the while statement examples in Section 5.2.2 using for statement:
- Calculating the average of a list of numbers.
- Calculating the standard deviation of a list of numbers.
- Calculating the factorial of a number.
- Taylor series expansion of sin(x).
- Implement the for statement examples in Section 5.2.4 using while statement:
- Words with vowels and consonants.
- Run-length encoding.
- Permutation.
- List split.
- Matrix multiplication.
- What does the variable c hold after executing the following Python code?

```
c = list("address")
for i in range(0, 6):
    if c[i] == c[i+1]:
        for j in range(i, 6):
            c[j] = c[j+1]
```

- Write a Python code that removes duplicate items in a list. E.g. [12, 3, 4, 12] should be changed to [12, 3, 4]. The order of items should not change.
- Write a Python code that replaces a nested list of numbers with the average of the numbers. E.g. [[1, 3], [4, 5, 6], 7, [10, 20]] should yield [2, 5, 7, 15].
- Write a Python code that finds all integers dividing a given integer without a remainder. E.g. for number 21, your code should compute [1, 3, 7].
- Write a Python code that converts an integer (less than 128) into binary string using a for/while statement. E.g. 21 should be represented as 00010101.
- Write a Python code that convers a binary string like 00010101 into integer.