Sinan Kalkan
Onur T. Şehitoğlu
Göktürk Üçoluk

Programming with Python for Engineers

Springer

Programming with Python for Engineers

Sinan Kalkan · Onur T. Şehitoğlu ·
Göktürk Üçoluk

Programming with Python for Engineers

Springer

Sinan Kalkan
Department of Computer Engineering
Middle East Technical University
Ankara, Türkiye

Onur T. Şehitoğlu
Department of Computer Engineering
Middle East Technical University
Ankara, Türkiye

Göktürk Üçoluk
Department of Computer Engineering
Middle East Technical University
Ankara, Türkiye

ISBN 978-3-031-57147-3 ISBN 978-3-031-57148-0 (eBook)
https://doi.org/10.1007/978-3-031-57148-0

This Springer imprint is published by the registered company Springer Nature Switzerland AG
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

We dedicate this book to our lonely and beautiful country which we love passionately.

Foreword

Questions like “is there an algorithm for this?”, “can we compute that?”, or “can we approximate the next thing?” used to be the domain of computer scientists; now, almost every train of thought can be enhanced by asking versions of these questions. They’re variants of the practical question “how can I do this better/efficiently/safely?”, which is behind every advance since at least the fire-hardened spear. You really can’t do science or engineering if you ignore computational thinking, and to do science or engineering well requires hearing answers to computational questions in your head by reflex. The most usual (and usually the best) way to start fostering this reflex is to practice programming. This book has important features that will help you practice:

- You can browse it online, at http://pp4e.online/, if you run into problems.
- There is interactive content, where you can provide input and see what happens.
- You can run each example in each chapter, using the Jupyter Notebook version of the book.
- And there are lots of examples and exercises.

Most people can learn to write simple programs quite well with practice and with careful guidance. But programming is a deep and subtle art. You’ll find as you proceed to learn that writing programs that do what you want them to do can be surprisingly delicate. First, one has to get used to all that syntax. This takes a bit of patience, and you should be willing to forgive yourself silly mistakes as long as you don’t make too many—I’ve been programming for at least 40 years now, and I occasionally use the wrong bracket, omit an indentation, drop a semicolon, or whatever. Then you’ll need to learn to express yourself, so that the program does what you want. This mostly just takes practice, but the next step is hard. You’ll find very often that you didn’t clearly understand what you wanted the program to do, and the program you wrote reflects that failure of understanding—it’s actually doing what you wanted, but you didn’t want the right thing in the first place. While everyone finds this experience annoying on occasion, being forced to very clearly think through what is the right thing is a valuable discipline.

An important help with all these problems is practice. Strong programmers have written and read many different programs. Some lucky people just have gifts, but most strong programmers got that way by practice and imitation. Some types of problem—for example, sorting a list of numbers—turn up again and again in practice, so strong programmers remember patterns and learn from one another.

Practice will help you develop a valuable habit. Suspect that you might make mistakes, and think about how you can find and fix them (Chap. 9 is good on this). Strong programmers know what behaviors their code should exhibit—for example, squared numbers are never negative—and check it is doing so. So if you have something that should be the square of a number and it's negative, there's a problem. Where the problem is, could get interesting; but there is one.

You'll find that producing programs of even moderate size that are known to be correct is challenging. Among other difficulties, specifying the correct behavior of a program can be difficult. Embracing these challenges has produced astonishing artifacts that have had an impact on a very large number of lives—for example, the fly-by-wire software that is a key part of modern air transportation allows people to visit their relatives cheaply and safely.

As I looked through this book, I found the lovely line "There is no servant without fault, love me with the faults I have". It captures my own experience of learning to program and continuing to program rather well. I still find it frustrating that programs do what I wrote, rather than what I should have written; and I still find trying to express what I know (or what I suspect) using code to be creative and fruitful. I close by commending the authors for putting so much time and effort into helping you learn to program. My life has certainly been enriched by learning to program, and I hope yours will be too.

June 2023

David Forsyth
Fulton Watson Copp Chair in Computer Science
University of Illinois at Urbana-Champaign
Champaign, IL, USA

Preface

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 endeavor requires a delicate balance between providing in-depth details on computers and programming in a complete manner and the programming needs of science and engineering disciplines. With the hope of providing a suitable balance, the book uses Python as the programming language, since it is easy to learn and program with. Moreover, to keep the balance, the book is made up of three topics:

- The Basics of Computers and Computing (Chaps. 1–3): The book starts with the definitions of computation and an introduction to modern hardware and software infrastructure on which programming is commonly performed and the spectrum of programming languages.
- Programming with Python (Chaps. 4–9): These chapters present the basic building blocks of Python programming and continue with the ground formation for solving a problem in 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.
- Using Python for Science and Engineering Problems (Chaps. 10–12): The book reserves the last chapters to practical and powerful tools that are commonly used in 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.

A Note on the Python Version Used in This Book

The Python version referred to in this book is 3.6. However, to the best of our knowledge, the version at the time of printing (version 3.11) is compatible with the codes presented in this book.

Python is a language that was initially introduced in 1991 and has seen regular version upgrades, with new versions typically being released approximately once a year. Starting from Python 3.0, each major version has been maintained for about five years. It's important to note that Python 3.0 introduced backward incompatibilities, meaning that Python 2 code does not run unmodified with a Python 3 interpreter. However, there are tools available that help in transitioning Python 2 code to Python 3. As of now, all Python 3.x versions maintain backward compatibility. The expectation is that if a future Python 4.0 version is released, it will strive to maintain backward compatibility.

For detailed information on the version history of Python, we encourage readers to consult the following resources:

- https://docs.python.org/3.11/whatsnew/index.html for the current version (as of writing this book).
- https://devguide.python.org/versions/.
- https://en.wikipedia.org/wiki/History_of_Python.
- https://en.wikibooks.org/wiki/Python_Programming/Version_history.

Please refer to these resources to explore the specific changes and features introduced in different Python versions.

How to Use the Book

This book takes an "interactive" and "minimalist" approach, focusing on direct interaction with examples and problems rather than emphasizing every detail or specialized subject. It is not intended to serve as a comprehensive reference manual, but rather as a "first things first" and "hands-on" guide. To provide additional information on skipped details, the book will include links to further reading. With this in mind, readers are strongly encouraged to thoroughly read and engage with all of the content of the book.

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 reader to complete/write a piece of Python program, run it, and inspect the result, from time to time. The reader is encouraged to complete these minor tasks. Such tasks and interactions are of great help in gaining an acquaintance with Python and building up self-confidence in solving problems with Python.

The printed version of the book and in parallel the e-book extensively contain four types of boxes. They are easily recognizable by their background color and numbered like figures or tables as follows:

[1] https://jupyter.org.

Interactive Example 0.1

http://pp4e.online

- This indicates a dynamic content, which can be in the form of a video or a visual tutorial web application, where the output updates in real-time, based on your input. In the e-book version, you can switch to the Web application by clicking the title of the box. However, in the printed version, this interactive feature is not available and the box displays a static image snapshot. It is highly recommended that the reader temporarily switches to the e-book version to access the relevant box and to activate its interactive functionality by clicking on it.

Hands-on Code 0.1

https://pp4e.online/c0s1

- This indicates a code snippet that is executable and modifiable in the e-book version. In the printed book, the code and its output will be displayed. In the e-book version, when you click on the box, you will be directed to an interactive Python environment. Essentially, you will be taken to a portal containing the Python interpreter with some pretyped code provided. There, you can run the code and experiment with it by making modifications and rerunning it.

Installation Notes 0.1

- After completing your learning period, you will likely find yourself writing code on standalone computers. While it is possible to continue programming on Jupyter Notebook, doing so may introduce additional web connectivity overhead. Moreover, standalone software deployment is often not feasible in those environments. Therefore, it is advisable to learn how to set up a Python run-time environment on your local computer. These boxes provide you with information on this matter.

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

- The gray boxes, such as the one above, are integral components of the written tutoring material. They are static and intended for reading purposes only. They do contain sample Python code runs.

Interactive Web Page of the Book

To facilitate dynamic contents and interaction with code segments, all chapters are served at the following Web page and executable in your web browser without installing any software: https://pp4e.online/.

Solutions Manual

A pdf solutions manual for problems posed in this book is available for download by instructors adopting this book for use in their courses from https://sites.google.com/springernature.com/extramaterial/lecturer-material.

Textual Conventions in the Book

- Codes and outputs are presented in `teletype` font.
- The first introduction of a term or a keyword is emphasized in *italic*.
- The definitions of the key concepts are also provided as a glossary at the end of the book.
- Python function names, whether built-in or user-defined, are always followed by a pair of parentheses "()" when referenced outside of the code.
- External references, such as websites or external resources, are provided as footnotes or end-notes.
- Occasionally, when a term description is lengthy, angle brackets are employed to enclose the description, similar to their use in the syntax of the language.

Ankara, Türkiye

Sinan Kalkan
Onur T. Şehitoğlu
Göktürk Üçoluk

Acknowledgements

We would like to express our sincere gratitude and appreciation to the following individuals.

First and foremost, we are deeply indebted to Prof. Dr. Fatoş Yarman Vural for her unwavering support and constant encouragement throughout this journey. Her belief in this project has been invaluable, inspiring us to strive for excellence.

We are profoundly grateful to Burçin Sapaz, Hasan Ulusoy, Dr. Hüseyin Aydın, Jiaee Cheong, Dr. Levent Eksert, and Dr. Özcan Dülger for their meticulous proofreading. Their keen eyes for detail and constructive criticism have been very valuable in shaping the final content.

Furthermore, we would like to thank Dr. Özgür Kaya for his assistance with the technical infrastructure and his unwavering friendship, which have been greatly important. His expertise and willingness to lend a helping hand have made a significant difference.

We are immensely grateful to Prof. David Forsyth, from UIUC, for his time, effort, and dedication in crafting the foreword. We feel privileged to have his endorsement, which has undoubtedly contributed to the overall visibility of this work.

Last but not least, we would also like to extend our heartfelt thanks to our life partners, Dr. Gökçe, Prof. Dr. Solmaz, and Dr. Gülnur. Their constant support and understanding have made this endeavor possible. They have not only tolerated the sacrifices of time and energy that this project demanded, but also provided a continuous stream of love, motivation, and inspiration.

Contents

Computing and Computers 1

In this chapter, we will first introduce the fundamental concepts for our book: computing, computer, and programming. Then, we will provide an overview of the internals of a modern computer. To do so, we will first describe the 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".

1.1 An Overview

1.1.1 What Is Computing?

Computing can be broadly described as the process of inferring data from data. Through such a process, we are interested in solving a problem, called the *task*, wherein, when the task is solved, the data of interest is produced. In this context, the original data is called the *input (data)* and the inferred one is the *output (data)*.

To better grasp these concepts, let us look at some examples:

- Multiplying two numbers, X and Y, and subtracting 1 from the multiplication result is a *task*. The two numbers X and Y are the *input* and the result of $X \times Y - 1$ is the *output*.
- Detecting 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 expected, the positions of the pixels 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, Height, Gender**>. The *task*, an optimization problem in essence, is to find the curve (i.e., the function) that goes through these tuples in a three-dimensional space spanned by Age, Height, and Gender. As you may have already guessed, the *output* are the parameters that define the function and an error that describes 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.

S. Kalkan et al., *Programming with Python for Engineers*,
https://doi.org/10.1007/978-3-031-57148-0_1

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

1.1.2 Are all “Computing Machinery” Alike?

It is tempting to view modern computers as the only computing machinery. However, this is a common misconception. There are diverse architectures based on totally different physical phenomena that can compute. A biological brain is a mesmerizing example of this. To better understand the concept of computing and computers, let us briefly look at how a brain computes.

A brain relies on completely different mechanisms compared to the *microprocessors* in our laptops, desktops, mobile phones, and calculators. A brain is a densely connected web of a huge number of small processing units, called *neurons*. A *neuron* is a cell that has several input channels, called *dendrites*, and a single output channel, called *axon*. An axon branches out like a tree towards its end and connects the neuron to the dendrites of other neurons (Fig. 1.1).

Neurons are connected to each other at *synapse*s. A synapse functions like a valve, enabling the flow of molecules, called *neurotransmitters*, to transmit a signal from one neuron to another. Neuro-

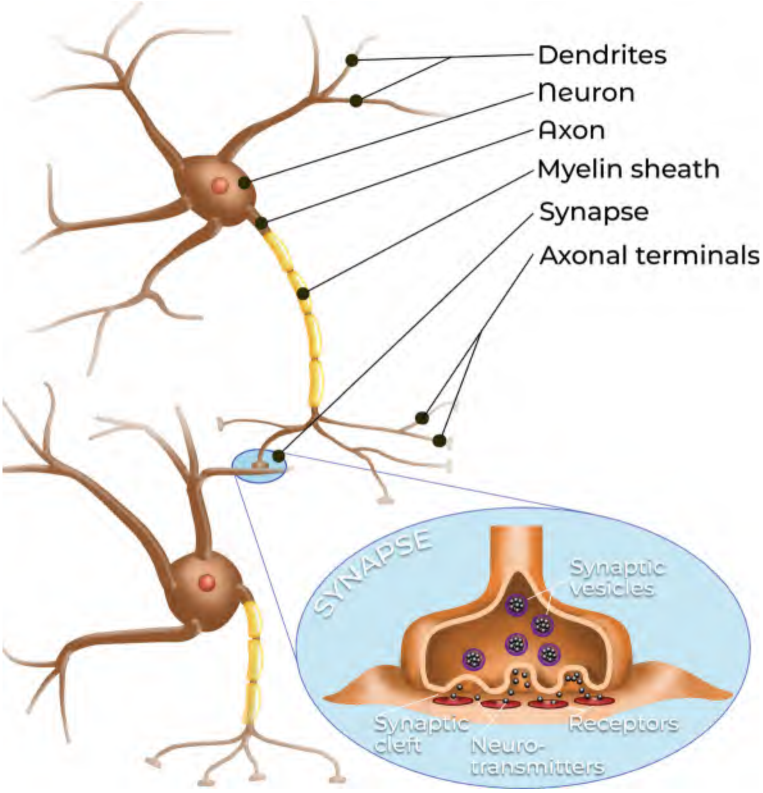


Fig. 1.1 Our brains are composed of simple processing units, called neurons. Neurons receive signals (information) from other neurons, process those signals, and produce an output signal to other neurons. Neurons “communicate” with each other by transmitting neurotransmitters via synapses

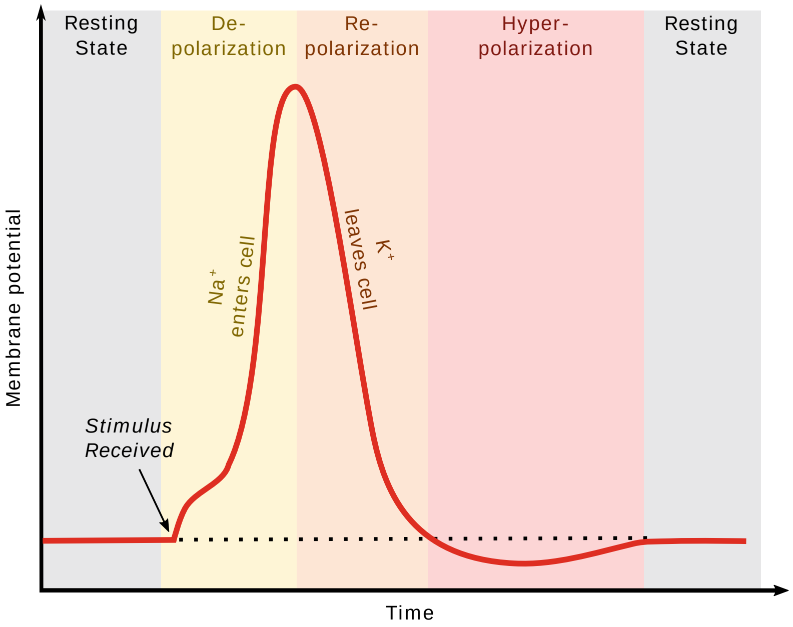


Fig. 1.2 When a neuron receives a "sufficient" amount of signals, i.e., stimulated, it emits an electrical charge along its axon, i.e., it fires

transmitters are released into the synapse as the "output" and they "interact" with the dendrite (i.e., the "input") of the other neuron.

All neurotransmitters flown in through the input channels (dendrites) have an accumulative effect on the (receiving) neuron. When the amount of messages (stimulus) received through the neuron's dendrites is above a threshold, the neuron becomes stimulated (its voltage level increases) and emits an electrical burst through its axon via charged elements, as illustrated in Fig. 1.2. Note that such an excitation is more sophisticated than a "what-comes-in-goes-out" process. After a burst, the neuron returns to its resting (normal) state and becomes ready to be excited again.

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

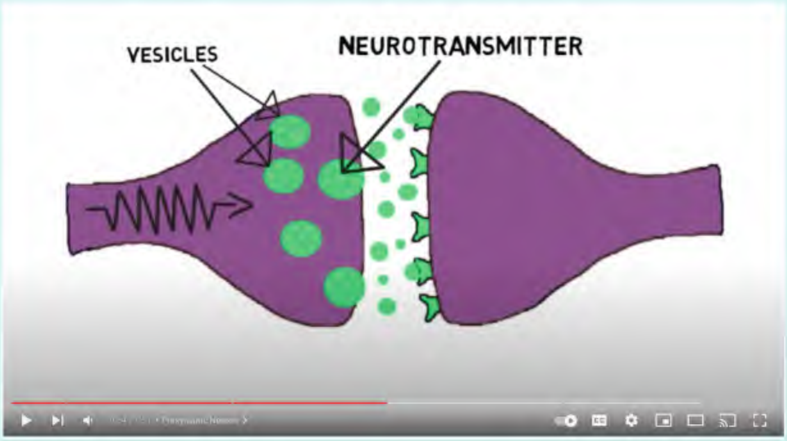
Some neurons are specialized in creating neurotransmitter emissions under certain physical events. For example, neurons in the retina create neurotransmitters if light falls on them. Some, on the other hand, create physical effects, like creating an electric potential that activates a muscle cell. These 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.

Interactive Example 1.1

http://youtu.be/WhowH0kb7n0

The following is a short video that provides more information about synaptic communication between neurons:



A snapshot of the “2-Minute Neuroscience: Synaptic Transmission” video at http://youtu.be/WhowH0kb7n0
[Provided with an explicit permission from the owner, Marc Dingman]

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

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

In contrast 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 centrally in a unit. The microprocessor consists of sub-units but they are extremely specialized in function and far less in number compared to the 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 the rest of the chapter, 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 examples include the “analog computer”, the “quantum computer”, and the “connection machine”.

1.1.3 What Is a “Computer”?

As you may have 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 concept, namely, a device that has a "microprocessor".

A computer...

- is based on binary (`0/1`) representations such that all inputs are converted to `0`s and `1`s and all outputs are converted from `0/1` representations to a desired form, mostly a human-readable one. The processing takes place on `0`s and `1`s, 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 V, a signal).
- consists of two clearly distinct entities: The Central Processing Unit (CPU), also known as the microprocessor (μP), and a *Memory*. In addition to these, the computer is connected to or incorporates other electronic units, known as "peripherals", mainly for input–output purposes.
- 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 always produce the same result. It is possible to include randomization in this process only by making use of a peripheral that provides electronically random inputs.

1.1.4 What Is Programming?

The CPU (the microprocessor—μP) 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 (e.g., shifting digits to the left or right; inverting `0`s and `1`s).
- Transferring to/from memory.
- Comparing numbers (e.g., whether a number n_1 is 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 technical terms, tasks are solved by a CPU by executing a sequence of binary instructions. Such sequences of binary instructions are called machine codes. Constructing machine codes for a CPU is called "machine code programming".

But, a *program* has a broader meaning:

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

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.

1.2 The Organization of Contemporary Computers

1.2.1 The von Neumann Architecture

1.2.1.1 John von Neumann

The following is a short biography written with OpenAI's ChatGPT AI language model (generated with the following prompt on March 2023: "Write a short biography of John von Neumann focusing on his contributions to computer organization"):

John von Neumann (1903–1957)

"John von Neumann was a Hungarian-American mathematician who made significant contributions to computer organization, including the development of the von Neumann Architecture. This architecture, which is still used in most modern computers, is based on the principles of a stored-program computer.

Von Neumann's work on the architecture of computers was groundbreaking. He recognized the value of a flexible stored program, which allowed programs to be easily changed without altering the computer's basic circuits. This concept is now a fundamental principle of computer organization and is used in the design of all modern computers.

Von Neumann was also instrumental in the development of the Electronic Numerical Integrator and Computer (ENIAC), the first electronic computer. He helped design the computer and was responsible for programming it. His work on the ENIAC led to the development of other computers, including the UNIVAC, which was the first commercial computer.

Von Neumann's research extended beyond computer architecture. He contributed to the development of game theory and quantum mechanics, and he was involved in the Manhattan Project, a project to build an atomic bomb. He was a prolific author, and his works include 'Theory of Games and Economic Behavior', 'Mathematical Foundations of Quantum Mechanics', and 'The Computer and the Brain'.

Von Neumann's contributions to computer organization have had a lasting impact on the field. His work on the von Neumann Architecture provided a framework for the development of modern computers, and his ideas continue to shape the way we think about computers and their applications in the modern world."

1.2.1.2 Components of the von Neumann Architecture

The *von Neumann Architecture* (Fig. 1.3) 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).

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. 1.3). These buses 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 machines conforming to the von Neumann Architecture are working on electricity, and therefore these zeros and ones correspond to voltages. A one usually indicates the presence of a 5 V and a zero denotes the absence of it.

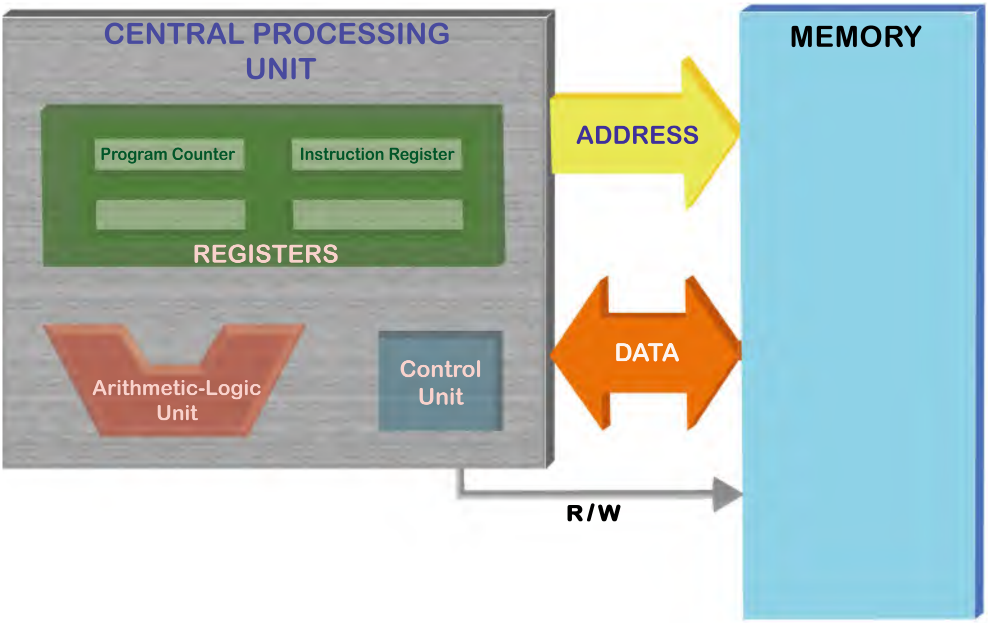
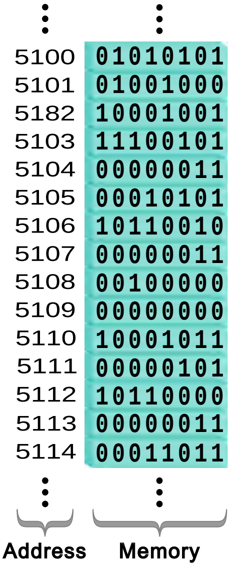


Fig. 1.3 A block structure view of the von Neumann Architecture. [Reproduced with permission from: G. Üçoluk, S. Kalkan, Introduction to Programming Concepts with Case Studies in Python, Springer, 2012 [1]]

1.2.1.3 The Memory

The memory can be imagined as pigeonholes organized as a stack of rows (Fig. 1.4). Each row has eight pigeonholes, each being able to hold a zero (0) or one (1). In electronic terms, each pigeonhole



Address	Memory
⋮	⋮
5100	01010101
5101	01001000
5182	10001001
5103	11100101
5104	00000011
5105	00010101
5106	10110010
5107	00000011
5108	00100000
5109	00000000
5110	10001011
5111	00000101
5112	10110000
5113	00000011
5114	00011011
⋮	⋮

Fig. 1.4 The memory is organized as a stack of rows such that each row has an associated address. [Reproduced with permission from: G. Üçoluk, S. Kalkan, Introduction to Programming Concepts with Case Studies in Python, Springer, 2012 [1]]

is capable of storing a voltage (can you guess what type of an electronic component a pigeonhole is?). Each such row is named to be of the size *byte*, i.e., a byte means 8 bits.

Each byte of the memory has a unique address. When the address input (also called address bus—Fig. 1.3) 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 R/W wire being set to Write (`1`) or Read (`0`), the action that is carried out on the memory byte differs:

- **R/W wire is set to WRITE (1)**:
  The binary content on the data bus is copied into the 8-bit location whose address is provided on the address bus, the former content is *overwritten*.
- **R/W wire is set to READ (0)**:
  The data bus is set to a copy of the content of the 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 lives in the memory happily, until the power is turned off.

The memory is also referred to 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 today's RAMs, this access time is around 50 ns.
- 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. 1.3, the CPU does. How this is done by the CPU side will become clear in the next section.

1.2.1.4 The Central Processing Unit (CPU)

The central processing unit can be considered as the "brain" of a computer. Its sole responsibility is to execute machine instructions, which are binary sequences describing an operation and the operands of the operation. The CPU achieves its responsibility with the help of the following units:

- **Control Unit (CU)**, which is responsible for fetching instructions from the memory, interpreting ("decoding"), and executing them. After finishing the execution of an instruction, 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 a size of 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.

1.2.1.5 The Fetch–Decode–Execute Cycle

The CPU is, in fact, a *state machine*, a machine that has a representation of its current *state* (Fig. 1.5). 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. 1.6) where each step of the cycle is based on its state. The *control unit* is responsible for the functioning of the cycle.

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—see Fig. 1.3).

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:

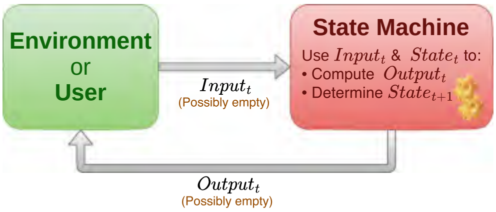


Fig. 1.5 The CPU is a state machine that uses the current input and the current state to compute an output and update its state

FETCH
DECODE
EXECUTE

Fig. 1.6 The CPU constantly follows the fetch–decode–execute cycle while the computer is running a program

- *Data manipulation instructions*: Arithmetic/logic operations on/among registers.
- *Data transfer instructions*: Memory-to-register, register-to-memory, register-to-register transfers.
- *Execution flow control instructions*: Instructions that stop execution, jump to a different part of the memory for the 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-bit instruction that has the first 4 bits 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 4 bits then contain the parameters of the instruction, which are the data to be operated on, the address in the memory or the codes (denoting names) of the registers, etc.

Let us assume that this 8-bit example instruction (with 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 affected. When the instruction completes, the PC is updated to the address of the next instruction. However, if the instruction was for changing the flow of execution (i.e., jumping to a different memory location), then 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 to compute 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[1] of the CPU are 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 important point to note here is the mechanism of the transitions of the CPU 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 observed when each phase of the fetch–decode–execute cycle is completed in one-and-only-one clock cycle. On modern CPUs, this is true for the addition instruction, for example. But there are instructions (like floating-point division) that take about 40 clock cycles.

What is the 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 5 GHz (that is 5×10^9 pulses per second).

[1] Circuits or devices constructed using transistors, microchips, and other hardware components.

Since (period) = 1/(frequency), a clock cycle is 200 picoseconds for this processor. 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. These are out of the scope of this book.

1.2.1.6 The Stored Program Concept

In order for the CPU to compute something, the corresponding instructions for performing 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 in 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 of a program. This program multiplies two integer numbers in two different locations of 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 some data follow it or not, so we cannot say where the second instruction starts. However, the CPU which these instructions were written for, would know this, having it 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 names created by the programmer, in this case for 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, “`movl`” starts 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 of an *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. From now on, we will call the binary code program 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 assembly programming languages are insufficient for efficient and fast programming. They lack some high-level constructs and tools necessary for solving problems. 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.

1.2.1.7 Pros and Cons of the von Neumann 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 considerable control of the memory organization. So, s/he can optimize the memory usage to its full extent.

Disadvantages

- The sequential instruction processing nature makes parallel implementations difficult. Parallelization is achieved by rapidly transitioning between tasks, albeit in a sequential manner.
- 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”[2] which could not withstand the test of time due to critical disadvantages compared to the von Neumann Architecture.

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

1.2.2 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 considered 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 briefly.

All devices are electronically listening to the buses (the address and data buses) 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 any other device that listens to the buses 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 is not talking 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 two different devices from having the same address.

To communicate with the peripherals, the CPU can occasionally 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 sent in irregular intervals): You cannot know when there will be a keyboard button press so, in polling, you have to ask very frequently the keyboard device for the existence of any data.

To avoid this inefficiency of polling, another mechanism based on interrupts is built into the CPU. The *interrupt* mechanism relies on 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. This way, the device gets a chance to send/receive data to/from the CPU.

1.3 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.

1.3.1 Start-Up Process

We have learned that at the core of a computer is the von Neumann Architecture. Now, we will see how a machine code finds its way into the memory, and gets settled there, so that the CPU starts executing it.

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 the memory that, 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* (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 16 KB,[3] nowadays it is about 1000 times larger, 16 MB.

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 disk 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.

BIOS nowadays is modernized into the Unified Extensible Firmware Interface (UEFI) which introduces more capabilities such as faster hardware check, better user interface, and more security. With UEFI, MBR is replaced by a modernized version, called GPT (globally unique identifier partition table) to allow larger disks, larger partitions (drives), and better recovery options.

1.3.2 The Operating System (OS)

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**: In modern computers, there is more than one machine code program loaded into the memory. Some programs are initiated by the user (like an Internet browser, document editor, office programs, 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 finishes execution, 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 JPEG image file) then the process makes this demand to the memory manager, which grants it or denies it.
- **Process (Time) Management**: As stated above, the memory of a modern computer 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?

[3] A byte (B) is 8 bits. 1 kilobyte (KB) is $2^{10} = 1024$ bytes. 1 megabyte (MB) is $2^{10} = 1024$ KB. 1 gigabyte (GB) is $2^{10} = 1024$ MB.

When a process gets the turn, the following actions are executed:

1. The current state of the CPU (the contents of all registers) is saved to some secure position in the memory, in association with the last executing process.
2. From that secure position, the last saved state information of the process which is going to take the turn is found and the CPU is set to that state.
3. Then the CPU, for a period of time, executes that process.
4. At the end of that period, the CPU switches over to the time manager. The time manager makes a decision about which process gets the next turn, and Steps 1–4 are repeated.

Determining the next process to execute is a complex task and an active research area. The time manager collects some statistics about each individual process and its system resource utilization. Moreover, 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. This is a highly 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 functionalities. This software is usually provided by the device manufacturer and is loaded into the operating system as a part of the device installation process. These plug-ins are named *device drivers*.
  An operating system performs device communication by means of these drivers. It does the following activities for device management:

  - Keeps track 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, and audio-visual data are handled. Handling data also includes *storing* and *retrieving* it on some external recording device. Examples of 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 metadata (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, and 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, layout policies,* and some other aspects are concerned. Files are most often managed in a hierarchy. This is achieved by a concept of *directory* or *folder*. On the surface (as far as the user sees them), a file system usually consists of files separated into directories where directories can contain files or other directories.
  The file manager is responsible for the creation and 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 that the system holds, 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 security. The second item especially 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 holds the processes, and the processes should not interfere with each other unless specifically intended. In fact, 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 into 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 harm. Sometimes it is the case that bugs in 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, and 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 past, 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 conducted via moving and clicking a pointing device which is another icon (usually a movable arrow) on the screen. The Xerox Alto[4] introduced in 1973 was the first computer that had a GUI. The first personal computer with a GUI was Apple Lisa,[5] introduced in 1983 with a price of $10,000. Almost 3 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 of a type-able window, called the terminal, which is still very much favored among programming professionals because it provides more control for the OS.

1.4 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 of 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.

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

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

1.5 Further Reading

- More on the CPU and the memory:
  - CPU and its history: https://en.wikipedia.org/wiki/Central_processing_unit [5]
  - Instruction sets: https://en.wikipedia.org/wiki/Instruction_set_architecture [9]
  - Bus: https://en.wikipedia.org/wiki/Bus_(computing) [4]
  - Memory: https://en.wikipedia.org/wiki/Random-access_memory [13]
- Computer architectures:
  - von Neumann Architecture: http://en.wikipedia.org/wiki/Von_Neumann_architecture [14]
  - Harvard architecture: http://en.wikipedia.org/wiki/Harvard_architecture [7]
- Alternatives to digital computation:
  - Analog computer: https://en.wikipedia.org/wiki/Analog_computer [2]
  - Quantum computer: http://en.wikipedia.org/wiki/Quantum_computer [12]
  - Optical computer: https://en.wikipedia.org/wiki/Optical_computing [11]
  - Chemical computer: http://en.wikipedia.org/wiki/Chemical_computer [6]
- Running of a computer:
  - Booting a computer: https://en.wikipedia.org/wiki/Booting [3]
  - Operating systems: https://en.wikipedia.org/wiki/Operating_system [10]
  - History of operating systems: https://en.wikipedia.org/wiki/History_of_operating_systems [8].

1.6 Exercises

1. To gain more insight, play around with the von Neumann machine simulator at http://vnsimulator.altervista.org
2. Can you think of any shortcomings of the von Neumann Architecture? If you had the chance, what would you fix about the von Neumann Architecture?
3. Make a list of similarities and dissimilarities between the biological brain (with the context provided in Sect. 1.1) and the von Neumann Architecture.
4. Find the following information for your desktop/laptop using a search engine or the website of your machine's manufacturer:

   a. Memory (RAM) size.
   b. CPU type and clock frequency.
   c. Data bus size.
   d. Address bus size.
   e. Size of general-purpose registers in the CPU.
   f. Hard disk or SSD size and random access time.

Opcode	Affected address
0001	0100

5. Assume that we have a CPU that can execute instructions with the format and size given above.

   a. What is the number of different instructions that this CPU can decode?
   b. What is the maximum number of rows in the memory that can be addressed by this CPU?

6. Using a search engine, discover how a zero or one can be represented by a memory device.
7. Using a search engine, discover the different types of memory:

   a. Volatile versus non-volatile memory.
   b. Dynamic versus static memory.
   c. Read-only memory.
   d. Sequential memory.

8. Using a search engine, learn about Moore's law.
9. Using a search engine, learn about (i) why Moore's law does not hold anymore and (ii) what potential solutions are being developed to address this issue.
10. Using a search engine, learn about the von Neumann bottleneck.

References

[1] G. Üçoluk, S. Kalkan, *Introduction to Programming Concepts with Case Studies in Python* (Springer Science & Business Media, 2012)
[2] Wikipedia contributors, "Analog computer," Wikipedia, The Free Encyclopedia (2023). https://en.wikipedia.org/w/index.php?title=Analog_computer&oldid=1171254284. Accessed 28 Aug 2023
[3] Wikipedia contributors, "Booting," Wikipedia, The Free Encyclopedia (2023). https://en.wikipedia.org/w/index.php?title=Booting&oldid=1171047111. Accessed 28 Aug 2023
[4] Wikipedia contributors, "Bus (computing)," Wikipedia, The Free Encyclopedia (2023). https://en.wikipedia.org/w/index.php?title=Bus_(computing)&oldid=1172515590. Accessed 28 Aug 2023
[5] Wikipedia contributors, "Central processing unit," Wikipedia, The Free Encyclopedia (2023). https://en.wikipedia.org/w/index.php?title=Central_processing_unit&oldid=1172444772. Accessed 28 Aug 2023
[6] Wikipedia contributors, "Chemical computer," Wikipedia, The Free Encyclopedia (2023). https://en.wikipedia.org/w/index.php?title=Chemical_computer&oldid=1135445564. Accessed 28 Aug 2023
[7] Wikipedia contributors, "Harvard architecture," Wikipedia, The Free Encyclopedia (2023). https://en.wikipedia.org/w/index.php?title=Harvard_architecture&oldid=1171100943. Accessed 28 Aug 2023
[8] Wikipedia contributors, "History of operating systems," Wikipedia, The Free Encyclopedia (2023). https://en.wikipedia.org/w/index.php?title=History_of_operating_systems&oldid=1165693013. Accessed 28 Aug 2023
[9] Wikipedia contributors, "Instruction set architecture," Wikipedia, The Free Encyclopedia (2023). https://en.wikipedia.org/w/index.php?title=Instruction_set_architecture&oldid=1170025514. Accessed 28 Aug 2023
[10] Wikipedia contributors, "Operating system," Wikipedia, The Free Encyclopedia (2023). https://en.wikipedia.org/w/index.php?title=Operating_system&oldid=1170338346. Accessed 28 Aug 2023
[11] Wikipedia contributors, "Optical computing," Wikipedia, The Free Encyclopedia (2023). https://en.wikipedia.org/w/index.php?title=Optical_computing&oldid=1171140708. Accessed 28 Aug 2023
[12] Wikipedia contributors, "Quantum computing," Wikipedia, The Free Encyclopedia (2023). https://en.wikipedia.org/w/index.php?title=Quantum_computing&oldid=1172606301. Accessed 28 Aug 2023
[13] Wikipedia contributors, "Random-access memory," Wikipedia, The Free Encyclopedia (2023). https://en.wikipedia.org/w/index.php?title=Random-access_memory&oldid=1172248814. Accessed 28 Aug 2023
[14] Wikipedia contributors, "Von neumann architecture," Wikipedia, The Free Encyclopedia (2023). https://en.wikipedia.org/w/index.php?title=Von_Neumann_architecture&oldid=1163023411. Accessed 28 Aug 2023

2 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 cryptic binary machine code instructions we introduced in the previous chapter: We can use human-readable instructions and languages to make programming easier.

2.1 How Do We Solve Problems with Programs?

The von Neumann machine, on which modern computer design is based, makes a clear distinction between instruction and data (do not get confused by the machine code that holds 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). This bifurcation is not merely a technical detail; it significantly influences how computational approaches are devised for problem-solving (Fig. 2.1). When tasked with solving a world problem, a programmer first pinpoints the specific data that needs to be manipulated. Following this identification, they must develop a structured sequence of operations—an algorithm—that systematically transforms this data to render a solution. Thus, the process of programming is deeply rooted in the act of classifying the information (data) and subsequently orchestrating a series of logical steps (instructions) that work in concert to address the given challenge.

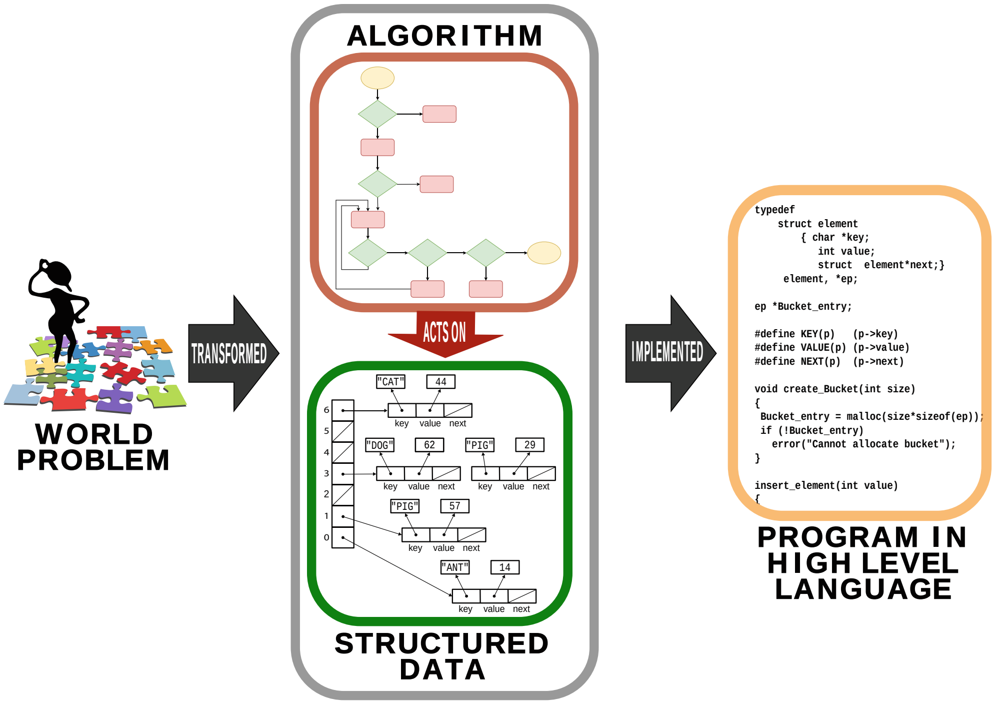
2.2 Algorithm

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

Algorithms can be thought of as recipes for cooking. This analogy is intuitive since we would define a recipe as a step-by-step procedure for cooking something: Each step describes a little action (cutting, slicing, stirring, etc.) that brings us closer to the outcome: The dish.

This is true for algorithms as well: At each step, we make small progress towards the desired 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.

S. Kalkan et al., *Programming with Python for Engineers*,
https://doi.org/10.1007/978-3-031-57148-0_2



```
typedef
    struct element
        { char *key;
           int value;
           struct  element*next;}
     element, *ep;

ep *Bucket_entry;

#define KEY(p)   (p->key)
#define VALUE(p) (p->value)
#define NEXT(p)  (p->next)

void create_Bucket(int size)
{
 Bucket_entry = malloc(size*sizeof(ep));
 if (!Bucket_entry)
   error("Cannot allocate bucket");
}

insert_element(int value)
{
```

Fig. 2.1 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. [Reproduced with permission from: G. Üçoluk, S. Kalkan, Introduction to Programming Concepts with Case Studies in Python, Springer, 2012 [1]]

2.2.1 The Origin of the Word "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 wrote a book on algebra titled "The Compendious Book on Calculation by Completion and Balancing" during 813–833 which presented the first systematic solution of linear and quadratic equations. His contributions laid the foundations of algebra, which stems from his method of "al-jabr" (meaning "completion" or "rejoining"). The reason the word 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; etc.")—i.e., using what we call today as algorithms.

2.2.2 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. In other words, we can design and study algorithms without using computers, e.g., with just a pen and paper. A program, on the other hand, is just an *implementation* (realization) of an algorithm in a programming language. In other words, algorithms are *designs* and programs are the written realizations of these designs in programming languages.

2.2.3 How to Write Algorithms?

As we have discussed above, before programming our solution, we first need to design it. While designing our solution as 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. These descriptions are not as specific or restricted as the descriptions in a programming language but they are not as free as the language we use for communicating with other humans: Since pseudo-codes are meant for describing actions for computers, pseudo-codes should be composed of precise and feasible steps and avoid ambiguous descriptions.
   Here is an example pseudo-code:

Algorithm 2.1 Average Calculation of Numbers Provided by the User

```
Input: None
Output: $Result$ — The average of the provided numbers
Get how many numbers will be provided and store that in a variable $N$
Create a variable named $Result$ with initial value 0
Execute the following step $N$ times:
    Get the next number and add it to $Result$
Divide $Result$ by $N$ to obtain the average
return $Result$
```

2. **Flowcharts**. As an alternative to pseudo-codes, we can use flowcharts while designing algorithms. Flowcharts are diagrams composed of small computational elements that describe the steps of algorithms. The example in Fig. 2.2 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.

2.2.4 How to Compare Algorithms?

If two algorithms find the same solution, are they of the same quality? To get a better feeling, let us go over an example. Do you recall “Guess My Number”, a game we used to play when we were in primary school?

The rules are 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 with one of the 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 gamers:

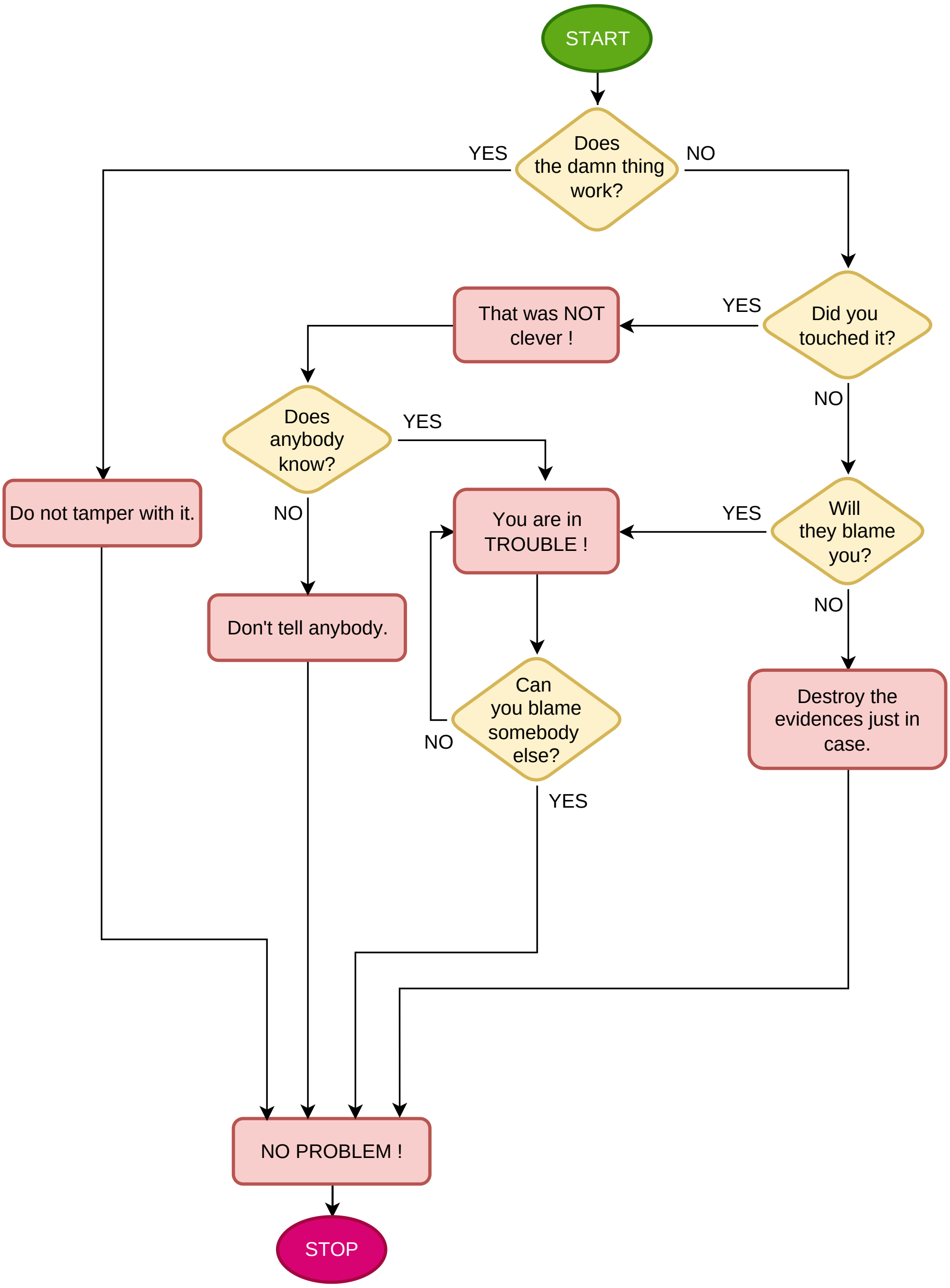


Fig. 2.2 Flowcharts describe the flow of execution by using basic geometric symbols and arrows. The program’s start or end is depicted with an oval. A rectangular box denotes a simple action or status. Decision-making is represented by a diamond and a parallelogram represents the input/output process

1. *Random guessers:* 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.:

   - Guesser: *Is it 1000?* Answer: *LESSER*.
   - Guesser: *Is it 999?* Answer: *LESSER*.
   - Guesser: *Is it 998?* Answer: *LESSER*.
   - ...

   Certainly, at some point, such players do get a HIT. There may be players who decrease (or increase) the number by two or three as well. With a first GREATER reply, they start to increment (decrement) 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.:

   - Guesser: *Is it 500?* Answer: *LESSER*.
   - Guesser: *Is it 250?* Answer: *LESSER*.
   - Guesser: *Is it 125?* Answer: *GREATER*.
   - Guesser: *is it 187? (equal to (125+250)/2)* Answer: *GREATER*.
   - ...

All three categories actually adopt different algorithms, which will find the answer at the end. However, as you may have realized 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 "betterness"? The answer is yes.

Let us look at this for the 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, $1000/2$, $1000/2^2$, ... Therefore, 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 (why?).

We call such an algorithm of "order of log(n)" or, more technically, $\mathcal{O}(\log n)$. In our case 1000 determines the "size" of the problem. This is symbolized with n. $\mathcal{O}(\log(n))$ 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, a 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 $\mathcal{O}(n)$.

Certainly the algorithm that has $\mathcal{O}(\log(n))$ is better than the one with $\mathcal{O}(n)$, as illustrated in Fig. 2.3. In other words, an $\mathcal{O}(\log(n))$ algorithm requires less number of steps and will run faster than the one with $\mathcal{O}(n)$ complexity.

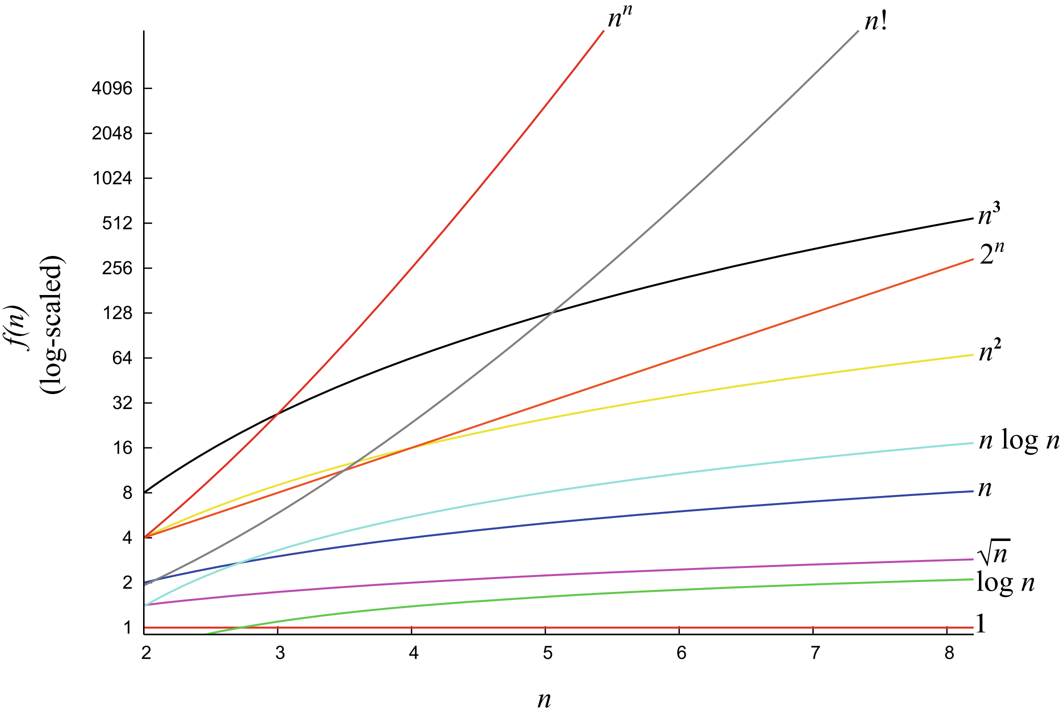


Fig. 2.3 A plot of various complexities

2.3 Data Representation

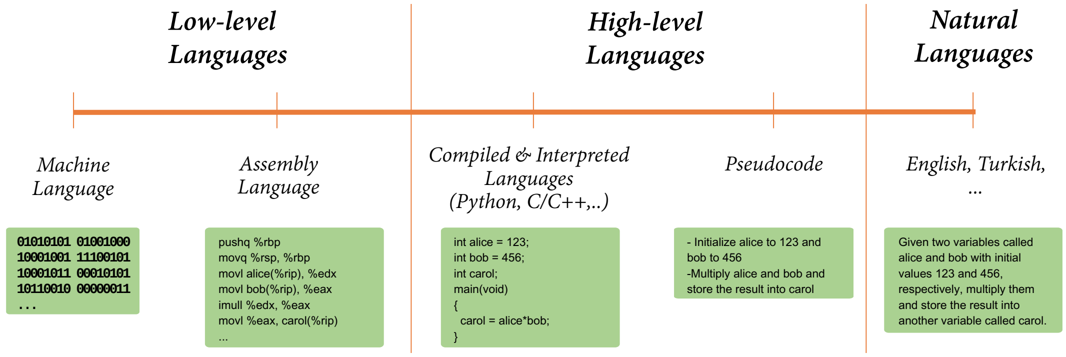
The other crucial component of our solutions to world problems (as shown in Fig. 2.1) is the data representation, which deals with putting together information regarding the problem in a form that is most suitable for our algorithm.

For example, if our problem is the calculation of the average of the 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. We will discuss these in Chap. 3.

2.4 The World of Programming Languages

Since the advent of computers, many programming languages have been developed with different design choices and levels of complexity. In fact, there are about 700 programming languages—see, e.g., [2] for an up-to-date list—that offer different abstraction levels (hiding the low-level details from the programmer) and computational benefits (e.g., providing built-in rule-search engine).

In this section, we will give a flavor of programming languages in terms of abstraction levels (low level versus high level—see Fig. 2.4) as well as the computational benefits they provide.



```
01010101 01001000
10001001 11100101
10001011 00010101
10110010 00000011
...
```

```
pushq %rbp
movq %rsp, %rbp
movl alice(%rip), %edx
movl bob(%rip), %eax
imull %edx, %eax
movl %eax, carol(%rip)
...
```

```
int alice = 123;
int bob = 456;
int carol;
main(void)
{
  carol = alice*bob;
}
```

- Initialize alice to 123 and bob to 456
-Multiply alice and bob and store the result into carol

Given two variables called alice and bob with initial values 123 and 456, respectively, multiply them and store the result into another variable called carol.

Fig. 2.4 The spectrum of programming languages, ranging from low-level languages to high-level languages and natural languages

2.4.1 Low-Level Languages

In the previous chapter, we introduced the concept of a 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 the 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
       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. For example, an instruction like `ADDI` represents integer addition.
- Numerical constants can be written down in human-readable, base-10 format, the assembler does the conversion to the internal format.
- Names can be used to refer to 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.

2.4.2 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 (as well as 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 with the syntax. The syntax is based on keywords from a human language (for historical reasons, English). Using human-readable keywords eases 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 name we have chosen for a variable that will hold the input value.

High-level languages implement many concepts that are not present at the machine code programming level. The 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 *(values or 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)*.

2.4.3 Implementing with a High-Level Language: Interpreter Versus 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. 2.5). The outcome is a machine code program that can be run at any time (by asking the OS to do so) and does the job described in the high-level language program.
   Conceptually, this process is correct, but actually, there is an additional step involved. The compiler produces an almost complete machine code with some holes in it. These holes are about the parts of the code which are 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.
2. **Interpretive Approach**. In this approach, a machine code program, named as *interpreter*, inputs and processes the high-level program line by line (Fig. 2.6). After taking a line as input, the 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 that 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.

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 computationally intensive big projects or time-tight tasks.

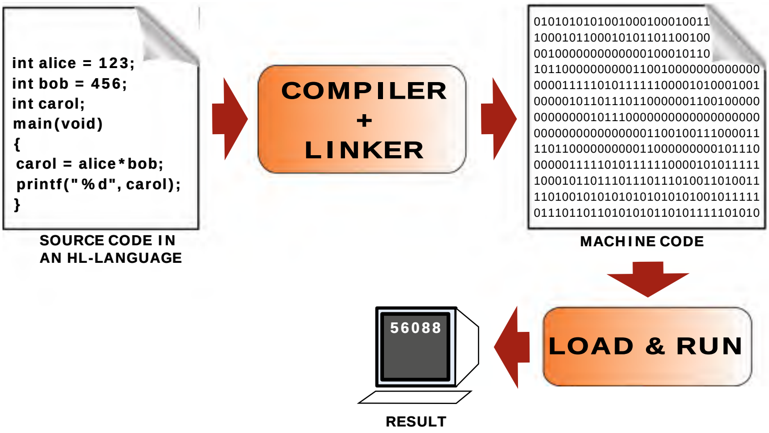


Fig. 2.5 A program code in a high-level language is first translated into machine-understandable binary code (machine code) which is then loaded and executed on the machine to obtain the result. [With permission from: G. Üçoluk, S. Kalkan, Introduction to Programming Concepts with Case Studies in Python, Springer, 2012 [1]]

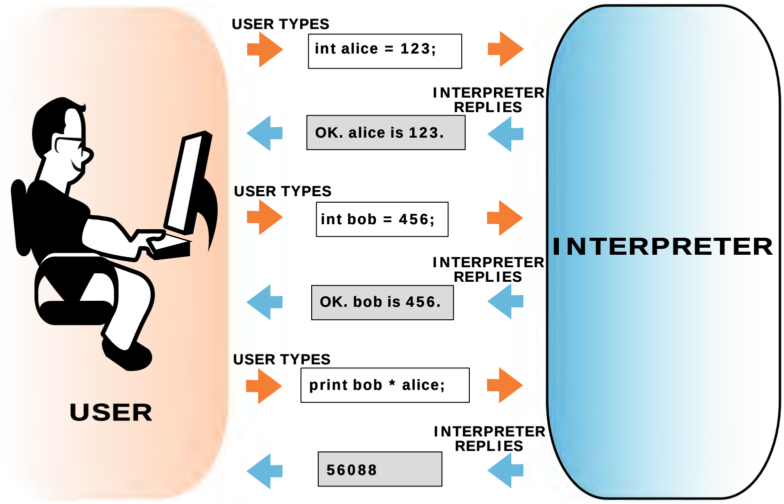


Fig. 2.6 Interpreted languages (e.g., Python) come with interpreters that process and evaluate each action (statement) from the user on the run and return an answer. [With permission from: G. Üçoluk, S. Kalkan, Introduction to Programming Concepts with Case Studies in Python, Springer, 2012 [1]]

2.4.4 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 that 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 programming languages, different strategies or world views about programming have also developed. These worldviews are reflected in the programming languages.

For example, one worldview 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 worldviews is coined as *programming paradigms*. In a nutshell:

programming paradigm

is a worldview, a philosophy, in software development that governs how solutions are formulated and expressed in programs through specific patterns and methodologies.

Below is a list of some major programming paradigms:

- **Imperative**: This is a paradigm where programming statements and their compositions are very aligned with the fetch–decode–execute cycle of the machine. Therefore, program segments can be directly mapped to the machine code segments.
- **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 in parallel. This paradigm is very suitable for problems that can be solved by a divide-and-conquer strategy.
- **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) are 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.

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 into his or her particular program. As it is with many "worldviews" among humans, in the field of programming, fanaticism exists too. You can meet individuals that do only functional programming or object-oriented programming. We may consider them outliers.

Python, the subject language of this book, supports the imperative, functional, and object-oriented programming paradigms. It also provides some functionality in other paradigms with the help of some modules.

2.5 Introducing Python

After having provided some background on the world of programming, let us introduce Python: Although it is widely known to be a relatively recent programming language, Python's design, by Guido van Rossum [3], dates back to 1980s, as a successor of the ABC programming language. The first version was released in 1991 and when the second version was 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 both easy to understand and code in, while also being powerful, functional, practical, and fun. This has deep roots in its design philosophy (a.k.a. the Zen of Python [4]):

> "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 a *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 Chap. 7. Thanks to its wide acceptance, especially in 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" [5]. While van Possum was developing Python, he read the scripts of Monty Python's "Flying Circus" and thought "python" was "short, unique and mysterious" [6] for the new language. To make Python more fun to learn, earlier releases heavily used phrases from Monty Python in programming code examples.

With version 3.10 being released in 2023 as the latest version, Python is one of the most popular programming languages in a wide spectrum of disciplines and domains. With 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 what 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 receive and process our computational demands, e.g.:

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

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

2.6 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 versus compilative approach to programming.
- Programming paradigms.

2.7 Further Reading

- The World of Programming chapter of G. Üçoluk, S. Kalkan, Introduction to Programming Concepts with Case Studies in Python, Springer, 2012 [1].
- Programming Languages:
  - For a list of programming languages: https://en.wikipedia.org/wiki/List_of_programming_languages [2].
  - For a comparison of programming languages: http://en.wikipedia.org/wiki/Comparison_of_programming_languages [7].
  - For more details: Daniel P. Friedman, Mitchell Wand, Christopher Thomas Haynes: Essentials of Programming Languages, The MIT Press 2001 [8].
- Programming Paradigms:
  - Introduction: http://en.wikipedia.org/wiki/Programming_paradigm [9].
  - 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 [10].
  - Comparison between Paradigms: http://en.wikipedia.org/wiki/Comparison_of_programming_paradigms [11].

2.8 Exercises

1. Draw the flowchart for the following algorithm:

Algorithm 2.2 Average of all Elements of a List of Numbers

```
Input: $Numbers$ — A list of $N$ numbers
Output: The average of all elements in $Numbers$
Create a variable named $Sum$ with initial value 0
For each number $i$ in $Numbers$, execute the following lines:
    If $i > 0$ then Add $i$ to $Sum$
    If $i < 0$ then Add the square of $i$ to $Sum$
Divide $Sum$ by $N$
return $Sum$
```

2. Devise an algorithm to find the maximum of a list of three numbers using simple greater than or less than comparisons. Represent your algorithm with a pseudo-code and a flowchart.
3. Devise an algorithm to find the median of a list of three numbers using simple greater than or less than comparisons. Represent your algorithm with a pseudo-code and a flowchart.

4. Given a 2D rectangle (with corners at (x_1, y_1), (x_2, y_2), (x_3, y_3), (x_4, y_4)) horizontal to the x-axis, devise an algorithm that checks whether a point (x', y') is in the rectangle or not. Represent your algorithm with a pseudo-code and a flowchart.
5. Given a 2D triangle (with corners at (x_1, y_1), (x_2, y_2), (x_3, y_3)), devise an algorithm that checks whether a point (x', y') is in the triangle or not. Represent your algorithm with a pseudo-code and a flowchart.
6. What is the complexity of Algorithm 2.1 (in Sect. 2.2.3)?
7. What is the complexity of the following algorithm?

Algorithm 2.3 Standard Deviation of all Elements of a List of Numbers

```
Input: $Numbers$ — A list of $N$ numbers
Output: $Std$ — The standard deviation of all elements of $Numbers$
Get a list of $N$ numbers in a variable named $Numbers$
Create a variable named $Mean$ with initial value 0
For each number $i$ in $Numbers$, execute the following line:
    Add $i$ to $Mean$
Divide $Mean$ by $N$
Initialize a variable named $Std$ with value 0
For each number $i$ in $Numbers$, execute the following line:
    Add the square of $(i - Mean)$ to $Std$
Divide $Std$ by $N$ and take its square root
return $Std$
```

8. 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	$\mathcal{O}(\log n)$	$\mathcal{O}(n)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^3)$	$\mathcal{O}(2^n)$	$\mathcal{O}(n^n)$
$n = 10^2$							
$n = 10^3$							
$n = 10^4$							
$n = 10^5$							

9. Assume that we have a parser that 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 not, why not?
10. For each situation below, try to identify which paradigm is more suitable compared to the others:

    a. Writing a program that should take an image as input, an RGB color value, and find all pixels in the image that match the given color.
    b. Writing a program to prove theorems.
    c. Writing an autopilot program flying an airplane.
    d. Writing a document editing program as an alternative to Microsoft Word.
    e. Writing a first-person shooter game.

References

[1] G. Üçoluk, S. Kalkan, *Introduction to Programming Concepts with Case Studies in Python* (Springer Science & Business Media, 2012)

[2] Wikipedia contributors, "List of programming languages by type," Wikipedia, The Free Encyclopedia (2023). https://en.wikipedia.org/w/index.php?title=List_of_programming_languages_by_type&oldid=1173532133. Accessed 5 Sept 2023

[3] Wikipedia contributors, "Guido van rossum," Wikipedia, The Free Encyclopedia (2023). https://en.wikipedia.org/w/index.php?title=Guido_van_Rossum&oldid=1152945687. Accessed 5 Sept 2023

[4] Wikipedia contributors, "Zen of python," Wikipedia, The Free Encyclopedia (2023). https://en.wikipedia.org/w/index.php?title=Zen_of_Python&oldid=1145967392. Accessed 5 Sept 2023

[5] Wikipedia contributors, "Monty python," Wikipedia, The Free Encyclopedia (2023). https://en.wikipedia.org/w/index.php?title=Monty_Python&oldid=1172419961. Accessed 5 Sept 2023

[6] Python Documentation, "Why is it called python?" (2023). https://docs.python.org/2/faq/general.html#why-is-it-called-python. Accessed 5 Sept 2023

[7] Wikipedia contributors, "Comparison of programming languages," Wikipedia, The Free Encyclopedia (2023). https://en.wikipedia.org/w/index.php?title=Comparison_of_programming_languages&oldid=1173931849. Accessed 5 Sept 2023

[8] D.P. Friedman, M. Wand, C.T. Haynes, *Essentials of Programming Languages* (MIT Press, 2001)

[9] Wikipedia contributors, "Programming paradigm," Wikipedia, The Free Encyclopedia (2023). https://en.wikipedia.org/w/index.php?title=Programming_paradigm&oldid=1167849453. Accessed 5 Sept 2023

[10] P. Van Roy et al., Programming paradigms for dummies: what every programmer should know. New comput. Parad. Comput. Music **104**, 616–621 (2009)

[11] Wikipedia contributors, "Comparison of programming paradigms," Wikipedia, The Free Encyclopedia (2023). https://en.wikipedia.org/w/index.php?title=Comparison_of_programming_paradigms&oldid=1170935398. Accessed 5 Sept 2023

3 Representation of Data

As we discussed in detail in Chap. 2, in order to solve a world problem using a computer, we have to determine what data is involved in this problem and how we can process this data (i.e., devise an algorithm) to obtain the solution—let us recall this with Fig. 3.1 from Chap. 2.

At this stage, you may wonder 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: Suppose we have a table containing multiple rows, where each row represents an angle value and its corresponding cosine value expressed as decimal numbers. How would you structure the storage of these rows in memory? Two straightforward options are

(a) Row-by-row:

$$
\begin{array}{l}
Anglevalue_1 \\
Cosinevalue_1 \\
Anglevalue_2 \\
Cosinevalue_2 \\
\vdots \\
Anglevalue_n \\
Cosinevalue_n
\end{array}
$$

(b) Column-by-column:

$$
\begin{array}{l}
Anglevalue_1 \\
Anglevalue_2 \\
\vdots \\
Anglevalue_n \\
Cosinevalue_1 \\
Cosinevalue_2 \\
\vdots \\
Cosinevalue_n
\end{array}
$$

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

S. Kalkan et al., *Programming with Python for Engineers*,
https://doi.org/10.1007/978-3-031-57148-0_3

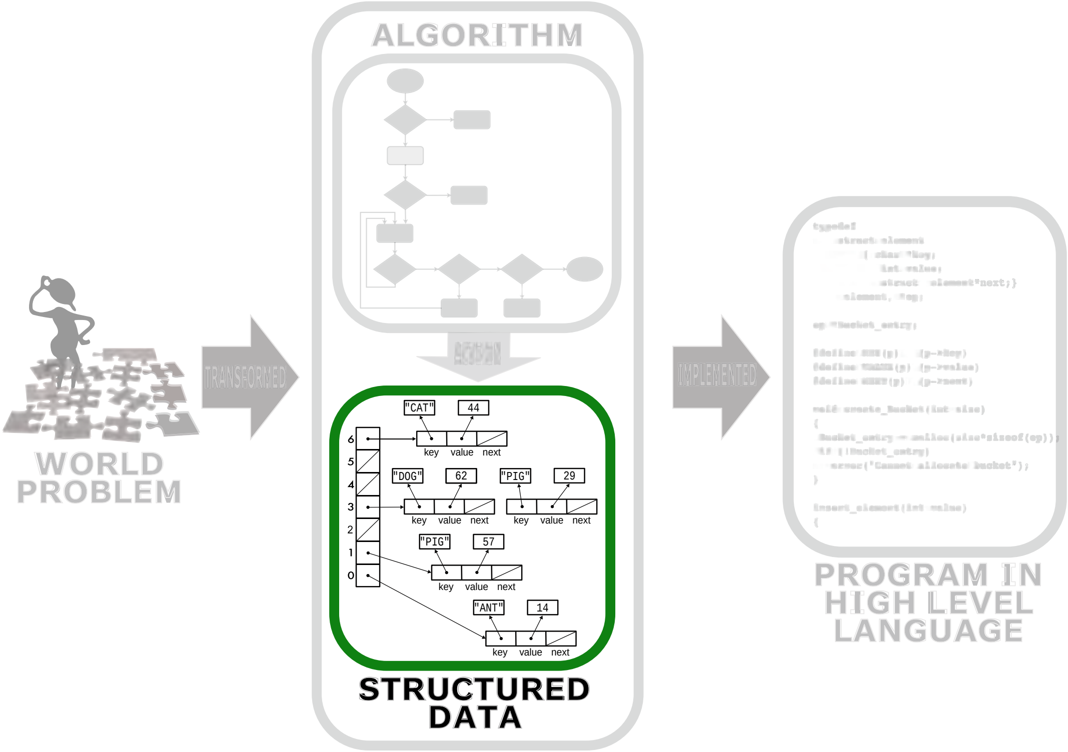


Fig. 3.1 Solving a world problem with a computer requires first designing how the data is going to be represented and specifying the steps which yield the solution when executed on the data. This design of the solution is then written (implemented) in a programming language to be executed as a program such that, when executed, the program outputs the solution for the world problem. [Reproduced with permission from: G. Üçoluk, S. Kalkan, Introduction to Programming Concepts with Case Studies in Python, Springer, 2012 [1]]

Although there are alternative methods to organize data, even for simple examples like the table mentioned above, they fall outside the scope of this book.

Now, we will look into the atomic structure of data representation: i.e., how integers, real (floating point) numbers, and symbols (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 symbolize this absence and presence using “0” and “1”. Consequently, any data intended for processing on this architecture needs to be converted into a representation consisting of “0”s and “1”s. It’s important to note that each “one” or “zero” in our representation corresponds to the presence or absence of a voltage at a specific location within the electronic circuitry.

3.1 Representing Integers

Mathematics already provides a solution for representing integers using binary notation, which involves *base-2* calculations 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

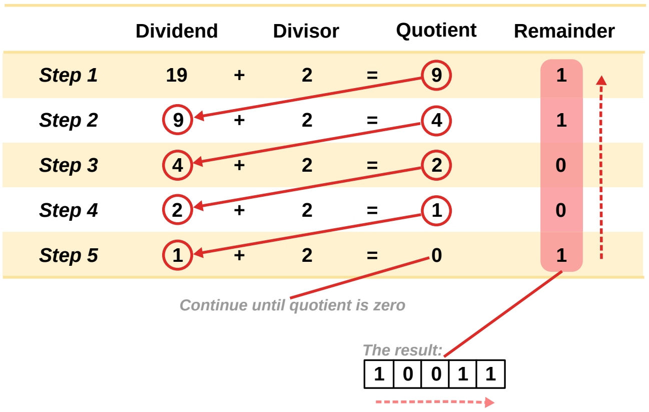
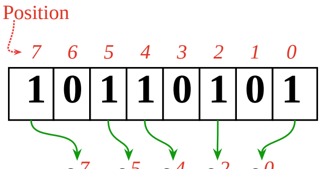


Fig. 3.2 The division method for converting a decimal number (e.g. 19) into binary

1. Counting starts from the right-most digit. A “1” in a position k means “additively include a value of 2^k”:

$Position$	n	$n-1$	...	2	1	0
$Meaning$	2^n	2^{n-1}	...	2^2	2^1	2^0

The following is an example that illustrates how to express the integer **181** in base-2:



$$2^7 + 2^5 + 2^4 + 2^2 + 2^0 =$$

$$128 + 32 + 16 + 4 + 1 = \mathbf{181}$$

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. 3.2. In 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 (as we did above).

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.

3.1.1 Sign-Magnitude Notation

Decimal arithmetic uses the minus sign (−) to represent negativity. However, the electronics of the von Neumann machinery require that the minus is also represented by either a “1” or “0”. We can reserve a bit, for example, the leftmost digit, for this purpose. If we do this, when the leftmost digit is a “1”, the rest of the digits encode 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*.

For example, in four bits, $(+2)_{10}$ and $(-2)_{10}$ can be represented as follows:

- $(0\ 010)_2 \Longrightarrow (+2)_{10}$
- $(1\ 010)_2 \Longrightarrow (-2)_{10}$

Although this notation is straightforward and intuitive, it 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*

A naive electronics implementation of addition may yield incorrect results (let us assume a 4-bit representation):

$$\begin{array}{lc|c} & \text{Sign-magnitude representation} & \text{Decimal value} \\ \hdashline & (0010)_2 & (+2)_{10} \\ + & (1101)_2 & (-3)_{10} \\ \hline & (1111)_2 & (-7)_{10} \end{array} \tag{3.1}$$

To be able to perform the addition correctly, 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 larger magnitude.
  2. Subtract the smaller magnitude from the larger one, and obtain the result.
  3. If the larger magnitude integer was negative, the result is negative (set the sign bit to "`1`"). Otherwise, set the sign bit in the result to "`0`".

This was only for the case of addition. A similar electronic circuitry is needed to subtract two integers.

This technique has the following drawbacks:

- It requires separate electronics for subtraction.
- It requires electronics for magnitude-based comparison and an algorithm implementation to set the sign bit.
- Electronically, it has to differentiate between addition and subtraction.

2. Representing the 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:

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

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

3.1.2 Two's Complement Notation

The aforementioned drawbacks of the sign-magnitude notation can be mitigated by carefully defining a mapping from positive and negative integers to binary numbers as follows:

- Positive integers are represented by their base-2 representation, with a leading "`0`".
- 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) such 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 that the result is the encoding of a negative integer.

There are two preferred alternatives to 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 (for a 32-bit computer, $n = 32$; for a 64-bit computer, it is 64) to represent numbers, then we can represent integer values in the range $[-2^{n-1}, 2^{n-1} - 1]$. Then, the two's complement representation of an integer (with magnitude, or absolute value, p) can be obtained as follows:

- If the integer is positive, simply convert it to base-2.
- If it is negative, then

  1. Convert p to base-2.
  2. Negate this base-2 representation by flipping all `1`s to `0`s and all `0`s to `1`s: `1` ↔ `0`.
  3. Add `1` to the result of the negation.

Here is a simple example for representing $(-2)_{10}$ in two's complement notation:

- Take the binary representation of the magnitude (2): $(0010)_2$.
- Flip (inverse) the bits: $(0010)_2 \Longrightarrow (1101)_2$.
- Add `1`: $(1101)_2 + (1)_2 = (1110)_2$.
- The result as the two's complement representation of $(-2)_{10}$ is $(1110)_2$.

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

Decimal value	Two's complement	Pos. or Neg.	Action	Binary result
0	`00000000`	–	Direct base-2 encoding	
1	`00000001`	*Positive*	Direct base-2 encoding	

−1	`11111111`	*Negative*	*magnitude*:
			inverse:
			+1:
				`00000001`
				`11111110`
				`11111111`

−2	`11111110`	*Negative*	*magnitude*:
			inverse:
			+1:
				`00000010`
				`11111101`
				`11111110`

18	`00010010`	*Positive*	Direct base-2 encoding	

−18	`11101110`	*Negative*	*magnitude*:
			inverse:
			+1:
				`00010010`
				`11101101`
				`11101110`

−47	`11010001`	*Negative*	*magnitude*:
			inverse:
			+1:
				`00101111`
				`11010000`
				`11010001`

−111	`10010001`	*Negative*	*magnitude*:
			inverse:
			+1:
				`01101111`
				`10010000`
				`10010001`

111	`01101111`	*Positive*	Direct base-2 encoding	
112	`01110000`	*Positive*	Direct base-2 encoding	

−128	`10000000`	*Negative*	*magnitude*:
			inverse:
			+1:
				`10000000`
				`01111111`
				`10000000`

−129	Not calculated	*Negative*	(Out of valid range)	
128	Not calculated	*Positive*	(Out of valid range)	

3.1.3 Why Does Two's Complement Work?

The two's complement notation 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 Chap. 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:

$$\begin{array}{|c|c|c|c|c|c|c|c|}\hline 1&1&1&1&1&1&1&1\\\hline\end{array}$$

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

$$\begin{array}{|c|c|c|c|c|c|c|c|}\hline 0&0&0&0&0&0&0&0\\\hline\end{array}$$

Because we lose the 9th bit since the CPU cannot fit that into the 8-bit representation:

$$\begin{array}{rc}
 & \begin{array}{|c|c|c|c|c|c|c|c|}\hline 1&1&1&1&1&1&1&1\\\hline\end{array}\\
+ & \begin{array}{|c|c|c|c|c|c|c|c|}\hline &&&&&&&1\\\hline\end{array}\\
\hline
\cancel{1} & \begin{array}{|c|c|c|c|c|c|c|c|}\hline 0&0&0&0&0&0&0&0\\\hline\end{array}
\end{array}$$

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

$$(2^8 - 1) + 1 \equiv 0. \qquad (\text{mod } 2^8) \tag{3.2}$$

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:

$$-(a) \equiv 0 - a \equiv 2^n - a. \qquad (\text{mod } 2^n) \tag{3.3}$$

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

$$2^n - a \equiv 2^n - 1 + 1 - a \equiv \underbrace{(2^n - 1 - a)}_{\text{Invert the bits of } a} + \underbrace{1}_{\text{Add 1 to the inverted bits}} \qquad (\text{mod } 2^n) \tag{3.4}$$

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

3.1.4 Benefits of the Two's Complement Notation

Let us revisit the limitations of the sign-magnitude representation (Sect. 3.1.1):

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

$$\begin{array}{rr|l} & (0010)_2 & (+2)_{10} \\ + & (1101)_2 & (-3)_{10} \\ \hline & (1111)_2 & (-1)_{10} \end{array} \tag{3.5}$$

- **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:

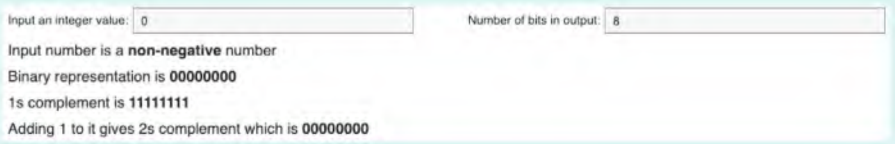
  * $(+0)_{10} = (0000)_2$
  * $(-0)_{10} = -(0000)_2 = (1111)_2 + (1)_2 = (0000)_2$, where we used two's complement to convert $-(0000)_2$ into $(1111)_2 + (1)_2$ by flipping the bits and adding 1.

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

Interactive Example 3.1

https://pp4e.online/c3d1

Please visit the Web page of the chapter for a practical session on the two's complement representation.



Input an integer value: 0

Number of bits in output: 8

Input number is a **non-negative** number

Binary representation is **00000000**

1s complement is **11111111**

Adding 1 to it gives 2s complement which is **00000000**

3.2 Representing Real Numbers

Floating point is the data type used to represent non-integer real numbers. On modern computers, all non-integer real numbers are represented using the floating-point data type. Since all integers are real numbers, you can represent integers 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 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 (point).

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

$$\begin{array}{cccccccccccc} 10^4 & 10^3 & 10^2 & 10^1 & 10^0 & . & 10^{-1} & 10^{-2} & 10^{-3} & 10^{-4} & 10^{-5} & 10^{-6} \\ \hline 1 & 2 & 2 & 6 & 3 & . & 9 & 2 & 1 & 8 & 7 & 5 \end{array} \tag{3.6}$$

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

$$\begin{array}{cccccccccccccccccccc} 2^{13} & 2^{12} & 2^{11} & 2^{10} & 2^9 & 2^8 & 2^7 & 2^6 & 2^5 & 2^4 & 2^3 & 2^2 & 2^1 & 2^0 & . & 2^{-1} & 2^{-2} & 2^{-3} & 2^{-4} & 2^{-5} & 2^{-6} \\ \hline 1 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & . & 1 & 1 & 1 & 0 & 1 & 1 \end{array} \tag{3.7}$$

How did we obtain the fractional part? By multiplying the fraction (0.921875) by two until we obtain zero for the fraction part. This is illustrated in Fig. 3.3 for a smaller fraction. Note that this method is in certain ways the reverse of what we did to convert the whole part to binary, in Fig. 3.2.

At this stage, it is worth mentioning the following: 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

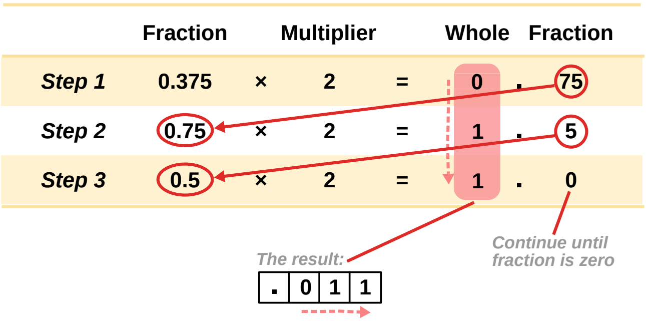


Fig. 3.3 The multiplication method for converting a fractional number into binary

period is moved. If it is moved to the left, it has a positive value, otherwise, it has a negative value. 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

$$\begin{array}{r|cccccccccccccccccccc} & 2^{0} & . & 2^{-1} & 2^{-2} & 2^{-3} & 2^{-4} & 2^{-5} & 2^{-6} & 2^{-7} & 2^{-8} & 2^{-9} & 2^{-10} & 2^{-11} & 2^{-12} & 2^{-13} & 2^{-14} & 2^{-15} & 2^{-16} & 2^{-17} & 2^{-18} & 2^{-19} \\ \hline 2^{+13}\times & 1 & . & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 1 \end{array} \tag{3.8}$$

Since there is always a leading "1" for all values (except for 0.0, which will be represented with an exception), there is no need to keep a record of the whole part, that is, the only remaining "1" value to the left of the period. Moreover, as the period is always there, 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

3.2.1 The IEEE 754 Representation

In the IEEE 754 format, the exponent of the multiplicative factor, namely, n (which can be negative) in 2^n, becomes a part of the representation—see Fig. 3.4. To be able to represent negative exponent values, 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 3.1. 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 feature 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 mantissa parts to compare numbers.

In our example, the factor was 2^{13}. Adding $k = 127$ yields $13 + 127 = 140$, which has an 8-bit representation of $(10001100)_2$. This will form the exponent portion (the 8 green bits in Fig. 3.4) of the IEEE 754 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 an IEEE 754 representation as follows:

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

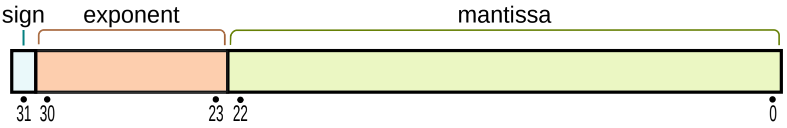


Fig. 3.4 The 32-bit IEEE 754 floating-point representation

Table 3.1 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

How is the real number "0.0" represented?

In the IEEE 754 standard, the floating-point number zero is represented with all zero bits in the positional range [1–31]. The zeroth bit, namely, the sign bit, can be either "0" or "1".

3.2.2 Information Loss in Floating-Point Representations

The condition for a floating-point number to be represented "exactly" by the IEEE 754 standard is for the number to be equal to:

$$f = \sum_{i=n}^{m} digit_i \times 2^i, \tag{3.9}$$

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 i value in the sum above), so that $|m - n| \leq 24$ is satisfied.

In fact, many rational numbers are not expressible in the form of the summation above: As an example, try deriving the representation of 4.1 and see if you can represent it in binary with a finite number of bits. Furthermore, we have infinitely many irrational numbers which do not have finite fractions: e.g., $\sqrt{2}$, $\sqrt{3}$, π, e. While using these numbers in computers, we can only work with their approximations. 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, the result you obtain has a high *imprecision*. This is also the case for the multiplication and division of relatively large 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. The result is a bad surprise! In contrast to our expectations, 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 $\sum_{i=1}^{N}(digit_i \cdot 2^{-i}), \quad digit_i \in \{0, 1\}$ 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, representing 0.9 is problematic. Number 0.9 cannot be expressed as a similar sum where N stays in the IEEE representation limit. N extents to $+\infty$ for 0.9. Hence, the $digit_n$ sequence has to be truncated before it can be 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. The bottom line is that many numbers cannot be represented exactly in the IEEE 754 format, and this causes a loss of precision.
- Things can even get worse. Consider the following interaction with Python:

  ```
  >>> 2000.0041 - 2000.0871
  -0.0829999999998563

  >>> 2.0041 - 2.0871
  -0.08299999999999974
  ```

  Actually, both results should have been -0.0830. In addition to having 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. This is because 2000 needs more bits to be represented compared to 2.
- The number π (pi) is a transcendental number. The fractional part never stops in any base. Let us give it a shot on the Python interpreter:

  ```
  >>> PI = 3.141592653589793238462643383279502884197169399375105820974944592307816406286
  >>> print(PI)
  3.141592653589793
  ```

  From this, we can deduce that the IEEE representation can only allocate a limited number of bits for the fractional part, specifically 15 decimal places. That seems 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 numerical error for the sine value, which is different from 0.0. However, when it comes to the cosine value, we were fortunate that the imprecision somewhat canceled out and gave us the correct result.

- We have been taught 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 result 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 that has nothing to do with the precision loss mentioned. Let us assume that you provide your program with the number π as 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 fifth digit after the decimal point in the number π which will propagate through your calculations. It can get worse (for example, if you subtract two very close numbers and use it in the denominator), but it can never get better. Your best chance is to get a correct result on the fourth digit after the decimal point. The following digits, as far as precision is concerned, are bogus.

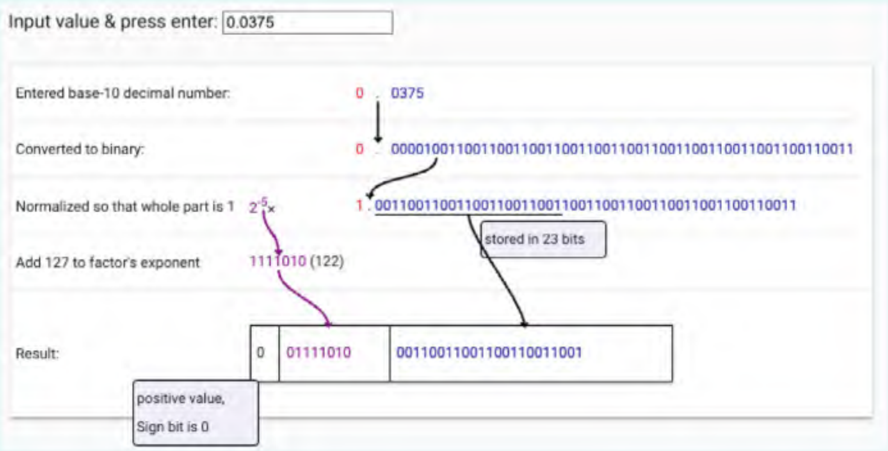
So, what can we do? Here are some rules of thumb for using floating-point numbers:

- It is in your best interest to refrain from using floating points. If 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 floating-point data types, use them.
- Use lower precision floating points only when you are short of memory.
- Subtracting two floating points close in value has a potential danger.
- If you add or subtract two numbers that 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.
- It is strongly recommended to use widely used and reputable libraries for floating-point calculations instead of developing your own algorithms from scratch.

Interactive Example 3.2

https://pp4e.online/c3d2

Please visit the Web page of the chapter for a practical session on the IEEE 754 floating point representation.



Input value & press enter: 0.0375

Entered base-10 decimal number: 0 . 0375

Converted to binary: 0 . 00001001100110011001100110011001100110011001100110011

Normalized so that whole part is 1 2^{-5}× 1 . 0011001100110011001100110011001100110011001100110011

stored in 23 bits

Add 127 to factor's exponent 1111010 (122)

Result: 0 | 01111010 | 0011001100110011001

positive value,
Sign bit is 0

3.3 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 large-integer representation silently when needed. You do not have to worry about it. Furthermore, keep 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 with decimal points 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 specified 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 signify complex numbers, but this is the most natural way considering your previous knowledge from high school.

3.4 Representing Truth Values (Booleans)

Boolean is another data type that has roots in the very structure of the CPU. The answers to all questions asked to the CPU are either *true* or *false*. The logic of a CPU is strictly based on the 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 concepts of binary `0` and `1`: 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`.

3.5 Representing Text

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

3.5.1 Characters

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

How can we represent graphemes and unprintables in binary? Graphemes are highly 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 have 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 (not 8-bit-long!) codes. Since the 7-bit-long code can also be interpreted as a number, for convenience, this number is also displayed in the ASCII table—see Table 3.2.

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

Table 3.2 The ASCII table. Dec: Decimal value. Bin: Binary representation. Char: Character being represented

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	ˆ	126	0111 1110	~
31	0001 1111	[US]	63	0011 1111	?	95	0101 1111	_	127	0111 1111	[DEL]

- 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	Uppercase letters
91–96	Punctuations
97–122	Lowercase letters
123–127	Punctuations

- There is no logic in the distribution of the punctuations.
- It is based on the English alphabet; characters from other languages are simply not there. Moreover, there is no mechanism for diacritics.
- Letters are ordered in the table, and uppercase letters come first (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 from its ASCII value.
- The table is only and only about 128 characters, neither more nor less. There is nothing like Turkish-ASCII or French-ASCII. The extensions, where the 8th bit is set, have nothing to do with the ASCII table.
- The older versions of Python (v1 and v2) used the ASCII character representations.

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 (with version 3), Java, Perl, TCL, Ada95, and C#, gaining wide popularity.

3.5.2 Strings

Strings are sequences of characters that are used to represent text data. Text data is as vital as numerical data in the world of programming, but we have a problem here. As we discussed above, numbers (integers and floating points) have a niche in the CPU. There are instructions designed for them: With instructions, we can store and retrieve them to/from the memory; we can perform arithmetical operations among them. Character data can be represented and processed as well because they are mapped to 1-byte integers through ASCII or alternative tables. But, when it comes to strings, the CPU does not have any facility for them.

How can we represent a string, i.e., a sequence of characters? The only reasonable way is to store the codes of all characters that make up a string in the memory in consecutive bytes. In other words, the string “Python rocks!” can be represented using the ASCII codes of the characters as follows:

80	121	116	104	111	110	32	114	111	99	107	115	33

Does this solve the problem of “representation”? Unfortunately, not. The trouble is determining how to know where the string ends, for which there are two possible solutions:

1. Store string length: In front of the string’s characters, store the length (the number of characters in the string) as an integer of a fixed number of bytes. This solution would represent our example string as follows, with the count 13 in the front:

13	80	121	116	104	111	110	32	114	111	99	107	115	33

2. Store an end mark: Store a special byte value (number zero in general), which is not used to represent any other character, at the end of the string characters. This solution would represent our example string as follows with the marker “0” at the end:

80	121	116	104	111	110	32	114	111	99	107	115	33	**0**

Different languages adopt different solutions between these options.

3.6 Containers

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

Also, note that whatever is placed in the memory has an “address” and the value of the address can also become a part of the organization. As far as strings are concerned, the usage of the “address concept” is 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 that include addresses of other data organizations. In other words, it is possible to jump from one group of data to some other data in the memory. These types of organizations are named *Data Structures* in Computer Science. In other words, string is a data structure. As far as Python is concerned, there are several other data structures. They are coined as *containers* in Python. In addition to strings, Python provides lists, tuples, sets, and dictionaries as containers. These containers will be covered in detail in the next chapter.

3.7 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 IEEE 754 standard for representing real numbers.
- Precision loss in representing floating-point numbers.
- Representing characters with the ASCII table.
- Representing truth values.

3.8 Further Reading

- Two’s Complement: http://en.wikipedia.org/wiki/Two%27s_complement [2].
- The Method of Complements: https://en.wikipedia.org/wiki/Method_of_complements [3].
- Excess-k Representation: https://en.wikipedia.org/wiki/Offset_binary [4].
- IEEE 754 Floating-Point Standard: http://en.wikipedia.org/wiki/IEEE_754-2008 [5].
- ASCII: http://en.wikipedia.org/wiki/ASCII [6].
- UTF–UCS Transformation Format: http://en.wikipedia.org/wiki/UTF-8 [7].

3.9 Exercises

1. Represent 6 and (-7) in a 5-bit sign-magnitude notation and add them up in binary, without looking at their signs.
2. By hand, find the 5-bit two's complement representation of the following numbers: 4, -5, 1, -0, 11.
3. Compute the decimal values of the following 8-bit two's complement representations:

   a. 01111000
   b. 10110011
   c. 10000001
   d. 11111110

4. Perform the following calculations on decimal numbers using only 8-bit two's complement representation and binary addition (you are not allowed to use binary subtraction). Verify your results by performing the operations using decimal values.

   a. $101 - 111$
   b. $10 - 20$
   c. $-20 - 17$
   d. $118 - 18$

5. Find the IEEE 754 32-bit representation of the following floating-point numbers: 3.3, 3.37, 3.375.
6. Compare the largest values that can be stored using two's complement notation and IEEE 754 floating-point representation in 32 bits. Considering that both representations make use of all possible 32-bit combinations (2^{32} in total), what is the cause of the (huge) difference?
7. Let us assume that we modify the IEEE 754 32-bit representation by increasing the mantissa by 1 bit to 24 bits and decreasing the exponent by 1 bit to 7 bits. What will we gain and what will we lose?
8. Draw the flowchart for an algorithm for comparing two 32-bit IEEE 754 floating points, based on their sign bits, exponents, and mantissa subfields.
9. Convert the following IEEE 754 floating-point numbers to their decimal values.

   a. 11100100110110000000000000000000
   b. 01000001010000010000000000000000
   c. 11000001010000010110000000000000
   d. 10111111110010000000000000000000

10. If we had some variable-sized floating-point representation but finite memory size, would we be able to represent:

    a. All real numbers?
    b. All irrational numbers?
    c. All rational numbers?

    For each case, if your answer is yes, explain how; if not, explain why not.

References

[1] G. Üçoluk, S. Kalkan, *Introduction to Programming Concepts with Case Studies in Python* (Springer Science & Business Media, 2012)

[2] Wikipedia contributors, “Two’s complement,” Wikipedia, The Free Encyclopedia (2023). https://en.wikipedia.org/w/index.php?title=Two%27s_complement&oldid=1173822030. Accessed 5 Sept 2023

[3] Wikipedia contributors, “Method of complements,” Wikipedia, The Free Encyclopedia (2023). https://en.wikipedia.org/w/index.php?title=Method_of_complements&oldid=1166622150. Accessed 5 Sept 2023

[4] Wikipedia contributors, “Offset binary,” Wikipedia, The Free Encyclopedia (2022). https://en.wikipedia.org/w/index.php?title=Offset_binary&oldid=1113602470. Accessed 5 Sept 2023

[5] Wikipedia contributors, “Ieee 754-2008 revision,” Wikipedia, The Free Encyclopedia (2023). https://en.wikipedia.org/w/index.php?title=IEEE_754-2008_revision&oldid=1166623991. Accessed 5 Sept 2023

[6] Wikipedia contributors, “Ascii,” Wikipedia, The Free Encyclopedia (2023). https://en.wikipedia.org/w/index.php?title=ASCII&oldid=1173875977. Accessed 5 Sept 2023

[7] Wikipedia contributors, “Utf-8 — c,” Wikipedia, The Free Encyclopedia (2023). https://en.wikipedia.org/w/index.php?title=UTF-8&oldid=1173289191. Accessed 5 Sept 2023

4 Dive into Python

After having covered some background on how a computer works (Chap. 1), how we solve problems with computers (Chap. 2), and how we can represent data in binary (Chap. 3), we are ready to interact with, learn to represent data, and perform actions in Python.

The Python interpreter that we introduced in Sect. 2.5 is a program waiting for our computational demands if not executing something already. Those demands that describe an algorithm must 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 basis for our solutions are going to be performed with 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. This is achieved by using actions that enable interaction with the “environment”.

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

1. **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.
    - *function*s acting on expressions.

2. **Statement**: Unlike an expression, a statement does *not* return data as a result. It can either be 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 subsequent 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 control the execution of those statements in some way.

S. Kalkan et al., *Programming with Python for Engineers*,
https://doi.org/10.1007/978-3-031-57148-0_4

Naming and Printing Data

For the sake of clarity, we will make use of two concepts (variables and printing data) before they are introduced in more detail in the second part of the chapter. 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
  ```

  Here, we asked Python to add 10 with a value referred to by the name `a`, and Python produced the result as `13`. We call such names *variable*s, and the action `a = 3` is called an assignment. We defer a more detailed coverage until Sect. 4.4.1.
- **Printing data**: Python provides the `print()` function to display data items on the screen:

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

  where `item` can be of any data type. For example:

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

4.1 Basic Data

The following are the basic data types we frequently use:

- Numbers
  - Integers
  - Floating point numbers
  - Complex numbers
- Booleans

In Python, arithmetic operations between numbers of the same type are provided in a way that is consistent with our mathematical expectations. Furthermore, mixed-type operations (e.g., subtracting a floating point number form an integer) are also possible.

In programming, being able to ask questions about data is vitally important. 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 `True` or `False`.

4.1.1 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 in 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 silently switches from the CPU-imposed fixed-size integers to some elaborated big-integer representation when needed. You do not have to worry about it. Furthermore, keep 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 that have a 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.

More on Integers and Floating Point Numbers
Python provides the `int` data type for integers and `float` data type for floating-point numbers. You can easily play with the `int` and `float` numbers and check their types as follows:

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

where < `class 'int'` > indicates type `int`.

As we mentioned above, 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: (i) `int`, which used the fixed-length two's complement representation (covered in Sect. 3.1.2) supported by the CPU, and (ii) `long`, which was unbounded. Since this book is based on Python version 3, we assume that `int` refers to an unbounded representation.

As for the `float` type, Python uses the 64-bit IEEE 754 standard (covered in Sect. 3.2.1), which can represent numbers in the range `[2.2250738585072014E-308, 1.7976931348623157E+308]`.

Useful Operations:
The following functions can be useful when working with numbers:

- `abs(<Number>)`: Takes the absolute value of the number.

```
>>> abs(-3.2)
3.2
>>> abs(3.2)
3.2
```

- `pow(<Number1>, <Number2>)`: Takes the power of `<Number1>`, i.e., $\text{<Number1>}^{\text{<Number2>}}$. Python also provides an operator (`**`) for taking the power of numbers: `<Number1> ** <Number2>`.

```
>>> pow(3, 2.4)
13.966610165238235
>>> pow(3, -1)
0.3333333333333333
```

- `round(<FloatNumber>)`: Rounds the floating point number to the *closest* integer.

```
>>> round(4.2)
4
>>> round(4.6)
5
```

- Functions from the `math` library: `sqrt()`, `sin()`, `cos()`, `log()`, etc. (see the Python documentation[1] 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
```

4.1.2 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'>
```

Python converts several instances of other data types to Boolean values, if used in a place where a Boolean value is expected. For example,

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

are interpreted as `False`. All other values are interpreted as `True`.

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

Useful Operations:
With Boolean values, we can use the "`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 operands are `True`, otherwise it returns `False`. `or` returns `True` if one or both of its operands are `True`. The following table gives the 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`

4.2 Container Data (str, tuple, list, dict, set)

Up to this point, we have seen the basic data types. They are certainly needed in our computations, 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 equivalences. Graphs are necessary for many spatial problems. Trees are vital for representing hierarchical relational structures, action logics, and 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). The elements of a list 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 efficient data type that implements 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 carried out in almost constant

time (irrespective of how many elements the dictionary has). In Computer Science terms, a dictionary is called the *hash table* data structure.

5. **Set** (`set`): The `set` type is equivalent to sets in mathematics. The order of the elements is undefined. (*We de-emphasize 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. A dictionary is not sequential and element indexes are arbitrary. For brevity, 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). In the following, we will walk through some examples to explain them.

Mutability Versus Immutability

Some container types are created as "frozen". After creating them, you can completely 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, it is possible to add new elements and change or delete existing ones. As we will see throughout the book, mutable data types will make programming easier for us.

4.2.1 Accessing Elements in Sequential Containers

All containers except `set` reveal and provide access to their individual elements by an *indexing* mechanism with brackets, as illustrated in Fig. 4.1.

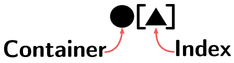


Fig. 4.1 How elements of a container are accessed

For the s-containers, the *index* is an ordinal number where counting starts from zero. For dictionaries, the index is a Python data item from the source (domain) set. A *negative index* (usable only on s-containers) means that the (counting) value is relative to the end. A negative index can be converted to a positive index by adding the length of the container to the negative value.

Below we have an s-container that has a length of $(n + 1)$ (note that indexing starts at 0!) (Fig. 4.2):

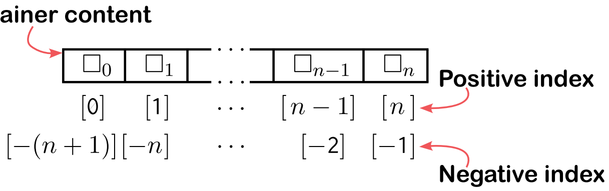


Fig. 4.2 $+/-$ indexing of an s-container

Observe that, 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. This mechanism allows us to define a start index and an end index and access the items that lie in between (Fig. 4.3):

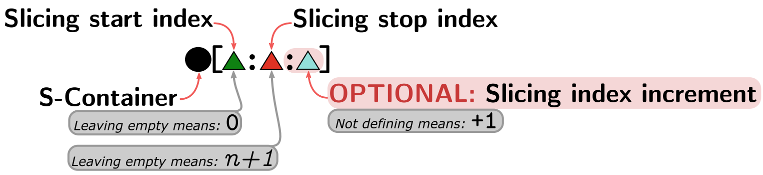


Fig. 4.3 Accessing multiple elements of an s-container is possible via the slicing mechanism, which specifies a starting index, an ending index, and an index increment between elements

- 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—that is, the end index is not inclusive).
- It is also possible to optionally define an increment (a "jump" amount) between the indexes. After the element at $[start]$ is accessed first, $[start + increment]$ is accessed next. This goes on until the accessed position is equal to 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.

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.

4.2.2 Useful and Common Container Operations

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 the "+" 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 the "*" operation:

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

where the container is copied `<Number>` 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, the `in` operator tests whether the element is present in the domain set, while, for other data structures, it checks if the element is a member.

4.2.3 String (str)

As explained in Sect. 3.5.2, a string is used to hold a sequence of characters. Essentially, it is a container where each element is a character. However, Python does not have a special representation for a single character. Single characters, if necessary, 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 (`’Example String’`) or double quotes (`"Example String"`). A string surrounded with triple double quotes (`"""Example String"""`) 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 4.1 provides the full list.

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

Table 4.1 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*

that begins and ends with the literal double quote character. Finally, `\` can be used to escape itself: `\\` is the literal backslash character. For `'\nnn'`, *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 10–15).

In Python v3, all strings use the Unicode representation where all international symbols can be used, 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 on the Web page of this chapter. Feel free to change the text and rerun the examples.

Hands-on Code 4.1

https://pp4e.online/c4s1

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

Output

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

Hands-on Code 4.2

https://pp4e.online/c4s2

```
my_beautiful_string[0]
```

Output

```
'T'
```

Hands-on Code 4.3

https://pp4e.online/c4s3

```
my_beautiful_string[4]
```

Output

```
'q'
```

Hands-on Code 4.4

https://pp4e.online/c4s4

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

Output

```
'The '
```

Hands-on Code 4.5

https://pp4e.online/c4s5

```
my_beautiful_string[:4]
```

Output

```
'The '
```

Hands-on Code 4.6

https://pp4e.online/c4s6

```
my_beautiful_string[4:]
```

Output

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

Hands-on Code 4.7

https://pp4e.online/c4s7

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

Output

```
'brown'
```

Hands-on Code 4.8

https://pp4e.online/c4s8

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

Output

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

Hands-on Code 4.9

https://pp4e.online/c4s9

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

Output

```
'laz'
```

Hands-on Code 4.10

https://pp4e.online/c4s10

```
my_beautiful_string[:]
```

Output

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

Hands-on Code 4.11

https://pp4e.online/c4s11

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

Output

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

Hands-on Code 4.12

https://pp4e.online/c4s12

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

Output

```
'zal'
```

Hands-on Code 4.13

https://pp4e.online/c4s13

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

Output

```
'Teqikbon'
```

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

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

Useful Operations with Strings:
In the following, we go over some common operations with strings. After having introduced object-oriented programming, we will cover more useful operations in Sect. 7.3.

- 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 really fun '
>>> 'fun' in 'Python'
False
>>> 'on' in 'Python'
True
```

- Evaluating 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, modifying them directly is not possible. The only way to modify a string is by creating a new string, which involves utilizing slicing and concatenation operations (using "+"). The new string is then replaced in the same location as 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'
```

4.2.4 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 discussed at the beginning of Sect. 4.2, 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:

- `[("aerys","targaryen",("deceased", 1996)), ("brandon","stark", "not born")]`
- `(["aerys","targaryen",("deceased", 1996)], ["brandon","stark", "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 use cases for lists and tuples:

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

Useful Operations with Lists and Tuples

In the following, we go over some common operations with lists and tuples. After having introduced object-oriented programming, we will cover more useful operations in Sect. 7.3.

1-*Deletion from lists*

As far as deletion is concerned, you can use two methods for the time being:

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

```
>>> L = [111,222,333,444,555,666]
>>> L[1:5] = []
>>> 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[1:5]
>>> 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 programming features (i.e., `<data>.function(…)`), which will be covered in Chap. 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()` function 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* `list()` *functions*: Similar to other data types, the tuple and list data types provide two functions for creating data from other data types. An example is provided below, after the last item.

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. An example is provided below, after the last item.

5-*Membership*: Similar to strings, `in` and `not in` operations can be used to check whether a tuple/list contains an element. An example is provided below, after the last item.

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])
>>> 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 mentioned, the examples with some containers included two types of constructs that were not covered yet: One is the use of “functions” on containers, e.g., the use of `len()`. With your high school background, the use of functions should be 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 a 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 *member functions* and called (applied) on that object (in our case the container) by means of this dot notation:

$$\bullet.\,f(\Box) \quad \text{has the conceptual meaning of} \quad f(\bullet,\Box) \tag{4.1}$$

Do not try this explicitly as it will not work. The equivalence is just “conceptual”: The function receives the object internally, as a “hidden” argument. All these will be covered in detail in Chap. 7. Till then, for the sake of completeness, from time to time, we will be referring to this notation. For the time being, simply interpret the use of member functions 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. We can benefit from representing them in a regular grid structure that we call matrix:

$$A = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & & \ddots & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{pmatrix}, \tag{4.2}$$

which has n columns and m rows. We generally shorten this as $m \times n$ and say that matrix A has size $m \times n$.

Solutions to many different problems can be formulated as a set (system) of equations that describe relations among a set of variables. The following is a simple example:

$$\begin{aligned} 3x + 4y + z &= 4, \\ -3x + 3y + 5 &= 3, \\ x + y + z &= 0. \end{aligned} \tag{4.3}$$

These equations can be represented with matrices as follows:

$$\begin{pmatrix} 3 & 4 & 1 \\ -3 & 3 & 5 \\ 1 & 1 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} 4 \\ 3 \\ 0 \end{pmatrix}. \tag{4.4}$$

This represents Eq. 4.3 in terms of matrices and matrix multiplication. It is okay if you are not familiar with matrix multiplication—we will briefly introduce the concept and use it as an example in the next chapter.

Writing a problem in a matrix form as 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 the coefficients in a matrix, which allows us to analyze and manipulate the matrix of coefficients (the first matrix in Eq. 4.4) 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.

Let us now explore how matrices can be represented in Python. A simple and effective approach that allows for modifying elements of the matrix later is by using nested lists, as shown below:

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

Compare this list with the first matrix in Eq. 4.4 and note that each row is represented as a list which is a member of the outer list. This would allow us to access entries like this:

```
>>> A[1]  # 2nd row
[-3, 3, 5]
>>> A[1][0] # 2nd row, 1st element
-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 Chap. 10. For an extended coverage of matrices and matrix operations, we refer to the Matrix algebra for beginners[2] or Chap. 2.2 of “Mathematics for Machine Learning”.[3]

[2] J. Gunawardena, “Matrix algebra for beginners”, lecture notes, 2006. http://vcp.med.harvard.edu/papers/matrices-1.pdf.

[3] M. P. Deisenroth, A. A. Faisal and C. S. Ong, “Mathematics for Machine Learning”, Chap. 2.2, Cambridge University Press, 2020. https://mml-book.com.

4.2.5 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. 4.4).

The keys can only be immutable data types, i.e., numbers, strings, or tuples (with only immutable elements); some other immutable types that we do not cover in this book can also be used as keys. Since lists and dictionaries are mutable, they cannot be used as keys for indexing. Regarding 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. Dictionaries, as well as their individual elements, are mutable (you can replace them). Moreover, it is possible to add and remove items from the mapping.

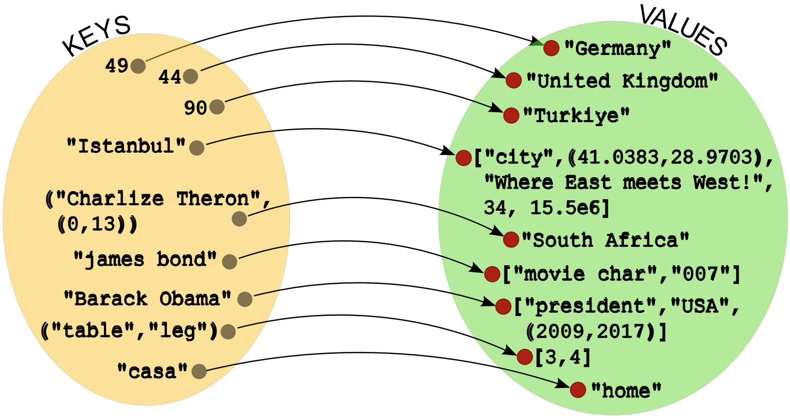


Fig. 4.4 A dictionary provides a mapping from keys to values

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. 4.4 would be denoted as:

```
{49: "Germany",
44: "United Kingdom",
90: "Turkey",
"Istanbul": ["city", (41.0383,28.9703), "Where East meets West!", 34, 15.
↪5e6],
("Charlize Theron", (0,13)): "South Africa",
"james bond": ["movie char", "007"],
"Barack Obama": ["president", "USA", (2009, 2017)],
("table", "leg"): [3, 4],
"casa": "home"}
```

Similar to other containers, we usually access dictionaries via variables. Let us assume the dictionary above was assigned to a variable with the name `conno` and look at some examples:

Hands-on Code 4.14

https://pp4e.online/c4s14

```
conno = {49: "Germany", 44: "United Kingdom", 90: "Turkey", "Istanbul": ["city", (41.0383,28.9703),
↪"Where East meets West!", 34, 15.5e6], ("Charlize Theron", (0,13)): "South Africa", "james bond": [
↪"movie char", "007"], "Barack Obama": ["president", "USA", (2009, 2017)], ("table", "leg"): [3, 4],
↪"casa": "home"}
print(conno["james bond"])
print(conno["Istanbul"])
```

Output

```
['movie char', '007']
['city', (41.0383, 28.9703), 'Where East meets West!', 34, 15500000.0]
```

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

Hands-on Code 4.15

https://pp4e.online/c4s15

```
>>> print(conno["london"])
print(conno["london"])
```

Output

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
Cell In[17], line 1
----> 1 print(conno["london"])
KeyError: 'london'
```

Ups, that was bad! To avoid such a situation, we have a simple method to test for the existence of a key in a dictionary, by using the `in` operation:

Hands-on Code 4.16

https://pp4e.online/c4s16

```
>>>  print("london" in conno)
```

Output

```
False
```

It is also possible to remove/insert key–value pairs from/to a dictionary:

Hands-on Code 4.17

https://pp4e.online/c4s17

```
print("conno has this many keys:", len(conno))
conno["london"] = ["capital", "london eye", 44]
print("conno has this many keys now:", len(conno))
print(conno["london"])
print(conno["james bond"])
del conno["james bond"]
print("james bond" in conno)
print("After 'james bond' is deleted we have this many keys:", len(conno))
```

Output

```
conno has this many keys: 9
conno has this many keys now: 10
['capital', 'london eye', 44]
['movie char', '007']
False
After 'james bond' is deleted we have this many keys: 9
```

The benefit of using a dictionary is “time efficiency”. The functionality of a dictionary could be attained by using (key, value) tuples inserted into a list. In such a representation, 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 time proportional to the length of the list (the 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, as illustrated for the `conno` dictionary introduced above:

```
>>> conno.values()
dict_values(['Germany', 'United Kingdom', 'Turkey', ['city', (41.0383, 28.
↪9703), 'Where East meets West!', 34, 15500000.0], 'South Africa', ['movie␣
↪char', '007'], ['president', 'USA', (2009, 2017)], [3, 4], 'home'])
>>> conno.keys()
dict_keys([49, 44, 90, 'Istanbul', ('Charlize Theron', (0, 13)), 'james bond
↪', 'Barack Obama', ('table', 'leg'), 'casa'])
>>> for key in conno.keys(): print(f"key: {key} => value: {conno[key]}")
key: 49 => value: Germany
key: 44 => value: United Kingdom
key: 90 => value: Turkey
key: Istanbul => value: ['city', (41.0383, 28.9703), 'Where East meets West!
↪', 34, 15500000.0]
key: ('Charlize Theron', (0, 13)) => value: South Africa
key: james bond => value: ['movie char', '007']
key: Barack Obama => value: ['president', 'USA', (2009, 2017)]
key: ('table', 'leg') => value: [3, 4]
key: casa => value: home
```

4.2.6 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. Since sets are mutable, sets themselves cannot be elements of other sets.

Here is a small example with sets:

Hands-on Code 4.18

https://pp4e.online/c4s18

```
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)
```

Output

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

Lists can undertake most functionalities of sets. Furthermore, lists do not possess the restrictions that sets do. On the other hand, membership tests especially 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)`).

Here are some examples:

```
>>> odd = {1, 3, 5, 7}
>>> even = {0, 2, 4, 6, 8}
>>> odd
{1, 3, 5, 7}
>>> even
{0, 2, 4, 6, 8}
>>> odd_and_even = odd | even
>>> odd_and_even
{0, 1, 2, 3, 4, 5, 6, 7, 8}
>>> odd <= odd_and_even # odd is a subset of odd_and_even?
True
>>> odd <= even # odd is a subset of even?
False
>>> odd_and_even - odd == even # (odd_and_even \ odd) =? even
True
```

The followings 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 the element from the set.
- `S.pop()`: Remove an arbitrary element from the set.

4.3 Expressions

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

Operations can be graphically illustrated as follows:

$$\square_1 \odot \square_2 \tag{4.5}$$

where $\odot$ is called the operator, and $\Box_1$ and $\Box_2$ are called the operands. In this example, the operator is binary, i.e., it acts on two operands.

We can also have *unary* operators:

$$\odot\Box \tag{4.6}$$

or operators that have more than two operands.

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

4.3.1 Arithmetic, Logic, Container, and Comparison Operations

Python provides various operators 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. Some operators might be applied to different container data types with different meanings. An operator compatibility matrix is given in Table 4.2

Table 4.2 Python operators and valid operand types are marked with a circle, ●. An empty cell means the data type is not supported by the operator

	bool	int	float	str	list	tuple	dict	set
`+`	●[1]	●[1]	●[1]	●[2]	●[2]	●[2]		
`-`	●[1]	●[1]	●[1]					●[8]
`*`	●[1]	●[1]	●[1]	●[3]	●[3]	●[3]		
`/`	●[1]	●[1]	●[1]					
`//`	●[4]	●[4]	●[4]					
`%`	●[5]	●[5]	●[5]	●[6]				
`**`	●[1]	●[1]	●[1]					
`==`	●[1]	●[1]	●[1]	●[7]	●[7]	●[7]	●[7]	●[7]
`<`	●[1]	●[1]	●[1]	●[9]	●[9]	●[9]		●[8]
`>`	●[1]	●[1]	●[1]	●[9]	●[9]	●[9]		●[8]
`<=`	●[1]	●[1]	●[1]	●[9]	●[9]	●[9]		●[8]
`>=`	●[1]	●[1]	●[1]	●[9]	●[9]	●[9]		●[8]
`!=`	●[1]	●[1]	●[1]	●[7]	●[7]	●[7]	●[7]	●[7]
`in`				●[10]	●[10]	●[10]	●[10]	●[10]
`[]`				●[11]	●[11]	●[11]	●[12]	
&	●[13]	●[13]						●[8]
|	●[13]	●[13]					●[8]	●[8]
ˆ	●[13]	●[13]						●[8]
~	●[13]	●[13]						
»	●[13]	●[13]						
«	●[13]	●[13]						

[1] Arithmetic operator with the default meaning. bool values are mapped to integers 0 and 1. Comparison operators return `bool`. Others return `int` if both operands are `int`, otherwise return `float`

[2] Concatenates two containers of the same type and returns the new container with the same type

[3] Repeats the elements of the container a given number of times to return a new container. The right operand should be an integer

[4] Integer part of the division. Returns `int` for `int` operands, `float` otherwise

[5] Remainder of the integer division. Returns `int` for `int` operands, `float` otherwise

[6] First operand is the format string. Returns the formatted string with the values from the right operand

[7] Equality of containers with the same data type. Element by element comparison. Returns `bool`

[8] Set operation, returns `set`. Comparison operators are mapped to subset ($\subset$) relation and return `bool`

[9] Lexicographic or dictionary order. The result depends on the leftmost differing element. Returns `bool`

[10] If the container contains the value provided as the right operand. For dictionaries, it searches the value in the keys only. Returns `bool`

[11] Member selection with an integer index. Slicing (`[str:stp:inc]` syntax) is supported. May return any type

[12] Member selection with an immutable index type. No slicing. May return any type

[13] Bitwise logical operation applied on bit by bit basis for integer types. Returns `int`

with their semantics (for Python 3.9). Note that Python is an evolving language and the operator compatibility in Table 4.2 may change from version to version, especially with major versions.

Below are some illustrative examples:

Hands-on Code 4.19

https://pp4e.online/c4s19

```
print("[1,2,3] + [3,4] -> ", [1,2,3] + [3,4])
print("(1,2,3) + (3,4) -> ", (1,2,3) + (3,4))
print("[1,2] * 3 -> ", (1,2) * 3)
print("'hello' * 3 -> ", 'hello' * 3)
S1 = "Four"
S2 = "Five"
print("len(S1) < len(S2) -> ", len(S1) < len(S2) )
print("S1 != S2  -> ", S1 != S2 )
print("S1 > S2  -> ", S1 > S2 )
print("[1,2,3] == [1,2,3] -> ", [1,2,3] == [1,2,3])
print("[1,2,3] == (1,2,3) -> ", [1,2,3] == (1,2,3))
print("[1,2,3] < [1,2,5] -> ", [1,2,3] <[1,2,5])
print("{1,2,3} < {1,2,5} -> ", {1,2,3} < {1,2,5})
print("{1,2} < {1,2,5} -> ", {1,2} < {1,2,5})
print("4 in [1,2,3,4] -> ", 4 in [1,2,3,4])
print("'i' in 'team'  -> ", 'i' in 'team')
print("4 in {4:'a', 5:'b'} -> ", 4 in {4: 'a', 5:'b'})
print("'a' in {4:'a', 5:'b'} -> ", 'a' in {4: 'a', 5:'b'})
```

Output

```
[1,2,3] + [3,4] ->  [1, 2, 3, 3, 4]
(1,2,3) + (3,4) ->  (1, 2, 3, 3, 4)
[1,2] * 3 ->  (1, 2, 1, 2, 1, 2)
'hello' * 3 ->  hellohellohello
len(S1) < len(S2) ->  False
S1 != S2  ->  True
S1 > S2  ->  True
[1,2,3] == [1,2,3] ->  True
[1,2,3] == (1,2,3) ->  False
[1,2,3] < [1,2,5] ->  True
{1,2,3} < {1,2,5} ->  False
{1,2} < {1,2,5} ->  True
4 in [1,2,3,4] ->  True
'i' in 'team'  ->  False
4 in {4:'a', 5:'b'} ->  True
'a' in {4:'a', 5:'b'} ->  False
```

4.3.2 Bitwise Operators

Bitwise operators are logical operators defined for the integer data type. In contrast to the `bool` type, bitwise logical operations are carried out in bit by bit manner. Each bit of the two's complement representation of an integer value takes part in a logical operation with the same bit position of the other operand. "&" denotes the logical "`and`", "`|`" denotes the logical "`or`", and unary operator "~" denotes the logical "`not`". Exclusive or, "ˆ" has no Boolean counterpart. It gives 0 for the case where both values are the same, 1 otherwise. In addition to the binary logical operators, there are `left shift` (<<) and `right shift` (>>) operators that shift all bits to the left or right and pad 0 for the missing values. The right operand for a shift operator is an integer denoting how many times the bits will be shifted.

Bitwise operators are used in limited areas of programming like "cryptography", "low-level I/O", and hardware control. The following examples show how these operators work:

Hands-on Code 4.20

https://pp4e.online/c4s20

```
a = 85
b = 15
print('a:     ', '{:8b}'.format(a), a)
print('~a:    ', '{:8b}'.format(~a), ~a)
print('b:     ', '{:8b}'.format(b), b)
print('a & b: ', '{:8b}'.format(a & b), a & b )
print('a | b: ', '{:8b}'.format(a | b), a | b )
print('a ^ b: ', '{:8b}'.format(a ^ b), a ^ b )
print('a >> 3:', '{:8b}'.format(a >> 3), a >> 3 )
print('a << 3:', '{:8b}'.format(a << 3), a << 3 )
```

Output

```
a:       1010101 85
~a:     -1010110 -86
b:          1111 15
a & b:       101 5
a | b:   1011111 95
a ^ b:   1011010 90
a >> 3:     1010 10
a << 3: 1010101000 680
```

4.3.3 Exercise

Give an example for each row in Table 4.2. Please complete this exercise in the interactive Python notebook version of the chapter.

4.3.4 Evaluating Expressions

In the previous section, we have seen simple uses of operators in expressions. 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 the first operator to be evaluated. For example, 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, the 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 4.3.

Therefore, according to Table 4.3, a sophisticated expression such as `2**3**4*5-2//2-1` is equivalent to `2**81*5-1-1` which is equivalent to `12089258196146291747061760-2` which is `12089258196146291747061758`.

Table 4.3 Precedence and associativity for the operators in Table 4.2

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

Below are some notes and explanations regarding expression evaluation:

(i) The *Special* keyword in Table 4.3 refers to a treatment which is common to mathematics but not to programming. Python is unique among commonly used programming languages. If $\odot_i$ is any Boolean comparison operator and $\Box_j$ is any numerical expression, the sequence of

$$\Box_1 \odot_1 \Box_2 \odot_2 \Box_3 \odot_3 \Box_4 \cdots \Box_{n-1} \odot_{n-1} \Box_n \tag{4.7}$$

is interpreted as

$$\Box_1 \odot_1 \Box_2 \ \texttt{and} \ \Box_2 \odot_2 \Box_3 \ \texttt{and} \ \Box_3 \odot_3 \Box_4 \quad \cdots \quad \Box_{n-1} \odot_{n-1} \Box_n \tag{4.8}$$

(ii) It is **always** possible to override the precedence by making use of parentheses. This is precisely how we grouped calculations in our middle school math courses. For example, for an expression such as $3 + 4 * 5$, if we want the addition to be performed first, we can use parentheses like $(3 + 4) * 5$.

(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 (e.g., as in 3 + 4.5), then the operation is performed according to the following rules (see also Sect. 4.3.5):

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

(v) Except for the logical operators `and` and `or`, all operators have an evaluation scheme that is coined as *eager evaluation*. Eager evaluation is the strategy where all operands are evaluated first and then the semantics of the operators kick in, providing the result.
Here is an example: Consider the mathematical expression "`0*(2**150-3**95)`". As intelligent beings, our immediate reaction would be as follows: Anything multiplied with `0` (zero) is `0`, therefore, we do not have to compute the two huge exponentiation operations within the parentheses. This is taking a *short-cut* in evaluation and is certainly **far from** eager evalua-

tion. Eager evaluation would evaluate the operations within the parentheses, obtain the value `−693647454339354238433323618063349607247325483` and then multiply this with `0` to obtain `0`.

Python would follow this "less intelligent" way and perform 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).

For a conjunctive expression like:

$$\square_1 \ \texttt{and} \ \square_2 \ \texttt{and} \ \square_3 \ \texttt{and} \ \cdots \ \texttt{and} \ \square_n \tag{4.9}$$

The evaluation proceeds as illustrated in Fig. 4.5:

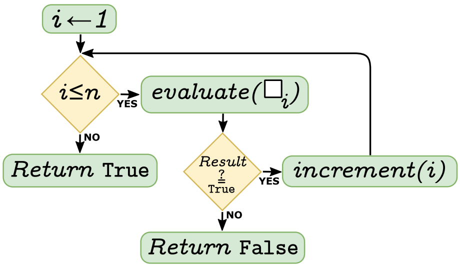


Fig. 4.5 Logical AND evaluation scheme

Similarly, a disjunctive expression:

$$\square_1 \ \texttt{or} \ \square_2 \ \texttt{or} \ \square_3 \ \texttt{or} \ \cdots \ \texttt{or} \ \square_n \tag{4.10}$$

has the following evaluation scheme in Fig. 4.6.

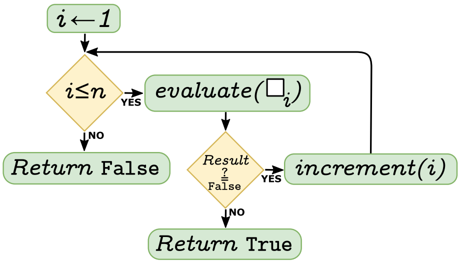


Fig. 4.6 Logical OR evaluation scheme

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 (i) is readable and understandable by other programmers and (ii) does not include any ambiguity.

4.3.5 Implicit and Explicit Type Conversion (Casting)

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

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

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

Although Python can do such conversions, it is a good programming practice to make these conversions explicit to ensure the clarity of the intention to the reader. Explicit type conversion, also called *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. Implicit conversions are allowed only for the basic data types as illustrated in Fig. 4.7.

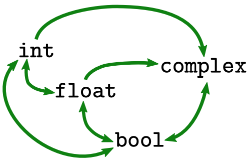


Fig. 4.7 Type casting among numeric and Boolean data types

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

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

4.4 Basic Statements

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

4.4.1 Assignment Statement and Variables

After obtaining a result in Python, we often print and/or keep the result for further computations. *Variables* help us here as named memory positions in which we can store data. You can imagine variables as pigeonholes that can hold **single** data items.

There are two methods to create and store data into a variable. Here, we will mention the one that is overwhelmingly used. The other is more implicit and will be introduced when we cover functions.

A variable receives data for storage using 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 prior value, if there is one, is purged).

After the assignment, the value can be used repetitively. 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 in the second line uses the integer `3` 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 `4`, is stored in the variable `a`. The variable `a` had a previous value of `3`; this value is purged and replaced by `4`. The former value is not kept anywhere and cannot be recovered.

Multiple Assignments

Multiple variables can be assigned to the same value at once. For example,

```
>>> 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 at `4`.

Multiple Assignments 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 is 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 that you want to swap the values of two 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 of variable `b`: Is it updated? On the fourth line, there is an expression using the variable `b`: Which value is it referring to? `4` or `5`? When we use the variable `b` in the following expression, will the “definition” be retrieved and recalculated?

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

- **ANSWER:** *No. Statements are executed only once here, when they are entered into the Python interpreter (it is possible to repetitively execute a statement but that is not the case in this example). Each assignment in the example above is executed only once. No re-evaluation 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. This is very similar to that, right? But I am confused having seen a line like `a = a + 1`. What is happening? The `“a”` s cancel out and we are left with `0 = 1`?

- **ANSWER:** *The use of the equal sign (=) is confusing to some extent. It does not stand for a balance between the left-hand side and the right-hand side. Do not interpret it as an "equality" in mathematics. It has absolutely different semantics. As discussed above, it only means:*

  1. **First** *calculate the right-hand side,*
  2. **Then** *store the result into an (electronic) pigeonhole which has the name label given to the left-hand side of the equal sign.*

- **QUESTION:** I typed 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 might be missing the time flow. The statements are executed in the following 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 (owing to the third step). "y" now holds 3.*
  5. **Fifth:** *print both the values in "x" and "y". Both are "3", as displayed on screen.*

4.4.2 Variables and Aliasing

There is something peculiar about lists (and other mutable data types) that we need to be careful about while assigning them to variables. First, consider the following code:

"The first example"

Hands-on Code 4.21

https://pp4e.online/c4s21

```
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)
```

Output

```
address of 5:  4333935280
address of a:  4333935280
address of b:  4333935280
address of a:  4333935216
b is:  5
```

Here, we used the `id()` function to display the addresses of the variables in the memory to help us understand what happened in those assignments:

- The second line creates `5` in the memory and links that with `a`.
- The fourth line links the content of `a` with the variable `b`. Hereby, 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, let us keep the task the same but change the data from an integer to a list:

“The second example”

Hands-on Code 4.22

https://pp4e.online/c4s22

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

Output

```
addresses of a and b:  4364549568 4364549568
b is:  [5, 1, 7]
addresses of a and b:  4364490688 4364549568
b is:  [5, 1, 7]
```

In other words, this works similarly to the first example, as expected.

In contrast, consider the following slightly different example:

“The third example”

Hands-on Code 4.23

https://pp4e.online/c4s23

```
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)
```

Output

```
addresses of a and b:  4364489280 4364489280
b is:  [5, 1, 7]
addresses of a and b:  4364489280 4364489280
b is:  [3, 1, 7]
```

To our surprise, changing the first element of `a` (with `a[0] = 3`) also affected `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` since they are just different names for the same memory location.

This behavior is called *aliasing*. 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 context:

- If you carelessly assign a mutable variable to another variable, changes made on one variable are going to affect the other one. If this is not intended and the location in the code where the aliasing

is initiated cannot be identified, you may lose hours or days trying to identify the source of the problem in your code.
- Aliasing can be beneficial, especially when we want changes made to one variable to be reflected in another. This will be useful for passing data to functions and getting the results of computations from functions.

4.4.3 Naming Variables

Programmers usually choose a name for a variable such that the name signifies what the content will be. In Python, 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 uppercase letters is allowed, bear in mind that programmers reserve starting with an upper case to differentiate a property (i.e., *scope*) of the variable which you will learn later (when we introduce functions).

y	y1	Y	Y_1	_1
temperature	temperature_today	Temperature Today	Cumulative	coordinate_x
a1b145c	a_1_b1_45_c	s_s_	_	*s*___

As you might have noticed, though they are perfectly valid, the five examples in the last row may not be sensible as they do not provide any information about the content they store.

When naming variables, you can consider the following guideline:

- It is a better practice to name variables that reflect 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
```

  where the values stored by the variables are more clear, making the computation and the code easier to follow.
- You should use different variables for different data, which is named as the "Single Responsibility Principle" in Computer Science. For example, even if you could use the same variable in one statement to count students and in another statement to hold the largest grade, this is not recommended. In programming, we do not economize with variables. Rather, we strive to make the code readable, and for this purpose, we introduce different variables, when necessary, to reflect the semantics and the context, uniquely.
- Variable names should be pronounceable, making them easier to remember.
- Avoid variable names that could misguide you or others looking at your code.
- Use `i,j,k,m,n` only for counting: Programmers implicitly recognize such variables as holding integers (for historical reasons).
- Use `x,y,z` for coordinate values or multi-variate function arguments.
- Avoid the single character variable `l` as it can be easily confused with `1` (one).
- If you are using multiple words as a variable name, choose one of the following:

- Make all words lowercase, combine them using _ as separator; e.g., `highest_midterm_grade`, `shortest_path_distance_up_to_now`.
- Capitalize the first letter of each word, except for the first, and combine all words directly without using a separator; e.g., `highestMidtermGrade`, `shortestPathDistanceUpToNow`.

Reserved Names

The following keywords are being used by Python already and, therefore, cannot be used as variable names. It is possible to use the name of a built-in function as a variable name (e.g., `len = 20`). This, however, loses access to such a built-in function until the interpreter is restarted.

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	

4.4.4 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 Python version 2. However, this has changed, and `print` is a function in Python version 3. Therefore, `print` has a value (which we will see later).

4.5 Compound Statements

Like other high-level languages, Python provides statements that combine other 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.

4.6 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. For these purposes, we can use the following.

4.6.1 Actions for Input

In Python, you can use the `input()` function to obtain input from the user:

```
input(<prompt>)
```

where `<prompt>` is an optional informative string displayed to the user before getting the input. The following is a simple example:

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

The `input()` function gives the programmer a string. If you expect the user to enter an expression, you can evaluate the expression in the string using the `eval()` function as we explained in Sect. 4.2.3.

4.6.2 Actions for Output

To display data on 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 (marked with `{int}`) for data items that 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
```

where the placeholder with the integer index *i* is filled in with the parameter of the `format()` function at index *i*.

Alternatively, instead of using integers for the placeholders, we can give them names or labels (as `{label}`):

```
>>> 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?
```

where a placeholder is filled in with the matching labeled parameter of the format function.

The `format()` function provides many more functionalities than the ones we have illustrated. However, the use exemplified here should be sufficient for general use and this book. The reader interested in complete coverage is referred to Python's documentation on string formatting.[4]

A more handy approach for printing is to tell Python (using `f"..."`) to use existing variables to fill in the placeholders:

```
>>> age = 20
>>> height = 1.70
>>> eye_color = "brown"
>>> print(f"I am {height} tall, {age} years old and have {eye_color} eyes")
I am 1.7 tall, 20 years old, and have brown eyes
```

Note that, in this example, the `format()` function is not used. Instead, Python directly used the variables defined before the `print()` function.

4.7 Actions That Are Ignored

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

4.7.1 Comments

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

- **Comments with #**: When Python encounters the symbol # in your code, it assumes that the current line is finished, ignores the rest of the line, evaluates and 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**: To provide comments longer than one line, you can use triple quotes:

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

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

Multi-line 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 to automatically generate documents for codes.

As we have seen before, multi-line comments are actually strings in Python. Therefore, if you use triple quotes to provide a comment, make sure that it does not overlap with an expression or a statement line in your code.

4.7.2 pass Statement

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 placeholder (to be filled in later) so that the interpreter does not complain about missing a statement. For example:

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

4.8 Actions and Data Packaged in Libraries

Python, like many high-level programming languages, provides 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 to practically cover and explain here. The interested reader is directed to check the comprehensive list at Python docs.[5]

The `import` statement that we used above *loads* the library and makes its contents directly accessible 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` every time 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 Chap. 7 where we introduce the concept of objects and object-oriented programming.

To find out what is available in a library or any data item, you can use the `dir()` function:

```
>>> 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']
```

4.9 Providing Actions to the Interpreter

Python provides different options to provide your actions and get them executed.

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

4.9.1 Directly Interacting with the Interpreter

As we have seen until now, you can interact with and type our actions into the interpreter directly. When you are done with the interpreter, you can finish the session and quit the interpreter using the functions `quit()` or `exit()` or by pressing `CTRL-D`. For example:

```
$ 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 makes it difficult to manage. Moreover, when you exit the interpreter, all your actions and executions are lost, and to be able to run them again, you need to retype them from scratch. This can be tedious, redundant, impractical, and inefficient.

4.9.2 Writing Actions in a File (Script)

Alternatively, you can place our actions in a file with the "`.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] = eval(input("Enter a list of two numbers as [a, b]: "))
print("You have provided: ", a, b)
result = a + b
print("The sum is: ", result)
```

Assuming you have saved these lines in a file (named `test.py`), you can execute the actions in a terminal shell by running the command “`python3 test.py`” (“`cat test.py`” only displays the content of the file):

```
$ 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")
[a, b] = eval(input("Enter a list of two numbers as [a, b]: "))
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 as [a, b]: [3, 4]
```

(continues on next page)

(continued from previous page)

```
You have provided:  3 4
The sum is:  7
```

A more advanced feature is to provide *command-line arguments* to your script and use the arguments provided 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 with the command-line arguments:

```
$ 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.

4.9.3 Using Actions from Libraries (Modules)

Another mechanism for executing your actions is to place them into libraries (modules) and provide them to Python using the `import` statement, as we have illustrated in Sect. 4.8. 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.

4.10 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 on your codes.
- Using libraries.

4.11 Further Reading

- A detailed history of Python and its versions: https://en.wikipedia.org/wiki/History_of_Python [1].
- The concept of aliasing: http://en.wikipedia.org/wiki/Aliasing_%28computing%29 [2].

4.12 Exercises

1. Without using Python, determine the results of the following expressions and validate your answers with Python:

   a. `2 - 3 ** 4 / 8 + 2 * 4 ** 5 * 1 ** 8`
   b. `4 + 2 - 10 / 2 * 4 ** 2`
   c. `3 / 3 ** 3 * 3`

2. Assuming that `a` is `True`, `b` is `True`, and `c` is `False`, what would the values of the following expressions be?

   a. `not a == b + d < not a`
   b. `a == b <= c == True`
   c. `True <= False == b + c`
   d. `c / a / b`

3. 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.
4. To convert Celsius to Fahrenheit, we multiply Celsius by $9/5$ and then add 32. Write a Python code that reads a Celsius value and prints the equivalent Fahrenheit.
5. Discover the procedure to convert Fahrenheit to Celsius. Then, write a Python code that reads a Fahrenheit value and prints the Celsius equivalent.

6. The formula for compound interest is the following:

   $$A = P\left(1 + \frac{r}{n}\right)^{nt},$$

   where:

   A = final amount,
   P = initial principal value,
   r = annual interest rate,
   n = the number of times interest is applied per year,
   t = the number of years.

   Write a Python code that reads P, r, n, t from the user and prints A as the result.
7. A sphere of radius R_1 is made of plastic. There is a sphere-shaped hole of radius R_2 ($R_2 < R_1$) in the plastic sphere. Write a Python code that reads R_1, R_2 from the user, and prints the volume of plastic used.
8. Write a Python code that reads a string from the user and prints `True` if the string is a palindrome, and `False` if not. A string is a palindrome if it is symmetrical. For example, "`madam`" and "`kek`" are palindromes whereas "`mamma`" and "`carcar`" are not.
9. *Body Mass Index* (BMI) is a measure of a person's weight (in kilograms) relative to the person's height (in meters), defined as follows:

   $$\text{BMI} = \frac{weight}{height^2}.$$

   A BMI of 25.0 or more is considered overweight.
   Write a Python code that reads a weight and a height from the user, calculates the BMI value, and prints `OVERWEIGHT` or `NORMAL`, accordingly.
10. Given the variable `perplexed` with a list as follows:

```
perplexed = [[[[1, 2], [3, 4]], [[5, 6], [7, 8]]], \
             [[[9, 10], [11, 12]], [[13, 14], [15, 16]]]]
```

   Construct an expression that fetches out the number `12`, then print it on the screen.

References

[1] Wikipedia contributors, "History of python," Wikipedia, The Free Encyclopedia (2023). https://en.wikipedia.org/w/index.php?title=History_of_Python&oldid=1172730884. Accessed 5 Sept 2023

[2] Wikipedia contributors, "Aliasing (computing)," Wikipedia, The Free Encyclopedia (2023). https://en.wikipedia.org/w/index.php?title=Aliasing_(computing)&oldid=1133707419. Accessed 5 Sept 2023

5 Conditional and Repetitive Execution

Many circumstances require a program's execution (i) to change its flow based on the truth value of a condition, or (ii) to repeat a set of steps for a certain number of times or until a condition is no longer true. These two kinds of actions are crucial components of programming solutions, for which Python provides several alternatives. As we will see in this chapter, some of these alternatives are compound statements, while others are expressions.

5.1 Conditional Execution

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

5.1.1 The `if` Statement

The syntax of the conditional (`if`) statement is as follows:

`if` *Boolean expression* `:` *Statement*

The semantics is rather straightforward: If the *Boolean expression* evaluates to `True`, then the *Statement* is carried out; otherwise, the *Statement* is not executed. Here is a simple example:

```
>>> 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.

S. Kalkan et al., *Programming with Python for Engineers*,
https://doi.org/10.1007/978-3-031-57148-0_5

How to group statements?

It is quite often that we want to do more than one action instead of executing a single statement. This can be performed in Python as follows:

- Switch to a new line, indent by one tab, write your first statement to be executed,
- Switch to the next line, *indent* by the same amount (usually your Python-aware editor will do it for you), write your second statement to be executed, and
- Continue doing these as many times as necessary.
- To finish this writing task, and presumably enter a statement which is *not* part of the group action, just do not indent.

It is very important that these statements, that are part of the `if` part (statements after the colon), have the same amount and type of whitespace for indentation. For example, if the first statement has four spaces, all the following statements in the if statement need to start with four spaces. If the first indentation is a tab character, all the 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 choose a style of indentation (e.g., either four 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 the “...” prompts appear when you sit in front of the interpreter. They will not show up if you run your programs from a file. If you type in these small programs (we call them *snippets*) just ignore “>>>” and “...” from the examples, as follows:

Hands-on Code 5.1

https://pp4e.online/c5s1

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

Output

```
42
Hurray
6
```

As you continue to practice 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 such a programming construct were not provided, we would have to ask the same question again, but this time negated:

```
>>> if x == 6: print("it is still six")
>>> if not (x == 6): print("it is no longer 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` $\boxed{\textit{Boolean expression}}$ `:` $\boxed{\textit{Statement}_1}$

`else :` $\boxed{\textit{Statement}_2}$

Making use of the `else` part, the last example can be re-written as:

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

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

5.1.2 Exercise

In the following code cell, you are expected to insert a few 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):

Hands-on Code 5.2

https://pp4e.online/c5s2

```
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
```

5.1.3 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.

Nested if–else statements allow us to code a structure called a *decision tree*. A decision tree is a set of binary questions and answers where, at each instance of an answer, either a new question is asked or an action is carried out. Figure 5.1 displays such a decision tree for *forecasting rain* based on the temperature, wind, and air pressure values.

The tree in Fig. 5.1 can be coded with nested `if–else` statements as follows:

```
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
```

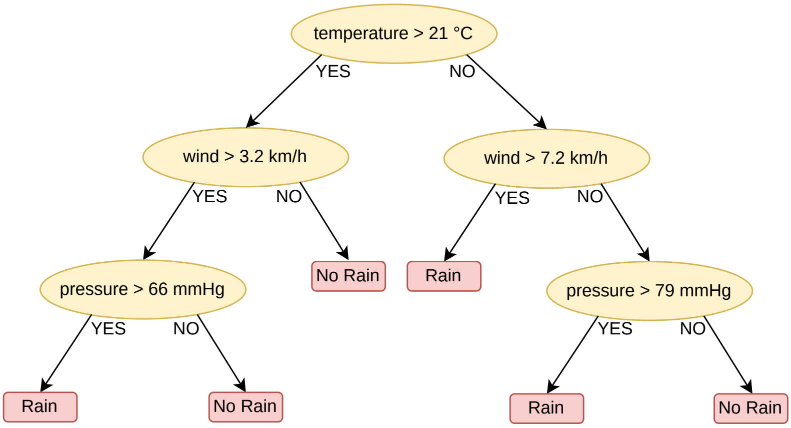


Fig. 5.1 An example decision tree

Sometimes the `else` case contains another `if`, which, in turn, has its own `else` case. This new `else` case can also have its own `if` statement, and so on. For such a cases, `elif` serves as a combined keyword for the `else if` action, as follows:

`if` $\boxed{\mathit{Boolean\ expression}_1}$: $\boxed{\mathit{Statement}_1}$

`elif` $\boxed{\mathit{Boolean\ expression}_2}$: $\boxed{\mathit{Statement}_2}$

⋮

`elif` $\boxed{\mathit{Boolean\ expression}_n}$: $\boxed{\mathit{Statement}_n}$

`else` : $\boxed{\mathit{Statement}_{otherwise}}$

The flowchart in Fig. 5.2 explains the semantics of the `if-elif-else` combination.

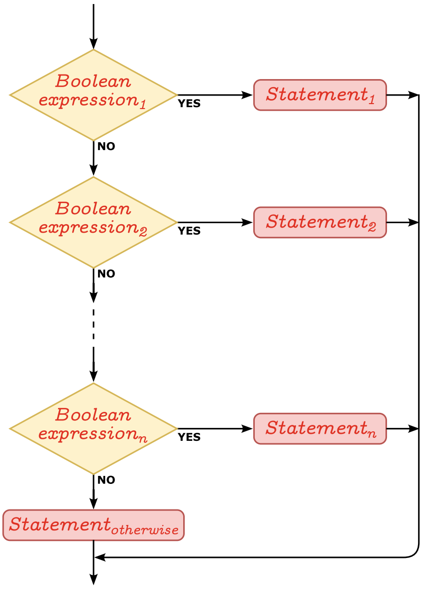


Fig. 5.2 The semantics of the `if-elif-else` combination

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

5.1.4 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

$$s = \begin{cases} (x+1)^2, & x < 1 \\ x - 0.5, & 1 \leq x < 10 \\ \sqrt{x+0.5}, & 10 \leq x < 100 \\ 0, & otherwise \end{cases} \tag{5.1}$$

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

Hands-on Code 5.3

https://pp4e.online/c5s3

```
#@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)
```

Output

```
s is:  3.24037034920393
```

5.1.5 Conditional Expression

The `if` statement, like other statements, does not return a value. A non-statement alternative that has a return value is the *conditional expression*.

A conditional expression also uses the keywords `if` and `else`. However, this time, `if` is not at the beginning of the action, 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 that yields a value. Therefore, 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).

Although it is not compulsory, it is wise to use the conditional expression enclosed within parentheses to make your code more readable and to prevent unintentional 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 the sake of readability, it is strongly advised not to nest conditional expressions. Instead, it is better to restructure the code into nested `if-else` statements (by making use of intermediate variables).

5.2 Repetitive Execution

Repeating a sequence of actions over and over is another fundamental programming ingredient. An action that repeats other actions is called *iteration* or *loop*. Iteration is carried out either for a known number of times or until a criterion is met.

We use iteration:

Type I. To systematically deal with all possible cases or all elements of a collection, or

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

Some examples are:

- Computing letter grades of a class (Type I, for all students).
- Finding the shortest path from city A to city B on a map (Type II).
- Root finding by Newton–Raphson method (Type II).
- Finding the darkest and brightest point(s) in an image (Type I, for all pixels).
- Computing a function value using Taylor expansion (Type I, for a number of orders).
- 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: `while` and `for`. `for` is used mainly for (Type I) iterations, whereas `while` is used for both types.

5.2.1 The `while` Statement

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

`while` *Boolean expression* : *Statement*

Statement is executed *while* *Boolean expression* is `True`.

Similar to the `if` statement, it is possible to have a statement group as part of the `while` statement. The semantics can be expressed with the flowchart in Fig. 5.3.

In Sect. 5.2.5, we will introduce the (`break` and `continue`) statements that are allowed in a `while` or `for` statement. These statements 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) as illustrated below:

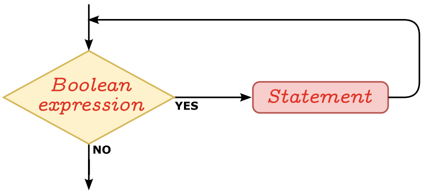


Fig. 5.3 The flowchart illustrating how a `while` statement is executed

```
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.

5.2.2 Examples with the `while` Statement

Example 1: Finding the Average of Numbers in a List

Let us 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:

$$\text{avg}(L) = \frac{1}{N}\sum_{i=1}^{N} L_i, \qquad (5.2)$$

where L_i is the ith number in the list L and N is the number of items in L.

Let us look at the algorithm before implementing the solution in Python (note that, in the equation, indexing starts with 1, compare this with the implementation):

Algorithm 5.1 Average of all Elements of a List of Numbers

```
Input: $L$—A list of $N$ numbers
Output: $avg$—The average of numbers in $L$
1: Initialize a $total$ variable with value 0
2: Initialize an index variable, $i$, with value 0
3: While $i < N$ do execute steps 4–5
4:     $total \leftarrow total + L[i]$
5:     $i \leftarrow i + 1$
6: $avg \leftarrow total/N$
7: return $avg$
```

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

Here is the implementation in Python:

Hands-on Code 5.4

https://pp4e.online/c5s4

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

total = 0                # Step 1
i = 0                    # Step 2

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

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

Output

```
1215.25
```

Have you noticed the remarkable similarity between Python code and the algorithm? Python's principles of simplicity make it significantly easier to write code that closely resembles pseudo-code.

Example 2: Standard Deviation

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

$$\mathrm{std}(L) = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(L_i - \mathrm{avg}(L))^2}. \tag{5.3}$$

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

Hands-on Code 5.5

https://pp4e.online/c5s5

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

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

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

avg = total / N

# CALCULATE THE STD NOW
total = 0
i = 0

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

std = (total / N)**0.5

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

Output

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

Exercise

Write the algorithm of the second example as a pseudo-code.

Example 3: Factorial

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

$$n! = \begin{cases} n \times (n-1) \times \cdots 2 \times 1, & \text{if } n > 0 \\ 1, & \text{if } n = 0. \end{cases} \tag{5.4}$$

Let us first write down the algorithm:

Algorithm 5.2 Factorial Computation

```
Input: $n$—A non negative integer
Output: $n\_factorial$—The factorial if $n$
1: $n\_factorial \leftarrow 1$
2: If $n > 0$ then execute steps 3–6
3:     Initialize an index variable, $i$, with 1
4:     While $i \leq n$ do execute steps 5–6:
5:         $n\_factorial \leftarrow n\_factorial \times i$
6:         $i \leftarrow i + 1$
7: return $n\_factorial$
```

which can be implemented in Python as follows:

Hands-on Code 5.6

https://pp4e.online/c5s6

```python
# Let us set an n value:
n = 5

n_factorial = 1          # Step 1
if n > 0:                # 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)
```

Output

```
n! is 120
```

Exercise

Modify our factorial implementation so that the variable `i` starts from `n`.

Example 4: Computing Sine with Taylor Expansion

Let us assume that we want to compute $\sin(x)$, for a given value of x, up to a given precision. Actually, today's CPUs have embedded math coprocessors that calculate Sine values at a microcode level. We will not use this convenience and perform the computation ourselves.

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

$$\sin(x) \approx x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} \cdots , \tag{5.5}$$

which could also be written as

$$\sum_{k=0}^{\infty} \frac{(-1)^k}{(2k+1)!} x^{2k+1}. \tag{5.6}$$

Let us implement this. We will use the factorial code we have developed above as part of 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 code is readable, but at the same time, it is quite inefficient due to the following aspects:

- `2*k+1` is computed several times. Do we need `2*k+1` at all?
- It uses factorial computation that does not make use of its preceding computations. For example, 9! is actually $9 \times 8 \times 7!$ and 7! could be reused while computing 9!. Therefore, there is significant inefficiency regarding the use of the factorial in the nested iteration.
- Similar to the inefficiency of factorial computation, the computation of $x^{(n+2)}$ does not make use of the already computed x^n value.
- There is a similar problem with $(-1)^{\mathrm{k}}$. Do we need to compute $(-1)^{\mathrm{k}}$ to get an alternating sign? Can there be a simpler way to program this?

Let us investigate the ratio between two consecutive terms ($k = n$) and ($k = n + 1$) to see what changes in each iteration:

$$\frac{term_{n+1}}{term_n} = \frac{(-1)^{(n+1)} x^{2(n+1)+1}}{(2(n+1)+1)!} \times \frac{(2n+1)!}{(-1)^n x^{2n+1}} = \frac{-x^2}{(2n+2)(2n+3)}. \tag{5.7}$$

This is interesting because it suggests 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 find that there is nothing magical about $(2n + 1)$. It can be simply called $(d \equiv 2n + 1)$ which increments by 2 for each consecutive term. Taking these into account, we can have the following:

$$\frac{term_{n+1}}{term_n} = \frac{-x^2}{(d+1)(d+2)}. \tag{5.8}$$

Now, let us see the more efficient implementation:

Hands-on Code 5.7

https://pp4e.online/c5s7

```
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))
```

Output

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

5.2.3 The `for` Statement

The syntax of the `for` statement is:

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

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 question “Are we at the end?”.
3. Provide the next value in the line.

The `for` statement is a mechanism that allows traversing all values provided by an *Iterator*, assigning each value to the *Variable* one at a time and then executing the given *Statement* for each value.

The semantics of the for statement is displayed in Fig. 5.4.

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. The sequence starts at 0 (default) and with increments of 1 (default), continues until a given stop number, excluding the stop number:

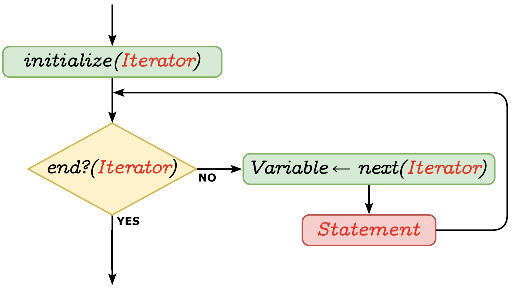


Fig. 5.4 The flowchart illustrating how a `for` statement is executed

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 increment

3. Users can define their own iterators. Since this is an introductory book, we will not cover how this is done.

5.2.4 Examples with the `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.

Hands-on Code 5.8

https://pp4e.online/c5s8

```
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)
```

Output

```
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 a property: For example, a clear sky in an image is actually many pixels with the color “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]`. As an example, the text `"aaaaaaxxxxmyyyaaaassssssssstttuivvvv"` should be encoded as

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

Let us write a Python code that produces this encoding for us:

Hands-on Code 5.9

https://pp4e.online/c5s9

```python
text = "aaaaaaxxxxmyyyaaaassssssssstttuivvvv"
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)
```

Output

```
[['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 ordered list: The ith element of the list stores the old position where the ith element of the new sequence will come from. In our implementation, counting starts from 0 (zero).

Hands-on Code 5.10

https://pp4e.online/c5s10

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

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

print(new_list)
```

Output

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

Example 4: List Split

In this example, 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.

Hands-on Code 5.11

https://pp4e.online/c5s11

```
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)
```

Output

```
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**:

$$\mathbf{u} \cdot \mathbf{v} = \sum_{i=1}^{n} u_i v_i \,, \tag{5.9}$$

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

Hands-on Code 5.12

https://pp4e.online/c5s12

```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)
```

Output

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

Exercise

Implement the 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:

$$\mathbf{u} \cdot \mathbf{v} = ||\mathbf{u}||\ ||\mathbf{v}||\ \cos(\theta), \tag{5.10}$$

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

$$||\mathbf{u}|| = \sqrt{\mathbf{u} \cdot \mathbf{u}} = \sqrt{\sum_{i=1}^{n} u_i^2}. \tag{5.11}$$

The angle can be derived from these as follows:

$$\theta = \arccos\left(\frac{\mathbf{u} \cdot \mathbf{v}}{||\mathbf{u}||\ ||\mathbf{v}||}\right). \tag{5.12}$$

Let us implement this:

Hands-on Code 5.13

https://pp4e.online/c5s13

```python
from math import sqrt, acos

# Define two vectors. These 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)
```

Output

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

Example 7: Matrix Multiplication

Our last example is about matrix multiplication. A matrix A can be multiplied by a 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 the row count equals that of A and the column count equals that of B:

$$(A \cdot B)_{ij} = \sum_{k=0}^{m-1} A_{ik} \cdot B_{kj}. \tag{5.13}$$

As we discussed in the previous chapter, since Python does not provide two-dimensional containers, but a very flexible list container, matrices are mostly represented as lists of lists each of which represents a row. For example, a 3 × 4 matrix:

$$\begin{pmatrix} -2 & 3 & 5 & -1 \\ 0 & 3 & 10 & -7 \\ 11 & 0 & 0 & -8 \end{pmatrix}, \tag{5.14}$$

can be represented as:

```python
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 the two matrices given in this representation and prints the result.

Hands-on Code 5.14

https://pp4e.online/c5s14

```
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)
```

Output

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

5.2.5 The `continue` and `break` Statements

It is possible to alter the flow of execution in a `while` or `for` statement using the `continue` or `break` statements.

Let us assume that you are executing a sequence of statements. Somewhere in the process, without waiting for the terminating condition to become `False` in a `while` statement or exhausting all items of the iterator in a `for` statement, you decide to terminate the iteration. The `break` statement exactly serves this purpose: The `while` (or `for`) statement is stopped immediately and the execution continues with the statement after the `while` (or `for`) statement.

Similarly, at some point in the process, if you decide not to execute the remaining statements within a group, and instead want to continue testing the termination condition in the case of a `while` statement or proceed with the next item in the case of a `for` statement, you can utilize the `continue` statement. See Figs. 5.5 and 5.6 for how the `continue` and `break` statements change the flow of execution.

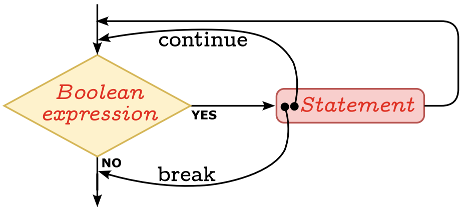


Fig. 5.5 How `continue` and `break` statements change execution in a `while` statement

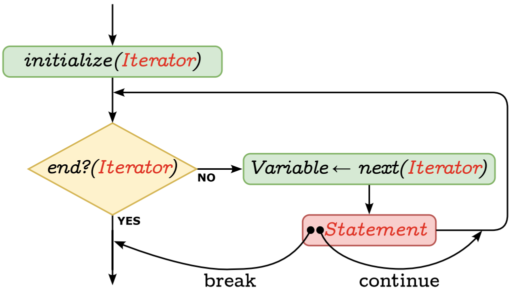


Fig. 5.6 How `continue` and `break` statements change execution in a `for` statement

5.2.6 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 including a variable,
2. A colon or vertical bar separator, and
3. A logical predicate.

For example:

$$\{x^3 \mid x \in \{0, 1, \ldots, 7\}\} \tag{5.15}$$

defines the following set:

$$\{0, 1, 8, 27, 64, 125, 216, 343\} \tag{5.16}$$

A more elaborate example is provided below:

$$\left\{x^3 \mid x \in \{0, 1, \ldots, 7\} \wedge (x \bmod 2) = 1\right\} \tag{5.17}$$

defining the following set:

$$\{1, 27, 125, 343\} \tag{5.18}$$

Python is one of the few languages that provides a notation with the same semantics for both sets and lists (the one for lists is called *list comprehension*):

```
>>> [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}
```

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 or container. Here are more examples on matrices.

Hands-on Code 5.15

https://pp4e.online/c5s15

```
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)] )
```

Output

```
[[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]]
```

5.3 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.
- `break` and `continue` statements to change the flow of iterations.
- Set and list comprehension as a form of iterative computation in an expression.

5.4 Further Reading

- “Managing the Size of a Problem” (Chap. 4) of G. Üçoluk, S. Kalkan, Introduction to Programming Concepts with Case Studies in Python, Springer, 2012 [1].

5.5 Exercises

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

1. Implement the `while` statement examples in Sect. 5.2.2 using `for` statement:
   a. Calculating the average of a list of numbers.
   b. Calculating the standard deviation of a list of numbers.
   c. Calculating the factorial of a number.
   d. Taylor series expansion of sin(x).

2. Implement the `for` statement examples in Sect. 5.2.4 using `while` statement:
   a. Words with vowels and consonants.
   b. Run-length encoding.
   c. Permutation.
   d. List split.
   e. Matrix multiplication.

3. 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]
```

4. Write a Python code that removes duplicate items in a list. For example, `[12, 3, 4, 12]` should be changed to `[12, 3, 4]`. The order of the items should not change.
5. Write a Python code that replaces a nested list of numbers with the average of the numbers. For example, `[[1, 3], [4, 5, 6], 7, [10, 20]]` should yield `[2, 5, 7, 15]`.
6. Write a Python code that finds all integers that divide a given integer without a remainder. For example, for the number `21`, your code should compute `[1, 3, 7]`.
7. Write a Python code that uses a `for`/`while` loop to convert a positive integer less than 128 into a binary string representing the binary representation of the number. For example, the number 21 should be converted into the binary string `"00010101"`.
8. Write a Python code that converts a binary string like `"00010101"` into integer.
9. Given a list of integers in a variable named `Subjects` and another list of integers in a variable `AvoidMultiples`, print the sum of elements which are in `Subjects` and are **not** multiples of any element in `AvoidMultiples`.
   Hint: Make use of list comprehension.
   Example:

```
Subjects = [10, 12, 70, 75, 42, 17, 28, -24, 77, -12, 13]
AvoidMultiples = [3, 7]
```

   The value to be printed:

```
40
```

10. Write a program that takes a list of strings and prints the frequency of each word in the list.
    Hint: Make use of a dictionary.
    Example: For the following list,

```
["apple", "banana", "apple", "cherry", "orange", "banana",
"cherry", "elderberry", "orange", "fig", "fig", "fig"]
```

    Your code should print the following:

```
Frequency of words:
apple: 2
banana: 2
cherry: 2
orange: 2
```

```
elderberry: 1
fig: 3
```

11. You are given a dictionary of *food* and *price* information, stored under a variable `Menu`. Here is an example:

```
Menu = {"soup":22.5, "fries":15.0, "coke":10.0,  "beer":30.0,
        "bread":5.0, "hamburger":65.0, "scrambled egg":27.5,
        "coffee":18.0, "pastry":23.5, "large pizza":70.0,
        "medium pizza":40.0, "small pizza":25.0, "salad":7.75}
```

Write a program that takes a list of items (from the menu), i.e., the bill as:

```
["hamburger", "fries", "beer", "medium pizza", "beer"]
```

and then, prints the sum as follows:

```
180.0
```

Now, modify your program so that the same bill above can be entered as:

```
["hamburger", "fries", (2,"beer"), "medium pizza"]
```

12. A dictionary named `Nominals` contains nominal values for blood sample measurements. Each nominal value is a tuple with two elements. The first element represents the lower end of the range, and the second element represents the upper end of the range. Here is an example:

```
Nominals = {"EO%":(0.5,7),"BA%":(0.0,1),"RBC":(3.5,5.7),
            "WBC":(4,10),"LY#":(0.8,4),"LY%":(20,40),
            "MO%":(3,12),"MO#":(0.12,1.2),"NE#":(2,7),
            "EO#":(0.02,0.7),"BA#":(0,0.1),"NE%":(50,70),
            "HGB":(13.2,17.3),"HCT":(39,52),"MCV":(80,100),
            "RDW":(11.8,14.3),"MPV":(6.5,12),"PCT":(0.1,0.4),
            "PDW":(0.15,17),"PLT":(150,450),"IMG#":(0,0.02),
            "IMG%":(0,0.2),"MCH":(27,35),"MCHC":(32,36)}
```

Your program takes a list of measurements (*name*, *value*) as tuples and prints each value with a suffix of `"LOW"`, `"NORMAL"`, or `"HIGH"` based on the nominal values.

Example: For the following list:

```
[("WBC",7.4),("LY#",4.61),("MO#",0.58),("NE#",4.78),
 ("EO#",0.35),("BA#",0.04),("LY%",21.9),("MO%",7.9),
 ("NE%",64.9),("EO%",4.7),("BA%",0.6),("RBC",4.8),
 ("HGB",13.2),("HCT",40.3),("MCV",83.9),("MCH",27.4),
 ("MCHC",32.7),("RDW",15.4),("PLT",56.0),("MPV",13.5),
 ("PCT",0.02),("PDW",0.14),("IMG#",0.01),("IMG%",0.002)]
```

Your program should print:

```
WBC 7.4 NORMAL
LY# 4.61 HIGH
MO# 0.58 NORMAL
NE# 4.78 NORMAL
EO# 0.35 NORMAL
BA# 0.04 NORMAL
LY% 21.9 NORMAL
MO% 7.9 NORMAL
NE% 64.9 NORMAL
EO% 4.7 NORMAL
BA% 0.6 NORMAL
RBC 4.8 NORMAL
HGB 13.2 NORMAL
HCT 40.3 NORMAL
MCV 83.9 NORMAL
MCH 27.4 NORMAL
MCHC 32.7 NORMAL
RDW 15.4 HIGH
PLT 56.0 LOW
MPV 13.5 HIGH
PCT 0.02 LOW
PDW 0.14 LOW
IMG# 0.01 NORMAL
IMG% 0.002 NORMAL
```

13. Write a program that inputs a list of numbers `L` and a positive integer `n` that is less than the length of `L`. Considering each element of `L` only once, construct a sorted list of the biggest `n` elements of `L` and print it. You are **not allowed** to make a copy of the list and/or sort it.
14. Write a program that inputs an integer ($0 \le N \le 9999$) from the user and converts that number into its written form.

    Example: For the numbers 1606 and 111, the outputs should be:

    ```
    onethousandsixhundredandsix
    onehundredandeleven
    ```

15. Write a program that inputs an integer ($0 < N < 4000$) and converts that number into its Roman value.

    Example: For the numbers 928 and 800, the outputs should be:

    ```
    CMXXVIII
    DCCC
    ```

16. Write a program that inputs a Roman number ($0 < N < 4000$) as a string and converts it to its decimal value.

17. Write a program that inputs three numbers, representing a day, a month, and a year, and returns the weekday. You can assume that year > 1789.

    Example: For the numbers 15, 12, 1999, the output should be:

    ```
    WEDNESDAY
    ```

18. A *quadratic equation* has the following form:

    $$ax^2 + bx + c = 0.$$

    To find x that satisfies such an equation, first the *discriminant* (Δ) is computed:

    $$\Delta = b^2 - 4ac.$$

    Based on Δ, we can reason about the solutions of the quadratic equation:

    If $\Delta < 0$: There is no real solution.
    If $\Delta = 0$: There is a single solution: $x_0 = -\frac{b}{2a}$.
    If $\Delta > 0$: There are two distinct solutions: $x_{1,2} = \frac{-b \pm \sqrt{\Delta}}{2a}$.

    Write a program that inputs values for a, b, and c from the user, consecutively. If there are no real solutions, print `"NO ROOT"`. Otherwise, print the solution(s) on the screen.
19. You are given an ordered list of (frequency, station-name) tuples of radio stations assigned to a variable `MyRadio`. An example for `MyRadio` would be:

    ```
    MyRadio = [(171, 'Morocco'), (198, 'Ouargla'), (227, 'Polskie'),
               (234, 'Luxembourg'), (540, 'Solt'), (585, 'Madrid'),
               (621, 'Batra'), (683, 'Beograd'), (801, 'Ajloun'),
               (909, 'BBC'), (1089, 'Rossii'), (1440, 'Dammam'),
               (1521, 'Duba')]
    ```

    Given this variable, write a program that:

    - Inputs from the user a frequency (not necessarily one of those above).
    - Finds the radio station that has the closest frequency to the input frequency.
    - Prints the found station's name, prefixed with a word `"Radio"`, separated with a single blank.

    Example: Given input 580, the output should be:

    ```
    Radio Madrid
    ```

20. Solve the previous exercise with the following restriction: If the length of `MyRadio` is N, you have the right to look-up its content (with `MyRadio[■]`) at most $\log_2(N) + 1$ times.
21. Write a program that inputs a list of 2D coordinates of 4 points:
    `[[x₁,y₁], [x₂,y₂], [x₃,y₃], [x₄,y₄]]`
    where x_i and y_i are numeric values.

Your program should determine whether these four points are on a circle or not and output `"CIRCLE"` or `"NOT CIRCLE"`, respectively.

Example: Given the following list,

```
[[1.75,-1.918698582794336],[3.1,4.948187929913334],
 [0.5,4.372281323269014],[5.5,-0.30277563773199456]])
```

Your program should output:

```
CIRCLE
```

Hints and notes:

- The four points are distinct.
- Use floating-point arithmetic. If you need to do an equality test with floating points, **do not use** the `'=='` comparison operator. You should consider two values x and x' to be equal if:

$$\left|\frac{x - x'}{x + x'}\right| \leq 10^{-3}.$$

22. Consider your high school knowledge of lenses. Let us recall the main governing equation:

$$\frac{1}{f} = \frac{1}{i} + \frac{1}{o},$$

with the magnification as $-i/o$. To refresh your knowledge, please check Wikipedia for “Lens (optics)”.

Consider the x-axis. Given that, at the origin, there is a real object with height h cm. The image is along the positive y-axis. At $x = x_1$, a lens with a diopter of d_1 is placed, and at $x = x_2$ another lens with a diopter of d_2 is positioned. Assume that x_1 and x_2 are both positive values whereas $d_{1,2}$ can be positive or negative, for converging or diverging lenses, respectively.

Write a program that inputs a list with exactly five elements, namely, `[`$\texttt{d}_1$`,` $\texttt{x}_1$`,` $\texttt{d}_2$`,` $\texttt{x}_2$`, h]`, and outputs a 3-tuple `(`$position$`,` $type$`,` $height$`)` for the *image position*, the *image type* (`"virtual"` or `"real"`), and the *image height* formed by the rays that are refracted by both lenses (there is only one such image). You can assume that $x_1 < x_2$.

Reference

[1] G. Üçoluk, S. Kalkan, *Introduction to Programming Concepts with Case Studies in Python* (Springer Science & Business Media, 2012)

6 Functions

A *function* (or sometimes synonymously referred to as subroutine, procedure, or sub-program) is a group of actions under a given name that performs a specific task. `len()`, `print()`, `abs()`, `type()` are functions that we have covered before. Python adopts a syntax that resembles mathematical functions.

A function, i.e., "named" pieces of code, can receive *parameters*. These parameters are variables that carry information from the "calling point" (e.g., when you write `N = len("The number of chars")`) to the callee, i.e., the piece of code that *defined* the function (`len()` in our example). The life span of these variables (parameters) is as long as the function is active (for that call).

6.1 Why Define Functions?

There are four main reasons for using functions while solving world problems with a programming language:

1- **Reusability**
A function is defined at a single point in a program, but it can be called from several different points. To better illustrate this, let us consider an example: Assume that you need to calculate the run-length encoding of a string (the example code from Sect. 5.2.4) in your program for *n* different strings in different positions in your code. In a programming language without functions, you would have to copy-paste that run-length encoding code piece in *n* different places. Thanks to functions, we can place the code piece in a function once (named `encodeRunLen` for example) and use it in *n* places via simple function calls like `enc = encodeRunLen(inpstr)`.

2- **Maintenance**
Using functions makes it easier to make updates to an algorithm. Let us explain this with the run-length encoding example: Assume that we found an error in our calculation or that we discovered a better algorithm than the one we are using. Without functions, you would need to go through all your code, find the locations where your run-length encoding algorithm is implemented, and update each and every one of them. This is very impractical and tedious. With functions, it is sufficient just to change the function that implements the run-length encoding.

S. Kalkan et al., *Programming with Python for Engineers*,
https://doi.org/10.1007/978-3-031-57148-0_6

3- **Structuredness**
Let us have a look at a perfectly valid Python expression:

```
(b if b<c else c) if (a if a<b else b)<(b if b<c else c) else (a if a<b else b)
```

which certainly takes some time for a human to parse and understand but it just performs the following:

```
max(min(a,b), min(b,c))
```

The version with the `min` and `max` functions has a structure that is easier to parse and follow. This in turn makes the code more readable: Even if the definitions of `max` and `min` are not given yet, one can still grasp what the expression is up to. It is not only because the expression is smaller, but because function names can give clues about the purposes of the actions.

4- **Benefits of the functional programming paradigm**
If a programmer sticks to the functional paradigm (briefly covered in Sect. 2.4.4), then he or she has to plot a functional decomposition so that the solution of the problem is obtained by means of functional composition, something that is well known and understood in mathematics. This type of solution does not suffer from unexpected side effects or variable alterations (e.g., due to the aliasing problem covered in Sect. 4.4.2); therefore, testing and debugging become easier.

Another benefit of the functional paradigm is *recursion*: In other words, while defining a function, we can call the function that is being defined—a seemingly confusing concept, which we will cover in this chapter. Moreover, we can define *higher-order functions* that take functions as input and apply them to a sequence of data. All these benefits come free with using functions, as we will illustrate below.

Even when we do not limit ourselves to the functional programming paradigm, it is essential to use functions in professional programming. Generic and general algorithms coded as functions have become very valuable because they can be reused again and again in new programs. Collections formed by such general purpose functions are called *libraries*. Libraries have been created for almost all application areas over decades.

6.2 Defining Functions

For defining a function, Python adopts the following syntax:

`def` $\boxed{\mathit{FunctionName}}$ `(` $\boxed{\mathit{Parameter}_1}$ `,` $\boxed{\mathit{Parameter}_2}$ `,` $\cdots$ `)` `:` $\boxed{\mathit{Statement}}$

The *Statement*, which can be replaced by multiple statements as we did with the other compound statements, can make use of the *Parameter*s as variables to access the actual arguments (data) sent to the function at the call-time. If the function is going to give a resulting value in return to the call, this value is provided by a `return` statement.

FunctionName is the name of the function being defined. Python follows the same naming rules and conventions used for naming variables—if you do not remember them, we recommend you go back and check the restrictions for naming variables in Sect. 4.4.3.

Here is a straightforward example. Let us assume that we want to implement the following mathematical function:

$$F_{gravity}(m_1, m_2, r) = G\frac{m_1 m_2}{r^2}. \tag{6.1}$$

The following is the one-line Python code that defines it:

```
def F_gravity(m_1, m_2, r): return G * m_1 * m_2 / (r * r)
```

This function, `F_gravity()`, returns a value (i.e., the result of `G * m_1 * m_2 / (r * r)`). This is achieved with the `return` statement. As you may have observed, the values m_1, m_2 and r are provided as the first, second and third parameters, named as `m_1`, `m_2` and `r`, respectively.

What about `G`? It is not provided in the arguments and is used in the return expression for its value. If the definition were a multi-statement definition (indented multi-statements following the `def` line), then Python would seek for a defined value for `G` among them. As `G` is not defined in the function, Python would look for the variable in the `global` environment, the environment in which `F_gravity()` is called.

Therefore, before calling `F_gravity()`, we must make sure that a value is set for a global variable `G` (which apparently is the *Gravitational Constant*), e.g., as follows:

Hands-on Code 6.1

https://pp4e.online/c6s1

```
def F_gravity(m_1, m_2, r): return G * m_1 * m_2 / (r * r)

G = 6.67408E-11
print(F_gravity(1000, 20, 0.5), "Newton")
```

Output

```
5.3392640000000006e-06 Newton
```

We will cover these concepts in detail in the **Scope of Variables** section (Sect. 6.4).

6.3 Passing Parameters to Functions

When a function is called, first its arguments are evaluated. The function will receive the results of these evaluations. For this, the parameter variables used in the definition are created, and set to the results of the argument evaluations. Then, the statement that follows the column (`:`) after the argument parenthesis is executed. Certainly, this statement can and will make use of the parameter variables (which are set to the results of the argument evaluations).

Let us look at an example:

Hands-on Code 6.2

https://pp4e.online/c6s2

```
def F_gravity(m_1, m_2, r): return G * m_1 * m_2 / (r * r)

G = 6.67408E-11

S = 100
Q = 2000

print(F_gravity(Q+S, Q-S, ((504.3-66.1)**2+(351.1-7.7)**2)**0.5))
```

Output

```
8.59177215925003e-10
```

This is a legitimate piece of code that performs a sequence of actions in a specific order.

- Defines the function `F_gravity()`.
- Sets the value of the global variable `G` and `S` and `Q`.
- Calls the (internally defined) `print()` function that takes one argument. This argument is a function call to `F_gravity()`. Therefore, to have a value to print, this function call must be performed.
- A call to `F_gravity()` is prepared: It has three arguments, each has to be evaluated and boiled down to a value. For this, in a left-to-right order, the arguments are evaluated:
  - `Q+S` is evaluated, and the result is stored in `m_1`.
  - `Q-S` is evaluated, and the result is stored in `m_2`.
  - The Euclidean distance, $\sqrt{(504.3-66.1)^2+(351.1-7.7)^2}$, is evaluated and the result is stored in `r`.
- Since all arguments are evaluated, the statement defining the function `F_gravity()` can be executed.
- This statement is "`return G * m_1 * m_2 / (r * r)`", which requires the expression following the `return` keyword to be evaluated and the result to be returned as the result of the call to the function.
- Now the `print()` function has got its parameter evaluated. The defining action of the `print()` function can be carried out (this is an internally defined function, i.e., we cannot see its definition, but the documents defining the language explain it).

To summarize, prior to the function call, there is a preparation phase where all arguments are evaluated and assigned a value. This process is known in Computer Science as *call-by-value*.

6.3.1 Default Parameters

While defining a function, we can provide default values for parameters as follows:

$$\texttt{def}\ \boxed{FunctionName}\ \texttt{(}\ \boxed{Parameter_1 = Exp_1}\ \texttt{,}\ \boxed{Parameter_2 = Exp_2}\ \texttt{,}\ \cdots\ \texttt{)}\ \texttt{:}\ \boxed{Statement}$$

where Exp_i is an expression. When calling the function, we may omit providing the parameters for which the function has default values.

Let us take a look at a simple example:

Hands-on Code 6.3

https://pp4e.online/c6s3

```
def norm(x1, x2, norm_type="L1", verbose=True):
    result = None
    if norm_type == "L1": result = abs(x1) + abs(x2)
    elif norm_type == "L2": result = (x1**2 + x2**2)**0.5
    elif verbose: print("Norm not known:", norm_type)

    if verbose: print(f"The {norm_type} norm of", [x1, x2], "is:", result)
    return result

norm(3, 4)                                      # CASE 1
norm(3, 4, "L2")                                # CASE 2
norm(3, 4, norm_type="L2")                      # CASE 3
norm(3, 4, verbose=False)                       # CASE 4: Does not print the value
norm(3, 4, verbose=True, norm_type="L1")        # CASE 5
norm(x2=3, x1=4, verbose=True, norm_type="L1")  # CASE 6
```

Output

```
The L1 norm of [3, 4] is: 7
The L2 norm of [3, 4] is: 5.0
The L2 norm of [3, 4] is: 5.0
The L1 norm of [3, 4] is: 7
The L1 norm of [4, 3] is: 7
```

Let us look at some cases that lead to errors:

Hands-on Code 6.4

https://pp4e.online/c6s4

```
norm(3, 4, norm_type="L1", True)                # CASE 7
```

Output

```
File "<ipython-input-19-128dc3e43256>", line 2
    norm(3, 4, norm_type="L1", True)                # CASE 7
                              ^
SyntaxError: positional argument follows keyword argument
```

In this example, you should notice the following crucial points:

- We can call a function without providing any value for the parameters that have default values (`CASE 1`-`CASE 4`).
- We can change the order of default parameters when we call the function (`CASE 5` and `CASE 6`).
- When calling a function, we can give values to non-default parameters by using their names (`CASE 6`).
- Parameters after default parameters also need to be given names, while both defining and calling the function (`CASE 7`).

6.4 Scope of Variables

We extensively use variables while programming our solutions. Not all variables are accessible from everywhere in our solutions: There is a governing set of rules that determine which parts of a code can access a variable. The code segment from which a variable is accessible is called its *scope*.

In Python, there are four categories of scopes. As will be explained shortly, they are abbreviated with their initials, which then combine to form the rule mnemonic LEGB which stands for:

*L*ocal Scope $<$ *E*nclosing Scope $<$ *G*lobal Scope $<$ *B*uilt-in Scope

1- Local Scope

Defining a variable in a function makes it *local* in Python. The variable `G` in the following function is a local variable of the function `F_gravity()`:

```python
def F_gravity(m_1, m_2, r):

    G = 6.67408E-11
    return G * m_1 * m_2 / (r * r)
```

Local variables are the most "private" ones: Any code that is external to that function cannot see it. If you make an assignment anywhere in the function to a variable, it is registered as a local variable. If you attempt to refer to such a local variable prior to assignment, Python will generate a run-time error.

The local variables are created each time the function is called (it is activated), and they are annihilated (destructed) the moment the function returns (finishes).

2- Enclosing Scope

As we mentioned before, it is possible to define functions inside functions. They are coined as *local functions* and can refer to (but not change) the values of the local variables of its parent function (the function in which the local function is defined). The scope of the outer function is the enclosing scope for the inner function. Though the parent function's variables are visible to the local function, the converse is not true. The parent function cannot access its local function's local variables. Therefore, the accessibility is from inward to outward and for referring (reading) purposes only.

This is illustrated below (`distance()` and `particle_list` are not defined for brevity):

```python
def n_body_interaction(particle_list): # a function enclosing F_gravity
    G = 6.67408E-11 # G is defined in an enclosing scope of the function F_gravity

    def F_gravity(m_1, m_2, r): return G * m_1 * m_2 / (r * r)

    for p1 in particle_list:
        p1['forces'] = []
        for p2 in particle_list:
            p1['forces'] += F_gravity(p1['mass'], p2['mass'], distance(p1['position'], p2['position']))
```

3- Global Scope

The default scope provided by the Python interpreter is the *global scope*. If a variable is not defined in a function, then it is a *global variable*, i.e., a variable in the global scope. A global variable can be accessed both outside and inside of any function. In the following, the variable `G` and the function `F_gravity()` are in the global scope:

```python
def F_gravity(m_1, m_2, r): return G * m_1 * m_2 / (r * r)

G = 6.67408E-11
```

If a global variable is accessed inside a function, it cannot be altered there. If it is altered anywhere in the function, this is recognized at the declaration-time and that variable is declared (created) as a local variable. A local variable that has the same name as a global variable conceals the global variable (because of the LEGB ordering among the scopes).

If you want to change the value of a global variable inside a function, then you have to *explicitly* declare it as a global variable by a statement:

`global` *Variable*

Then an assignment to *Variable* will not register it as a local variable but will merely refer to the global *Variable*. As said, since the system will recognize them automatically, there is no need to use the `global` statement for global variables which are referred to only for their values. However, it is a good programming habit to do so to enhance readability and prevent accidental local variable creations (due to careless assignments to global variables).

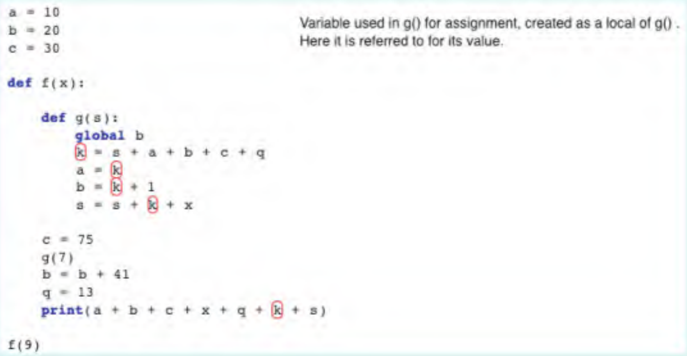
4- Built-in Scope

The *built-in scope* is a special Python scope that is created or loaded whenever you run a script or open an interactive session. This is actually the scope in which the Python interpreter starts. It contains names such as keywords, built-in functions, exceptions, and some other attributes. Entities in this scope are available from everywhere in your code.

Interactive Example 6.1

https://pp4e.online/c6d1

In the interactive Jupyter Notebook page of the chapter, an interactive example is provided to get a better understanding of different scopes. You can hover over the variables with your mouse to see where each variable is accessible.



```
a = 10
b = 20
c = 30

def f(x):

    def g(s):
        global b
        k = s + a + b + c + q
        a = k
        b = k + 1
        s = s + k + x

    c = 75
    g(7)
    b = b + 41
    q = 13
    print(a + b + c + x + q + k + s)

f(9)
```

Variable used in g() for assignment, created as a local of g() .
Here it is referred to for its value.

A variable occurrence has different meanings for different positions in the program:

1. *First-time occurrence of the variable at the left-hand side of an assignment*: This tells Python to declare a variable in the current scope, the current function, or the global scope. If there is a variable with the same name in an outer scope, it is hidden as long as the new variable is active. This declaration will be active for the scope in which it is created, including statements preceding it.
2. *Second and subsequent occurrences of the variable at the left-hand side of an assignment*: They simply update the previously declared variable.
3. *Occurrences of the variable at the right-hand side of an assignment or any other place in an expression*: Python places the value (content) of the variable in the expression position. The variable is searched in all scopes following the LEGB ordering, and the first found variable is used. If the declaration of the variable is made after its reference in the current scope, Python will raise the error `'Reference before assignment'`.

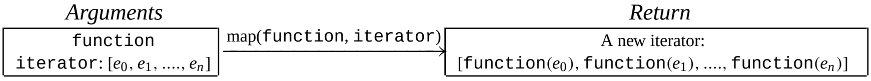
Note that a `global` declaration changes the behavior above. It makes all occurrences refer to the global variable and does not declare a variable in the current scope with the globally defined names.

6.5 Higher-Order Functions

Python provides several flavors of the functional paradigm, one of the most powerful of which is the ability to easily pass functions as parameters to other functions. No special syntax or rule is needed for this: If you use a parameter as if it is a function and when calling the higher-order function, pass a function name as a parameter, Python will handle the rest.

Python provides two convenient higher-order functions for us:

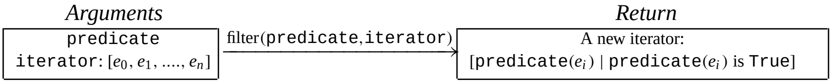
- **map** function: The `map()` function has the following syntax:
  `map(function, iterator)`
  which yields an iterator where each element is the result of applying the `function` on the corresponding element in the `iterator` as illustrated below:

  
  Arguments
`function`
`iterator`: $[e_0, e_1, ..., e_n]$
map(`function`, `iterator`)
Return
A new iterator:
$[$`function`(e_0), `function`(e_1), ..., `function`$(e_n)]$

  Here is a simple example:

```
>>> abs_it = map(abs, [10, -4, 20, -100])
>>> for x in abs_it: print(x)
10
4
20
100
```

- **filter** function: The `filter()` function has a similar syntax to the `map()` function:
  `filter(predicate, iterator)`
  where `predicate` is a function with a Boolean return value. The `filter()` function applies `predicate` to each element of the `Iterator` and only keeps the ones for which `predicate` returns `True`, as illustrated below:

  
  Arguments
`predicate`
`iterator`: $[e_0, e_1, ..., e_n]$
filter(`predicate`,`iterator`)
Return
A new iterator:
[`predicate`(e_i) | `predicate`(e_i) is `True`]

  For example:

```
>>> def positive(x): return x > 0
>>> for x in filter(positive, [10, -4, 20, -100]): print(x)
10
20
```

Of course, the functional capabilities of Python are more diverse than we can cover in this introductory textbook. The reader is referred to Python docs[1] for a more complete coverage. The curious reader is especially recommended to look at *lambda expressions* and the *reduce* function.

[1] https://docs.python.org/3/howto/functional.html.

6.6 Functions in Programming Versus Functions in Mathematics

It is crucial to note that functions have a differentiated meaning compared to functions in mathematics, as listed below:

- In mathematics, a function is a mapping from a domain to a range (target); a function returns a value.

  In Python: Functions may not return a value at all. If so, they are intended to code a parametrized action. A few examples of this are:

  – Drawing a *red* line from (x_0, y_0) to (x_1, y_1).
  – Printing a separating line made of n "minus" signs.
  – Turning off the lights.
  – Sending a digital initialization string to a device connected through the USB port.

- In mathematics, a function definition is global. You do not have function definitions local to some function.

  In Python: Since a function definition is done by a Python *Statement* (or a group of statements—grouped by indentation), it is quite possible to have another definition as part of the defining statements. Such a function can only be usable in the *Statement* group that defines the outer function and accesses its local variables and parameters. Outside of the function, this interior function (called a *local function*) will not be visible or usable.
- In mathematics, an element from the domain is mapped to one and only one target element by the function. The mapping does not change. Moreover, a mathematical function returns a value that depends only on the arguments. Therefore, for the same arguments, the result is the same.

  In Python: Even if such a mapping exists (i.e., a function returns a value), it is possible that the function acts differently due to different settings of the variables (outside of the function) that can be accessed in the function. Consider the F_gravity() function again:

  ```
  def F_gravity(m_1, m_2, r): return G * m_1 * m_2 / (r * r)
  ```

  A different value for G (the Gravitational Constant) will yield a different F_gravity() result even when the parameters remain the same.
- In mathematics, the operation of a function is reflected in the value it returns: It does not have side effects. In other words, a mathematical function does not change the mathematical environment that is external to it (variables or the ways other functions are evaluated).

  In Python: This may not be true. Consider the G value of the F_gravity() function above. Hypothetically, it is possible that, at each call of F_gravity(), the G value is increased slightly (1% for example):

```
>>> def F_gravity(m_1, m_2, r):
...     global G          # since we alter it
...     G = 1.01 * G
...     return G * m_1 * m_2 / (r * r)

>>> G = 6.67408E-11

>>> print(F_gravity(1000, 20, 0.5), "Newton")
5.39265664e-06 Newton

>>> print(F_gravity(1000, 20, 0.5), "Newton")
5.4465832064e-06 Newton

>>> print(F_gravity(1000, 20, 0.5), "Newton")
5.501049038464e-06 Newton
```

In these aspects, a function defined in Python has greater freedom. As previously stated in Sect. 2.4.4, there are paradigms of programming. One of these paradigms, the functional paradigm, promotes function usage to be restricted to the mathematical sense. In other words, the functional paradigm followers deliberately renounce the additional freedom programming languages like Python provide in function definitions and usages with respect to the mathematical sense. The main reason is that deviating from the mathematical sense makes programming more error-prone.

6.7 Recursion

Recursion is a function definition technique in which the definition makes use of calls to the function being defined. Although this might be difficult to grasp at first, it allows compact and more readable code for recursive algorithms, relations, or mathematical functions.

To explain recursion, it is almost customary to start with the infamous example of the recursive definition of factorial. First, let us put it down in mathematical terms:

$$\begin{aligned} N! &= (N-1)! \times N, \qquad \text{for } N \in \mathbf{N}, \quad N > 0 \\ 0! &= 1. \end{aligned}$$

Based on this, we can realize that:

- $6!$ is $6 \times 5!$
- $5!$ is $5 \times 4!$
- $4!$ is $4 \times 3!$
- $3!$ is $3 \times 2!$
- $2!$ is $2 \times 1!$
- $1!$ is $1 \times 0!$
- $0!$ is 1

Substituting any factorial value, based on this rule, one ends up with the following:

$$6! = 6 \times 5 \times 4 \times 3 \times 2 \times 1 \times 1. \qquad (6.2)$$

Making use of $(N-1)!$ while defining $N!$ is making a recursive definition. The Python equivalent of the above recursive mathematical definition of the factorial would be:

```
def factorial(N):

    if N == 0: return 1
    else: return N * factorial(N-1)
```

High-level languages are designed so that each call to the `factorial()` function does not overwrite any previous call(s) even if they are still active (unfinished). For every new call, fresh positions for all local variables are created in the memory without overwriting (forgetting) the former values of variables for the unfinished calls.

From the programmer's point of view, the only concern is the algorithmic answer to the following question:

> Having the recursive call returned with the correct answer to the smaller problem, how can I cook up the answer to the actual problem?

It requires a kind of "stop thinking" (a leap of faith) action: "Stop thinking how the recursive call comes up with the correct answer". Instead, just concentrate on the next step as stated above ("How can I cook up the answer to the actual problem?").

Not to get into an infinite, non-stopping recursive loop, you should be careful about two aspects:

1. The recursive call should be made with a decrescent (getting smaller) argument. In the factorial example, the calculation of $N!$ recursively called $(N-1)!$, i.e., a smaller value than $N!$.
2. The function definition must start with a test for the smallest possible argument case (in the factorial example, this was 0 (zero)) where a value for this case is directly returned (in the factorial example, this was 1 (one)). This is called the *termination condition* or *base condition*.

Let us consider another example: We have a list of unordered numbers. Nothing is pre-known about the magnitude of the numbers. The problem is to find the maximum of the numbers in that list.

Let us say that you are given a list of 900 numbers. For a second, assume that you have a very loving grandpa. Your grandfather has a rule of his own: Not to corrupt you, he will not solve the problem for you but, if you reduce the problem slightly, his heart softens and provides the answer for the smaller problem.

Presumably, the simplest "reduction" that you can do on the 900-element list is to remove an element from it, making it a list of 899 numbers. As far as Python lists are concerned, the simplest of the simple is to remove the first element: If `L` is your list, `L[0]` is the first element and `L[1:]` is the list of remaining elements. So, the trick is to remove the first element, take `L[1:]` to your grandfather (that is, making the recursive call), and ask for the solution. Your grandfather is very cooperative and gives you the solution (the highest number) for the list (`L[1:]`) you gave him.

We are not finished yet: On the one hand, you have what your grandfather returned to you as the highest number of the slightly reduced list (`L[1:]`), and on the other hand, you have the removed (first) element `L[0]`. Now, which one will you return as the solution for the 900 number list? Certainly, the larger one, since we are interested in finding the largest number in `L`.

Here is the solution in Python:

Hands-on Code 6.5

https://pp4e.online/c6s5

```python
def find_max(L):
    if len(L) == 1: return L[0]   # Terminating condition
    else:
        grandpa_result = find_max(L[1:])   # Recursion is on L with the 0th element removed
        if grandpa_result > L[0]: return grandpa_result
        else: return L[0]

# Let's try with a simple list -- try changing M and check the result
M = [-100, 10, -10, 100]
print("The maximum element of M is:", find_max(M))
```

Output

```
The maximum element of M is: 100
```

As you may have recognized, the definition started with testing for the smallest case. That is, a single-element list for which the result (the biggest number in the list) is that single number itself.

Recursion helps writing elegant functions. However, it is not free. The price is in the time and memory overhead spent on creating a new function call. Time may be something tolerable, but memory is limited. For functions that make thousands of recursive calls, you should better seek algorithmic solutions that do not lean on recursion. Python has a default for the recursion depth, which is set to 1000. This can be altered.

Any recursive function can be implemented using iterations. The reverse is also true: Any iterative function can be implemented using recursion. The definition of the problem or the algorithm generally provides hints for whether recursion or iteration is better for implementing your solution.

6.8 Function Examples

Example 1: Computing average and standard deviation

In the previous chapter, we provided iterative definitions for calculating the average and the standard deviation of a list of numbers. Let us look at these definitions again with functions. For the sake of ease, the mathematical definitions are provided again. Moreover, we will use this opportunity to talk more about iterations as well.

The average of a set (S) of numbers is calculated by dividing the summation of the values in the set by their number:

$$avg(S) = \frac{1}{|S|}\sum_{x\in S} x, \tag{6.3}$$

which can be coded as:

Hands-on Code 6.6

https://pp4e.online/c6s6

```python
def avg(S):
    total = 0.0
    for x in S: total += x
    return total/len(S)

S = [11, 2, 9, 6]
print("The avg of S:", avg(S))
```

Output

```
The avg of S: 7.0
```

This is quite elegant. As you can see, it works at a higher abstraction level. This can be done because the `in` operator is very handy, and when used with an iterator, it iterates over the elements of the list.

Sometimes it is more desirable to work with the indices to reach out to each element. Recall that the indexing of container elements starts at zero. Therefore, a more index-based variation of the same formula would be:

$$avg(S) = \frac{1}{|S|} \sum_{i=0}^{|S|-1} S_i, \tag{6.4}$$

which can be implemented as follows:

Hands-on Code 6.7

https://pp4e.online/c6s7

```python
def avg(S):
    total = 0.0
    for i in range(0, len(S)): total += S[i]
    return total/len(S)

S = [11, 2, 9, 6]
print("The avg of S:", avg(S))
```

This is almost okay, but not perfect. `len()` is a built-in function that returns the element count of any container (string, list, tuple). As you have observed, `len(S)` is calculated twice, which is inefficient. Function calls have a price, time- and space-wise; therefore, unnecessary calls should be avoided. To fix this, we change the code to calculate the length once and store the result for further usage:

Hands-on Code 6.8

https://pp4e.online/c6s8

```python
def avg(S):
    total = 0.0
    length = len(S)
    for i in range(0, length): total += S[i]
    return total/length

S = [11, 2, 9, 6]
print("The avg of S:", avg(S))
```

Output

```
The avg of S: 7.0
```

`total`, `length`, `i` and the parameter `S` are all local variables of the function `avg()`. Contrary to many other programming languages, we do not have to declare locals in Python. They are automatically recognized. This comes in handy.

One point to note about the code is that $|S| - 1$ is missing. This is not an issue because the `range` iterator excludes the end value, as we discussed in Sect. 5.2.3.

Now, let us also introduce the code for standard deviation. First, the mathematical definition:

$$std(S) = \sqrt{\frac{1}{|S|} \sum_{i=0}^{|S|-1} (S_i - avg(S))^2}. \tag{6.5}$$

This time, we bear in mind that unnecessary function calls should be avoided:

Hands-on Code 6.9

https://pp4e.online/c6s9

```
def avg(S):
    total = 0.0
    length = len(S)
    for i in range(0, length): total += S[i]
    return total/length

def std(S):
    total = 0.0
    length = len(S)
    average = avg(S)
    for i in range(0, length): total += (S[i] - average)**2
    return (total/length)**0.5

S = [11, 2, 9, 6]
print("The avg of S:", avg(S))
print("The std of S:", std(S))
```

Output

```
The avg of S: 7.0
The std of S: 3.391164991562634
```

We implemented the solution using explicit indexing. As given below, this particular problem can be coded without iteration over the indices. Such problems are rare. Problems that require operations over vectors and matrices or problems that aim at optimizations usually require explicit index iterations.

For the sake of completeness, below, we provide an index-less implementation for calculating the average and standard deviation. This version will work on any container data type without the need for support of index selection and the `len()` function:

Hands-on Code 6.10

https://pp4e.online/c6s10

```
def avg(S):
    sum = 0.0
    length = 0
    for x in S:
        sum += x
        length += 1
    return sum/length

def std(S):
    sum = 0.0
    length = 0
    average = avg(S)
    for x in S:
        sum += (x - average)**2
        length += 1
    return (sum/length)**0.5

S = [11, 2, 9, 6]
print("The avg of S:", avg(S))
```

Output

```
The avg of S: 7.0
The std of S: 3.391164991562634
```

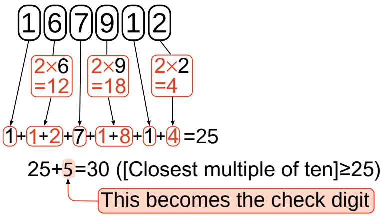

1 6 7 9 1 2
2×6 =12
2×9 =18
2×2 =4
1+1+2+7+1+8+1+4 =25
25+5=30 ([Closest multiple of ten]≥25)
This becomes the check digit

Fig. 6.1 How the check digit is calculated for a student number

Example 2: Computing check digit

What is a check digit?

A check digit serves as a redundancy check employed to detect errors in identification numbers, like social security numbers, bank account numbers, or student id's. These numbers are often entered manually in computer applications which is an error-prone action. The check digit is calculated using an algorithm based on the other digits or letters present in the identification number. Incorporating a check digit enables the identification of basic errors in a string of characters, typically digits. These errors could involve a single mistyped digit or the transposition of two consecutive digits.

Middle East Technical University, with which the authors are affiliated, also implements a check digit algorithm for student numbers. A student number has exactly six digits. The algorithm for generating the check digit is as follows:

- $total \leftarrow 0$
- For each digit in the student number (the leftmost digit is named as the "first") that has an odd position (first, third, fifth), take the digit and add it to the $total$.
- For each digit in the student number that has an even position (second, fourth, sixth), double the digit. If this result of doubling is a two-digit number, then add each of these digits to the $total$; otherwise, add the result of the doubling (which is a single-digit number itself) to the $total$.
- The one-digit number, which, when added to the $total$, results in a multiple of 10, is the check digit.

To illustrate the steps, consider the student number **167912** as an example in Fig. 6.1.

The check digit of the student number **167912** is found to be **5**, which is appended to the student number and becomes part of the student ID as **167912-5**.

This algorithm can be implemented as follows (assume that the student number, S, is a string):

Hands-on Code 6.11

https://pp4e.online/c6s11

```
def check_digit(S):
    total = 0
    for i in range(0, len(S)):
        digit = int(S[i])
        if i%2 == 0: total += digit
        else: total += (9 if digit==9 else (2*digit) % 9)
    n = total % 10
    return 0 if n == 0 else 10 - n

print(check_digit("167912"))
```

Output

```
5
```

We encourage the reader to investigate the code carefully. A very good technique to understand how a code progresses when it runs is to insert print actions to the points of interest. You can start by inserting:

```
print(digit, total)
```

as the last action in the `for` statement, right after the line of `else:` (left-aligned with the `else`). A few notes about the code:

- Remember that indexing starts in Python at zero. One group of digits, namely (first, third, fifth), are actually at `S[0]`, `S[2]` and `S[4]`. So, their indices are **even**. Or in mathematical terms, they are (modulo 2) zero.
- Summing the digits of a number is a trick we learned in primary school. It is actually nothing else but a quick way to do (modulo 9). So, we are actually taking the (modulo 9) of the doubled digit. With one exception: If the digit is 9, the double is 18 where we have to add a 1+8=9 to the `total`. But `(18 % 9)` would give us zero (not 9). So, here we have to insert an exception: For all digits except 9, we can use `(2*digit) % 9`. For 9, we ought to add a 9 to the `total`.
- The rule of constructing the check digit from the `total` has a similar exception case for `total` being a multiple of 10 (i.e., n (modulo 10) = 0). In that case, we would have a check digit of 10 (because 10–0 = 10) which is not a digit, but two digits. The designers decided to use a zero as the check digit for this particular case.

Example 3: Recovering a missing digit in a student ID

Now let us see how the check digit can be used to recover a missing digit in a student ID.
Consider a student ID such as **1□9503-7**. Now, can we discover what the missing digit (□) is?
For this purpose, we will write a function `find_q_mark()` that will take a string as an argument, in which the missing digit is indicated by a "?" (question mark). We are expecting that a call of `find_q_mark("1?9503-7")` will recover the missing digit and return the demystified string: "139503-7".
Note that the `check_digit()` definition above is still effective.

Hands-on Code 6.12

https://pp4e.online/c6s12

```
# places digit in the place of '?' in the string S
def replace_q_mark(S, digit):
    new_S =""
    for c in S: new_S += str(digit) if c=='?' else c
    return new_S

# finds the value for the position with '?'
def find_q_mark(MissingStdID):      # MissingStdID is something like "1?7912-5"
    six_digit_string = MissingStdID[:6]   # "1?7912"
    checksum_digit = int(MissingStdID[7])
    for missing_digit in range(0, 10):
        candidate = replace_q_mark(six_digit_string, missing_digit)
        if check_digit(candidate) == checksum_digit:
              return candidate + MissingStdID[6:]

print(find_q_mark("1?7912-5"))
```

Output

```
167912-5
```

A few notes about the code:

- `replace_q_mark()` function is called by `find_q_mark()`. It is called only once, therefore its actions could have been merged into `find_q_mark()`. Efficiency-wise, there would be no change (except for one extra function call). However, separating it, making it into a function, and naming it accordingly make the code more readable.
- The first two lines that define `find_q_mark()` actions are merely dissecting the string into a student ID and a check digit part.
- What we do in the `for` loop in `find_q_mark()` (on Lines 11–13) is coined in Computer Science as *brute force*. Practically, it is trying every possible case for being a solution. In this problem, the "?" mark can be any digit in the range [0, 9]. The `for` loop tries each of these possibilities out until the computed check digit is equal to the check digit appearing in the input string.

Example 4: Sequential Search

Searching for an item in a list or any other container is a frequently performed operation while solving world problems. For the sake of simplicity, let us just focus on situations where only numbers are involved; i.e., the searched item is a number and the list includes only numbers. Extension to other data types is trivial.

In a search problem, we have a query item (let us call it `x`) and a list of items (let us call it `L`). If `x` is a member of `L`, then we want the search algorithm to return `True`, optionally by returning its index in `L`. If `x` is not a member of `L`, we expect the algorithm to return `False`.

Consider the example list below:

$$\begin{array}{|c|c|c|c|c|}\hline 109 & -48 & 25 & 4 & -13 \\ \hline \end{array} \tag{6.6}$$

If `x` is 4, the search algorithm should return `True`. If it is 5, the answer should be `False`.

Let us start with the simplest algorithm we can use: *Sequential Search*. In sequential search, we start at the beginning of the list, go over each item in the list, and check if the query (`x`) is equal to any item that we go over.

The following is the pseudocode for the algorithm:

Algorithm 6.1 Sequential Search with While Iterations

```
Input: $x$ — The query number
       $L$ — the list of numbers
Output: $True$ (if $x$ is in $L$) or $False$ (if $x$ is not in $L$)
1: Create a variable named $Index$ with initial value 0
2: Create a variable named $N$ with value $length(L)$
3: While $Index < N$ is $True$ do execute Steps 4-5:
4:     If $x = L[Index]$ then return $True$ as the result
5:     $Index \leftarrow Index + 1$
6: return $False$ as the result
```

Let us implement this simple algorithm in Python:

Hands-on Code 6.13

https://pp4e.online/c6s13

```
# Sequential Search with while statement
def seq_search(x, L):
    index = 0                          # Step 1
    N = len(L)                         # Step 2
    while index < N:                   # Step 3
        if x == L[index]: return True    # Step 4
        index = index + 1                # Step 5
    return False                       # Step 6


# Now let us test the function with examples:
L = [109, -48, 25, 4, -13]
x = 4
print(f"seq_search({x}, {L}): ", seq_search(x, L))
x = 5
print(f"seq_search({x}, {L}): ", seq_search(x, L))
x = 5
L = []
print(f"seq_search({x}, {L}): ", seq_search(x, L))
```

Output

```
seq_search(4, [109, -48, 25, 4, -13]):  True
seq_search(5, [109, -48, 25, 4, -13]):  False
seq_search(5, []):  False
```

That looks good. However, we can have a more compact solution using `for` loops:

Hands-on Code 6.14

https://pp4e.online/c6s14

```
# Sequential Search with for statement
def seq_search(x, L):
    for y in L:
        if x == y: return True
    return False

# Now let us test the function with examples:
L = [109, -48, 25, 4, -13]
x = 4
print(f"seq_search({x}, {L}): ", seq_search(x, L))
x = 5
print(f"seq_search({x}, {L}): ", seq_search(x, L))
x = 5
L = []
print(f"seq_search({x}, {L}): ", seq_search(x, L))
```

Output

```
seq_search(4, [109, -48, 25, 4, -13]):  True
seq_search(5, [109, -48, 25, 4, -13]):  False
seq_search(5, []):  False
```

That's the beauty of Python. These two are iterative solutions. Let us have a look at a recursive solution. First, let us update the algorithm. The idea is essentially the same: We compare with the first item and return `True` if they match. Otherwise, we continue our search with the remaining items:

Algorithm 6.2 Sequential Search with Recursion

```
Input: $x$ — the query number
       $L$ — the list of numbers
Output: $True$ (if $x$ is in $L$) or $False$ (if $x$ is not in $L$)
1: If $L$ is empty then return $False$
2: If $x$ is equal to the first item of $L$, ($L[0]$) then return $True$
3: Make a recursive call on $L[1 : N]$, and return its answer
```

And here is the recursive implementation in Python:

Hands-on Code 6.15

https://pp4e.online/c6s15

```python
# Sequential Search with recursion
def seq_search(x, L):
    if L == []: return False      # Step 1: We have exhausted the list, x was not found
    if x == L[0]: return True     # Step 2: x was at the beginning of the list, yay!
    return seq_search(x, L[1:])   # Step 3: Search among the remaining items since x != L[0]

# Now let us test the function with examples:
L = [109, -48, 25, 4, -13]
x = 4
print(f"seq_search({x}, {L}): ", seq_search(x, L))
x = 5
print(f"seq_search({x}, {L}): ", seq_search(x, L))
x = 5
L = []
print(f"seq_search({x}, {L}): ", seq_search(x, L))
```

Output

```
seq_search(4, [109, -48, 25, 4, -13]):  True
seq_search(5, [109, -48, 25, 4, -13]):  False
seq_search(5, []):  False
```

Exercise

Write down the pseudocode for the Python function `seq_search()` with `for` statement.

Exercise

Extend the recursive implementation of `seq_search()` to search for an item `x` in a list `L` in the case where `L` can be nested. In other words, the extended search algorithm should return `True` for:

`seq_search_nested(5, [10, [4, [5]], -84, [3]])`

and `False` for:

`seq_search_nested(8, [10, [4, [5]], -84, [3]])`

Example 5: Binary Search

If the list of items (numbers in our case) is not sorted, sequential search is the only option to search for an item. If, however, the numbers are sorted (in an increasing or decreasing order), then we have a much more efficient algorithm, called *binary search*.

Assuming that `L`, our list of numbers, is sorted in increasing order: In binary search, we recursively execute the following steps (see also Fig. 6.2):

1. Compare `x` with the number that is at the center (middle) of the list. If the middle number is equal to `x`, then we obviously return `True`.
2. Otherwise, we have two options:

   a. `x` is smaller than the middle number: Then `x` has to be to the left of the middle number—we can continue our search on the left part of L.

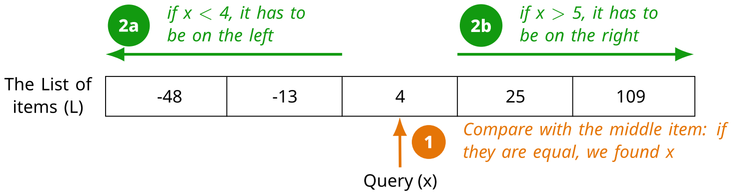

2a
if x < 4, it has to be on the left
2b
if x > 5, it has to be on the right
The List of items (L)
-48
-13
4
25
109
1
Compare with the middle item: if they are equal, we found x
Query (x)

Fig. 6.2 Binary search algorithm

b. `x` is larger than the middle number: Then `x` must be on the right side of the middle number—we can continue our search on the right part of L.

Here is the pseudocode that describes this algorithm:

Algorithm 6.3 Binary Search with Recursion

```
Input: $x$ — the query number
       $L$ — the sorted list of numbers (ascending order)
Output: $True$ (if $x$ is in $L$) or $False$ (if $x$ is not in $L$)
1: If $length(L) = 0$ then return $False$ since the list is empty
2: Create a variable $Mid\_index$ with value $length(L)/2$
3: If $x = L[Mid\_Index]$ then return $True$, we found the item
4: If $x < L[Mid\_index]$ then make recursive call with $L[0 : Mid\_index]$ and return its answer
5: Otherwise, make a recursive call with $L[Mid\_index + 1 : ]$
   and return its answer
```

The following is the implementation in Python:

Hands-on Code 6.16

https://pp4e.online/c6s16

```python
# Binary search with recursion
def bin_search(x, L):
    if len(L) == 0: return False                                # Step 1: Terminating condition
    Mid_index = len(L) // 2                                     # Step 2: Uses integer division
    if x == L[Mid_index]: return True                           # Step 3
    if x < L[Mid_index]: return bin_search(x, L[0:Mid_index])   # Step 4
    if x > L[Mid_index]: return bin_search(x, L[Mid_index+1:])  # Step 5: Can drop if statement

# Now let us test the function with examples:
L = [-48, -13, 4, 25, 109]
x = 4
print(f"bin_search({x}, {L}): ", bin_search(x, L))
x = 5
print(f"bin_search({x}, {L}): ", bin_search(x, L))
x = 5
L = []
print(f"bin_search({x}, {L}): ", bin_search(x, L))
```

Output

```
bin_search(4, [-48, -13, 4, 25, 109]):  True
bin_search(5, [-48, -13, 4, 25, 109]):  False
bin_search(5, []):  False
```

Exercise
Implement binary search using while statements.

Exercise
Identify the complexity of sequential search and binary search in the following cases:

- Best case: When the item that we are making our first comparison with is the one we were looking for.
- Worst case: The item we are looking for is the last item with which we performed the comparison.
- Average case.

Example 6: Sorting
Sorting pertains to ordering a list of items (L) so that each item in L satisfies the following constraint:

$$L_i < L_{i+1}, \tag{6.7}$$

which leads to an ascending (increasing) order of items in L and this is called *ascending sort*. If the order of the constraint is changed to $L_i > L_{i+1}$, the outcome is a descending (decreasing) order of the items in L and this is called *descending sort*.

An algorithm that can perform ascending sort can easily be changed to do descending sort; therefore, we will just focus on one of them, namely ascending sort. Moreover, we will focus on sorting a list of numbers; however, the same algorithms can be applied to other data types as long as an ordering constraint (< or >) can be defined.

Bubble Sort
Let us look at a very simple but inefficient sorting algorithm, called *Bubble Sort*. In Bubble Sort, we start from the beginning of the list and check whether the constraint $L_i < L_{i+1}$ is violated for an index i. If it is, we swap L_i with L_{i+1}. When we reach the end of the list, we start from the beginning of the list again and continue checking the constraint $L_i < L_{i+1}$. Once we reach the end of the list, we go back to the beginning and continue to check the constraint for each i. This continues until the constraint $L_i < L_{i+1}$ is not violated for any index i, which means that the numbers are sorted.

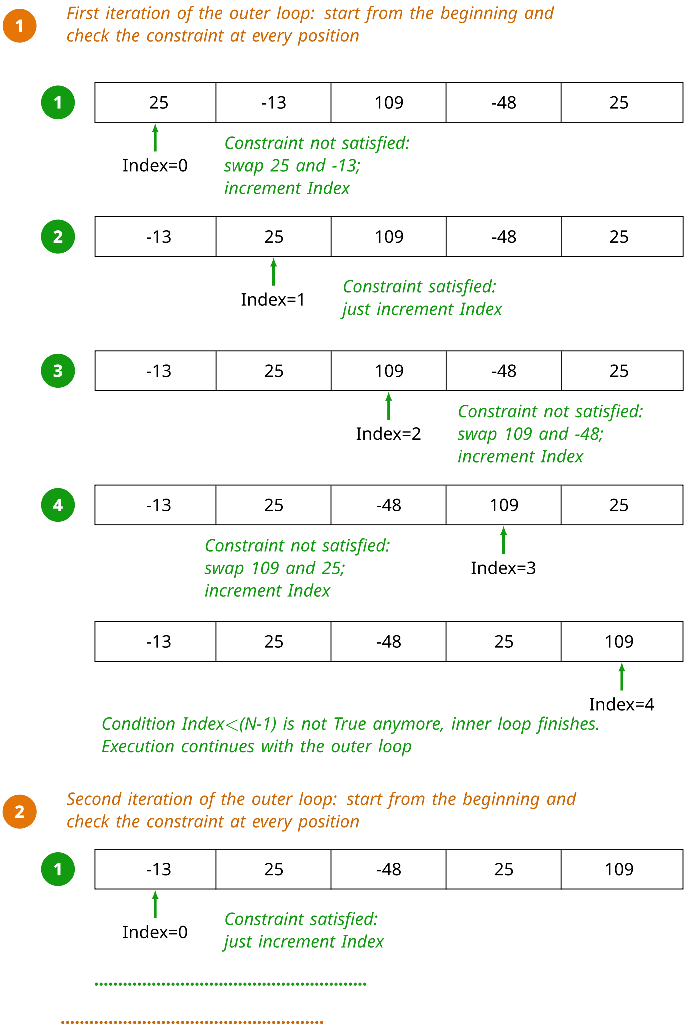
Therefore, in Bubble Sort, we have two loops, one outer loop that continues until all numbers are sorted, and an inner loop that goes through the list and checks the constraint. An outer iteration of the algorithm is illustrated on a small list of numbers below in Fig. 6.3.

With one outer iteration, the list appears to be one step closer to a sorted list. In the example in Fig. 6.3, the largest number has found its correct place after one outer iteration. With a second one, the second largest number will find its place. With a third iteration, the third largest number will find its place, etc.

Here is a pseudocode for the Bubble Sort algorithm:

Algorithm 6.4 Bubble Sort

```
Input: $L$ — the list of numbers (not sorted)
Output: $L$ — the sorted list of numbers (ascending order)
 1: Create a variable $Is\_sorted$ with value $False$
 2: Create a variable $N$ with value $length(L)$
 3: Create a variable $Index$
 4: While $Is\_sorted$ is $False$ do repeat steps 5–11:
 5:   $Is\_sorted \leftarrow True$
 6:   $Index \leftarrow 0$
 7:   While $Index < (N - 1)$ do repeat steps 8–11
 8:     If $L[Index] > L[Index + 1]$ then execute steps 9–10:
 9:       Swap $L[Index]$ and $L[Index + 1]$
10:       $Is\_sorted \leftarrow False$
11:     $Index \leftarrow Index + 1$
12: return $L$
```



1
First iteration of the outer loop: start from the beginning and check the constraint at every position
1
25 | -13 | 109 | -48 | 25
Index=0
Constraint not satisfied: swap 25 and -13; increment Index
2
-13 | 25 | 109 | -48 | 25
Index=1
Constraint satisfied: just increment Index
3
-13 | 25 | 109 | -48 | 25
Index=2
Constraint not satisfied: swap 109 and -48; increment Index
4
-13 | 25 | -48 | 109 | 25
Constraint not satisfied: swap 109 and 25; increment Index
Index=3
-13 | 25 | -48 | 25 | 109
Index=4
Condition Index<(N-1) is not True anymore, inner loop finishes. Execution continues with the outer loop
2
Second iteration of the outer loop: start from the beginning and check the constraint at every position
1
-13 | 25 | -48 | 25 | 109
Index=0
Constraint satisfied: just increment Index

Fig. 6.3 One outer iteration of the Bubble Sort algorithm

And here is the implementation in Python:

Hands-on Code 6.17

https://pp4e.online/c6s17

```python
# Bubble Sort

def bubble_sort(L):
    Is_sorted = False                                   # Step 1
    N = len(L)                                          # Step 2
    Index = 0                                           # Step 3: Can be removed in Python

    while Is_sorted is False:                           # Step 4
        Is_sorted = True                                # Step 5
        Index = 0                                       # Step 6
        while Index < (N-1):                            # Step 7
            if L[Index] > L[Index+1]:                   # Step 8
                (L[Index], L[Index+1]) = (L[Index+1], L[Index])  # Step 9
                Is_sorted = False                       # Step 10
            Index = Index + 1                           # Step 11
    return L                                            # Step 12

# Let us test our implementation with some sample lists
L = [109, -13, 4, -48, 25]
print(f"bubble_sort({L}): ", bubble_sort(L))

L = [-48, -13, 4, 25, 109]
print(f"bubble_sort({L}): ", bubble_sort(L))

L = []
print(f"bubble_sort({L}): ", bubble_sort(L))
```

Output

```
bubble_sort([109, -13, 4, -48, 25]):  [-48, -13, 4, 25, 109]
bubble_sort([-48, -13, 4, 25, 109]):  [-48, -13, 4, 25, 109]
bubble_sort([]):  []
```

6.9 Programming Style

With the topics covered in this chapter, we have started working with longer and more elaborate Python codes. Up to now, we have introduced crucial aspects that can be arranged differently among programmers:

- Naming variables and functions.
- Indentation.
- Function definition.
- Commenting.

Longing for brevity and readability, the creators of Python have defined some guidelines of style for those interested: Python Style Guide.[2]

There are alternative styles, and we strongly recommend that you choose one and stick to it as a programming habit.

[2] https://www.python.org/dev/peps/pep-0008/.

6.10 Important Concepts

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

- Defining functions.
- Function parameters and how to pass values to functions.
- Default parameters.
- Scopes of variables.
- Local, enclosing, global, and built-in scopes.
- Higher-order functions.
- Differences of functions between programming and mathematics.
- Recursion.

6.11 Further Reading

- Functional tools of Python: https://docs.python.org/3/howto/functional.html [1].
- For a more comprehensive coverage on recursion, see "Managing the Size of a Problem" (Chap. 4) of G. Üçoluk, S. Kalkan, Introduction to Programming Concepts with Case Studies in Python, Springer, 2012 [2].
- Python style guide: https://www.python.org/dev/peps/pep-0008/ [3].

6.12 Exercises

1. Write a Python function named `only_numbers()` that removes items in a list that are not numbers. For example, `only_numbers([10, "ali", [20], True, 4])` should return `[10, 4]`. Implement two versions: One using iteration and another using recursion.
2. Write a Python function named `flatten()` that removes nesting in a nested list and lists all elements in a single non-nested list. For example, `flatten([1, [4, 5, [6]], [["test"]]])` should return `[1, 4, 5, 6, "test"]`.
3. Write a function named `find_indices()` that takes two arguments. The first argument is a list. The second argument is the element being searched for. The return value should be a list of indices at which the searched element is found. For example,

   ```
   find_indices(["How", "much", "wood", "could", "a", "wood", "chuck",
                 "chuck", "if", "a", "wood", "chuck", "could", "chuck",
                 "wood?"], "chuck")
   ```

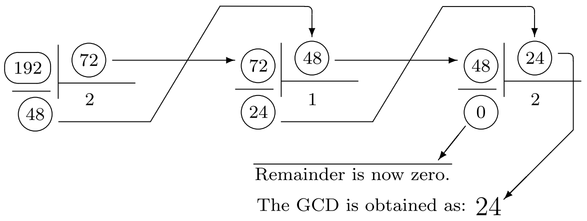
   should return `[6, 7, 11, 13]`. The function should return an empty list if the searched element is not in the list.
4. Write a Python function named `reverse()` that reverses the order of elements in a list. For example, `reverse([10, 5, 0])` should return `[0, 5, 10]`. Implement two versions: One using iteration and another using recursion.
5. A prime factor is a positive integer (bigger than 1) whose only factors are 1 and itself. Write a Python function named `prime_factors()` that finds the prime factors for a positive integer given as argument. `prime_factors()` should return the list of prime factors in increasing order. For example, `prime_factors(2184)` should return `[2, 2, 2, 3, 7, 13]`

6. Write a function called `combination()` that takes two integers n and k as arguments and returns the combination, which is also known as the binomial coefficient, $\binom{n}{k}$.

   In this exercise, you should:

   - Write the function using a helper function called `factorial()`, which you should also code.
   - Then, rewrite the function by attempting to minimize the number of mathematical operations.
   - Finally, implement the function using only the addition operation and recursion.

7. Write a Python function that accepts a string as an argument and counts the number of uppercase and lowercase letters (ignoring all other characters). The function should return a 2-tuple of: `(`*Count_of_uppercase_chars*, *Count_of_lowercase_chars*`)`
8. Write a function named `merge()` that takes two sorted lists with numeric elements, as arguments, and returns a merged, order-preserving (sorted), list. For example, `merge([-3, 3, 17, 29, 100], [-11, 1, 20, 22, 81, 85, 86, 99])` should return:
   `[-11, -3, 1, 3, 17, 20, 22, 29, 81, 85, 86, 99, 100]`.
9. Write a Python function named `greatest_divisor()` that finds the greatest divisor of an integer. The greatest divisor of an integer n is the largest integer $x < n$ that satisfies $n = k \cdot x$ for some integer k. For example, `greatest_divisor(12)` should return `6`.
10. Write a Python function named `GCD()` that finds the greatest common divisor of two integers. You should use the following algorithm for finding the GCD *(greatest common divisor)*. Let us assume we want to find `GCD(192,72)`:



    Implement two versions: One using iteration and another using recursion.

11. A fraction or an integer can be represented by means of a list. Here are a few examples:

Fraction/Integer	Representation
$351 \rightarrow$	`[351]`
$-351 \rightarrow$	`[-351]`
$\frac{2}{5} \rightarrow$	`[2,5]`
$-\frac{2}{5} \rightarrow$	`[-2,5]`
$3\frac{2}{5} \rightarrow$	`[3,2,5]`
$-3\frac{2}{5} \rightarrow$	`[-3,2,5]`

As you may have already noticed, a negative value is only represented by a negative value in the first member of the list.

You are expected to implement four functions `add()`, `sub()`, `mul()`, `div()`, which will take exactly two arguments (two values represented as above) and perform addition, subtraction, multiplication or division, respectively. These functions should return their results as a list which is in the same format as their inputs.

Here is an example:

```
>>> mul([2], mul(sub([1, 2, 3], [7, 4]), [520, 36]))
[-2, 11, 27]
```

12. In the realm of number theory, a *perfect number* is a positive integer that is equal to the sum of all its positive divisors except for itself. For example, 6 is a perfect number because its proper positive divisors are 1, 2, and 3, and their sum is 6. Other examples of perfect numbers include 28, 496, and 8128.

    Write a Python function named `is_perfect()` that takes a positive integer as argument and checks whether it is perfect or not. The function should return a Boolean value.

13. You are given a list of 3-tuple elements in a variable called `Cities`. Each 3-tuple has the structure `(`**cityname**, **latitude**, **longitude**`)` where:

    cityname is a string.
    latitude is a string of the form: $dd{:}mm{:}ss\mathcal{L}_1$, where dd is degrees, mm is minutes, ss is seconds and $\mathcal{L}_1$ is one of `N` or `S`.
    longitude is a string of the form: $dd{:}mm{:}ss\mathcal{L}_2$, where dd is degrees, mm is minutes, ss is seconds and $\mathcal{L}_2$ is one of `E` or `W`.

    Example of the first elements of `Cities` might be:

    ```
    Cities = [("ankara", "39:55:00N", "32:55:00E"),
              ("dublin", "53:20:10N", "06:15:15W"), ...]
    ```

    The task is to find the city, which we will call the *meeting place*, such that the sum of the flight distances from all the other cities to this *meeting place* is minimized. Write a function that takes `Cities` as argument and returns the **cityname** of the *meeting place*.

14. A date can be expressed as a 3-tuple of integers. Namely, day, month, and year. Write a function, named `delta_date()`, which takes two such date tuples as arguments and returns the distance (difference) between the two dates in days. If the first argument is later in time with respect to the second then the result should be a negative value.

15. Write a function named `dispense()` that takes two arguments. The first argument is a list of banknote values. The second argument of `dispense()` is a positive integer indicating the amount of money that has to be summed up in terms of banknotes. The return value should be a list of counts of banknotes.

    Example: `dispense([1, 5, 10, 20, 50, 100, 200], 2797)` should return `[2, 1, 0, 2, 1, 1, 13]`

    Notes: It is compulsory to give out the least number of banknotes possible. You can assume that there is an infinite stock of each banknote type.

References

[1] Python Documentation, Functional tools of python (2023). https://docs.python.org/3/howto/functional.html. Accessed 5 Sept 2023

[2] G. Üçoluk, S. Kalkan, *Introduction to Programming Concepts with Case Studies in Python* (Springer Science & Business Media, 2012)

[3] Python Documentation, Python style guide (2023). https://www.python.org/dev/peps/pep-0008/. Accessed 5 Sept 2023

7 A Gentle Introduction to Object-Oriented Programming

At this stage of programming, you must have realized that programming possesses some idiosyncrasies:

- Unless some form of randomization is explicitly built in, all computations are *deterministic*; i.e., the result is always the same for the same input.
- The logic involved is binary; i.e., there are two truth values: True and False.
- There is a clear distinction between *actions* and *data*. Actions are coded into expressions, statements, and functions. Data is coded into integers, floating point numbers, and containers (strings, tuples, and lists).

The first two can be dealt with in a controlled manner. Furthermore, it is mostly more preferable to have a crisp and deterministic outcome. However, the third is not as natural. The world we live in is not made up of data and independent actions acting on that data. This is merely an abstraction. The need for this abstraction stems from the very nature of the computing device we use, the von Neumann Machine. What you store in the *memory* is either some data (integer or floating point) or some instruction. The *processor* processes the data based on a series of instructions. Therefore, we have a clear separation of data and action in computers.

However, when we look around, we do not see such a distinction. We see *objects*. We see a tree, a house, a table, a computer, a notebook, a pencil, a lecturer, a student... Objects have some properties that would be quantified as data, but they also have some capabilities that would correspond to some actions. What about reuniting data and action under the natural concept of "object"? *Object-Oriented Programming*, abbreviated as OOP, is the answer to this question.

7.1 Properties of Object-Oriented Programming

OOP is a paradigm that comes with some properties:

- *Encapsulation*: Combining data and functions that manipulate that data under a concept that we name as "object" so that a rule of "need-to-know" and "maximal privacy" is satisfied.

S. Kalkan et al., *Programming with Python for Engineers*,
https://doi.org/10.1007/978-3-031-57148-0_7

- *Inheritance*: Defining an object and then using it to create “descendant” objects so that the descendant objects inherit all functions and data of their ancestors.
- *Polymorphism*: A mechanism that allows a descendant object to appear and function like its ancestor object when necessary.

7.1.1 Encapsulation

Encapsulation is the property that data and actions are glued together in a data-action structure called “object” that conforms to a rule of “need-to-know” and “maximal privacy”. In other words, an object should provide access only to data and actions that are needed by other objects. The data and actions that are not needed should be “hidden” and used by the object itself for its own merit.

This is important especially to keep the implementation modular and manageable: An object stores some data and implements certain actions. Some of these are private and hidden from other objects, whereas others are *public* to other objects so that they can access such public data and actions to suit their needs.

Public data and actions function as the interface of the object to the outside world. In this way, objects interact with each other’s interfaces by accessing public data and actions. This can be considered as a message-passing mechanism: Object1 calls Object2’s action f(), which calls Object3’s function g(), which returns a message (a value) back to Object2, which, after some calculation, returns another value to Object1.

In this modular approach, Object1 does not need-to-know how Object2 implements its actions or how it stores data. All Object1 needs to know is the public interface through which “messages” are passed to compute the solution.

As a realistic example of a university registration system, assume that `Student` object calls `register()` action of a `Course` object and it calls `checkPrerequisite()` action of a `Curriculum` object. `checkPrerequisite()` checks if the course can be taken by the student and returns the result. `register()` action performs additional checks and returns the success status of the registration to the `Student`.

As another example, assume that you need to implement a simple weather forecasting system. This hypothetical system gets a set of meteorological sensor data like humidity, pressure, and temperature from various levels of the atmosphere and tries to estimate the weather conditions for the next couple of days. The data of such a system may have a time series of sensor values. The actions of such a system would be a group of functions that add sensor data as they are measured and forecasting functions to obtain the future estimate of weather conditions. For example:

```
sensors = [{'datetime':'20201011 10:00','temperature':12.3,
            'humidity': 32.2, 'pressure':1.2,
            'altitute':1010.0},
            {'datetime':'20201011 12:00','temperature':14.2,
            'humidity': 31.2, 'pressure':1.22,
            'altitute':1010.0},
            ....]
def addSensorData(sensorarr, temp, hum, press, alt):
    '''Add sensor data to sensor array with
       current time and date'''
    ....
```

(continues on next page)

(continued from previous page)

```
def estimate(sensorarr, offset):
    '''return whether forecast for given
       offset days in future'''
    ...
...
addSensorData(sensors, 20.3, 15.4, 0.82, 10000)
...
print(estimate(sensors, 1))
...
```

In the implementation above, the data and actions are separated. The programmer should maintain the list containing the data and make sure that actions are available and called as needed with the correct data. This approach has a couple of disadvantages:

1. There is no way to ensure that actions are called with the correct sensor data format and values (i.e., `addSensorData('Hello world', 1, 1, 1, 1)`).
2. `addSensorData` implementation can ensure that the sensor list contains correct data; however, since sensor data can be directly modified, its integrity can be violated later on (i.e., `sensors[0] = 'Hello World'`).
3. When you want to forecast for more than one location, you need to duplicate all data and maintain them separately. Keeping track of which list contains which location requires extra special care.
4. When you need to improve your code and change data representation, such as storing each sensor type on a separate sorted list by time, you need to change the action functions. However, if some code directly accesses the sensor data, it may conflict with the changes you made to the data representation. For example, if the new data representation is as follows:

   ```
   sensors = {'temperature': [('202010101000',23),...],
              'humidity': [('2020101000',45.3),...],
              'pressure': [('2020100243',1.02),...]}
   ```

   Any access to `sensors[0]` as a dictionary directly through code segments will be incorrect.

With encapsulation, sensor data and actions are put into the same body of definition so that the only way to interact with the data would be through the actions. In this way:

1. Data and actions are kept together. The encapsulation mechanism guarantees that the data exists and has the correct format and values.
2. Multiple instances can be created for forecasting for multiple locations, and each location is maintained in its object as if it were a simple variable.
3. Since no code part accesses the data directly but calls the actions of the object, changing the internal representation and implementation of functions will not require changing code segments using this class.

The following is an example of OOP implementation for the problem at hand:

```
class WhetherForecast:
  # Data
  __sensors = None
```

(continues on next page)

(continued from previous page)

```
  # Actions acting on the data
  def __init__(self):
    self.__sensors = []   # this will create initial sensor data

  def addSensorData(self, temp, hum, press, alt):
    ....

  def estimate(self, offset):
    ...
    return {'lowest':elow, 'highest':ehigh,...}

ankara = WhetherForecast() # Create an instance for location Ankara
ankara.addSensorData(...)
....

izmir = WhetherForecast()  # Create an instance for location Izmir
izmir.addSensordata(...)
....

print(ankara.estimate(1))  # Work with the data for Ankara
print(izmir.estimate(2))   # Work with the data for Izmir
```

The above syntax will be more clear in the following sections; however, note how the newly created objects `ankara` and `izmir` behave. They contain their sensor data internally, and the programmer does not need to care about their internals. The resulting object will syntactically behave like a built-in data type of Python.

7.1.2 Inheritance

In many applications, the objects with which we will work will be related. For example, in a drawing program, we are going to work with shapes such as rectangles, circles and triangles which have some common data and actions, e.g.:

- Data:
  - Position
  - Area
  - Color
  - Circumference
- and actions:
  - draw()
  - move()
  - rotate()

What kind of data structure do we use for these data and how we implement the actions are important. For example, if one shape is using Cartesian coordinates (x, y) for position and another is using Polar coordinates (r, θ), a programmer can easily make a mistake by providing (x, y) to a shape using Polar coordinates.

As for actions, implementing such overlapping actions in each shape from scratch is redundant and inefficient. In the case of separate implementations of overlapping actions in each shape, we would have to update all overlapping actions if we want to correct an error in our implementation or switch to a more efficient algorithm for the overlapping actions. Therefore, it makes sense to implement the common functionalities in another object and reuse them whenever needed.

These two issues are handled in OOP via *inheritance*. We place common data and functionalities into an ancestor object (e.g., `Shape` object for our example) and other objects (`Rectangle`, `Triangle`, `Circle`) can inherit (reuse, derive) these data and definitions as if those data and actions were defined in their object definitions.

In real-life entities, you can observe many similar relations. For example:

- A `Student` is a `Person` and an `Instructor` is a `Person`. Updating personal records of a `Student` is not different from that of an `Instructor`.
- A `DCEngine`, a `DieselEngine`, and a `StreamEngine` are all `Engine`s. They have the same characteristic features like horsepower, torque, etc. However, `DCEngine` has power consumption in units of Watt, whereas `DieselEngine` consumption can be measured in liters per km.
- In a transportation problem, a `Ship`, a `Cargo_Plane` and a `Truck` are all `Vehicle`s. They have the same behavior of carrying a load; however, they have different capacities, speeds, costs, and ranges.

Assume that we would like to improve the forecasting accuracy by adding radar information in our `WhetherForecast` example above. We need to obtain our traditional estimate and combine it with the radar image data. Instead of duplicating the traditional estimator, it is wiser to use the existing implementation and *extend* its functionality with the newly introduced features. This way, we avoid code duplication and when we improve our traditional estimator, our new estimator will automatically use it.

Inheritance is a very useful and important concept in OOP. Together with encapsulation, it enhances reusability, maintenance, and reduces redundancy.

7.1.3 Polymorphism

Polymorphism is a property that enables a programmer to write functions that can operate on different data types uniformly. For example, calculating the sum of elements of a list actually behaves the same for a list of integers, a list of floats, and a list of complex numbers. As long as the addition operation is defined among the members of the list, the summation operation would be the same. If we can implement a *polymorphic* `sum()` function, it will be able to calculate the summation of distinct datatypes, and hence it will be polymorphic.

In OOP, all descendants of a parent object can act as objects of more than one type. Consider our example on shapes above: The `Rectangle` object inheriting from the `Shape` object can also be used as a `Shape` object since it contains the data and actions defined in a `Shape` object. In other words, a `Rectangle` object can be assumed to have two data types: `Rectangle` and `Shape`. We can exploit this to write polymorphic functions: If we write functions that operate on `Shape` with well-defined actions, they can operate on all descendants of `Shape` including `Rectangle`, `Circle`, and all objects inheriting `Shape`. Similarly, the actions of a parent object can operate on all its descendants if it uses a well-defined interface.

Polymorphism improves modularity, code reusability, and expandability of a program.

7.2 Basic OOP in Python

The way Python implements OOP is not to the full extent in terms of the properties listed in the previous section. Encapsulation, for example, is not implemented strongly. Thankfully, inheritance and polymorphism are provided more firmly. Moreover, operator overloading, a feature that is much demanded in OOP, is present.

In the last decade, Python has become increasingly popular and widely used as a standard language for scientific and engineering computations. Before Python, there were software packages for various computational purposes. Packages to do numerical computations, statistical computations, symbolic computations, computational chemistry, computational physics, and all sorts of simulations were developed over four decades. Today, many such packages, free or proprietary, are *wrapped* to be called through Python. This packaging is done mostly in an OOP manner. Therefore, it is vital to know some basics of OOP in Python.

7.2.1 The Class Syntax

In Python, an object is a code structure as illustrated in Fig. 7.1.

First, a piece of jargon:

- **Class**: A prescription that defines a particular object. The blueprint of an object.
- **Class Instance ≡ Object**: A computational structure that has functions and data fields built according to the blueprint, namely the class. Similar to the construction of buildings according to an architectural blueprint, in Python we can create *objects* (more than one) conforming to a class definition. Each of these objects will have its own data space and in some cases customized functions. Objects are equivalently called *class instances*.
  Each object provides the following:
  - **Methods**: Functions that belong to the object.
  - **Sending a message to an object**: Calling a method of the object.
  - **Member**: Any data or method that is defined in the class.

Now, let us look at how we can define a class. In Python, this is done by the keyword `class`:

```
class ClassName :
          Statement block
```

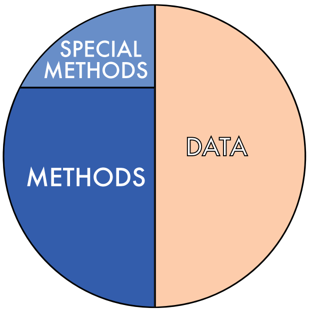


Fig. 7.1 An object includes both data and actions (methods and special methods) as one data item

Here is an example:

```
class shape:
   color = None
   x = None
   y = None

   def set_color(self, red, green, blue):
      self.color = (red, green, blue)

   def move_to(self, x, y):
      self.x = x
      self.y = y
```

This blueprint tells Python that:

1. The name of this class is `shape`.
2. Any object that will be created according to this blueprint has three data fields, named `color`, `x` and `y`. At the moment of creation, these fields are set to `None` (a special value of Python indicating that there is a variable here but no value is assigned yet).
3. Two member functions, the so-called *methods*, are defined: `set_color()` and `move_to()`. The first takes four arguments, constructs a tuple of the last three values and stores it into the `color` data field of the object. The second, `move_to()`, takes three arguments and assigns the last two of them to the `x` and `y` data_fields, respectively.

The peculiar keyword `self` in the blueprint refers to the particular instance (when an object is created based on this blueprint). The first argument to all methods (the member functions) has to be coded as `self`. That is a rule. The Python interpreter will fill it out when that function is activated.

To refer to any function or data field of an object, we use the dot (.) notation. Inside the class definition, `self.□` syntax should be used for this purpose. Outside of the object, the object is certainly stored somewhere (in a variable or a container), in which case the way (syntax) to access the stored object is followed. Then, this syntax is appended by the dot (.) which is then followed by the data field name or the method name.

For our example `shape` class, let us create two objects and assign them to two global variables `p` and `s`, respectively:

```
p = shape()
s = shape()
p.move_to(22, 55)
p.set_color(255, 0, 0)
s.move_to(49, 71)
s.set_color(0, 127, 0)
```

The object creation is triggered by calling the class name as if it is a function (i.e., `shape()`). This creates an *instance* of the class. Each instance has its private data space. In the example, two `shape` objects are created and stored in the variables `p` and `s`. As said, the object stored in `p` has its private data space and so does `s`. We can verify this by:

```
print(p.x, p.y)
print(s.x, s.y)
```

7.2.2 Special Methods and Operator Overloading

There are many more special methods than the ones we described above. For a complete reference, we refer to "Sect. 3.3 of the Python Language Reference".[1]

When a class is defined, there are a bunch of methods, which are automatically created to facilitate the integration of the object with the Python language. For example, what if we issue a print action on the object?

```
print(s)
```

```
<__main__.shape object at 0x7f295325a6a0>
```

Not very informative, is it? We can change this behavior by overwriting a special function in the class, named `__str__()`. By doing so, the print function can output the color and coordinate information, as follows:

```
shape object: color=(0,127,0) coordinates=(47,71)
```

`__str__()` is the method that is automatically activated when a `print()` function has an object to be printed. The built-in print function calls the `__str__()` member function (method) of the object (in the OOP jargon, the print function sends to the object an `__str__()` message). All objects, when created, have some *special methods* predefined. Many of them are out of the scope of this course, but `__str__()` and `__init__()` are among these special methods.

It is possible that the programmer, in the class definition, overwrites (redefines) these predefinitions. `__str__()` is set to a default definition so that when an object is printed such an internal location information is printed.

Another special method that we will illustrate is the `__init__()` method, also called the *constructor*. `__init__()` is the method that is automatically activated when the object is first created. As default, it will do nothing, but can also be overwritten. Observe the following statement in the code above:

```
s = shape()
```

The object creation is triggered by calling the class name as if it is a function. Python (and many other OOP languages) adopt this syntax for object creation. What is done is that the arguments passed to the class name are sent "internally" to the special member function `__init__()`. We will overwrite it to take two arguments at object creation, and these arguments will become the initial values for the x and y coordinates.

In the following code block, we change our class definition and illustrate the use of these special member functions:

[1] https://docs.python.org/3/reference/.

Hands-on Code 7.1

https://pp4e.online/c7s1

```
class shape:
    color = None
    x = None
    y = None

    def set_color(self, red, green, blue):
        self.color = (red, green, blue)

    def move_to(self, x, y):
        self.x = x
        self.y = y

    def __str__(self):
        return "shape object: color=%s coordinates=%s" % (self.color, (self.x,self.y))

    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __lt__(self, other):
        return self.x + self.y < other.x + other.y

p = shape(22,55)
s = shape(12,124)
p.set_color(255,0,0)
s.set_color(0,127,0)

print(s)
s.move_to(49,71)
print(s)

print(p.__lt__(s))
print(p < s)  # just the same as above but now infix

print(s.__dir__())
```

Output

```
shape object: color=(0, 127, 0) coordinates=(12, 124)
shape object: color=(0, 127, 0) coordinates=(49, 71)
True
True
['x', 'y', 'color', '__module__', 'set_color', 'move_to', '__str__', '__init__', '__lt__', '__dict__
↪', '__weakref__', '__doc__', '__repr__', '__hash__', '__getattribute__', '__setattr__', '__delattr_
↪_', '__le__', '__eq__', '__ne__', '__gt__', '__ge__', '__new__', '__reduce_ex__', '__reduce__', '__
↪subclasshook__', '__init_subclass__', '__format__', '__sizeof__', '__dir__', '__class__']
```

After defining a class, a critical issue that arises frequently is the behavior of the operators ($+$, $-$, $>$, $\ldots$) on the objects. For example, if you have an object, how will it behave under comparison? In Python, you can change the behaviors of the following operators by defining the corresponding member functions:

- `x<y` calls `x.__lt__(y)`,
- `x<=y` calls `x.__le__(y)`,
- `x==y` calls `x.__eq__(y)`,
- `x!=y` calls `x.__ne__(y)`,
- `x>y` calls `x.__gt__(y)`, and
- `x>=y` calls `x.__ge__(y)`.

Having learned this, let us look at the following function definition inside the `shape` class definition above.

```python
def __lt__(self, other):
    return self.x + self.y < other.x + other.y
```

With this, you have the (<) comparison operator available on `shape` objects. As you can see, the comparison result is based on the *Manhattan distance* from the origin. You can give it a test right away and compare the two objects `s` and `p` as follows:

```python
print(s<p)
```

How would you modify the comparison method so that it compares the Euclidean distances from the origin?

As we have created our first object in Python, we strongly advise that the encapsulation property of OOP is followed by the programmer, i.e., you. This property is that the data of an object is private to the object and not open to modification (or not even to inspection if encapsulation is taken extremely strictly) by a piece of code external to the object. Only member functions, the so-called methods, of that object can do this. Therefore, if you want to update the `color` value, for example, of an `shape` object, though you can do it "brutally" by, e.g.:

```python
p.color = (127,64,5)
```

However, you should not do so. Contrary to some other OOP programming languages, Python does not forbid this by brutal force. Therefore, it is not a "**you cannot**" but a "**you should not**" type of action. All data access should be done through messages (functions).

7.2.3 Example 1: Counter

Now, let us implement a very simple class called `Counter`. The counter has two important restrictions: It starts with zero and it can only be incremented. If you were to use a simple integer variable instead of `Counter`, it could be initialized to any value and it can directly be assigned to an arbitrary value without any checks or restrictions. Implementing a Python class for a `Counter` will let you enforce those restrictions since you define the initialization and the actions.

Here is the definition for such a counter:

Hands-on Code 7.2

https://pp4e.online/c7s2

```
class Counter:
    def __init__(self):
        self.value = 0    # this is the initialization

    def increment(self):
        '''increment the inner counter'''
        self.value += 1

    def get(self):
        '''return the counter as value'''
        return self.value

    def __str__(self):
        '''define how your counter is displayed'''
        return 'Counter:{}'.format(self.value)

stcnt = Counter()    # create the counter
stcnt.increment()
stcnt.increment()
print("# of students", stcnt)

sheep = Counter()
while sheep.get() < 1000:
    sheep.increment()

print("# of sheep",sheep)
```

Output

```
# of students Counter:2
# of sheep Counter:1000
```

7.2.4 Example 2: Rational Number

As our next example, let us try to define a new data type, *rational number*, as a Python class. It is possible to represent a rational number simply as a tuple of integers as `(numerator, denominator)`. However, this representation has a couple of issues. First, the denominator can be 0, which leads to an invalid value. Second, many distinct tuples represent practically the same value and they need to be interpreted in a special way in operators like comparison. We need to either normalize all into their simplest form or implement comparison operators to respect the values they represent. In the following implementation, we choose the first approach: We find the *greatest common divisor* of the numerator and the denominator and normalize them.

Hands-on Code 7.3

https://pp4e.online/c7s3

```
import math

class Rational:
    def __init__(self, n, d):
        '''initialize from numerator and denominator values'''
        if d == 0:
            raise ZeroDivisionError   # raise an error. Explained in following chapters
        self.num = n
        self.den = d
        self.simplify()

    def simplify(self):
        if self.num == 0:
            self.den = 1
        return
        gcd = math.gcd(self.num,self.den)
        if self.den < 0:             # flip their signs if denominator is negative
            self.den *= -1
            self.num *= -1
        self.num = self.num // gcd
        self.den = self.den // gcd   # normalize them by dividing by greatest common divisor

    def __str__(self):
        return '{}/{}'.format(self.num, self.den)

    def __mul__(self, rhs):        # this special method is called when * operator is used
         ''' (a/b)*(c/d) -> a*c/b*d in a new object '''
         retval = Rational(self.num * rhs.num, self.den * rhs.den) # create a new object
         return retval

    def __add__(self, rhs):        # this special method is called when + operator is used
         ''' (a/b)+(c/d) -> a*d+b*c/d*b in a new object '''
         retval = Rational(self.num * rhs.den + rhs.num * self.den,
                         self.den * rhs.den) # create a new object with sum
         return retval

    # -, /, and other operators left as exercise

    def __eq__(self, rhs):        # called when == operator is used
        '''a*d == b*c '''
        return self.num*rhs.den == self.den*rhs.num

    def __lt__(self, rhs):        # called when < operator is used
        '''a*d < b*c '''
        return self.num*rhs.den < self.den*rhs.num

    # rest can be defined in terms of the first two
    def __ne__(self, rhs):  return not self == rhs

    def __le__(self, rhs):  return self < rhs or self == rhs

    def __gt__(self, rhs):  return not self <= rhs

    def __ge__(self, rhs):  return not self < rhs
```

Hands-on Code 7.4

https://pp4e.online/c7s4

```
# Let us play with our Rational class

a = Rational(3, 9)
b = Rational(16, 24)
print(a, b, a*b+b*a)
print(a<b, a+b == Rational(1, 1))
```

Output

```
1/3 2/3 4/9
True True
```

This class definition only implements `*`, `+`, and comparison operators. The remaining operators are left as an exercise. Our new class `Rational` behaves like a built-in data type in Python, thanks to the special methods implemented.

7.2.5 Inheritance with Python

Now let us have a look at the second property of OOP, namely inheritance, in Python. We will do that by extending our `shape` example:

A `shape` is a relatively general term. We have many types of shapes: Triangles, circles, ovals, rectangles, and even polygons. If we were to incorporate them into software, for example, a simple drawing and painting application for children, each of them would have a position on the screen and a color. This would be common to all shapes. But then, a circle would be identified (additionally) by a radius; a triangle by two additional corner coordinates, etc.

Therefore, it is more efficient if a `triangle` object inherits all the properties and member functions of a `shape` object and defines only data and functions that are specific to the triangle shape.

This inheritance can be specified while defining the `triangle` class as follows:

```
class triangle(shape):
```

following the general syntax below:

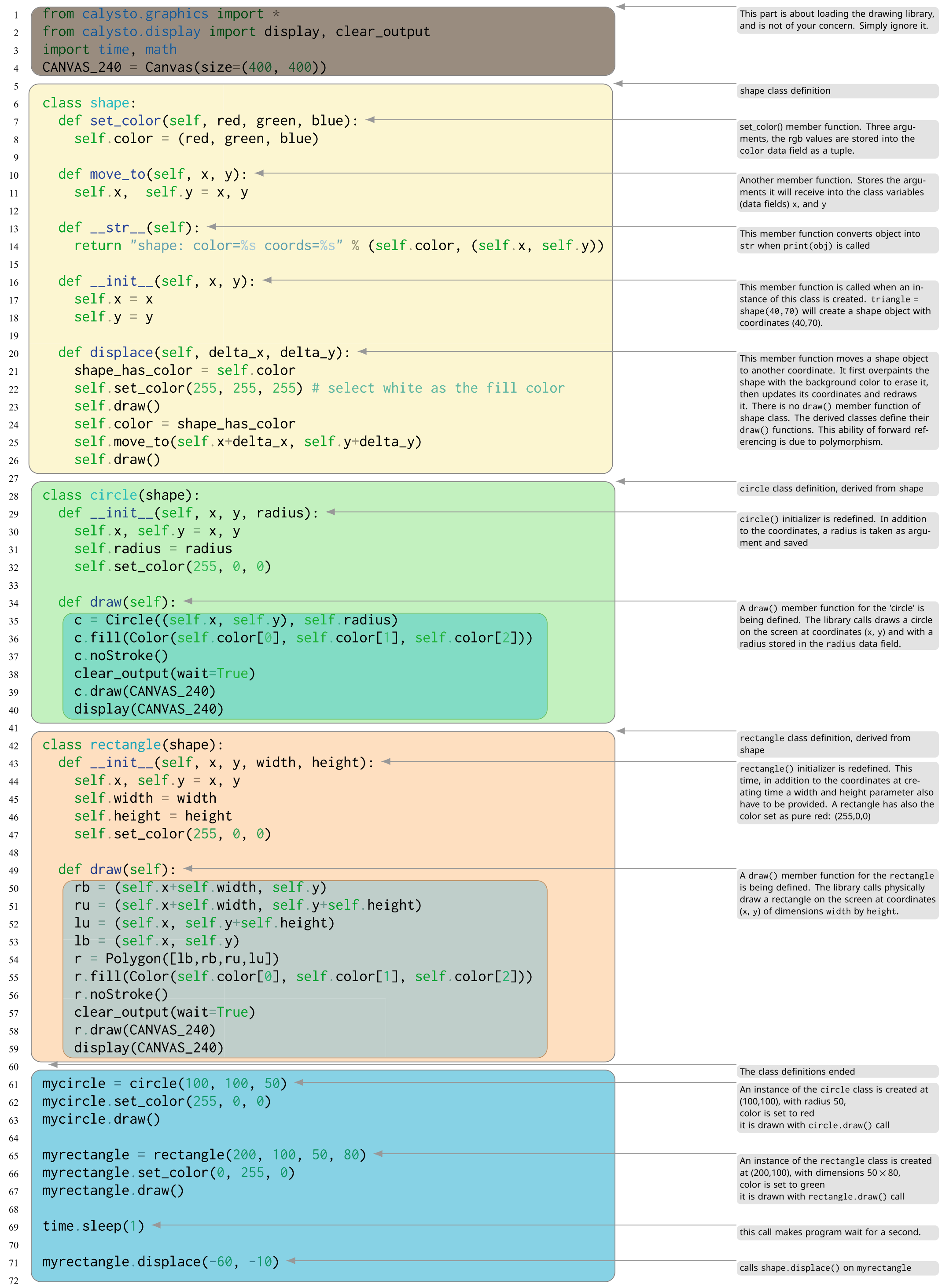
`class` $\boxed{ClassName}$ `(` $BaseClass_1$, $BaseClass_2$, ..., `):`
$\boxed{Statement\ block}$

Though multiple inheritance (inheriting from more than one class) is possible, it is seldom used and not preferred. It has elaborated rules for resolving member function name clashes, which is beyond the scope of this introductory book.

We will continue with an example that does inheritance from a single base class.

7.2.6 Interactive Example: A Simple Shape Drawing Program

In Fig. 7.2, you see a simple object-oriented Python code that draws a red circle and a green rectangle, and then moves the green rectangle towards the circle. You can visit the Web page of the chapter to interact with the program. You can get additional information by hovering your mouse over any piece of the code.



```
from calysto.graphics import *
from calysto.display import display, clear_output
import time, math
CANVAS_240 = Canvas(size=(400, 400))

class shape:
  def set_color(self, red, green, blue):
    self.color = (red, green, blue)

  def move_to(self, x, y):
    self.x,  self.y = x, y

  def __str__(self):
    return "shape: color=%s coords=%s" % (self.color, (self.x, self.y))

  def __init__(self, x, y):
    self.x = x
    self.y = y

  def displace(self, delta_x, delta_y):
    shape_has_color = self.color
    self.set_color(255, 255, 255) # select white as the fill color
    self.draw()
    self.color = shape_has_color
    self.move_to(self.x+delta_x, self.y+delta_y)
    self.draw()

class circle(shape):
  def __init__(self, x, y, radius):
    self.x, self.y = x, y
    self.radius = radius
    self.set_color(255, 0, 0)

  def draw(self):
    c = Circle((self.x, self.y), self.radius)
    c.fill(Color(self.color[0], self.color[1], self.color[2]))
    c.noStroke()
    clear_output(wait=True)
    c.draw(CANVAS_240)
    display(CANVAS_240)

class rectangle(shape):
  def __init__(self, x, y, width, height):
    self.x, self.y = x, y
    self.width = width
    self.height = height
    self.set_color(255, 0, 0)

  def draw(self):
    rb = (self.x+self.width, self.y)
    ru = (self.x+self.width, self.y+self.height)
    lu = (self.x, self.y+self.height)
    lb = (self.x, self.y)
    r = Polygon([lb,rb,ru,lu])
    r.fill(Color(self.color[0], self.color[1], self.color[2]))
    r.noStroke()
    clear_output(wait=True)
    r.draw(CANVAS_240)
    display(CANVAS_240)

mycircle = circle(100, 100, 50)
mycircle.set_color(255, 0, 0)
mycircle.draw()

myrectangle = rectangle(200, 100, 50, 80)
myrectangle.set_color(0, 255, 0)
myrectangle.draw()

time.sleep(1)

myrectangle.displace(-60, -10)
```

This part is about loading the drawing library, and is not of your concern. Simply ignore it.

shape class definition

set_color() member function. Three arguments, the rgb values are stored into the color data field as a tuple.

Another member function. Stores the arguments it will receive into the class variables (data fields) x, and y

This member function converts object into str when print(obj) is called

This member function is called when an instance of this class is created. triangle = shape(40,70) will create a shape object with coordinates (40,70).

This member function moves a shape object to another coordinate. It first overpaints the shape with the background color to erase it, then updates its coordinates and redraws it. There is no draw() member function of shape class. The derived classes define their draw() functions. This ability of forward referencing is due to polymorphism.

circle class definition, derived from shape

circle() initializer is redefined. In addition to the coordinates, a radius is taken as argument and saved

A draw() member function for the 'circle' is being defined. The library calls draws a circle on the screen at coordinates (x, y) and with a radius stored in the radius data field.

rectangle class definition, derived from shape

rectangle() initializer is redefined. This time, in addition to the coordinates at creating time a width and height parameter also have to be provided. A rectangle has also the color set as pure red: (255,0,0)

A draw() member function for the rectangle is being defined. The library calls physically draw a rectangle on the screen at coordinates (x, y) of dimensions width by height.

The class definitions ended

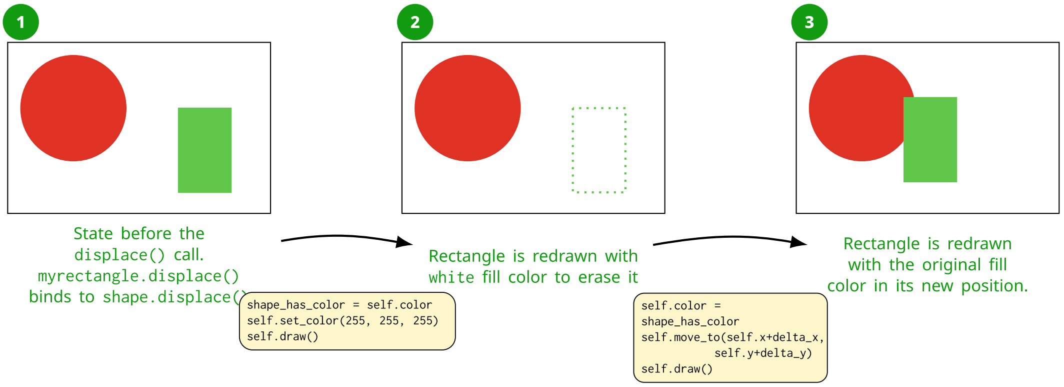
An instance of the circle class is created at (100,100), with radius 50,
color is set to red
it is drawn with circle.draw() call

An instance of the rectangle class is created at (200,100), with dimensions 50 × 80,
color is set to green
it is drawn with rectangle.draw() call

this call makes program wait for a second.

calls shape.displace() on myrectangle

Fig. 7.2 Interactive Python code for a simple-drawing program with an object-oriented paradigm


1
2
3
State before the displace() call. myrectangle.displace() binds to shape.displace()
Rectangle is redrawn with white fill color to erase it
Rectangle is redrawn with the original fill color in its new position.

```
shape_has_color = self.color
self.set_color(255, 255, 255)
self.draw()
```

```
self.color =
shape_has_color
self.move_to(self.x+delta_x,
             self.y+delta_y)
self.draw()
```

Fig. 7.3 Steps of the shape displacement example

Following this interactive code display, you can find a code box, the one with a small triangle on the left, which is the same code. You can run the code, and observe it live, running. In the exercise part, you will be asked to modify the code.

The dark brownish sections in the code contain calls to a drawing library (named `calysto`), which we will not explain in this book. Not to distract the reader intentionally, we have darkened it out.

Our object-oriented programming intention in this example is to define a general class that we will name `shape`. `shape` contains all the properties and functions that a drawable geometric object (circle, rectangle, triangle, ...) possesses. All of them are drawn on an imaginary canvas at a certain coordinate with some color. So, all shapes have a coordinate at which they are drawn and a color. The first lines of the `class shape` definition start with these variables.

In this example, we implement just two geometric shapes, the circle, and the rectangle. This piece of code with the green background is the definition of the `circle` class. It is *derived* from the *base* class `shape`. So, it *inherits* all the definitions of `shape` (but these definitions can be overwritten, in other words, can be redefined). As you observe, it defines a `radius` variable (in addition to the `color`, `x`, `y` variables that were inherited from the base class). In addition, it defines, for the first time, a `draw()` member function which has the duty to draw a `circle` on the canvas, based on the parameters `x`, `y`, `radius` and `color` using some functions of the library `calysto`. The details of this drawing implementation are not our focus here (therefore, it is darkened out).

The second geometric shape, which is implemented, is the rectangle. The code with the orange background defines this class. Again, it is a *descendent* of the base class `shape`. Hence, it *inherits* all the definitions of `shape` (just as the circle did). This time, it does not add a `radius` but a `width` and a `height` variables to the class. In addition to the redefinition of the initializer `__init__()`, which is activated the moment an instance is created, this class has its own definition of the `draw()` member function.

Now, for an instance, return to the `class shape` definition (the yellow box). There, you will see a `displace()` member function being defined. It is a function that displaces a geometric shape by an amount of $(\Delta x, \Delta y)$. Please inspect the function now. You will discover that the function:

- “memorizes” the color of the shape,
- changes the color to white,
- calls the `draw()` member function and draws a white shape exactly on top of the old one,
- restores the drawing color (from white to the memorized color),
- changes the `x`, `y` values by amounts `delta_x` and `delta_y`, respectively,

- calls the `draw()` member function and draws at the new `x,y` coordinates the shape with the original color.

Now we have an interesting phenomenon. `draw()` is not even defined in `shape`. It will be defined differently for each class that is derived from the `shape` base class. How can a function of `shape` make a reference to a function which does not exist for `shape` and will only be defined in classes that are descendants of this base class? Also note that the each of the descendant classes (`circle` and `rectangle`) are going to define it (the `draw()` function) their own way. This is called *forward-referencing and polymorphism*, a nice and powerful feature of OOP.

Note that the class definitions (`class shape`, `class circle`, `class rectangle`) are merely blueprints. They do not create any data. They define how they will be created and which functions will act how. The actual creations (based on these blueprints) are done in the code in the last (turquoise-colored) box.

In summary, the code in blue in Fig. 7.2 performs the following:

- An instance of the `circle class` is created (by calling its initializer) at coordinates (`100`, `100`) with a radius of `50`. This instance is stored in the variable `mycircle`.
- The color of this freshly created object is changed to the brightest red.
- The object's `draw()` function is called (in OOP jargon: a `draw` message is sent to the object stored in `mycircle`)
- Similarly, an instance of `rectangle class` is created at coordinates (`200`, `100`) and with `width = 50`, `height = 80`. This instance is stored in the variable `myrectangle`.
- The color of the rectangle is changed to the brightest green.
- The rectangles `draw()` function is called.
- The program pauses (waits) for `1` second.
- Finally, it calls the `displace()` function of the rectangle. Though it was defined in the base class, `displace()` locates the correct `draw()` (the draw of the rectangle) and performs the job.

Fig. 7.3 shows how shapes are drawn initially and how the rectangle is displaced by `myrectangle.displace(-60, -10)` call.

Please use the webpage of the chapter to see the interactive demo.

7.2.7 Useful Short Notes on Python's OOP

These notes are provided for completeness and as pointers for the curious reader. A detailed coverage of these notes is out of the scope of this introductory book.

- It is possible that a *derived* class overrides a member function of its base (parent) class, but still wants to access the (former) definition in the base class. This is possible by prefixing the function call by `super().` (no space after the dot). For example, `super().draw(x, y)`.
- There is no proper *destructor* in Python. This is because the Python engine does all the memory allocation and bookkeeping of data. Though entirely under the control of Python, sometimes the so-called *garbage collection* is carried out. At that moment, all unused object instances are wiped out of the memory. Before that, a special member function `__del__()` is called. If you want to do a special treatment of an object before it is wiped out forever, you can define the `__del__()` function. The concept of garbage collection is complex and it is wrong to assume that `__del__()`

will right away be called even if an object instance is deleted by the `del` statement (`del` will mark the object as unused but it will not necessarily trigger a garbage collection phase).

- *Infix* operators have special, associated member functions (those that start and end with double underscores). If you want your objects to participate in infix expressions, then you have to define those. For example “+” has the associated special member function `__add__()`. For a complete list and how-to-do’s, search for “special functions of Python” and “operator overloading”.
- You can restrict the accessibility of variables defined in a class. There are three types of accessibility modifications you can perform: *public*, *protected*, and *private*. The default is public.

  – Public access variables can be accessed anywhere inside or outside the class.
  – Protected variables can be accessed within the same package (file). A variable that starts with a single underscore is recognized by programmers as protected.
  – Private variables can only be accessed inside the class definitions. A variable that starts with two underscores is recognized by programmers as private.

7.3 Widely-Used Member Functions of Containers

Being acquainted with the OOP concepts, it is time to reveal the “object” properties of some Python components. We will do so only for containers.

7.3.1 Strings

In Table 7.1, you will find some of the very frequently used member functions of strings:

7.3.2 Lists

In Table 7.2, you will find some of the very frequently used member functions of lists.

7.3.3 Dictionaries

In Table 7.3, you will find some of the very frequently used member functions of dictionaries.

7.3.4 Sets

In Table 7.4, you will find some of the very frequently used member functions of sets.

Table 7.1 Widely-used member functions of strings. Assume S is a string. In the **Operation** column, anything in square brackets denotes that the content is optional-if you enter the optional content, do not type in the square brackets

Operation	Result
`S.capitalize()`	Returns a copy of S with its first character capitalized, and the rest of the characters lowercased
`S.count(sub [,start [, end]])`	Returns the number of occurrences of substring `sub` in string S
`S.find(sub [,start [,end]])`	Returns the lowest index in S where substring sub is found. Returns `-1` if sub is not found
`S.isalnum()`	Returns `True` if all characters in S are alphanumeric, `False` otherwise
`S.isalpha()`	Returns `True` if all characters in S are alphabetic, `False` otherwise
`S.isdigit()`	Returns `True` if all characters in S are digit characters, `False` otherwise
`S.islower()`	Returns `True` if all characters in S are lowercase, `False` otherwise
`S.isspace()`	Returns `True` if all characters in S are whitespace characters, `False` otherwise
`S.isupper()`	Returns `True` if all characters in S are uppercase, `False` otherwise
`separator.join(seq)`	Returns a concatenation of the strings in the sequence `seq`, separated by string `separator`, e.g., `"#".join(["a","bb","ccc"`]) returns `"a#bb#ccc"`
`S.ljust/rjust/center(width [,fillChar=’ ’])`	Returns S, left/right justified/centered in a string of length width. surrounded by the appropriate number of `fillChar` characters
`S.lower()`	Returns a copy of S converted to lowercase
`S.lstrip([chars])`	Returns a copy of S with leading `chars` (default: blank chars) removed
`S.partition(separ)`	Searches for the separator `separ` in S, and returns a tuple `(head, sep, tail)` containing the part before it, the separator itself, and the part after it
`S.replace(old, new [, maxCount = -1])`	Returns a copy of S with the first `maxCount` (`-1`: unlimited) occurrences of substring `old` replaced by `new`
`S.split([separator [, maxsplit]])`	Returns a list of the words in S, using `separator` as the delimiter string
`S.splitlines( [keepends])`	Returns a list of the lines in S, breaking at line boundaries
`S.startswith(prefix [, start [, end]])`	Returns `True` if S starts with the specified prefix, otherwise returns `False`. Negative numbers may be used for `start` and `end`. `prefix` can also be a tuple of strings to try
`S.strip([chars])`	Returns a copy of S with leading and trailing `chars` (default: blank chars) removed
`S.swapcase()`	Returns a copy of S with uppercase characters converted to lowercase and vice versa
`S.upper()`	Returns a copy of S converted to uppercase

Table 7.2 Widely-used member functions of lists. Assume `L` is a list. In the **Operation** column, anything in square brackets denotes that the content is optional-if you enter the optional content, do not type in the square brackets

Operation	Result
`L.append(x)`	same as `L[len(L) : len(L)] = [x]`
`L.extend(x)`	same as `L[len(L) : len(L)] = x`
`L.count(x)`	returns number of `i`’s for which `L[i] == x`
`L.index(x [, start [, stop ]])`	returns smallest `i` such that `L[i] == x`. `start` and `stop` limit search to only part of the list
`L.insert(i, x)`	same as `L[i:i] = [x]` if `i` $\geq$ `0`. if `i = -1`, inserts before the last element
`L.remove(x)`	same as `del L[L.index(x)]`
`L.pop([i])`	same as `x = L[i]; del L[i]; return x`
`L.reverse()`	reverses the items of `L` in place
`L.sort([cmp])` OR `L.sort([cmp=cmpFct] [,key=keyGetter] [,reverse=bool])`	sorts the items of `L` in place

Table 7.3 Widely-used member functions of dictionaries. Assume `D` is a dictionary. In the **Operation** column, anything in square brackets denotes that the content is optional-if you enter the optional content, do not type in the square brackets

Operation	Result
`D.fromkeys(iterable, value=None)`	Class method to create a dictionary with keys provided by the iterator, and all values set to `value`
`D.clear()`	Removes all items from `D`
`D.copy()`	A shallow copy of `D`
`D.has_key(k)` OR `k in D`	`True` if `D` has key `k`, else `False`
`D.items()`	A copy of `D`’s list of `(key, item)` pairs
`D.keys()`	A copy of `D`’s list of keys
`D1.update(D2)`	`for k, v in D2.items(): D2[k] = v`
`D.values()`	A copy of `D`’s list of values
`D.get(k, defaultval)`	The item of `D` with key `k`
`D.setdefault(k [, defaultval])`	`D[k] if k in D, else defaultval` (also setting it)
`D.iteritems()`	Returns an iterator over `(key, value)` pairs
`D.iterkeys()`	Returns an iterator over the mapping’s keys
`D.itervalues()`	Returns an iterator over the mapping’s values
`D.pop(k [, default ])`	Removes key `k` and returns the corresponding value. If key is not found, default is returned if given, otherwise `KeyError` is raised
`D.popitem()`	Removes and returns an arbitrary `(key, value)` pair from `D`

Table 7.4 Widely-used member functions of sets. Assume **T**, **T1**, **T2** are sets (unless otherwise stated). In the **Operation** column, anything in square brackets denotes that the content is optional-if you enter the optional content, do not type in the square brackets

Operation	Result
`T1.issubset(T2)`	`True` if every element in `T1` is in iterable `T2`
`T1.issuperset(T2)`	`True` if every element in `T2` is in iterable `T1`
`T.add(elt)`	Adds element `elt` to set `T` (if it doesn't already exist)
`T.remove(elt)`	Removes element `elt` from set `T`. `KeyError` if element not found
`T.discard(elt)`	Removes element `elt` from set `T` if present
`T.pop()`	Removes and returns an arbitrary element from set `T`; raises `KeyError` if empty
`T.clear()`	Removes all elements from this set
`T1.intersection(T2 [, T3 …])`	Synonym to `(T1 & T2)`. Returns a new Set with elements common to all sets (in the method `T2`, `T3`, …can be any iterable)
`T1.union(T2 [, T3 …])`	Synonym to `(T1 | T2)`. Returns a new Set with elements from either set (in the method `T2, T3,` … can be any iterable)
`T1.difference(T2 [, T3 …])`	Synonym to `(T1-T2)`. Returns a new Set with elements in `T1` but not in any of `T2, T3,` … (in the method `T2, T3,` … can be any iterable)
`T1.symmetric_difference(T2)`	Synonym to `(T1 ^ T2)`. Returns a new Set with elements from either of two sets but not in their intersection
`T.copy()`	Returns a shallow copy of set `T`

7.4 Important Concepts

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

- Encapsulation, inheritance, and polymorphism.
- Benefits of the Object-Oriented Paradigm.
- Concepts such as class, instance, object, member, method, message passing.
- Concepts such as base class, ancestor, and descendant.

7.5 Further Reading

- Special Methods in Python: https://docs.python.org/3/reference/datamodel.html#special-method-names [1].
- The "Object-oriented Programming" chapter of G. Üçoluk, S. Kalkan, Introduction to Programming Concepts with Case Studies in Python, Springer, 2012 [2].

7.6 Exercises

1. Define a class `Vec3d` to represent 3-dimensional vectors. `Vec3d` should encapsulate 3 `float` values, `x,y,` and `z`. You should implement the following methods for your class:

- `__init__(self, x,y,z)`: The constructor, which gets 3 values and construct a 3D vector.
- `__str__(self)`: Returns the string representation of vector as `(x,y,z)` so that print() and `str()` functions can display the vector in a human-readable form.
- `add(self, b)`: Constructs and returns a new `Vec3d` object which is the addition of the current object and `b` which is another `Vec3d` object. Vector addition is defined as the addition of all corresponding dimensions.
- `sub(self, b)`: Same as `add()` but dimensions are subtracted.
- `dot(self, b)`: Returns the **dot product** of two vectors as a scalar (`float`) value. Dot-product is defined as $x_1x_2 + y_1y_2 + z_1z_2$, i.e., the sum of products of the corresponding dimensions.
- `cross(self, b)`: Constructs and returns a new `Vec3d` object which is **cross product** of the current vector and `b`. The cross product is defined as $(y_1z_2 - z_1y_2, z_1x_2 - x_1z_2, x_1y_2 - y_1x_2)$
- `len(self)`: Returns length (norm) of the vector as a scalar: $\sqrt{x^2 + y^2 + z^2}$.
- `norm(self)`: Construct and return a vector in the same direction but the length is 1.0. Simply divide all dimensions by the length of the vector.

The test run and its output would look as follows:

```
a = Vec3d(1,0,0)
b = Vec3d(0,1,0)
c = Vec3d(0.7,0.7,0.7)
print(a.add(b), a.sub(b), a.dot(b), a.cross(b))
print(a.add(c), a.sub(c), a.dot(c), a.cross(c))
print(c.len(), c.norm())

(1, 1, 0) (1, -1, 0) 0 (0, 0, 1)
(1.7, 0.7, 0.7) (0.30000000000000004, -0.7, -0.7) 0.7 (0.0, -0.7, 0.7)
1.212435565298214 (0.5773502691896258, 0.5773502691896258, 0.
↪5773502691896258)
```

2. Modify the `Vec3d` class and replace `add`, `sub`, `dot`, and `cross` functions with the special functions `__add__()`, `__sub__()`, `__mul__()`, `__matmul__()`. This change will enable `+, -, *` and @ operators to be used on `Vec3d` objects. After this change, the test run should look like this:

```
a = Vec3d(1,0,0)
b = Vec3d(0,1,0)
c = Vec3d(0.7,0.7,0.7)
print(a + b, a - b,  a * b , a @ b)
print(a + c, a - c, a * c, a @ c)
print(c.len(), c.norm())
```

3. Define a class `Vector` to represent N-dimensional vectors. `Vector` should encapsulate N `float` values. Its constructor gets any object that can be iterated (typically a list or a tuple) and constructs a vector with the same size as the iterated value. Implement the same methods as the `Vec3d` class above except for cross product, which is not a binary operator in N dimensions.
Internally, you can use a tuple or a list, and easily construct the value to be represented by iteration as follows:

```
[x for x in constructorparam]
```

A sample test run can be given as:

```
a = Vec([1,2,3,4])
b = Vec([4,3,2,1])
c = Vec([1,0,0,0])
print(a+b, a-b, a*b)
print(a+c, a-c, a*c)
print(a.len(), a.norm())
```

4. Implement a class `sortedList` which will work like a Python `list` but keep all values sorted in increasing order. Implement the following methods:

   - `__init__(self, it)`: Constructs the current object from the iterated object. It basically inserts values from `it` in a loop. It should call `self.insert()` to make sure the values are stored sorted.
   - `insert(self, val)`: Inserts the value in its place. For this, you should first search for its position (the index of the first number larger than `val`), then insert the value in this position. You can use `insert(position, val)` in the internal list.
   - `extend(self, it)`: Adds values from the iterator `it` into the list. Similar to the constructor, `self.insert()` should be used.
   - `__getitem__(self, idx)`, `__str__(self)`, and `__delitem__(self, idx)` methods to direct the following calls to the internal list: `list[idx]`, `str(list)`, `del list[idx]` specifically.

   You can use binary search in Sect. 6.8 for an efficient implementation. A sample run and output can be given as follows:

```
a = sortedList([7,5,3,8])
print(a)
a.extend([7,8,2,1])
print(a)
del a[4]
print(a)

[3, 5, 7, 8]
[1, 2, 3, 5, 7, 7, 8, 8]
[1, 2, 3, 5, 7, 8, 8]
```

5. Implement a class for mobile subscription management called `mobilePlan`. Its constructor gets 3 float values for price per minute, price per SMS, and price per megabyte of Internet transfer. After construction, each mobile use by the user is recorded to the object with `call(seconds)`, `sms()` and `useInternet(megabytes)` member function calls. Each record is saved inside the object with its timestamp in a data structure (a list of tuples for example). For getting the timestamp, use `time.time()` call (first, you need to `import time`). Later, you can call `time.ctime(timestamp)` function to convert this timestamp to a human-readable form.
   The user can check the current billing amount by calling the `currentBill()` method. The amount is calculated by multiplying unit prices with usage amounts. For call-time calculation, charging is done in units of 10 s, which means that all remainder values between 0 and 10 are rounded up to

10 s. The method returns a quadruple of the amount for calls, the amount for SMS'es, the amount for Internet usage, and the total amount as the last value.

The last function you need to implement is `detailedBill()` which should generate in a string a table that lists usage information, time, amount, and price. In the last row, the bill summary should be given.

Sample run and output are given as follows:

```
a = mobilePlan(0.5, 0.5, 0.1)

a.call(12)
time.sleep(1)
a.sms()
time.sleep(1)
a.use(8.5)
time.sleep(1)
a.call(302)
time.sleep(1)
a.use(823)

print(a.detailedBill())
print(a.currentBill())
```

```
          Date              Type   Amount    Price
----------------------------------------------------
Fri May  5 14:23:03 2023  call        12.00     10.00
Fri May  5 14:23:04 2023  sms          1.00      0.50
Fri May  5 14:23:05 2023  internet     8.50      0.85
Fri May  5 14:23:06 2023  call       302.00    155.00
Fri May  5 14:23:07 2023  internet   823.00     82.30
----------------------------------------------------
Calls: 165.00, SMS: 0.50, Int: 83.15, Tol:    248.65
(165.0, 0.5, 83.15, 248.65)
```

6. Derive a class from `mobilePlan` above called `mobileFixPlan`. This new class gets 4 additional values in the constructor: Minutes, SMS'es, quota and some fixed amount. In this plan, the customer agrees to pay the minimum fixed amount. In return, the given amounts are covered by the plan. Only excess usages are subject to the price per minute, SMS etc.

   `mobileFixPlan`'s constructor gets 7 parameters and passes first 3 to `mobilePlan` by calling `super().__init__(cp, sp, ip)`. It inherits `call, sms, use` methods but overrides the `detailedBill()` and `currentBill()` methods with new bill calculation logic. The accumulated values up to the quota values cost 0, the rest will be billed per usage.

   A sample run would look like:

```
b = mobileFixPlan(0.5, 0.5, 0.1, 100, 300, 10, 500)

b.call(12)
time.sleep(1)
b.sms()
time.sleep(1)
b.use(8.5)
time.sleep(1)
b.call(302)
time.sleep(1)
```

(continues on next page)

(continued from previous page)

```
b.use(823)

print(b.detailedBill())
print(b.currentBill())

          Date              Type   Amount    Price
----------------------------------------------------
Fri May  5 16:09:17 2023  call       12.00      0.00
Fri May  5 16:09:18 2023  sms         1.00      0.00
Fri May  5 16:09:19 2023  internet    8.50      0.00
Fri May  5 16:09:20 2023  call      302.00      7.00
Fri May  5 16:09:21 2023  internet  823.00     33.15
----------------------------------------------------

             Usage: Calls: 314.0, SMS: 1, Int: 831.5
             Plan: Calls: 300.0, SMS: 10, Int: 500.0
 Calls: 15.00, SMS: 0.00, Int: 33.15, Tol:    148.15
(15.0, 0.0, 33.15, 148.15)
```

7. Implement a class `studentPlan` that is derived from `mobileFixPlan` with a given set of prices and quotas. `studentPlan`'s constructor should get no parameters other than `self` and call `super().__init__()` with the seven parameter values of your choice. This way, you can create any specific mobile plan object without supplying any parameters.
8. Revisit the `sortedList` class above. Instead of implementing it as a separate class, and containing a list in an internal variable, try to implement it as "`class sortedList(list):`". This way, you can implement the same class only by implementing a custom insert method and overloading the constructor, `append()` and `extend()` methods. The constructor should call `super().__init__()` and other methods should call the methods directly on `self` to manipulate the inner list from inheritance. Rewrite the example this way, and inherit other methods of Python `list`. Observe that you can use methods like `index, len` and iterate on your object without writing any specific code.
9. Go to the interactive example in Sect. 7.2.6 and modify it as follows:

    a. Add a `triange` class. It should take, at creation time, `(x,y,dx2,dy2,dx3,dy3)` as parameter, where `x,y` are the coordinates of one of the corners of the triangle on the canvas, and `dx2`,`dy2` are the increments relative to `x1`,`y1` to reach the second corner of the triangle (so, the absolute coordinate on the canvas of the second corner becomes (`x+dx2`,`y+dy2`)), `dx3`,`dy3` have a similar meaning to reach the third corner. You are expected to define all member functions present in `circle` or `rectangle`, this time for `triangle`. For the `draw()` member function, you can investigate the content of the `draw()` method of the `rectangle` class.
    b. Implement a `move_on_line(x_fin, y_fin, count_of_steps)` method that will work for all geometric shapes and displace a geometric shape starting from its current position on the canvas and ending at (`x_fin`,`y_fin`) coordinate in `count_of_steps` many displacements.
    c. Create a `class car` method, which is a child-drawing style car that consists of a rectangle and two circles below it. It should implement all member functions of `circle` or `rectangle`. Make use of the existing `rectangle` and `circle` classes.

References

[1] Python Documentation, Special methods (2023). https://docs.python.org/3/reference/datamodel.html#special-method-names. Accessed 5 Sept 2023

[2] G. Üçoluk, S. Kalkan, *Introduction to Programming Concepts with Case Studies in Python* (Springer Science & Business Media, 2012)

8 File Handling

All variables used in a program are kept in the main memory and they are *volatile*, i.e., their values are lost when the program ends. Even if you write a program that runs forever, your data will be lost in case of a shutdown or power failure.

Another drawback of the main memory is the capacity limitation. In the extreme case, when you need more than a couple of gigabytes for your variables, it will be difficult to keep all of them in the main memory. In particular, infrequently required data is better kept on an external storage device instead of the main memory.

Files provide a mechanism for storing data *persistently* on hard drives that provide significantly larger storage than the main memory. These devices are also called *secondary storage* devices. The data you put in a file will stay on the hard drive until someone overwrites or deletes the file (or when the hard drive fails, which is an unfortunate but rare case).

A *file* is a sequence of bytes stored on the secondary storage, typically hard drive (alternative secondary storage devices include CD, DVD, USB disk, tape drive). Data in a file have the following differences from data in memory (variables):

1. A file is just a sequence of bytes. Therefore, data in a file is unorganized, there is no data type, and there are no variable boundaries.
2. Data must be accessed indirectly, using I/O functions. For example, updating a value in a file requires reading it into the main memory, updating it in the main memory, and then writing it back into the file.
3. Accessing and updating data are significantly slower since it is on a slow external device.

Keeping data in a file instead of the main memory has the following use cases:

1. Data needs to be persistent. Data will be in the file when you restart your program, reboot your machine, or when you find your ancient laptop in the basement 30 years later (probably it will not be there when an AD 3000 archeologist finds your laptop on an excavation site. Hard disks are not that durable. Therefore, persistence is bounded).
2. You need to exchange data with another program. Examples:

   - You download data from the Web and your program gets it as input.
   - You would like to generate data in your program and put it on a spreadsheet for further processing.

S. Kalkan et al., *Programming with Python for Engineers*,
https://doi.org/10.1007/978-3-031-57148-0_8

3. You have a large amount of data that does not fit in the main memory. In this case, you will probably use a library or software like a database management system to access data in a faster and more organized way. Files are the most primitive, basic way of achieving it.

In this chapter, we will talk about simple file access so that you will learn about simple file operations such as opening, closing, reading, and writing. The examples of the chapter will create and modify files when run—we strongly encourage you to check the contents of the created files.

8.1 First Example

Let us quickly look at a simple example to get a feeling for the different steps involved in working with files.

Hands-on Code 8.1

https://pp4e.online/c8s1

```
fpointer = open('firstexample.txt',"w")
fpointer.write("hello\n")
fpointer.write("how are\n")
fpointer.write("you?\n")
fpointer.close()
```

The program above will create a file in the current directory with filename `firstexample.txt`. You can open it with your favorite text editor (there are plenty of text editors for all operating systems: notepad, wordpad, textedit, nano, vim) to see and edit it. The content will look like this:

```
hello
how are
you?
```

The first line of the program is `fpointer = open(’firstexample.txt’, "w")`. This line opens the file named `firstexample.txt` for writing to it. If the file exists, its content will be erased (and it will be an empty file afterward). The result of `open()` is a file object that we will use in the following lines. This object is assigned to the variable `fpointer`.

In the following lines, all functions we call with this file object `fpointer` will work on the corresponding file (i.e., `firstexample.txt`). This special *dot* notation helps us with calling functions in the scope of the file. `fpointer.`*functionname*`()` will call the *functionname* function for this file. `write(string)` function will write the `string` content to the file. Each call to `write(string)` will append the `string` to the file and the file will grow. At the end, when we are done, we call `close()` to finish accessing the file so that your operating system will know and take necessary actions. All open files will be closed when your program terminates. However, calling `close()` after finishing writing is a good programming practice.

Now, let us read this file:

Hands-on Code 8.2

https://pp4e.online/c8s2

```
fp = open("firstexample.txt","r")
content = fp.read()
fp.close()
print(content)
```

Output

```
hello
how are
you?
```

In this case, we called `open()` with argument `"r"` which tells the interpreter that we are going to read the file (or use it as an input source). If you skip the second argument in `open()`, it is assumed to be `"r"`, so `open("firstexample.txt")` will be equivalent.

The `read()` function gets an optional argument, which is the number of bytes to read. If you skip it, it will read the entire file content and return it as a string. Therefore, after the call, the `content` variable will be a string with the file content.

8.2 Files and Sequential Access

A file consists of bytes, and `read/write` operations access those bytes *sequentially*. In sequential access, the current I/O operation updates the file state so that the next I/O operation will resume from the end of the current I/O operation.

Assume that you have an old MP3 player that supports only the "*play me next 10 s*" operation on a button. Pressing it will play the next 10 s of the song. When you press again, it will resume from where it is left off and play for another 10 s. This follows until the song is over. The sequential access is similar. A *file pointer* keeps the current offset of the file and each I/O operation advances it so that the next call will read or write from this new offset—see Fig. 8.1.

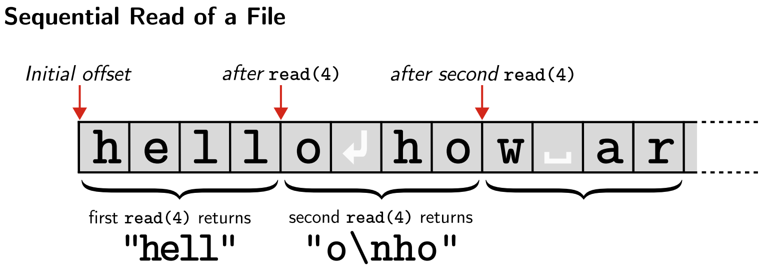


Fig. 8.1 Sequential reading of a file

The following is a sample program illustrating sequential access:

Hands-on Code 8.3

https://pp4e.online/c8s3

```
fp = open("firstexample.txt","r") # the example file we created above

for i in range(3):          # repeat 3 times
   content = fp.read(4)     # read 4 bytes in each step
   print("> ", content)     # output 4 bytes preceded by >

fp.close()
```

Output

```
>  hell
>  o
ho
>  w ar
```

The text in the file was:

```
hello
how are
you?
```

The first `read()` reads `'hell'`, the second reads `'o\nho'` (note that `\n` stands for a new line so that `ho` is printed on a new line), and the third reads `'w ar'`. After these operations, the file offset is left at a position so that the following reads will resume from content `'e\nyou\n'`.

We provided the example with 4-byte read operations. However, for text files, the typical scenario is reading characters line by line instead of fixed size strings.

8.3 Data Conversion and Parsing

A file, specifically a text file, consists of strings. However, especially in engineering and science, we work with numbers. A number is represented in a text file as a sequence of characters including digits, a sign prefix (`'-'` and `'+'`) and at most one occurrence of a dot (`'.'`). That means you may use π as `3.1416` in your Python program. However, in the text file, you store `'3.1416'`, which is a string consisting of chars `'3', '.', '1', '4', '1', '6'`.

Hands-on Code 8.4

https://pp4e.online/c8s4

```
pi = 3.1416
pistr = '3.1416'
print(pi+pi,':', pi * 3)
print(pistr+pistr,':', pistr *3)
```

Output

```
6.2832 : 9.4248
3.14163.1416 : 3.14163.14163.1416
```

Note that the second line of output above is a result of Python interpreting `+` operator as string concatenation, and `*` as adjoining multiple copies of a string. If we need to treat numbers as numbers, we need to convert them from strings. There are two handy functions for this: `int()` and `float()` convert a string into an integer and a floating point value, respectively. Here is an illustration:

Hands-on Code 8.5

https://pp4e.online/c8s5

```
pistr = '  0.31416E01  '
nstr = ' 47 '

# Convert numbers in the strings into numerical data types:
piflt = float(pistr)
nint = int(nstr)

print(piflt*2, nint*2)
```

Output

```
6.2832 94
```

Note that we cannot call `int('3.1416')` since the string is not a valid integer. That brings us another challenge of making sure that the strings we need to convert are actually numbers. Obviously, `int('hello')` and `float('one point five')` will not work either. The mechanisms for dealing with such errors are left for the next chapter. In this chapter, we assume that we have our data carefully created and all conversions work without any error.

Our next challenge is having multiple numbers in a string separated by special characters or simply spaces, e.g., `'10.0 5.0 5.0'`. In this case, we need to decompose a string into string pieces representing numbers, so that we will have `'10.0', '5.0', '5.0'` for the above string. The next step will be converting them into numbers:

$$\texttt{'10.0 5.0 5.0'} \xrightarrow{Step\ 1} \texttt{['10.0', '5.0', '5.0']} \xrightarrow{Step\ 2} \texttt{[10.0, 5.0, 5.0]}$$

For the first step, we will use the `split()` method of a string. A string, or the variable containing the string, is followed by `.split(delimiter)`, which returns a list of strings separated by the given delimiters. The delimiters are removed and all values in between are put in the list—for example:

Hands-on Code 8.6

https://pp4e.online/c8s6

```
print('a:b:c'.split(':'))
print('hello darkness, my old friend'.split(' '))
print('a <=> b <=> c'.split(' <=> '))
print('multiple       spaces          are         tricky'.split(' '))
a = '10.0 5.0 5.0'
print(a.split(' '))
```

Output

```
['a', 'b', 'c']
['hello', 'darkness,', 'my', 'old', 'friend']
['a', 'b', 'c']
['multiple', '', '', '', '', '', '', 'spaces', '', '', '', '', '', '', '', '', '', 'are', '', '', '', '',
↪ '', '', '', '', 'tricky']
['10.0', '5.0', '5.0']
```

For the second step, we will use the `float()` function on a list (or the `int()` function if you have a list of integers). We have a couple of options for this. One is to start from an empty list and append the converted value at each step:

Hands-on Code 8.7

https://pp4e.online/c8s7

```
instr = '10.0 5.0 5.0'
outlst = []

# Go over each substring
for substr in instr.split(' '):
  outlst += [float(substr)]    # Convert each element to float and append it to the list

print(outlst)
```

Output

```
[10.0, 5.0, 5.0]
```

A more practical and faster version will be list comprehension, which is the compact version of mapping a value into another as:

Hands-on Code 8.8

https://pp4e.online/c8s8

```
instr = '10.0 5.0 5.0'

outlst = [float(substr) for substr in instr.split(' ')]

print(outlst)
```

Output

```
[10.0, 5.0, 5.0]
```

As we have explained in Sect. 5.2.6, the syntax is similar to the set/list notation in mathematics:

$\{float(s) \mid s \in S\}$ versus `[float(s) for s in S]`

If you need to have multiple spaces within the values, you can use "`import re`" and call "`re.split(' +', inputstr)`" instead of "`inputstr.split(' ')`". This will split the `'multiple      spaces   are    tricky'` example above into four words without spaces. How it works is beyond the scope of the book. Curious readers can refer to the "`re`" and "`parse`" modules for more advanced forms of input parsing. These are not trivial modules for beginners.

Now, let us consider the reverse of the operation: Assume we have a list of integers and we like to convert that into a string that can be written to a file. For this, we follow the inverse of the steps:

$$\texttt{[10.0, 5.0, 5.0]} \xrightarrow{Step\ 1} \texttt{["10.0", "5.0", "5.0"]} \xrightarrow{Step\ 2} \texttt{"10.0 5.0 5.0"}$$

The first step will be handled with the `str()` function which converts any Python value into a human-readable string:

Hands-on Code 8.9

https://pp4e.online/c8s9

```
inlst = [10.0, 5.0, 5.0]

outlst = [str(num) for num in inlst]

print(outlst)
```

Output

```
['10.0', '5.0', '5.0']
```

The next step is to join those elements with a delimiter, which is the reverse of the `split()` operation. Not by accident, the name of this operation is `join()`. `join()` is a method of the delimiter string and list is the argument. `':'.join(['hello', 'how', 'are', 'you?'])` returns `'hello:how:are:you?'`.

Hands-on Code 8.10

https://pp4e.online/c8s10

```python
inlst = [10.0, 5.0, 5.0]

outlst = [str(num) for num in inlst]

print(' '.join(outlst))
```

Output

```
10.0 5.0 5.0
```

A more advanced way of converting values into strings is called *formatted output* and briefly introduced in Sect. 8.7.

8.4 Accessing Text Files Line by Line

Files consisting of human-readable strings are called *text files*. Text files consist of strings separated by the *end-of-line* character `'\n'`, also known as *new line*. The sequence of characters in a file contains the end-of-line characters so that a text editor will end the current line and show the following characters on a new line. We use end-of-line characters so that logically relevant data is on the same line. For example:

```
4
10.0 20.0
15.5 22.2
3 44
10 10.5
```

Let us assume the integer value 4 on the first line denotes how many lines will follow. Assume also that each of the following four lines has two real values, denoting *x* and *y* values of a point. In this way, we can represent our input separated by the end-of-line characters for each point and by the space character for each value in a line.

Let us create such a text file from a Python list. Please note that the file `read()` function returns a string and the `write()` function expects a string argument. In other words, calling `write(3.14)` will fail. In order to make the conversion, we use the `str()` function for numeric values and call `write(str(3.14))` instead. Another tricky point is that `write()` does not put the end-of-line character automatically. You need to insert it in the output string or call an extra `write("\n")`.

Hands-on Code 8.11

https://pp4e.online/c8s11

```
pointlist = [(0,0), (10,0), (10,10), (0,10)]

fp = open("pointlist.txt", "w")       # open file for writing
fp.write(str(len(pointlist)))         # write list length
fp.write('\n')

# Go over each point in the list
for (x,y) in pointlist:              # for each x,y value in the list
    fp.write(str(x))                   # write x
    fp.write(' ')                      # space as number separator
    fp.write(str(y))                   # write y
    fp.write('\n')                     # \n as line separator

fp.close()

# let us read the content to verify what we wrote
fp = open("pointlist.txt")           # open for reading
content = fp.read()
print(content)
fp.close()
```

Output

```
4
0 0
10 0
10 10
0 10
```

Using `read()` will get the whole content of the file; if the file is large, your program will use too much memory and processing the data will be difficult. In such a situation, we can access a text file line by line using the `readline()` function.

Let us write a program to read and output the content of a text file. We need a loop to read the file line by line and output. Here, when we are going to stop the loop is crucial. Python's `read()` and `readline()` functions return an empty string ('') when there is nothing left to read. We can use this to stop reading:

Hands-on Code 8.12

https://pp4e.online/c8s12

```
fp = open("pointlist.txt")              # open file for reading

nextline = fp.readline()                # read the first line
while nextline != '':                   # while read is successful
    print(nextline)                       # output the line
    nextline = fp.readline()              # read the nextline

fp.close()                              # when nextline == '' loop terminates
```

Output

```

4

0 0

10 0

10 10

0 10
```

Please note the empty lines between each output line. This is due to the `'\n'` character at the end of the string that `readline()` returns. In other words, `readline()` keeps the new line character it reads. `print()` puts an end-of-line after the output (this can be suppressed by adding an `end=''`

argument). As a result, we have the extra end-of-line at the end of each line. In order to avoid it, you can call `rstrip(’\n’)` on the returned string to remove the end-of-line. The new code will be:

Hands-on Code 8.13

https://pp4e.online/c8s13

```
fp = open("pointlist.txt")            # open file for reading

nextline = fp.readline()              # read the first line
while nextline != '':                 # while read is successful
    nextline = nextline.rstrip('\n')    # remove occurrences of '\n' at the end
    print(nextline)                     # output the line
    nextline = fp.readline()            # read the nextline

fp.close()
```

Output

```
4
0 0
10 0
10 10
0 10
```

Converting this file into the initial Python list `[(0,0), (10,0), (10,10), (0,10)]` is our next challenge. This requires conversion of a string as `"0 0\n"` into `(0,0)`. The first one is of type `str` whereas the second is a tuple of numeric values. We can use `int()` or `float()` functions to convert strings into numbers. Note that the string should contain a valid representation of a Python numeric value: `int("hello")` will raise an error.

The second issue is separating two numbers in the same string. We can use `split()` function followed by the separator string as in `nextline.split(’ ’)`. This call will return a sequence of strings from a string. If the separator does not occur in the string, it will return a list with one element. If there is one separator, it will return two elements. For n occurrences of the separator, it will return a list with $n - 1$ elements.

Here is the solution in Python:

Hands-on Code 8.14

https://pp4e.online/c8s14

```
fp = open("pointlist.txt")            # open file for reading

pointlist = []                        # start with empty list

nextline = fp.readline()              # read the first line
n = int(nextline)                     # find number of lines to read

for i in range(n):                    # repeat n times
    nextline = fp.readline()            # read the nextline
    nextline = nextline.rstrip('\n')    # remove occurrences of '\n' at the end
    (x, y) = nextline.split(' ')        # get x and y (note that they are still strings)
    x = float(x)                        # convert them into real values
    y = float(y)
    pointlist.append( (x,y) )           # add tuple at the end

fp.close()
print(pointlist)                      # output the resulting list
```

Output

```
[(0.0, 0.0), (10.0, 0.0), (10.0, 10.0), (0.0, 10.0)]
```

8.5 Termination of Input

There are two ways to stop reading input:

1. By reading a definite number of items.
2. By the end of the file.

In our previous examples, we read an integer that told us how many lines followed in the file. Then, we called `readline()` in a `for` loop with the given number of lines. This is an example of the first case which provides a definite number of items.

The alternative is to read lines in a `while` loop until a termination condition arises. The termination condition is usually the *end of file*, the case where functions like `read()` and `readline()` return an empty string `''`.

Hands-on Code 8.15

https://pp4e.online/c8s15

```
fp = open(”pointlist.txt”)              # open file for reading

pointlist = []                          # start with empty list
nextline = fp.readline()                # skip the first line (4) since we don’t need it

nextline = fp.readline()                # read the first line
while nextline != '':                   # until end of file
    nextline = nextline.rstrip('\n')    # remove occurrences of '\n' at the end
    (x, y) = nextline.split(' ')        # get x and y (note that they are still strings)
    x = float(x)                        # convert them into real values
    y = float(y)
    pointlist.append( (x,y) )           # add tuple at the end
    nextline = fp.readline()              # read the nextline

fp.close()
print(pointlist)
```

Output

```
[(0.0, 0.0), (10.0, 0.0), (10.0, 10.0), (0.0, 10.0)]
```

Note that the example above skips (reads and throws away) the first line in our input file so that the integer on the first line is ignored. When your input does not contain such an unnecessary value, you can delete this line.

Sometimes termination can be marked explicitly by a *sentinel value* which is a value marking the end of values. This is especially useful when you have multiple objects to read:

Hands-on Code 8.16

https://pp4e.online/c8s16

```python
# First, create a file named `twopointlists.txt`
fp = open("twopointlists.txt", "w")
fp.write("""3 0
3.4 2.1
5.1 3.2
EOLIST
1 1.5
2.0 2.5""")
fp.close()
```

Output

```
3 0
3.4 2.1
5.1 3.2
EOLIST
1 1.5
2.0 2.5
```

In this input, there are two lists with arbitrary sizes and we use a word of our choice, `EOLIST` to separate them. We can use the `EOLIST` word to input distinct lists as follows:

Hands-on Code 8.17

https://pp4e.online/c8s17

```python
fp = open("twopointlists.txt")
pntlst1 = []                        # start with empty list
pntlst2 = []                        # start with empty list

nextline = fp.readline()            # read the first line
while nextline != 'EOLIST\n':       # sentinel value
    nextline = nextline.rstrip('\n') # remove occurrences of '\n' at the end
    (x, y) = nextline.split(' ')    # get x and y (note that they are still strings)
    x = float(x)                    # convert them into real values
    y = float(y)
    pntlst1.append( (x,y) )         # add tuple at the end
    nextline = fp.readline()        # read the nextline

# first list has been read, now continue with the second list from the same file
nextline = fp.readline()
while nextline != '':               # until end of file
    nextline = nextline.rstrip('\n') # remove occurrences of '\n' at the end
    (x, y) = nextline.split(' ')    # get x and y (note that they are still strings)
    x = float(x)                    # convert them into real values
    y = float(y)
    pntlst2.append( (x,y) )         # add tuple at the end
    nextline = fp.readline()        # read the nextline

fp.close()
print('List 1:', pntlst1)
print('List 2:', pntlst2)
```

Output

```
List 1: [(3.0, 0.0), (3.4, 2.1), (5.1, 3.2)]
List 2: [(1.0, 1.5), (2.0, 2.5)]
```

8.6 Example: Processing CSV Files

CSV stands for *Comma-Separated Value*; it is a text-based format for exporting/importing *spreadsheet* (i.e., Excel) data. Each row in a CSV file is separated by a newline, and each column is separated by a comma (`,`). Actually, the format is more complex but for the time being, let us ignore commas that might be appearing in strings and focus on a simple form as follows:

```
Name,Surname,Age
Ada,Lovelace,37
John,von Neumann,53
Alan,Turing,42
Edsger W.,Dijkstra,72
Donald,Knuth,85
Leslie,Lamport,82
Dennis,Ritchie,70
```

Usually, the first line is reserved for the names of the columns in a spreadsheet. Now, let us create this file:

Hands-on Code 8.18

https://pp4e.online/c8s18

```
content = '''Name,Surname,Age
Ada,Lovelace,37
John,von Neumann,53
Alan,Turing,42
Edsger W.,Dijkstra,72
Donald,Knuth,85
Dennis,Ritchie,70'''
fp = open("input/first.csv", "w")     # open for writing
fp.write(content)               # write in a single operation, practical for small files
fp.close()
```

Our next task is to read this file into the memory as a list of dictionaries, as: `[{"Name":"Ada", "Surname":"Lovelace", "Age":"37"}, …]`

We need to read the file line by line, extract the components using the `split()` function, then create the dictionary. Then, we can append it to a result list. For example:

Hands-on Code 8.19

https://pp4e.online/c8s19

```
fp = open("first.csv","r")             # open for reading

line =  fp.readline()                  # read column names
line = line.rstrip('\n')               # get rid of new line
colnames = line.split(',')             # list of column names

result = []                            # resulting list of dictionaries
line = fp.readline()
while line != '':                      # end-of-file check
    line = line.rstrip('\n')
    entry = {}                         # start with empty dictionary
    c = 0                              # a counter to address column number
    for v in line.split(','):          # in a loop process each column of the row
        entry[colnames[c]] = v         # column name is index, value is from current row
        c += 1
    result.append(entry)               # add dictionary to result
    line = fp.readline()               # read next line

fp.close()
print(type(result))
print(result)
```

Output

```
<class 'list'>
[{'Name': 'Ada', 'Surname': 'Lovelace', 'Age': '37'}, {'Name': 'John', 'Surname': 'von Neumann', 'Age':
↪'53'}, {'Name': 'Alan', 'Surname': 'Turing', 'Age': '42'}, {'Name': 'Edsger W.', 'Surname': 'Dijkstra
↪', 'Age': '72'}, {'Name': 'Donald', 'Surname': 'Knuth', 'Age': '85'}, {'Name': 'Dennis', 'Surname':
↪'Ritchie', 'Age': '70'}]
```

Let us improve this example by adding a column as a result of a computation. Let us calculate the grade average and show the difference from the average as a new column. We need to go over all grade values in the list, convert them to real values (so that we can do arithmetic on them), calculate the average, then go over all rows to add a new column. Then, go over the list again to export/write it into a new CSV file.

Hands-on Code 8.20

https://pp4e.online/c8s20

```
n = 0
# Calculate the average
sum = 0

for entry in result:
    sum += float(entry['Age'])
    n += 1
average = sum / n

# Calculate the difference of each age from the average
for entry in result:
    entry['Avgdiff'] = str(float(entry['Age']) - average)

# Write the updated content into another CSV file
fp = open('second.csv', 'w')
colnames = entry.keys()               # this returns the keys (column names) of the CSV file
fp.write(','.join(colnames) + '\n')    # write this as the first line with comma separated values
for entry in result: # Go over each row
    vals = []
    for key in colnames: # Write each column on this row
        vals.append(entry[key])       # extract values of entry, entry.values() is a short version of this
    fp.write(','.join(vals) + '\n')

# Finished, close the file
fp.close()
```

The content of the file “second.csv”, after this code runs is as follows:

```
Name,Surname,Age,Avgdiff
Ada,Lovelace,37,-22.833333333333336
John,von Neumann,53,-6.833333333333336
Alan,Turing,42,-17.833333333333336
Edsger W.,Dijkstra,72,12.166666666666664
Donald,Knuth,85,25.166666666666664
Dennis,Ritchie,70,10.166666666666664
```

8.7 Formatting Files

Sometimes readability is important for text files, especially if data is in a tabular form. For example, seeing all related data in a column starting at the same position can improve readability significantly. The following shows the unformatted and formatted versions of the same data side by side:

```
        UNFORMATTED                                            FORMATTED
Name,Surname,Age,Avgdiff                   Name      , Surname             , Age  , Avgdiff
Ada,Lovelace,37,-22.833333333333336        Ada       , Lovelace            ,    37, -22.833
John,von Neumann,53,-6.833333333333336     John      , von Neumann         ,    53,  -6.833
Alan,Turing,42,-17.833333333333336         Alan      , Turing              ,    42, -17.833
Edsger W.,Dijkstra,72,12.166666666666664   Edsger W. , Dijkstra            ,    72,  12.167
Donald,Knuth,85,25.166666666666664         Donald    , Knuth               ,    85,  25.167
Dennis,Ritchie,70,10.166666666666664       Dennis    , Ritchie             ,    70,  10.167
```

In order to achieve this, you can use the `format()` method of a template string as in

`’{:10}, {:20}, {:3d}, {:7.3f}’.format(’Ada’, ’Lovelace’, 37, -22.833333333336)’`.

Each `{}` in the template matches a data value in the arguments and specifies how the data will be converted as follows:

- The value after `:` denotes the (minimum) width of the data. To make a text fit a specified size with a smaller number of characters, spaces are added to the right side, aligning the text to the left.
- For integers, the number is followed by a `d` to format it as a decimal value space-padded on the left (right aligned).
- When working with floating point values, you can format the output by specifying the number of digits after the decimal point. This is achieved by appending a period (.) followed by the desired digit count and the letter `f` to indicate the value as a float. The fraction part of the number will be rounded to match the specified number of digits.

The detailed description of `format()` is out of the scope of the book. For a detailed description, please refer to the Python reference manuals.

Let us rewrite the output part of the code using formatted output:

Hands-on Code 8.21

https://pp4e.online/c8s21

```
template = '{:10}, {:20}, {:5d}, {:7.3f}\n'
fp = open('third.csv', 'w')
colnames = entry.keys()                   # this returns the keys of the CSV file
fp.write('{:10}, {:20}, {:5}, {:7}\n'.format(*colnames) )      # header, composed of coloumn names
for entry in result:
    fp.write(template.format(entry['Name'],entry['Surname'],int(entry['Age']),float(entry['Avgdiff'])))
             # convert strings to numbers to respect number formatting
fp.close()
```

Producing the file “`third.csv`” with the following content:

```
Name      , Surname             , Age  , Avgdiff
Ada       , Lovelace            ,    37, -22.833
John      , von Neumann         ,    53,  -6.833
Alan      , Turing              ,    42, -17.833
Edsger W. , Dijkstra            ,    72,  12.167
Donald    , Knuth               ,    85,  25.167
Dennis    , Ritchie             ,    70,  10.167
```

8.8 Binary Files

So far, we have only looked at text files where all values are represented as human-readable text and all numerical values are represented as decimal strings. However, if you remember from Chapter 3, computers do not store and process numbers as decimal digit sequences. They store variables in binary format using the two’s complement method and the IEEE 754 floating point standard. In order to process, read and write decimal data in text, programming languages and libraries have to convert data from/to human-readable form to/from the internal form. Even though you will not notice the time spent in conversion in small amounts of data, if you read 10 million numbers, you start spending a significant amount of CPU time for converting data.

Binary files, on the other hand, store numbers as they are stored in the computer’s memory. They are still sequences of bytes, but in a more structured way. For example, a 4-byte integer is kept as a sequence of 4 bytes, each byte is a part of the number in two’s complement form. Reading a binary file is simply copying data to the memory; while doing so, either no conversion is performed or only the order of bytes is changed.

A floating point number `0` takes 1 byte in a text file, but the number `3.1415926535897932384626433832795028` takes 34 bytes. In a binary file, the total size of a number is fixed as the size of the IEEE 754 format, i.e., 4 bytes on a 32-bit computer. Both `0` and the number π are stored in 4 bytes for single precision, 8 bytes for double precision, in a binary file.

Keeping values in binary files have the following advantages:

1. It is more compact: Data occupies less space in the file.
2. No decimal-to-binary conversion is required. More efficient in terms of CPU usage.
3. Since sizes are fixed, randomly jumping to a location and reading relevant data is possible. In a text file, you have to start from the beginning and read all lines up to the relevant data. This kind of usage is a more advanced case and harder to understand for beginners.

On the other hand, using text files has the following advantages:

1. Files are human readable and editable. Users can change data using a standard editor. In binary files, special software has to be used.
2. File format is more flexible, using `variablename:value` patterns in the file, data can be stored in any order in a flexible way. This is why text files are often used as configuration files.

Most of the special formats with `.exe`, `.xls`, `.zip`, `.pdf` extensions are binary file formats.

Note: Binary files are kept out of the scope of this book. The following paragraphs give a couple of pointers for curious readers.

In order to use binary files:

1. You need to add ’b’ character in the second argument of the open() method as: open(’test.bin’, ’rb’) or open(’test.bin’, ’wb’).
2. Binary I/O requires bytes typed values instead of str typed values. bytes is a sequence of bytes. Elements of a byte sequence are not printable in contrast to str.
3. Python has the struct module for converting any value into bytes value. struct.pack(format, values) converts values into bytes. Computationally this conversion is much more cheaper than decimal-to-binary conversion.
4. struct.unpack(format, bytesval) can be used to convert bytes value into Python values. It is much cheaper than binary-to-decimal conversion.
5. read(), write() can be used as usual. In read(nbytes), data size should be given. struct.calcsize(format) can be used to calculate data size from format.

The following is an example of binary I/O. Assume the binary file contains an integer N, for the number of points, followed by $2 \times N$ floating point values. Let us write and then read this data:

Hands-on Code 8.22

https://pp4e.online/c8s22

```
import struct

points = [(1,1), (2.5, 3.4), (5.4,3.3), (2.2, 1.121)]

# 1- Open and write the binary file
fp = open("points.bin", "wb")
fp.write(struct.pack('i', len(points)))   # 'i' denotes a single integer value is converted into bytes

for (x,y) in points:
    fp.write(struct.pack('dd', x, y))     # 'dd' denotes two floating point values are converted into
↪bytes

fp.close()

# 2- Open and read the binary file
fp = open("points.bin", "rb")             # open same file for reading
content = fp.read(struct.calcsize('i'))   # read binary data with length sizeof integer bytes
(n,) = struct.unpack('i', content)        # unpack returns a tuple, 1tuple in this case

newpoints = []
for i in range(n):                        # n times
    content = fp.read(struct.calcsize('dd'))
    (x,y) = struct.unpack('dd', content)  # read two floats
    newpoints += [(x,y)]                  # append value at the end

fp.close()

# 3- Print the read and converted values
print("The read & converted points are:", newpoints)

print("This is what binary data looks like:")
fp = open("points.bin", "rb")
print(fp.read())
fp.close()
```

Output

```
The read & converted points are: [(1.0, 1.0), (2.5, 3.4), (5.4, 3.3), (2.2, 1.121)]
This is what binary data looks like:
b'x04x00x00x00x00x00x00x00x00xf0?x00x00x00x00x00x00xf0?x00x00x00x00x00x00x04@333333x0
b@x9ax99x99x99x99x99x15@ffffffn@x9ax99x99x99x99x99x01@Vx0e-xb2x9dxefxf1?'
```

8.9 Notes on Files, Directory Organization and Paths

Files are organized under directories so that you can put relevant files under the same category together. Operating systems provide a *filesystem hierarchy* consisting of directories (some prefer word *folder* instead) and regular files.

Directories can be arbitrarily nested. You may need to traverse *N* levels of directories to find your file. For example, on a UNIX originated operating system, your program can be in the `/Users/user/Desktop/Homeworks` directory. This means there is a `Homeworks` directory, under the `Desktop` directory, under the `user` directory, under the `Users` directory, under the root directory (`/`), which is the topmost level on your filesystem. The top-level directory and the separator in the MS Windows operating systems, is slightly different, being a backslash, “`\`”. However, “/” works in Python for Windows too.

In order to address a file, we use a *path* which is a sequence of directory names separated by “`/`”, ended by the name of the file. For example, `"/Users/user/Desktop/Homeworks/homework1.py"` is a path for the file named `"homework1.py"`.

A path can be either *full* (absolute) or *relative*. In the former case, a path starts with a slash (`/`). In the relative case, it is considered relative to the current working directory, i.e., the directory where you started your program. Full paths ignore your current directory whereas relative paths depend on it. For example, if you are currently in the `Desktop` directory, the path `"Homeworks/homework1.py"` will address the same full path above.

8.10 List of File Class Member Functions

In Table 8.1, you will find some of the very frequently used member functions of files.

Table 8.1 Frequently used member functions of files. Assume `F` is a file. In the **Operation** column, anything in square brackets denotes that the content is optional—if you enter the optional content, do not type in the square brackets

Operation	Result
`F.seek(offset[, whence=0])`	Set file `F`’s position, like stdio’s `fseek()`. `whence == 0` then use absolute indexing (using `offset`). `whence == 1` then `offset` relative to current pos. `whence == 2` then `offset` relative to file end
`F.tell()`	Return file `F`’s current position (byte offset)
`F.truncate([size])`	Truncate `F`’s size. If `size` is present, `F` is truncated to (at most) that size, otherwise `F` is truncated at current position (which remains unchanged)
`F.write(str)`	Write string *str* to file `F`
`F.writelines(list)`	Write *list* of strings to file `F`. No `EOL` are added
`F.close()`	Close file `F`
`F.fileno()`	Get fileno (fd) for file `F`
`F.flush()`	Flush file `F`’s internal buffer
`F.isatty()`	1 if file `F` is connected to a tty-like dev, else 0
`F.next()`	Returns the next input line of file `F`, or raises `StopIteration` when `EOF` is hit
`F.read([size])`	Read at most `size` bytes from file `F` and return as a string object. If `size` omitted, read to `EOF`
`F.readline()`	Read one entire line from file `F`. The returned line has a trailing `\n`, except possibly at `EOF`. Return `""` on `EOF`
`F.readlines()`	Read until `EOF` with `readline()` and return a list of lines read

8.11 Important Concepts

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

- Sequential access. File access.
- Text files. Reading and writing text files. Parsing a text file.
- End-of-file, new line.
- Formatting files.
- Binary files and binary file access.

8.12 Further Reading

- String formatting in Python: https://docs.python.org/3.4/library/string.html#formatspec [2].
- Working with binary data and files in Python: https://docs.python.org/3/library/binary.html [1].
- Comma-Separated Values (CSV) file format: https://en.wikipedia.org/wiki/Comma-separated_values [3].

8.13 Exercises

1. Write a function that reads a text file with the following format (ignore characters following #):

```
N                    # Number of students
                     # Empty line
Name Surname         # Fist student
M                    # Number of courses that the student has taken
Coursename1: Grade   # Grade is a real number
Coursename2: Grade
Coursename3: Grade
...
CoursenameM: Grade

                     # Empty line
Name Surname         # Second student
P                    # Number of courses that the student has taken
Coursename1: Grade   # Grade is a real number
Coursename2: Grade
Coursename3: Grade
...
CoursenameP: Grade
...
...
...
                     # Empty line
Name Surname         # Last student
Z                    # Number of courses that the student has taken
Coursename1: Grade   # Grade is a real number
Coursename2: Grade
Coursename3: Grade
...
CoursenameZ: Grade
```

The function gets the filename as a parameter and returns a list of dictionaries as follows:

```
[ { 'name':'John', 'surname':'Doe',
    'courses':[ ('ceng240', 'AA'), ('ceng301','BB')]},
  { 'name':'Jane', 'surname':'Doe',
    'courses':[ ('arch101', 'CC') ]},
  { 'name':'Bill', 'surname':'Smith',
    'courses':[ ] }
]
```

The input file for getting this example list is as follows:

```
3

John Doe
2
ceng240 AA
ceng301 BB

Jane Doe
1
arch101 CC

Bill Smith
0
```

2. Write a function that writes a list of dictionaries that have the following format into a text file. You may choose to write the number of elements at the top of the file.

```
{ "city": "Ankara",
  "plate code": "06",
  "max temperature (C)": 40,
  "min temperature (C)": -20,
  "population": 5700000
}
```

The text file should have a dictionary content in each line and all fields should be separated by space character.

```
Ankara 06 40 -20 5700000
Istanbul 34 40 -10 15800000
```

3. Write a function that reads the file that you have written in the previous question and returns a list of dictionaries.
4. You are given a text file with fixed-column size records on each line. The fields names, their widths in the input, and data types are given as follows:

- `Name:` 20, string.
- `Surname:` 20, string.
- `Age:` 4, integer.
- `Height:` 8, float.
- `Weight:` 8, float.

A sample file is as follows:

```
           Johnathan              Motley  28    2.06  104.00
             Metecan              Birsen  28    2.08  101.00
             Scottie            Wilbekin  30    1.88   80.00
               Melih          Mahmutoğlu  33    1.91   85.00
               Nigel               Hayes  28    2.03  103.00
             Dyshawn              Pierre  29    1.98  104.00
               Tonye              Jekiri  28    2.12  103.00
               Marko              Gudurić  28    1.98   91.00
               Tyler              Dorsey  26    1.93   82.00
               Devin              Booker  32    2.06  113.00
                Nick            Calathes  34    1.98   97.00
```

Write a function to read this into a list of dictionaries with the field names in the description. The function gets the file name as a parameter and returns the list of dictionaries. Instead of reading the file line by line and parsing afterward, try to read it field by field as:

```
    name = fp.read(20)
    surname = fp.read(20)
    age = fp.read(4)
    height = fp.read(8)
    weight = fp.read(8)
    rest = fp.readline()     # read remainder
```

Do not forget to convert the fields into their numerical types in the resulting dictionary.

5. Assume you would like to extend the above function into a generic fixed-column reader. Your field names, widths and types are given in a header line in the input as:

```
city,15,s:population,12,i:mintemp,8,f:maxtemp,8,f
          Ankara    5700000  -20.0     40.5
        Istanbul   15800000  -12.0     42.0
...
```

Write a function that reads the header in the first line, then read and construct the list of dictionaries based on this header.

6. Three-dimensional surfaces can be represented as a collection of polygons. A polygon consists of a list of 3D points and a 3D point consists of 3 `float` values. The following format defines a set of polygons (triangles, quadrilaterals, etc.) from a set of points. First, points (nodes) are defined as 3 coordinate values separated by space. Then, polygons are defined by referring to a point with their numbers. For example, face "`1 5 6`" defines a triangle from points 1, 5, and 6 in the input.

```
NODES
0 0 0
1 0 0
0.5 1 0
0 0 1
1 0 1
0.5 1 1
FACES
0 1 2
0 1 3 4
1 2 4 5
0 2 3 5
3 4 5
```

This input defines a triangle prism.

Write a function that converts the file above into a file with a list of points per polygon format as follows:

```
0,0,0 1,0,0 0.5,1,0
0,0,0 1,0,0 0,0,1 1,0,1
1,0,0 0.5,1,0 1,0,0 0.5,1,1
0,0,0 0.5,1,0 0.5,1,0 0.5,1,1
0,0,1 1,0,1 0.5,1,1
```

Each line consists of space-separated points where each point is comma-separated coordinate values. The function gets two parameters: The name of the input file and the name of the output file.

7. Write a function that reverses the conversion in the previous exercise. In other words, it should read a file with a list of points of a polygon format and write a file in NODES/FACES format. Note that points are repeated in the input. Put new points in a dictionary and match old points with their previous number in the dictionary. Generate the output afterward.
8. You are given inter-city distances in a file with the following format:

```
city1 city2 distance1
city3 city4 distance2
city2 city4 distance3
...
```

The input terminates with the end-of-file. The distance information is symmetrical and a pair of cities is only given once in the input. Write a function that reads an input file with the above format and writes a file in a distance matrix format described below.

The distance matrix consists of distance values organized into a tabular form. Since the distance is symmetric, the matrix contains only the upper triangular part of the table as:

```
              Paris  Istanbul    Madrid      Rome
Berlin         1054      2191      2319      1502
Rome           1421      2245      1957
Madrid         1274      3544
Istanbul       2731
```

Assume the constructed city list is `Paris, Istanbul, Madrid, Rome, Berlin`. Note that the cities in the rows are in reverse order. Also last items in the rows and columns are omitted. If two cities have no distance information, print a dash “`-`”.

9. Write a function that makes the reverse of the conversion above. Use “`re.split(’ +’, inputline)`” from the “`re`” library to split with multiple spaces.

10. Write a function that reads a file containing match details in the following format:

```
Team1:Team2:Day:City
ALBA Berlin:ASVEL Villeurbanne:13 April 2023:Berlin
Crvena Zvezda:Fenerbahce:13 April 2023:Belgrade
Olympiacos Piraeus:Cazoo Baskonia:13 April 2023:Piraeus
Anadolu Efes:AS Monaco:13 April 2023:Istanbul
ASVEL Villeurbanne:Cazoo Baskonia:15 April 2023:Lyon
AS Monaco:Crvena Zvezda:15 April 2023:Monaco
Olympiacos Piraeus:ALBA Berlin:15 April 2023:Piraeus
Fenerbahce:Anadolu Efes:15 April 2023:Istanbul
```

The function should convert the data into a "per team" report with the following format:

```
ALBA Berlin:
    ASVEL Villeurbanne:13 April 2023:Berlin
    Olympiacos Piraeus:15 April 2023:Piraeus
Crvena Zvezda:
    Fenerbahce:13 April 2023:Belgrade
    AS Monaco:15 April 2023:Monaco
Cazoo Baskonia:
    Olympiacos Piraeus:13 April 2023:Piraeus
    ASVEL Villeurbanne:15 April 2023:Lyon
Anadolu Efes:
    AS Monaco:13 April 2023:Istanbul
    Fenerbahce:15 April 2023:Istanbul
ASVEL Villeurbanne:
    ALBA Berlin:13 April 2023:Berlin
    Cazoo Baskonia:15 April 2023:Lyo
...
```

Note that each match is reported twice. The order of teams and games is not important.

11. Consider the reverse of the operation above. Write a function that reads the input in a "per team" report format and outputs all games in a colon-separated format. Assume the games are not repeated in the input and the team at the header is reported as the first team in the output. Note that the team header and other lines are distinguished by the leading space.
12. Write a function that reads the text file given below that contains two lists with arbitrary lengths separated by the word `EOLIST`.

```
3 0 3.4 2.1 5.1 3.2 EOLIST 1 1.5 2.0 2.5
```

The function should write these two lists to a binary file. The format of the binary file consists of the list size as an integer, let's call it N, followed by N floating point values. This pattern is repeated twice for the two lists. Hint: Example code in Sect. 8.8 writes a single list in a binary file with this format. Repeat this twice for the input.

13. Write a function that does the reverse of the operation above: reads the two lists from the binary file and writes them to a text file.

References

[1] Python Documentation, Binary data and files in python (2023). https://docs.python.org/3/library/binary.html. Accessed 5 Sept 2023

[2] Python Documentation, String formatting (2023). https://docs.python.org/3.4/library/string.html#formatspec. Accessed 5 Sept 2023

[3] Wikipedia contributors, Comma-separated values, {Wikipedia} {, } The Free Encyclopedia (2023). https://en.wikipedia.org/w/index.php?title=Comma-separated_values&oldid=1171747909. Accessed 5 Sept 2023

9 Error Handling and Debugging

As a Turkish song says "*There is no servant without fault, love me with the faults I have*", we all make mistakes, and programming is far from being an exception to this. It is a highly technical, error-intolerant task that requires full attention from the programmer. Even a single letter that you mistype can produce the complete opposite of what you aim to get.

The history of computing is full of examples of large amounts of money wasted on small programming mistakes (banks losing millions due to penny roundings, satellites turning into the most expensive fireworks, robots getting stuck on Mars's surface, etc.). Even more concerning, human lives can be affected by the proper functioning of a program.

For this reason, writing programs as error-free as possible is an important challenge and responsibility for a programmer. Programming errors can be classified into three groups:

1. Syntax errors.
2. Run-time errors.
3. Logical errors.

9.1 Types of Errors

9.1.1 Syntax Errors

Syntax errors are due to the strict well-formedness requirements of programming languages. In a natural language essay, we can use no punctuation at all; we can use silly abbreviations, mix the ordering of words, make typing mistakes, and still it will make sense to a reader (though your English teacher might barely give you points).

Computer programs are entirely different. When a programming language specification tells you to define blocks based on indentation, to match parentheses properly, to follow certain statements with some certain punctuation (i.e., loops and functions and "`:`"), you have to obey that. Because our compilers and interpreters cannot convert a program with syntax errors into a machine-understandable form.

S. Kalkan et al., *Programming with Python for Engineers*,
https://doi.org/10.1007/978-3-031-57148-0_9

The first step of the Python interpreter reading your program is to break it into parts and construct a machine-readable structure out of it. If it fails, you will get a syntax error, for example:

```
>>> for i in range(10)
...    print(i)

File "<ipython-input-1-12d72cac235a>", line 1
    for i in range(10)
                      ^
SyntaxError: invalid syntax
```

```
>>> x = float(input())
>>> a = ((x+5)*12+4

File "<ipython-input-2-dead5b360d91>", line 2
  a = ((x+5)*12+4
                 ^
SyntaxError: invalid syntax
```

```
>>> s = 0
>>> for i in range(10):
...  s += i
...    print(i)

File "<ipython-input-3-c3ef5d622e47>", line 4
  print(i)
  ^
IndentationError: unexpected indent
```

```
>>> while x = 4:
...    s += x

File "<ipython-input-4-befcf7769cec>", line 1
  while x = 4:
          ^
SyntaxError: invalid syntax
```

In the examples above, the following syntax errors are present:

1. "`:`" is missing at the end of the `for` header.
2. The first parenthesis does not have a matching closing parenthesis.
3. Different levels of indentation are used in the loop body.
4. `while` expects a Boolean expression, but an assignment is given ("`=`" is used instead of "`==`").

These are only a small sample of a large number of possible syntax errors one can do.

Syntax errors are the most *innocent* errors since you are notified of the error immediately when you start running your program. Running your program once will give you the exact spot (though sometimes matching parentheses and quote can be non-trivial, causing the interpreter to incorrectly point to an error in an adjacent line instead) in the error output. If you have learned the syntax of your language, you can fix a syntax error with a small effort.

9.1.2 Type Errors

Python is an interpreted language and it does not make strict type checks as a compiler would check type compatibility at compile time. Though, when it comes to performing an operation that is not compatible with the current type of data, Python will complain. For example, when you try to override a string element with an integer value, use an integer in a string context, or attempt to index a float variable, etc., you will get an error:

```
astr = 'hello'
bflt = 4.56
cdict = {'a':4, 'b':5}

print(astr ** 3)       # third power of a string
print(bflt[1])         # select first member of a float
print(cdict * 2)       # multiply a dictionary by two
cdict < astr           # compare a dictionary with a string
```

These will lead to the following errors:

```
>>> print(astr ** 3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for ** or pow(): 'str' and 'int'
>>> print(bflt[1])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'float' object is not subscriptable
>>> print(cdict * 2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for *: 'dict' and 'int'
>>> cdict < astr
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: '<' not supported between instances of 'dict' and 'str'
```

Compiled languages and some interpreters enforce type compatibility at compile time and treat all such errors as syntax errors, providing a safer programming experience at run time. However, Python and most other interpreters wait until the command is first executed and raise the error at the execution time, causing a *run-time error*.

9.1.3 Run-Time Errors

Run-time errors are sneakier compared to syntax and other compile-time errors. Your program initiates execution smoothly; traverses numerous loops and functions without encountering any errors; generates intermediate output along the way; and then, at the most unexpected juncture, it abruptly throws an error, leaving you with a disheartening error message instead of the anticipated satisfactory result.

The following example gets an input value and counts how many of the integers in the range [1, 1000] are divisible by the input value.

Hands-on Code 9.1

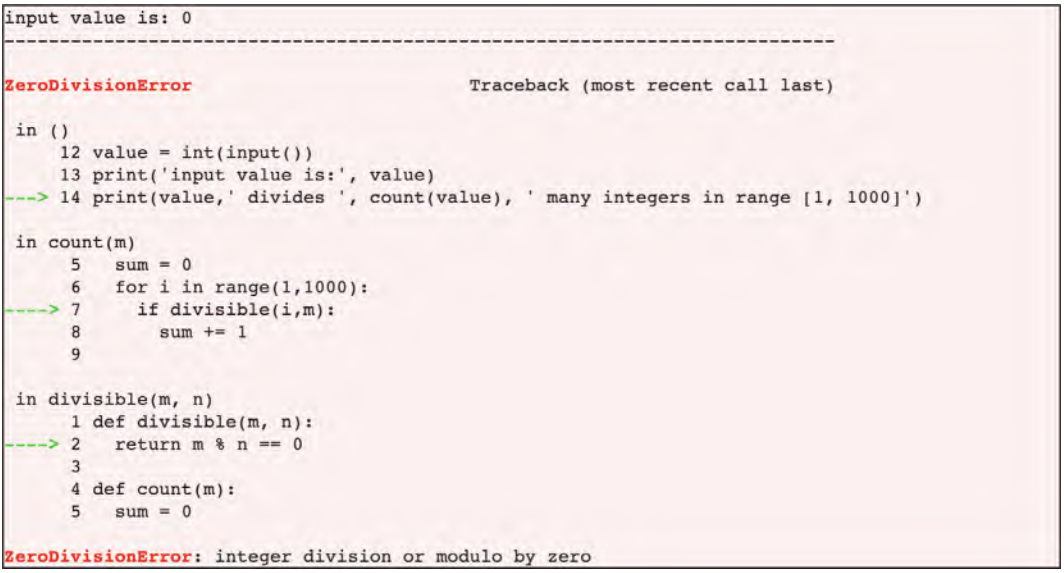
https://pp4e.online/c9s1

```
def divisible(m, n):
    return m % n == 0

def count(m):
    sum = 0
    for i in range(1,1000):
        if divisible(i, m):
            sum += 1

    return sum

value = int(input())
print('input value is:', value)
print(value,' divides ', count(value), ' many integers in range [1, 1000]')
```

It works without any problem for non-zero input values. However, if you enter `0`, it outputs the long message in Fig. 9.1:

This output is called an *Error Traceback*. When a run-time error occurs, you will get one of these and the execution of the program will be terminated. Depending on the settings, the output may be more brief but you will get line numbers and the line leading to the error.

The classification of the error is defined as an *Exception* and colored red in the error output. It is displayed in the first line and repeated with its explanation in the last line. Traceback also reports the functions involving the error, from the outermost to the innermost. Each arrow (colored green) marks the program line causing the error. The last item in the traceback is the actual position where the error occurred; the 2nd line having `return m % n == 0` in this case. The `divisible()` function containing the error line is called by the 7th line, the `count()` function, which is called by the 14th line of the program.



```
input value is: 0
---------------------------------------------------------------------------

ZeroDivisionError                          Traceback (most recent call last)

 in ()
     12 value = int(input())
     13 print('input value is:', value)
---> 14 print(value,' divides ', count(value), ' many integers in range [1, 1000]')

 in count(m)
      5    sum = 0
      6    for i in range(1,1000):
----> 7      if divisible(i,m):
      8        sum += 1
      9

 in divisible(m, n)
      1 def divisible(m, n):
----> 2    return m % n == 0
      3
      4 def count(m):
      5    sum = 0

ZeroDivisionError: integer division or modulo by zero
```

Fig. 9.1 A traceback example for a run-time error

By looking at the error output, you will see the whole chain of calls that led your program to the error state. The last one is the most important one, showing where Python tried to execute an operation and failed.

The reason for failure is `ZeroDivisionError` in this case. If you trace the program with the input value 0, you will see that Python tried to evaluate `m % 0 == 0` for some `m` value. The remainder operator is division based and dividing something by 0 is mathematically impossible. Most programming languages reject this operation with a run-time error. The Python style is to reject this operation with a `ZeroDivisionError` exception.

The following are the most common exceptions and run-time errors in Python:

Exception	Reason
`KeyboardInterrupt`	User pressed `Ctrl-C`; not an error but user intervention
`ZeroDivisionError`	Right-hand side of `/` or `%` is 0
`AttributeError`	Object/class does not have a member
`EOFError`	`input()` function gets End-of-Input by user
`IndexError`	Container index is not valid (negative or larger than length)
`KeyError`	`dict` has no such key
`FileNotFoundError`	The target file of `open()` does not exist
`TypeError`	Wrong operand or parameter types, or wrong number of parameters for functions
`ValueError`	The given value has the correct type but the operation is not supported for the given value

The following code illustrates each of these errors (except for the user-generated ones):

```python
b = 0
a = a / b                # ZeroDivisionError

x = [1,2,3]
print(x.length)          # AttributeError: lists don’t have a length attribute (use len(x))

print(x[4])              # IndexError: last valid index of list x is 2

person = { 'name' : 'Ada', 'surname': 'Lovelace'}
print(person['Name'])    # KeyError: person does not have 'Name' key but 'name'

fp = open("example.txt") # FileNotFoundError: file "example.txt" does not exist

print([1,2,3] / 12)      # TypeError: Division is not defined for lists

def f(x, y):
  return x*x+y

print(f(5))              # TypeError:  Only one argument is supplied instead of 2.

print(int('thirtytwo'))  # ValueError: string value does not represent an integer

a,b,c = [1,2,3,4]        # ValueError: too many values on the right hand side
```

9.1.4 Logical Errors

Logical errors are the worst among all errors because Python does not raise an error. It single-mindedly keeps running your program; however, your program has a mistake about what it intends to compute. In other words, it simply does not do what it is supposed to do because of an error.

Such an error can be due to a small typo, improper nesting of blocks, bad initialization of variables, and many other reasons. However, the code segment containing the error has correct syntax; hence, it does not cause a syntax or run-time error. As a result, you will not know that such an error exists until you observe some discrepancies in your output.

The following are a few examples of logical errors:

```
y = x / x+1             # you meant y = x / (x+1), forgetting about precedence

lastresult = 0
def missglobal(x):
    result = x*x+1                # you intend to update the global variable
    if x > result:                # but you assign a local variable instead.
        lastresult = result       # you should have used "global lastresult"


def returnsnothing(x, y):
    y = x*x+y*y
    if x <= y:
        return x                  # if x > y, the function returns nothing
print(returnsnothing(0.1, 0.1))   # does not have any value. prints "None"

s = 1
while i < n:                      # you forgot incrementing i as i+=1
   s += s*x/i                     # loop will run forever. "infinite loop"
```

9.2 How to Work with Errors

In programming, errors are an inevitable part of the process. As you accumulate experience, the frequency of errors tends to decrease, although a few may still occur. Thankfully, there are techniques available to mitigate the occurrence of errors, but achieving a state of absolute error-free programming is not attainable.

The following is a list of strategies for eliminating the errors:

1. Program with care.
2. Place controls in your code.
3. Handle exceptions.
4. Write verification code and raise exceptions.
5. Debug your program.
6. Write test cases.

Programmers at different experience levels can employ these strategies. In more critical scenarios, guaranteeing program quality and performing software testing become vital elements of the software engineering field, and dedicated engineers fulfill these responsibilities.

9.2.1 Program with Care

The best way to write error-free programs is not to cause one in the first place. As an analogy, instead of cleaning your living room every day, you may just choose to keep it tidy in the first place and not to leave garbage around. Programming is similar, a large percentage of errors are due to the careless acts of programmers.

Programming is *not* an evolutionary process where you start from an ugly code and improve it as you correct errors. This way of programming is possible but not efficient. You better spend some time before starting to write code to design your solution and develop a strategy: You should determine which functions you need, which data structures you use, which algorithms you use, and in which order you will write the code. It is a good practice to divide your problem into pieces (functions for example). Write one function at a time and test it before going to the next step.

As you gain more experience as a programmer, this strategy will become more effective. However, there's no reason to refrain from being extra attentive on your very first day of coding.

9.2.2 Place Controls in Your Code

The values that a variable can have or a function may return are not known in advance. Especially, user-supplied values as inputs are completely untrustable. Moreover, there may be other environmental issues. For example, a required file may not exist.

If you have an untrustworthy value, subsequent operations may fail as a result, as illustrated below:

```python
a = [1,2,3]
age = {'Han': 30, 'Leia': 20, 'Luke': 20}

# CASE 1
n = int(input())
print(a[n])          # will fail for n > 2 or n < -2

# CASE 2
name = input()
print(age[name])     # will fail names other than 'Han', 'Leia', 'Luke'

# CASE 3
x = float(input())
y = math.sqrt(x)     # will fail for x < 0
y = 1 / x            # will fail for x == 0
```

In order to deal with such errors, you can check all values, especially the user-supplied ones. Checking all input values before starting a computation is called *Input Sanitization*. For the above example cases, we can sanitize inputs as follows:

Hands-on Code 9.2

https://pp4e.online/c9s2

```python
a = [1,2,3]
age = {'Han': 30, 'Leia': 20, 'Luke': 20}

# CASE 1 with sanitization
n = int(input())
if -len(a) <= n < len(a):
    print(a[n])
else:
    print("n is not valid:", n)

# CASE 2 with sanitization
name = input()
if name in age:      # membership test for dictionaries
    print(age[name])
else:
    print("dictionary does not have member:", name)

# CASE 3 with sanitization
x = float(input())
if x >= 0:
    y = math.sqrt(x)
else:
    print("invalid for sqrt operation: ", x)

if x != 0:
    y = 1 / x
else:
    print("divisor cannot be 0")
```

Initially, writing such conditions may appear to be a tedious task. However, encountering issues related to these conditions while the program is running can be even more troublesome for users. If you are developing a program that aims to be genuinely useful, it is essential to incorporate these types of input checks.

9.2.3 Handle Exceptions

This is an alternative to placing check conditions in your code. Sometimes there are too many conditions, one for each step of your computation, so that each line of code opens a new `if` block and nests like the branches of a tree. Instead of writing a whole nested sequence of `if` ...`else` statements, exceptions give you a chance to handle all errors in the same place. Especially, they allow you to handle the errors in the caller function rather than the original place where the error occurred.

Before we can see how this works, let us look at its syntax first:

```
try:
  ......    # a block with possible errors
  ......    # if there are function calls here
  ......    # and an error occurs in the function, we can handle it here
except exceptionname:     # exceptionname is optional
  .....     # this is error handling block.
  .....     # when there is an error, execution jumps here
```

The `try-except` statement consists of two parts: a block of code (after `try:`) that may generate run-time errors, and one or more `except` clauses to handle those errors. When an error occurs within the `try` block, the execution jumps to the corresponding `except` clause that matches the type of the error. The code within the matching `except` clause is then executed. Multiple `except` clauses can be used to handle different types of exceptions. The `except:` clause, without specifying an exception name, catches all errors and can be used to handle any type of exception within a single block.

The following is the same example of the previous section, but the errors are handled in the same location.

Hands-on Code 9.3

https://pp4e.online/c9s3

```
import math

a = [1,2,3]
age = {'Han': 30, 'Leia': 20, 'Luke': 20}

try:
    n = int(input())
    print(a[n])          # will fail for n > 2 or n < -2

    name = input()
    print(age[name])     # will fail names other than 'Han', 'Leia', 'Luke'

    x = float(input())
    y = math.sqrt(x)     # will fail for x < 0
    y = 1 / x            # will fail for x == 0
except IndexError:
    print('List index is not valid')
except KeyError:
    print('Dictionary does not have such key')
except ValueError:
    print('Invalid value for square root operation')
except ZeroDivisionError:
    print('Division by zero does not have value')
except:
    print('None of the known errors. Somethings happened even if nothing happened')
```

You can try the above example with different inputs:

1. `-3`
2. `2 ’Obi’`
3. `2 ’Han’ -2`
4. `2 ’Han’ 0`
5. `2 ’Han’ 1`

The first will cause `a[n]` to raise an `IndexError`. The second will cause `age[name]` to raise a `KeyError`. The third will cause `math.sqrt(x)` to raise a `ValueError`. The fourth will cause `1 / x` to raise a `ZeroDivisionError`. The last one will have no error and finish the `try` block and continue with the next instruction jumping over `except` clauses. You can also create exceptions with the `raise` statement. `raise ValueError` will create the error.

Exceptions save you from nested `if` …`else` statements for data validation, e.g.:

```
if cond1:
   ..1..
   if cond2:
     ..2..
     if cond3:
       ..3..
       ..4..       # success at last
     else:
       # report error
   else:
     # report error
else:
  # report error
```

is harder to read compared to the following:

```
try:
  if !cond1:
    raise Error

  ..1..

  if !cond2:
    raise Error

  ..2..

  if !cond3:
    raise Error

  ..3..
  ..4..  # success
except :
  ... # Error handling
```

This is flatter and allows you to focus on the actual code.

9.2.4 Write Verification Code and Raise Exceptions

Even if you sanitize the user input and check all data for valid values, your program can still have logical errors and it might calculate incorrect intermediate/final values. If your program consists of multiple steps, a logical error in step one will cause step two to calculate an incorrect value and this will create a *snowball effect*, potentially causing all steps to produce incorrect values.

For example, you write a function for solving second-order equations $ax^2 + bx + c = 0$. You name the function as `solvesecond(a, b, c)` which returns `(x1, x2)` as the roots. You are (*almost*) sure that you always send correct `a, b, c` values to this function. However, it will be safer if you add a check like the following:

```
def solvesecond(a,b,c):
    det = b*b - 4*a*c
    # the following is the verification code
    if det < 0:
        print("Equation has no real roots for", a, b, c)
        raise ValueError
  ....
  ...
```

The `math.sqrt()` function would have raised the exception anyway. However, you added an extra error message about what caused the problem. Unfortunately, in cases of logical errors, there are usually no run-time errors but only false generated results and this may lead to serious problems.

9.2.5 Debug Your Code

The process of identifying and resolving programming errors is known as *debugging*. It entails locating the exact position of the error (the bug), determining the underlying causes, and modifying the code to prevent its recurrence.

Debugging methods are explained in detail in their own sections below.

9.2.6 Write Test Cases

To ensure that your program is functioning correctly and free of logical errors, it is essential to perform thorough testing. Testing is a crucial but challenging aspect of all engineering disciplines.

In order to test a program, you should create a set of inputs, run your program for each input case, and collect the outputs. Then you inspect them. If there is a way of verifying the outputs, you shall verify them. For instance, when solving an equation, you verify its validity by substituting the determined variable back into the equation and checking if it actually holds.

```
(x1, x2) = findrootsecond(a,b,c)

if a*x1*x1 + b*x1 + c != 0 or a*x2*x2 + b*x2 +c != 0:
   print('test failed for', a, b, c, 'roots', x1, x2)
```

You can generate millions of such numbers and automatically verify them. If the problem is not verifiable, you can get a set of known solutions and compare your solutions against the known solutions.

9.3 Debugging

Debugging is an act of looking for errors in a code, finding what causes the errors and correcting them. Since even modest programs can extend to hundreds of lines of code, debugging is not an easy task. Even if a run-time error gives you the exact location of the error, the variable values causing the error could be set in different parts of your code and you will not easily determine how your variable got into that error-causing state.

Debugging involves an iterative process. You break down your program into smaller parts and systematically eliminate each part, ruling out potential error locations one by one. By narrowing down possibilities step by step, you can pinpoint the exact step where the mistake occurred. The outputs of run-time errors assist in expediting this process.

There are several methods for debugging. A programmer may use one or more of them for debugging.

The following is a sample program with an error (you may jump to the Jupyter Notebook page of the chapter and run this interactive example):

Hands-on Code 9.4

https://pp4e.online/c9s4

```python
def startswith(srcstr, tarstr):
    '''check if tarstr starts with srcstr
      like srcstr="abra" tarstr="abracadabra" '''
    for i in range(len(srcstr)): # check all characters of srcstr
        if srcstr[i] != tarstr[i]:  # if does not match return False
            return False
    return True   # if False is not returned yet, it matches


def findstr(srcstr, tarstr):
    '''Find position of srcstr in tarstr'''
    for i in range(len(tarstr)):
        # if scrstr is same as tarstr from i to rest
        # return i
        if startswith(srcstr, tarstr[i:]):
                return i
    return -1

print(findstr("ada", "abracadabra"))
print(findstr("aba", "abracadabra"))
```

Output

```
5
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-3-b00405b7708c> in <module>()
     18
     19 print(findstr("ada", "abracadabra"))
---> 20 print(findstr("aba", "abracadabra"))

1 frames
<ipython-input-3-b00405b7708c> in startswith(srcstr, tarstr)
      3       like srcstr="abra" tarstr="abracadabra" '''
      4     for i in range(len(srcstr)): # check all characters of srcstr
----> 5         if srcstr[i] != tarstr[i]:  # if does not match return False
      6             return False
      7     return True   # if False is not returned yet, it matches

IndexError: string index out of range
```

The first call returns 5, which is printed on the screen, telling us that `"ada"` is at position 5 of `"abracadabra"`, matches after `"abrac"`. However, the second call should return −`1` since the `"aba"` substring does not exist in `"abracadabra"` but it generates an error.

9.3.1 Debugging Using Debugging Outputs

This is one of the simplest and oldest but most effective methods for debugging a program. In this method, you add extra output lines that shed light on the behavior of your program. This way, you can trace where your program diverted from the expected behavior. For example (you may check the Jupyter Notebook page of the chapter and run this interactive example):

Hands-on Code 9.5

https://pp4e.online/c9s5

```
def startswith(srcstr, tarstr):
    '''check if tarstr starts with srcstr
      like srcstr="abra" tarstr="abracadabra" '''
    for i in range(len(srcstr)): # check all characters of srcstr
        print("check if ", srcstr, '!=', tarstr, 'for i=', i)     # <-- DEBUG OUTPUT -----
        if srcstr[i] != tarstr[i]:  # if does not match return False
            return False
    return True   # if False is not returned yet, it matches


def findstr(srcstr, tarstr):
    '''Find position of srcstr in tarstr'''
    for i in range(len(tarstr)):
        # if scrstr is same as tarstr from i to rest

        # return i
        print("calling startswith", srcstr, tarstr[i:])     # <--- DEBUG OUTPUT ----
        if startswith(srcstr, tarstr[i:]):
                return i
    return -1

print(findstr("aba", "abracadabra"))
```

Output

```
calling startswith aba abracadabra
check if  aba != abracadabra for i= 0
check if  aba != abracadabra for i= 1
check if  aba != abracadabra for i= 2
calling startswith aba bracadabra
check if  aba != bracadabra for i= 0
...SOME  OUTPUT LINES ARE DELETED....
check if  aba != abra for i= 2
calling startswith aba bra
check if  aba != bra for i= 0
calling startswith aba ra
check if  aba != ra for i= 0
calling startswith aba a
check if  aba != a for i= 0
check if  aba != a for i= 1
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-6-b415ffe46a64> in <module>()
     19     return -1
     20
---> 21 print(findstr("aba", "abracadabra"))

1 frames
<ipython-input-6-b415ffe46a64> in findstr(srcstr, tarstr)
     15         # return i
     16         print("calling startswith", srcstr, tarstr[i:])
---> 17         if startswith(srcstr, tarstr[i:]):
     18                 return i
     19     return -1

<ipython-input-6-b415ffe46a64> in startswith(srcstr, tarstr)
      4     for i in range(len(srcstr)): # check all characters of srcstr
      5         print("check if ", srcstr, '!=', tarstr, 'for i=', i)
----> 6         if srcstr[i] != tarstr[i]:  # if does not match return False
      7             return False
      8     return True   # if False is not returned yet, it matches

IndexError: string index out of range
```

In the output, you can see that the execution failed after the output "`check if aba != a for i= 1`". Therefore, we can reason that an error occurred at the next step. The following test is

`if srcstr[i] != tarstr[i]` for "`aba`" and "`a`" for `i = 1`.

Apparently, attempting to get `tarstr[1]` if `tarstr` is "`a`" fails. The length of `tarstr` is 1 and, hence, the only valid index is 0. There are different ways to get rid of this error. One quick solution is to test if lengths of `srcstr` and `tarstr` are compatible. `len(tarstr) >= len(srcstr)` should hold in `startswith()`. The programmer should have considered and handled this case. Now, we can correct it as a result of our debugging session.

Of course, debugging output should be added in the correct places with sufficient descriptive information. If you add too much debugging output, you may be lost in the output lines. If there is not sufficient output, you may not locate the error.

For generic tracing of programs with multiple functions, you can use the following Python magic called *decorator*, which reports all function calls with parameters when a function is decorated as (you may check the Jupyter Notebook page and run this interactive example):

Hands-on Code 9.6

https://pp4e.online/c9s6

```
def tracedec(f):
    def traced(*p, **kw):
        print('  ' * tracedec.level + "->", f.__name__,'(',p, kw,')')
        tracedec.level += 1
        val = f(*p, **kw)
        tracedec.level -= 1
        print('  ' * tracedec.level + "<-", f.__name__, 'returns ', val)
        return val
    return traced
tracedec.level = 0

@tracedec
def startswith(srcstr, tarstr):
    '''check if tarstr starts with srcstr
      like srcstr="abra" tarstr="abracadabra" '''
    for i in range(len(srcstr)):    # check all characters of srcstr
        if srcstr[i] != tarstr[i]:  # if does not match, return False
            return False
    return True   # if False is not returned yet, it matches

@tracedec
def findstr(srcstr, tarstr):
    '''Find position of srcstr in tarstr'''
    for i in range(len(tarstr)):
        # if scrstr is same as tarstr from i to rest
        # return i
        if startswith(srcstr, tarstr[i:]):
                return i
    return -1


print(findstr("aba", "abracadabra"))
```

Output

```
-> findstr ( ('aba', 'abracadabra') {} )
  -> startswith ( ('aba', 'abracadabra') {} )
  <- startswith returns  False
  -> startswith ( ('aba', 'bracadabra') {} )
  <- startswith returns  False
  ... SOME OUTPUT LINES ARE DELETED ...
  -> startswith ( ('aba', 'bra') {} )
  <- startswith returns  False
  -> startswith ( ('aba', 'ra') {} )
  <- startswith returns  False
  -> startswith ( ('aba', 'a') {} )
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-2-f5cc3f720290> in <module>()
     30
     31
---> 32 print(findstr("aba", "abracadabra"))

3 frames

  ... SOME OUTPUT LINES ARE DELETED ...

<ipython-input-2-f5cc3f720290> in startswith(srcstr, tarstr)
     15       like srcstr="abra" tarstr="abracadabra" '''
     16     for i in range(len(srcstr)): # check all characters of srcstr
---> 17         if srcstr[i] != tarstr[i]:  # if does not match return False
     18             return False
     19     return True   # if False is not returned yet, it matches


IndexError: string index out of range
```

Putting “`@tracedec`” before the function definitions will give you the entry and return states of these functions. How this decorator works and the “@” syntax are far beyond the scope of this book. You may adapt and use the decorator above if it suits you.

9.3.2 Handle the Exception to Get More Information

One issue with run-time errors is that they do not offer information about the program’s current state, specifically the set of variables. They provide a traceback, indicating the line that caused the exception and the sequence of functions that led to that point. To retrieve variable values, you can handle the exception and include log output in the handler, as shown in the following example.

Hands-on Code 9.7

https://pp4e.online/c9s7

```
def startswith(srcstr, tarstr):
    '''check if tarstr starts with srcstr
      like srcstr="abra" tarstr="abracadabra" '''
    try:
      for i in range(len(srcstr)): # check all characters of srcstr
          if srcstr[i] != tarstr[i]:  # if does not match return False
              return False
    except IndexError:
      print('Error: srcstr: ', srcstr, ', tarstr:', tarstr, ', i:',i)   #<-- DEBUG OUTPUT --
      raise IndexError          # if you like, generate an additional exception
    return True   # if False is not returned yet, it matches

def findstr(srcstr, tarstr):
    '''Find position of srcstr in tarstr'''
    for i in range(len(tarstr)):
        # if scrstr is same as tarstr from i to rest
        # return i
        if startswith(srcstr, tarstr[i:]):
                return i
    return -1

findstr("aba", "abracadabra")
```

Output

```
Error: srcstr:  aba , tarstr: a , i: 1
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-11-b4556283ff77> in startswith(srcstr, tarstr)
      5       for i in range(len(srcstr)): # check all characters of srcstr
----> 6           if srcstr[i] != tarstr[i]:  # if does not match return False
      7               return False

IndexError: string index out of range

During handling of the above exception, another exception occurred:

IndexError                                Traceback (most recent call last)
2 frames
<ipython-input-11-b4556283ff77> in startswith(srcstr, tarstr)
      8     except IndexError:
      9       print('Error: srcstr: ',srcstr, ', tarstr:', tarstr, ', i:',i)
---> 10       raise IndexError          # if you like generate error again
     11     return True   # if False is not returned yet, it matches
     12

IndexError:
```

This way, we get the output in the first line:

```
Error: srcstr: aba , tarstr: a , i: 1
```

which gives us the state of the variables at the moment of the error.

9.3.3 Use the Python Debugger

All decent programming environments come with a *debugger software* which helps a programmer to execute a program step by step, observe the current state, add *breakpoints* (stops) to inspect, and provide a controlled execution environment.

For compiled languages, debuggers are external programs getting the user's program as input. For Python, it is a module called *pdb*, which stands for *Python DeBugger*. If you use an *integrated development environment* (IDE), a debugger is embedded in the tool so you can use it via the graphical user interface controls. Otherwise, you can use the command line interface.

In the Python command line, the only thing you need to do is to input:

```
import pdb
```

at the beginning of your program, and then type:

```
pdb.set_trace()
```

at any point you want the execution to stop and go into the *debugging mode*. When you run your program or call a function, your program will start executing and when the execution hits one of the breakpoints, execution will stop and a debugger prompt "`(Pdb)`" will be displayed:

```
> <ipython-input-14-110393975fb5>(7)startswith()
-> for i in range(len(srcstr)): # check all characters of srcstr
(Pdb) h

Documented commands (type help <topic>):
========================================
EOF    c          d        h         list      q        rv       undisplay
a      cl         debug    help      ll        quit     s        unt
alias  clear      disable  ignore    longlist  r        source   until
args   commands   display  interact  n         restart  step     up
b      condition  down     j         next      return   tbreak   w
break  cont       enable   jump      p         retval   u        whatis
bt     continue   exit     l         pp        run      unalias  where

Miscellaneous help topics:
==========================
exec  pdb
```

The first line tells at which function and line the execution has been stopped. For the following example let us assume that we inserted the `set_trace()` call in the first line of `startswith()` function. When the execution stops, it is possible to get help on the commands that can be entered. When `h` or `help` is typed on the debugger prompt, a list of all the debugger commands is displayed.

Single- or two-letter commands are abbreviated forms of the longer ones (e.g., `a` for `args`, `b` for `break`, `c` for `cont`, `cl` for `clear`, $\cdots$).

After this point, you can use the `next` (`n`) command to execute the program line by line. If the current statement contains a function call, you may choose to go into the function call using the `step` (`s`) command. Otherwise, `next` will call all functions, wait for their return, and execute the current line until the end in a single step. During debugging, the `print` (`p`) command can be used to display the content of a variable. The following is a debugging session output summarizing some of the useful commands of the debugger. Each time execution hits a breakpoint, the debugger prompt will show the current position. You can use the `cont` command to continue execution:

Command	Description
`next`	Execute the current line and stop at the next statement, jump over functions
`step`	Execute the current line, if there is a function enter it
`args`	show arguments of the current function
`break`	Add a new breakpoint. Execution will stop at that line too
`clear`	Remove the breakpoint(s)
`cont`	Continue execution until hitting a breakpoint
`print`	print the current value of the variable
`display`	display the variable value whenever it changes in the current function
`list`	list program at the current line
`ll`	list the current function
`return`	continue execution until the current function returns
`where`	show currently active functions: which function calls brought the code execution flow here

The following is an example run. Note that Web-based interactive environments, like the Jupyter Notebook, do not provide an effective debugger yet. You need to execute this in a local environment:

```
import pdb

def startswith(srcstr, tarstr):
    '''check if tarstr starts with srcstr
      like srcstr="abra" tarstr="abracadabra" '''
    pdb.set_trace()
    for i in range(len(srcstr)): # check all characters of srcstr
        if srcstr[i] != tarstr[i]:  # if does not match return False
            return False
    return True   # if False is not returned yet, it matches


def findstr(srcstr, tarstr):
    '''Find position of srcstr in tarstr'''
    for i in range(len(tarstr)):
        # if scrstr is same as tarstr from i to rest
        # return i
        if startswith(srcstr, tarstr[i:]):
                return i
    return -1

print(findstr("aba", "abra"))
```

when it is run, it stops execution and outputs a prompt "`(Pdb)`". The following is a sample interaction with the debugger. All words following the `(Pdb)` prompt are user input:

```
> <ipython-input-18-bf80973a0bf6>(7)startswith()
-> for i in range(len(srcstr)): # check all characters of srcstr
(Pdb) cont
> <ipython-input-18-bf80973a0bf6>(7)startswith()
-> for i in range(len(srcstr)): # check all characters of srcstr
(Pdb) help

Documented commands (type help <topic>):
========================================
EOF    c          d        h         list      q        rv       undisplay
a      cl         debug    help      ll        quit     s        unt
alias  clear      disable  ignore    longlist  r        source   until
args   commands   display  interact  n         restart  step     up
b      condition  down     j         next      return   tbreak   w
break  cont       enable   jump      p         retval   u        whatis
bt     continue   exit     l         pp        run      unalias  where

Miscellaneous help topics:
==========================
exec  pdb

(Pdb) cont
> <ipython-input-18-bf80973a0bf6>(7)startswith()
-> for i in range(len(srcstr)): # check all characters of srcstr
(Pdb) cont
> <ipython-input-18-bf80973a0bf6>(7)startswith()
-> for i in range(len(srcstr)): # check all characters of srcstr
(Pdb) cont
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-18-bf80973a0bf6> in <module>()
     20     return -1
     21
---> 22 print(findstr("aba", "abra"))

1 frames
<ipython-input-18-bf80973a0bf6> in findstr(srcstr, tarstr)
     16         # if scrstr is same as tarstr from i to rest
     17         # return i
---> 18         if startswith(srcstr, tarstr[i:]):
     19                 return i
     20     return -1

<ipython-input-18-bf80973a0bf6> in startswith(srcstr, tarstr)
      5       like srcstr="abra" tarstr="abracadabra" '''
      6     pdb.set_trace()
----> 7     for i in range(len(srcstr)): # check all characters of srcstr
      8         if srcstr[i] != tarstr[i]:  # if does not match return False
      9             return False

IndexError: string index out of range
```

This example selects the debugger intervention point with a call to `pdb.set_trace()`. If you want to start the debugging right away with the script start, execute your script at the operating system command line as

```
python3 -m pdb yourscript.py
```

Assume `yourscript.py` contains the script without pdb related calls:

```
OS Command Line Prompt> python3 -m pdb yourscript.py
> /tmp/yourscript.py(1)<module>()
-> def startswith(srcstr, tarstr):
(Pdb) cont
Traceback (most recent call last):
  File "/usr/lib/python3.9/pdb.py", line 1705, in main
    pdb._runscript(mainpyfile)
  File "/usr/lib/python3.9/pdb.py", line 1573, in _runscript
    self.run(statement)
  File "/usr/lib/python3.9/bdb.py", line 580, in run
    exec(cmd, globals, locals)
  File "<string>", line 1, in <module>
  File "/tmp/yourscript.py", line 1, in <module>
    def startswith(srcstr, tarstr):
  File "/tmp/yourscript.py", line 15, in findstr
    if startswith(srcstr, tarstr[i:]):
  File "/tmp/yourscript.py", line 5, in startswith
    if srcstr[i] != tarstr[i]:  # if does not match return False
IndexError: string index out of range
Uncaught exception. Entering post mortem debugging
Running 'cont' or 'step' will restart the program
> /tmp/yourscript.py(5)startswith()
-> if srcstr[i] != tarstr[i]:  # if does not match return False
(Pdb) where
  /usr/lib/python3.9/pdb.py(1705)main()
-> pdb._runscript(mainpyfile)
  /usr/lib/python3.9/pdb.py(1573)_runscript()
-> self.run(statement)
  /usr/lib/python3.9/bdb.py(580)run()
-> exec(cmd, globals, locals)
  <string>(1)<module>()
  /tmp/t.py(1)<module>()
-> def startswith(srcstr, tarstr):
  /tmp/t.py(15)findstr()
-> if startswith(srcstr, tarstr[i:]):
> /tmp/t.py(5)startswith()
-> if srcstr[i] != tarstr[i]:  # if does not match return False
(Pdb) list
  1   def startswith(srcstr, tarstr):
  2       '''check if tarstr starts with srcstr
  3          like srcstr="abra" tarstr="abracadabra" '''
  4       for i in range(len(srcstr)): # check all characters of srcstr
  5  ->          if srcstr[i] != tarstr[i]:  # if does not match return False
  6                return False
  7       return True   # if False is not returned yet, it matches
  8
  9
 10   def findstr(srcstr, tarstr):
 11       '''Find position of srcstr in tarstr'''
(Pdb) print(srcstr,tarstr,i)
aba a 1
(Pdb) quit
```

In this run, execution stops as soon as the script is started. The `cont` command tells the debugger to continue execution. Execution stops again when the run-time error occurs. All debugger commands are available in this state. The user can inspect the current values of variables or variables in the upper call levels.

Debuggers are incredibly powerful tools, particularly when dealing with complex code that involves numerous modules and functions. However, they require time and effort to master and wield effectively. The good news is that if you utilize an integrated development environment (IDE), they become more user-friendly and accessible. The following is a list of some of the free IDE software supporting Python debugging:

PyCharm	https://www.jetbrains.com/pycharm
PyDev	https://www.pydev.org/
Spyder	https://www.spyder-ide.org/
Thonny	https://thonny.org/
VSCode	https://code.visualstudio.com/

9.4 Important Concepts

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

- Different types of errors: Syntax, type, run-time, and logical errors.
- How to deal with errors.
- Exceptions and exception handling.
- Debugging by "printing" values, exception handling, and a debugger.

9.5 Further Reading

- Python's built-in exceptions: https://docs.python.org/3/library/exceptions.html [1].
- Python's document on errors and exceptions: https://docs.python.org/3/tutorial/errors.html [2].

9.6 Exercises

We have included numerous exercises throughout the chapter. Please take the time to study and answer them.

1. Find the errors and exception types in the following code:

```
students = { 131223: "john doe", 2314123: "jane doe", 2334233: "john smith"}
files = ["a.txt", "b.txt", "c.txt", "d.txt"]
s = "hello"
f = 7.1231231
i = 1

d = i / (i-1)
p = s + "/" + files[4]
print(students["john doe"])
print(f.split('.'))
print(s * 3)
s[0] = "H"
print(':'.join(files,s)
j = int(str(f))
```

2. Whenever possible, add a conditional test before each error line in the above program so that the code works without any error. Replace any

```
error generating line
```

with:

```
if sometest :
    error generating line
```

3. Put each error generating line in a `try: ... except:...` block and handle the exception in the above program. For each exception, the program should print an error message and continue with the next statement.
4. The following program reads an input file and calculates an average value. The first line of the input file is a number `N` denoting the count of student records. The following `N` lines are student records in the format:

`id:name:surname:grade`

where `id` is an integer and `grade` is a float. After the student records, the input file has 0 or more lines providing student IDs or student full names. The file terminates with the end-of-file.

Possible sources for error are marked in comments as ####. Those lines may generate an error depending on the input or current state of the program.

```
def tostudent(line):
    stid, name, surname, grade = line.split(':')   ####
    stid = int(stid)                               ####
    grade = float(grade)
    return {'stid': stid, 'fullname': name+' '+surname, 'grade':grade}

fp = open("students.txt")                          ####

stbyname = {}
stbyid = {}
numstudents = int(fp.readline())                   ####
for i in range(numstudents):
    st = tostudent(fp.readline())
    stbyname[st['fullname']] = st                  ####
    stbyid[st['stid']] = st                       ####

# following lines read student names or
# ids until end-of-file

sum = 0
```

(continues on next page)

(continued from previous page)

```
i = 0
while True:
    idorname = fp.readline().strip()
    if idorname == '':  # EOF test
        break
    st = stbyname[idorname] if stbyname[idorname] else stbyid[idorname]   ####
    sum += st['grade']
    i += 1

average = sum/i                              #####

print('Average of',i,'students are',average)
```

Correct the code above so that it will report and ignore the errors whenever possible (i.e., skip over invalid student records). The updated program should execute until the last line. Use `if` conditionals instead of `try:.. except:` whenever possible.

5. Solve the exercise above only with `try:... except:` blocks. Compare your solution with the solution to the previous exercise. Also, compare the handling of exceptions generated in the `tostudent()` function inside the function body or in the main program.
6. You are given the following binary search function, which searches a value in a list. At each step, it finds a mid-point of the search range. If the value is smaller than the mid-point, it narrows down the search in the first half, otherwise in the second half.

```
def bsearch(val, lst):
    if not issorted(lst):
        return -1
    start = 0
    end = len(lst)
    while start != end:
        mid = (start+end) // 2
        if val == lst[mid]:
            return mid
        elif val > lst[mid]:
            start = mid
        else:
            end = mid-1
```

This code makes a sanity check before starting the search. It calls the `issorted()` function to ensure the list is sorted in increasing order. First, implement this function.
Unfortunately, the `bsearch()` function does not work. It finds the correct result for some values but fails in many cases. It does not generate a run-time error but goes into an infinite loop. Use the debugging methods covered in the chapter to find and correct this error.
7. The following function gets two sorted lists as arguments and returns a sorted list by merging them. However, the code generates a run-time error.

```python
def merge(a,b):
    na, nb = len(a),len(b)
    if not (issorted(a) and issorted(b)):
        print("error")
        return []
    rlist = []     # value to be returned
    i,j = 0,0      # indices for a and b

    while i < na or j < nb:
        if a[i] < b[j]:     # a is smaller pick it
            rlist.append(a[i])
            i += 1
        else:               # b is smaller pick it
            rlist.append(b[j])
            j += 1
    return rlist
```

Debug and correct this function. Note that this is a debugging exercise, not a puzzle. Focus on applying the debugging techniques even after fixing the error. Try all techniques described in the chapter.

References

[1] Python Documentation: “Built-in exceptions” (2023). https://docs.python.org/3/library/exceptions.html. Accessed 5-September-2023

[2] Python Documentation: “Tutorial on errors and exceptions” (2023). https://docs.python.org/3/tutorial/errors.html. Accessed 5-September-2023

Scientific and Engineering Libraries

10

In this chapter, we cover several libraries that are very functional in scientific & engineering-related computing problems. In order to keep our focus on the practical usages of these libraries and considering that this is an introductory textbook, the coverage of this chapter is not intended to be comprehensive.

10.1 Numerical Computing with NumPy

NumPy can be considered as a library for working with vectors and matrices. NumPy calls vectors and matrices as *arrays*.

Installation Notes 10.1

To be able to use the NumPy library, you will need to download it from numpy.org[1] and install it on your computer. If you are using a Python package manager (e.g., pip), you can install it directly using: `$ pip install numpy`. If you are using a Windows/Mac machine, you should install anaconda[2] first. If you are using Colab or another Jupyter Notebook viewer, the platform may already have numpy installed.

[1] http://www.numpy.org

[2] https://docs.anaconda.com/anaconda/install/

10.1.1 Arrays and Their Basic Properties

Let us consider a simple vector and a matrix:

$$array1 = \begin{pmatrix} 1 & 2 & 3 \end{pmatrix}, \tag{10.1}$$

and

$$array2 = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}. \tag{10.2}$$

S. Kalkan et al., *Programming with Python for Engineers*,
https://doi.org/10.1007/978-3-031-57148-0_10

Let us see how we can represent and work with these two arrays in NumPy:

```
>>> import numpy as np          # Import the NumPy library
>>> array1 = np.array([1, 2, 3])
>>> array2 = np.array([[1, 2, 3], [4, 5, 6]])
>>> type(array1)
<class 'numpy.ndarray'>
>>> type(array2)
<class 'numpy.ndarray'>
>>> array1
array([1, 2, 3])
>>> array2
array([[1, 2, 3],
       [4, 5, 6]])
>>> print(array2)
[[1 2 3]
 [4 5 6]]
```

We see from this example that we can pass lists of numbers as arguments to the `np.array()` function which creates a NumPy array for us. If the argument is a nested list, each element of the list is used as a row of a 2D array (as in the `array2` case).

Arrays can contain any type of data as elements; however, we limit ourselves to numbers (integers and real numbers) in this chapter.

Shapes, Dimensions and Number of Elements of Arrays.
The first thing we can do with a NumPy array is check its *shape*. For our example `array1` and `array2`, we can do so as follows:

```
>>> array1.shape
(3,)
>>> array2.shape
(2,3)
```

where we can see that `array1` is a one-dimensional array with 3 elements and `array2` is a 2×3 array (a 2D matrix).

For a 2D array, a shape value `(R, C)` denotes the number of rows first (`R`), which is sometimes also called the first *dimension*, and then the number of columns (`C`), which is the second dimension. For nD arrays with $n > 2$, the meaning of the shape values is the same except that there are n values in the shape.

In NumPy, we can easily create a new array with a new shape but the same content (notice that the shape parameter is a tuple):

```
>>> array1.reshape((3,1))
array([[1],
       [2],
       [3]])
>>> array2.reshape((1,6))
array([[1, 2, 3, 4, 5, 6]])
```

Note that this operation does not change `array1` or `array2`.

For many applications, we will need to access the number of dimensions of an array. For this purpose, we can use the `<array>.ndim` value:

```
>>> array1.ndim
1
>>> array2.ndim
2
```

The number of elements in an array is another important value that we are frequently interested in. To access that, we can use the `<array>.size` value:

```
>>> array1.size
3
>>> array2.size
6
```

Accessing Elements in Arrays.
NumPy allows for the same indexing mechanisms as those you can use with Python's native container data types (e.g., tuples, lists). For NumPy, let us look at some examples:

```
>>> array1[-1]
3
>>> array2[1][2]
6
>>> array2[-1]
array([4, 5, 6])
```

Creating Arrays.

We have already seen that we can create arrays using the `np.array()` function. However, there are other ways of creating arrays that conform to certain predefined specifications. We can, for example, create arrays filled with zeros or ones (note that the argument is a tuple describing the shape of the matrix):

```
>>> np.zeros((3, 4))
array([[0., 0., 0., 0.],
       [0., 0., 0., 0.],
       [0., 0., 0., 0.]])
>>> np.ones((2,6))
array([[1., 1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1., 1.]])
```

Alternatively, we can create an array filled with a range of values using the `np.arange()` function—notice that the usage is very similar to the `range()` function we introduced in Sect. 5.2.3 to create an iterator:

```
>>> np.arange(1,10)     # 1: starting value. 10: ending value (excluded)
array([1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> np.arange(1,10,2)    # 1: starting value. 10: ending value. 2: Increment
array([1, 3, 5, 7, 9])
>>> np.arange(1,10).reshape((3,3))
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
```

10.1.2 Working with Arrays

Now let us see the different types of operations we can perform with arrays.

Arithmetic, Relational, and Membership Operations with Arrays
The arithmetic operations (`+`, `-`, `*`, `/`, `**`), the relational operations (`==`, `<`, `<=`, `>`, `>=`), and the membership operations (`in`, `not in`) that we can apply on numbers and other data types in Python can be applied on arrays with NumPy. These operations are performed *element-wise*. Therefore, the arrays that are provided as operands to a binary arithmetic operator need to have the same shape.

Let us look at some examples of arithmetic operations:

```
>>> A = np.arange(4).reshape((2,2))
>>> A
array([[0, 1],
       [2, 3]])
>>> B = np.arange(4, 8).reshape((2,2))
>>> B
array([[4, 5],
       [6, 7]])
>>> print(B-A)
[[4 4]
 [4 4]]
>>> print(B+A)
[[ 4  6]
 [ 8 10]]
>>> A
array([[0, 1],
       [2, 3]])
>>> B
array([[4, 5],
       [6, 7]])
```

Note that these operations create a new array whose elements are the results of applying the operation. Therefore, the original arrays are not modified. If you are interested in in-place operations that modify an existing array during the operation, you can use combined statements such as `+=`, `-=`, `*=`.

Relational and membership operations are also applied element-wise, and we can easily anticipate the outcomes of such operations, e.g., as follows:

```
>>> A < B
array([[ True,  True],
       [ True,  True]])
>>> B > A
array([[ True,  True],
       [ True,  True]])
>>> A > B
array([[False, False],
       [False, False]])
>>> 4 in B
True
>>> 10 in B
False
```

Useful Functions

NumPy arrays provide several useful functions. These include:

- Standard mathematical functions such as exponent, sine, cosine, square-root: `np.exp(<array>)`, `np.sin(<array>)`, `np.cos(<array>)`, `np.sqrt(<array>)`.
- Minimum and maximum: `<array object>.min()` and `<array object>.max()`.
- Summation, mean and standard deviation: `<array object>.sum()`, `<array object>.mean()` and `<array object>.std()`.

Note that minimum, maximum, summation, mean, and standard deviation can be applied on the whole array as well as along a pre-specified dimension (specified with an `axis` parameter). Let us see some examples to clarify this important aspect:

```
>>> A
array([[0, 1],
       [2, 3]])
>>> A.sum()
6
>>> A.sum(axis=0)
array([2, 4])
>>> A.sum(axis=1)
array([1, 5])
>>> A.sum(axis=2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/usr/local/lib/python3.7/site-packages/numpy/core/_methods.py", line 47, in _sum
    return umr_sum(a, axis, dtype, out, keepdims, initial, where)
numpy.AxisError: axis 2 is out of bounds for array of dimension 2
```

where we understand from the error that the axes start being numbered from zero.

Splitting and Combining Arrays

For many problems, we will need to *split* an array into multiple arrays or combine multiple arrays into one. For splitting arrays, we can use functions such as `np.hsplit()` (for horizontal split), `np.vsplit()` (for vertical split), and `np.array_split()` (for more general split operations). Below is an example for `hsplit()` and `vsplit()`:

```
>>> L = np.arange(16).reshape(4,4)
>>> L
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15]])
>>> np.hsplit(L,2)  # Divide L into 2 arrays along the horizontal axis
[array([[ 0,  1],
       [ 4,  5],
       [ 8,  9],
       [12, 13]]), array([[ 2,  3],
       [ 6,  7],
       [10, 11],
       [14, 15]])]
>>> np.vsplit(L,2)
[array([[0, 1, 2, 3],
       [4, 5, 6, 7]]), array([[ 8,  9, 10, 11],
       [12, 13, 14, 15]])]
```

Note that the resultant arrays are provided as a Python `list`. Note also that the functions `hsplit()` and `vsplit()` work on even sizes (i.e., they can split into equally sized arrays). For more specific split operations, you can use `array_split()`.

To combine multiple arrays, we can use `np.hstack()` (for horizontal stacking), `np.vstack()` (for vertical stacking), and `np.stack()` (for more general stacking operations). Below is an example (for the arrays `A` and `B` that we have created before):

```
>>> A
array([[0, 1],
       [2, 3]])
>>> B
array([[4, 5],
       [6, 7]])
>>> np.hstack((A, B))
array([[0, 1, 4, 5],
       [2, 3, 6, 7]])
>>> np.vstack((A, B))
array([[0, 1],
       [2, 3],
       [4, 5],
       [6, 7]])
```

Iterations with Arrays

NumPy already provides us with many functionalities that we might need while working with arrays. However, in many circumstances, these will not be sufficient and we will need to be able to iterate over the elements of arrays and perform custom algorithmic steps.

Luckily, iteration with arrays is very similar to how we would perform iterations with other container data types in Python. Let us see some examples:

```
>>> L
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15]])
>>> for r in L:
...     print("row: ", r)
...
row:  [0 1 2 3]
row:  [4 5 6 7]
row:  [ 8  9 10 11]
row:  [12 13 14 15]
```

Here we see that iteration over a multidimensional array iterates over the first dimension. To iterate over each element, we have at least two options:

```
>>> for element in L.flat:
...     print(element)
...
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
>>> for r in L:
...     for element in r:
...             print(element)
...
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
```

Now let us implement one of the NumPy operations we have seen above in Python from scratch as an exercise:

Hands-on Code 10.1

https://pp4e.online/c10s1

```
import numpy as np

def horizontal_stack(A,B):
    """A function that combines two 2D arrays A and B.
       A and B need to have the same height.
    """
    # Let us get dimensions first and check whether A and B
    # are compatible for stacking.
    (H_A, W_A) = A.shape
    (H_B, W_B) = B.shape
    if H_A != H_B:
        print("Arguments A and B have incompatible heights!")
        return None
    # Let us create an empty `result` array:
    H_result = H_A #or H_B
    W_result = W_A + W_B
    result = np.zeros((H_result, W_result))

    # Now let us iterate over each position in A and B and place
    # their elements into the corresponding positions
    for i in range(H_A):
        for j in range(W_A):
            result[i][j] = A[i][j]
        for j in range(W_B):
            result[i][j+W_A] = B[i][j]

    return result

# Let us test our code:
M = np.random.randn(2,2) # Create a random 3x4 array
N = np.random.randn(2,3) # Create a random 3x6 array
print("Random array M is:\n", M)
print("Random array N is:\n", N)
print("Horizontal stacking of M and N yields:\n", horizontal_stack(M, N))
```

Output

```
Random array M is:
 [[ 0.39521366  1.41939618]
 [-0.13770429  1.19319949]]
Random array N is:
 [[-1.61066212e-01  8.15354855e-01  7.19234023e-01]
 [ 2.65985989e-04 -2.89111500e-01  4.52781848e-02]]
Horizontal stacking of M and N yields:
 [[ 3.95213656e-01  1.41939618e+00 -1.61066212e-01  8.15354855e-01
   7.19234023e-01]
 [-1.37704293e-01  1.19319949e+00  2.65985989e-04 -2.89111500e-01
   4.52781848e-02]]
```

10.1.3 Linear Algebra with NumPy

Now let us get a flavor of the linear algebra operations provided by NumPy. Note that some of these operations are provided in the module `numpy.linalg`.

Transpose

The *transpose* operation flips a matrix along the diagonal. For an example matrix A,

$$A = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}, \tag{10.3}$$

its transpose is the following:

$$A^T = \begin{pmatrix} 1 & 3 \\ 2 & 4 \end{pmatrix}. \tag{10.4}$$

In NumPy, the transpose of a matrix can be easily accessed by accessing its `.T` member variable or by calling its `.transpose()` member function:

```
>>> A
array([[1, 2],
       [3, 4]])
>>> A.T
array([[1, 3],
       [2, 4]])
>>> A
array([[1, 2],
       [3, 4]])
```

Note that `.T` is simply a member variable of the array object and is not defined as an operation. Try finding the transpose of `A` with `A.transpose()` and see that you obtain the expected result. Note also that the transpose operation does not change the original array.

Inverse

The *inverse* of an $n \times n$ matrix A is an $n \times n$ matrix A^{-1} that yields an $n \times n$ identity matrix I when multiplied:

$$A \times A^{-1} = I. \tag{10.5}$$

In NumPy, we can use `np.linalg.inv(<array>)` to find the inverse of a square array. Here is an example:

```
>>> A
array([[1, 2],
       [3, 4]])
>>> A_inv = np.linalg.inv(A)
>>> A_inv
array([[-2. ,  1. ],
       [ 1.5, -0.5]])
```

Let us check whether the inverse was correctly calculated:

```
>>> np.matmul(A, A_inv)
array([[1.0000000e+00, 0.0000000e+00],
       [8.8817842e-16, 1.0000000e+00]])
```

which is the identity matrix (I) except for some rounding error at index $(1, 0)$ due to the floating point approximation. However, as we have seen in Chap. 2, while writing our programs, we should compare numbers such that `8.8817842e-16` can be considered equal to zero.

Determinant, norm, rank, condition number, trace

While working with matrices, we often require the following properties, which can be easily calculated. For the sake of brevity and to keep the focus, we will omit their explanations:

Matrix Property	How to Calculate with NumPy
Determinant ($\|A\|$ or $det(A)$)	`np.linalg.det(A)`
Norm ($\|\|A\|\|$)	`np.linalg.norm(A)`
Rank ($rank(A)$)	`np.linalg.matrix_rank(A)`
Condition number ($\kappa(A)$)	`np.linalg.cond(A)`
Trace ($tr(A)$)	`np.trace(A)`

Dot Product, Inner Product, Outer Product, and Matrix Multiplication

The product of two vectors and matrices can be defined and calculated in different ways:

- `np.dot(a, b)`: For vectors (1D arrays), this is equivalent to the *dot product* (i.e., $\sum_i \mathbf{a}_i \times \mathbf{b}_i$ for two vectors **a** and **b**). For 2D arrays and arrays with more dimensions, the result is matrix multiplication.
- `np.inner(a,b)`: For vectors, this is the dot product (like `np.dot(a,b)`). For higher-dimensional arrays, the *inner product* is a sum product over the last axes. Consider the following example for clarification on this:

```
>>> A
array([[0, 1],
       [2, 3]])
>>> B
array([[4, 5],
       [6, 7]])
>>> np.inner(A,B)
array([[ 5,  7],
       [23, 33]])
```

  The last axes for both A and B are the horizontal axes. Therefore, each first-axis element of A (i.e., [0, 1] and [2,3]) is multiplied with each first-axis element of B ([4, 5] and [6, 7]).
- `np.outer(a, b)`: The *outer product* is defined on vectors such that `result[i, j] = a[i] * b[j]`.
- `np.matmul(a, b)`: Multiplication of matrices `a` and `b`. The result is simply calculated as $\text{result}_{ij} = \sum_k \mathbf{a}_{ik}\mathbf{b}_{kj}$, as also illustrated in Fig. 10.1.

Eigenvectors and Eigenvalues

Explaining *eigenvectors* and *eigenvalues* is beyond the scope of the book. However, we can briefly remind that eigenvectors and eigenvalues are the variables that satisfy the following for a matrix **A** which can be considered as a transformation in a vector space:

$$\mathbf{A}\mathbf{v} = \lambda\mathbf{v}. \tag{10.6}$$

In other words, **v** is such a vector that **A** just changes its scale by a factor λ.

In NumPy, eigenvectors and eigenvalues can be obtained by `np.linalg.eigh(<array>)` which returns a tuple with the following two elements:

1. Eigenvalues (an array of λ) in decreasing order.
2. Eigenvectors (**v**) as a column matrix. In other words, `v[:, i]` is the eigenvector corresponding to the eigenvalue $\lambda[i]$.

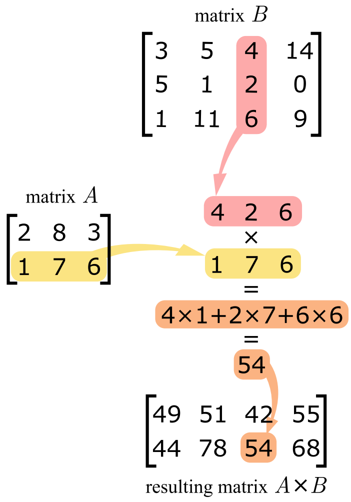


Fig. 10.1 Illustration of matrix multiplication of two matrices A and B.

Matrix Decomposition

NumPy also provides many frequently used matrix decomposition methods, as listed below:

Matrix Decomposition	How to Calculate with NumPy
Cholesky decomposition	`linalg.cholesky(A)`
QR factorization	`np.linalg.qr(A)`
Singular Value Decomposition	`linalg.svd(a)`

Solve a Linear System of Equations

Given a linear system of equations as follows:

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &= b_1, \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n &= b_2, \\ &\ldots \\ a_{n1}x_1 + a_{22}x_2 + \cdots + a_{nn}x_n &= b_n, \end{aligned} \tag{10.7}$$

which can be represented in compact form as

$$\mathbf{ax} = \mathbf{b}, \tag{10.8}$$

Such a system can be solved in NumPy using `np.linalg.solve(a, b)`.

Here is an easy example:

```
>>> A
array([[0, 1],
       [2, 3]])
>>> B
array([3, 4])
>>> np.linalg.solve(A,B)
array([-2.5,  3. ])
>>> X = np.linalg.solve(A,B)
>>> X
array([-2.5,  3. ])
```

which can be easily verified using multiplication:

```
>>> np.inner(A, X)
array([3., 4.])
```

which is equal to B.

10.1.4 Why Use NumPy? Efficiency Benefits

The previous sections have illustrated how capable the NumPy library is and how easily we can do many calculations with the pre-defined functionalities, e.g., in its linear algebra module. Since we can access each element of an array using Python's indexing mechanisms, we can be tempted to implement those functionalities ourselves, from scratch.

However, these functionalities in NumPy are *not* implemented in Python but in C, another high-level programming language where programs are directly compiled into machine-executable binary code. Therefore, they would execute much faster in comparison to what we would be implementing in Python.

The following example illustrates this. When executed, you should observe a significant difference in running time, typically on the order of approximately ~1000 times faster.

In other words, it is strongly recommended, whenever possible, to use existing NumPy functions and routines, and to write every operation in vector or matrix form as much as possible if you are working with matrices.

Hands-on Code 10.2

https://pp4e.online/c10s2

```python
import numpy as np
from time import time

def matmul_2D(M, N):
    """Custom defined matrix multiplication for two 2D matrices M and N"""
    (H_M, W_M) = M.shape
    (H_N, W_N) = N.shape
    if W_M != H_N:
        print("Dimensions of M and N mismatch!")
        return None

    result = np.zeros((H_M, W_N))
    for i in range(H_M):
        for j in range(W_N):
            for k in range(W_M):
                result[i][j] += M[i][k] * N[k][j]

    return result

# First let us check that our code works as expected
M = np.random.randn(2,3)
N = np.random.randn(3,4)

print("Our matmul_2D result: \n", matmul_2D(M, N))
print("Correct result: \n", np.matmul(M, N))

# Now let us measure the running-time performances
# Create two 2D large matrices
M = np.random.randn(100, 100)
N = np.random.randn(100, 100)

# Option 1: Use NumPy's matrix multiplication
t1 = time()
result = np.matmul(M, N)
t2 = time()
print("NumPy's matmul took ", t2-t1, "ms.")

# Option 2: Use our matmul_2D function
t1 = time()
result = matmul_2D(M, N)
t2 = time()
print("Our matmul_2D function took ", t2-t1, "ms.")
```

Output

```
Our matmul_2D result:
 [[-0.24208963  1.18790239 -1.5397112  -1.14521578]
 [-2.00697411  4.23675778 -3.74909068 -0.23854542]]
Correct result:
 [[-0.24208963  1.18790239 -1.5397112  -1.14521578]
 [-2.00697411  4.23675778 -3.74909068 -0.23854542]]
NumPy's matmul took  0.00040411949157714844 ms.
Our matmul_2D function took  1.0593342781066895 ms.
```

10.2 Scientific Computing with SciPy

SciPy is a library that includes many methods and facilities for Scientific Computing. It is closely linked to NumPy so much that NumPy needs to be imported first to be able to use SciPy.

Installation Notes 10.2

To be able to use the SciPy library, you will need to download it from scipy.org[1] and install it on your computer. If you are using a Python package manager (e.g., pip), you can install it directly using: `$ pip install scipy`. If you are using a Windows/Mac machine, you should install anaconda[2] first. If you are using Colab or another Jupyter Notebook viewer, the platform may already have scipy installed.

[1] http://www.scipy.org

[2] https://docs.anaconda.com/anaconda/install/

The following is a list of some modules provided by SciPy. Even a brief coverage of these modules is not feasible in a chapter. However, we will see an example in Chap. 12 using SciPy's `stats` and `optimize` modules.

Module	Description
cluster	Clustering algorithms
constants	Physical and mathematical constants
fftpack	Fast Fourier Transform routines
integrate	Integration and ordinary differential equation solvers
interpolate	Interpolation and smoothing splines
io	Input and Output
linalg	Linear algebra
ndimage	N-dimensional image processing
odr	Orthogonal distance regression
optimize	Optimization and root-finding routines
signal	Signal processing
sparse	Sparse matrices and associated routines
spatial	Spatial data structures and algorithms
special	Special functions
stats	Statistical distributions and functions

10.3 Data Handling and Analysis with Pandas

Pandas is a very handy library for working with files and analyzing data in different formats. To keep things simple, in this section, we will just look at CSV (Comma Separated Values) files (briefly covered in Sect. 8.6). Note that the facilities for other formats are very similar.

Installation Notes 10.3

To use the Pandas library, you will need to download it from pandas.pydata.org[1] and install it on your computer. If you are using a Python package manager (e.g., pip), you can install it directly using: `$ pip install pandas`. If you are using a Windows/Mac machine, you should install anaconda[2] first. If you are using Colab or another Jupyter Notebook viewer, the platform may already have Pandas installed.

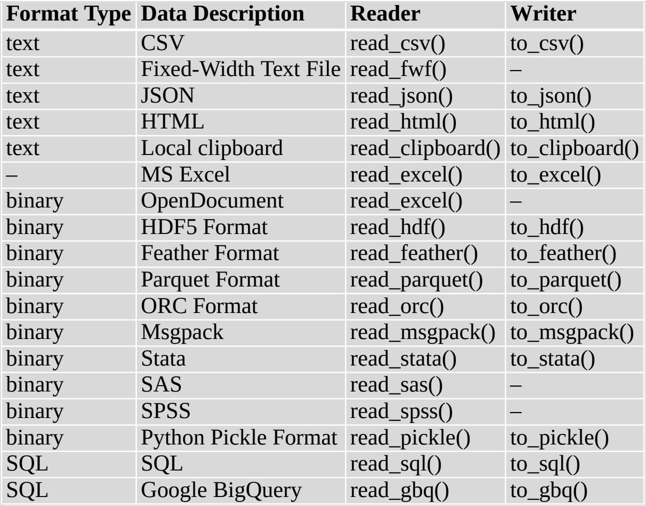
[1] https://pandas.pydata.org/

[2] https://docs.anaconda.com/anaconda/install/

10.3.1 Supported File Formats

As we have seen before, Python already provides facilities for reading and writing files. However, if you need to work with "structured" files such as CSV, XML, XLSX, JSON, HTML, the native Python features would need to be significantly extended.

That is where Pandas comes in. Pandas complements Python's native facilities to be able to work with structured files for both reading data from them and creating new files. Below is a list of the different file formats that Pandas supports and the corresponding functions for reading or writing them. The table is adapted from, and the reader is encouraged to check the Pandas IO Reference[1] to see a more up-to-date list:



Format Type	Data Description	Reader	Writer
text	CSV	read_csv()	to_csv()
text	Fixed-Width Text File	read_fwf()	–
text	JSON	read_json()	to_json()
text	HTML	read_html()	to_html()
text	Local clipboard	read_clipboard()	to_clipboard()
–	MS Excel	read_excel()	to_excel()
binary	OpenDocument	read_excel()	–
binary	HDF5 Format	read_hdf()	to_hdf()
binary	Feather Format	read_feather()	to_feather()
binary	Parquet Format	read_parquet()	to_parquet()
binary	ORC Format	read_orc()	to_orc()
binary	Msgpack	read_msgpack()	to_msgpack()
binary	Stata	read_stata()	to_stata()
binary	SAS	read_sas()	–
binary	SPSS	read_spss()	–
binary	Python Pickle Format	read_pickle()	to_pickle()
SQL	SQL	read_sql()	to_sql()
SQL	Google BigQuery	read_gbq()	to_gbq()

10.3.2 Data Frames

Pandas library is built on top of a data type called `DataFrame` which is used to store all types of data while working with Pandas. In the following, we will illustrate how you can get your data into a `DataFrame` object:

1. Loading data from a file.
If you have your data in a file that is supported by Pandas, you can use its reader to directly obtain a `DataFrame` object. Let us look at an example CSV file:

[1] https://pandas.pydata.org/pandas-docs/stable/user_guide/io.html.

Hands-on Code 10.3

https://pp4e.online/c10s3

```
# Import the necessary libraries
import pandas as pd

# Read the file named 'ch10_example.csv’
df = pd.read_csv(’intput/ch10_example.csv’)

# Print the CSV file’s contents:
print("The CSV file contains the following:\n", df, "\n")

# Check the types of each column
df.dtypes
```

Output

```
The CSV file contains the following:
       Name  Grade  Age
0     Jack   40.2   20
1   Amanda   30.0   25
2     Mary   60.2   19
3     John   85.0   30
4    Susan   70.0   28
5     Bill   58.0   28
6     Jill   90.0   27
7      Tom   90.0   24
8    Jerry   72.0   26
9   George   79.0   22
10  Elaine   82.0   23

Name      object
Grade    float64
Age        int64
dtype: object
```

Note the following critical details:

- The `read_csv()` function used the header (“Name”, “Grade”, and “Age”) from our CSV file. If your file does not have a header, you can call `read_csv()` with the `header` parameter set to `None` as follows: `pd.read_csv(filename, header=None)`.
- The `read_csv()` function reads all columns in the CSV file. If you wish to load only some of the columns (e.g., “Name”, “Age” in our example), you can relay this using the `usecols` parameter as follows: `pd.read_csv(filename, usecols=[’Name’,’Age’])`.

2. Convert Python data into a DataFrame.
Alternatively, you can provide an existing Python data object as an argument to a `DataFrame()` constructor, as illustrated in the following example:

Hands-on Code 10.4

https://pp4e.online/c10s4

```
lst = [('Jack', 40.2, 20), ('Amanda', 30, 25), ('Mary', 60.2, 19)]
df = pd.DataFrame(data = lst, columns=['Name', 'Grade', 'Age'])
print(df)
```

Output

```
     Name  Grade  Age
0    Jack   40.2   20
1  Amanda   30.0   25
2    Mary   60.2   19
```

In many cases, we will require the rows to be associated with names, or sometimes called keys. For example, instead of referring to a row as “the row at index 1”, we might require accessing a row with a non-integer value. This can be achieved as follows (note how the printed DataFrame looks different):

Hands-on Code 10.5

https://pp4e.online/c10s5

```python
names = ['Jack', 'Amanda', 'Mary']
lst = [(40.2, 20), (30, 25), (60.2, 19)]
df = pd.DataFrame(data = lst, index=names, columns=['Grade', 'Age'])
print(df)
```

Output

```
        Grade  Age
Jack     40.2   20
Amanda   30.0   25
Mary     60.2   19
```

Alternatively, we can obtain a DataFrame from a dictionary, which might be easier for us to create data column-wise—note that the column names are obtained from the keys of the dictionary:

Hands-on Code 10.6

https://pp4e.online/c10s6

```python
d = {'Grade': [40.2, 30, 60.2],
      'Age': [20, 25, 19]}
names = ['Jack', 'Amanda', 'Mary']
df = pd.DataFrame(data = d, index=names)
print(df)
```

Output

```
        Grade  Age
Jack     40.2   20
Amanda   30.0   25
Mary     60.2   19
```

10.3.3 Accessing Data with DataFrames

We can access data *column-wise* or *row-wise*.

1. Column-wise access.
For column-wise access, you can use `df["Grade"]`, which returns a sequence of values. To access a certain element in that sequence, you can either use an integer index (as in `df["Grade"][1]`) or a named index if it has been defined (as in `df["Grade"]["Amanda"]`). The following example illustrates this for the example `DataFrame` we have created above:

```
>>> print(df)
         Grade  Age
Jack      40.2   20
Amanda    30.0   25
Mary      60.2   19
>>> print(df['Grade'][1])
30.0
>>> print(df['Grade']['Amanda'])
30.0
```

2. Row-wise access.
To access a particular row, you can either use integer indexes with `df.iloc[<row index>]` or `df.loc[<row name>]` if a *named index* is defined. This is illustrated below:

```
>>> print(df)
        Grade  Age
Jack     40.2   20
Amanda   30.0   25
Mary     60.2   19
>>> print(df.iloc[1])
Grade    30.0
Age      25.0
Name: Amanda, dtype: float64
>>> print(df.loc['Amanda'])
Grade    30.0
Age      25.0
Name: Amanda, dtype: float64
```

You can also provide both column index and name index in a single operation, i.e., `[<row index>, <column index>]` or `[<row name>, <column name>]` as illustrated below:

```
>>> df.loc['Amanda','Grade']
30.0
>>> df.iloc[1, 1]
25
```

While accessing a `DataFrame` with integer indexes, you can also use Python's slicing; i.e., `[start:end:step]` indexing.

10.3.4 Modifying Data with DataFrames

Although modifying data in a `DataFrame` is simple, you need to be careful about one crucial concept: Let us say that you use column and row names to change the contents of a cell and you first access the column and then the corresponding row as follows:

```
>>> print(df)
            Grade  Age
Jack     40.2   20
Amanda   30.0   25
Mary     60.2   19
>>> df['Grade']['Amanda'] = 45
/usr/local/lib/python3.6/dist-packages/ipykernel_launcher.py:6: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame
See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html
↪#returning-a-view-versus-a-copy
```

This is called *chained indexing* and Pandas will warn you that you are modifying a `DataFrame` that was *returned* while accessing columns or rows in your original `DataFrame`: `df['Grade']` returns a direct access for the original `Grade` column or a copy of it. Depending on how your data is structured and stored in the memory, the end result may differ, and your change might not get reflected on the original `DataFrame`.

To prevent this from happening, you should use a single-access operation as illustrated below (you may also do this with `loc` and `iloc`):

```
>>> print(df)
        Grade  Age
Jack     40.2   20
Amanda   30.0   25
Mary     60.2   19
>>> df.loc['Amanda','Grade'] = 45
>>> df.iloc[1,1] = 30
>>> print(df)
        Grade  Age
Jack     40.2   20
Amanda   45.0   30
Mary     60.2   19
```

With these facilities, you can now access every item in a `DataFrame` and modify them.

10.3.5 Analyzing Data with DataFrames

Once we have our data in a `DataFrame`, we can use Pandas's built-in facilities for analyzing our data. One very simple way to analyze data is via *descriptive* statistics, which you can access with `df.describe()` and `df[<column>].value_counts()`:

Hands-on Code 10.7

https://pp4e.online/c10s7

```
print(df)
df.describe()
```

Output

```
        Grade  Age
Jack     40.2   20
Amanda   30.0   25
Mary     60.2   19
           Grade        Age
count   3.000000   3.000000
mean   43.466667  21.333333
std    15.362725   3.214550
min    30.000000  19.000000
25%    35.100000  19.500000
50%    40.200000  20.000000
75%    50.200000  22.500000
max    60.200000  25.000000
```

Apart from these descriptive functions, Pandas provides functions for sorting (using `.sort_values()` function), finding the maximum or the minimum (using `.max()` or `.min()` functions) or finding the largest or smallest n values (using `.nsmallest()` or `.nlargest()` functions:

Hands-on Code 10.8

https://pp4e.online/c10s8

```
print("Maximum grade is: ", df['Grade'].max())
print("\nRecords sorted according to age:\n", df.sort_values(by="Age"))
print("\n\nTop two grades are:\n", df['Grade'].nlargest(2))
```

Output

```
Maximum grade is:  60.2

Records sorted according to age:
         Grade  Age
Mary     60.2   19
Jack     40.2   20
Amanda   30.0   25


Top two grades are:
 Mary    60.2
Jack    40.2
Name: Grade, dtype: float64
```

Note that the displayed output includes the names because we selected names as the indices for accessing the rows.

10.3.6 Presenting Data in DataFrames

Pandas provides a very easy mechanism to plot your data through the `.plot()` function. Some examples are provided below to illustrate how you can plot your own data. This plotting facility is provided by Matplotlib, which we will see next. If you are not using Jupyter Notebook, you need to use the following method to show the plot:
`df.plot().figure.show()`

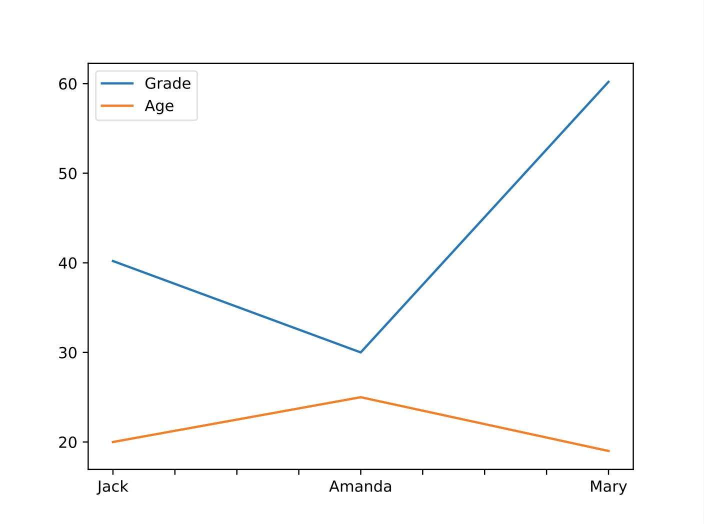
Hands-on Code 10.9

https://pp4e.online/c10s9

```
#Plots all columns in different colors
df.plot()
```

Output

```
<AxesSubplot:>
```



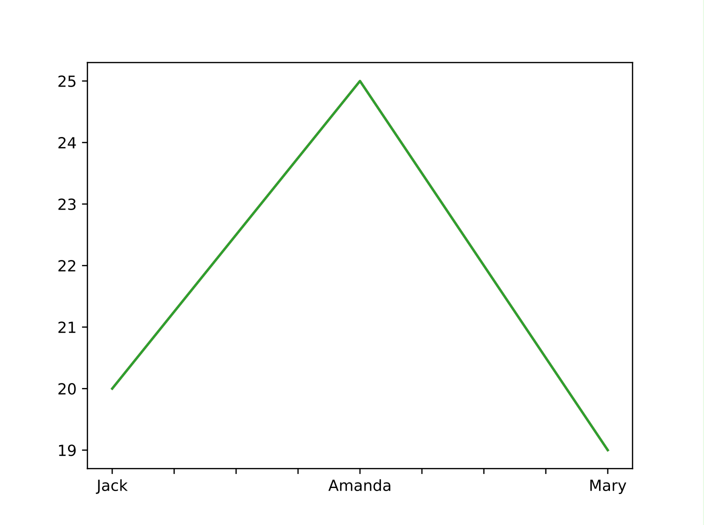
Hands-on Code 10.10

https://pp4e.online/c10s10

```
# Plots a single column
df['Age'].plot()
```

Output

```
<AxesSubplot:>
```



10.4 Plotting Data with Matplotlib

Matplotlib is a very capable library for drawing different types of plots in Python. It is very well integrated with NumPy, SciPy, and Pandas, and therefore, all these libraries are very frequently used together seamlessly.

Installation Notes 10.4

To be able to use the matplotlib library, you will need to download it from matplotlib.org[1] and install it on your computer. If you are using a Python package manager (e.g., pip), you can install it directly using: `$ pip install matplotlib`. If you are using a Windows/Mac machine, you should install anaconda[2] first. If you are using Colab or another Jupyter Notebook viewer, the platform may already have matplotlib installed.

[1] https://matplotlib.org/

[2] https://docs.anaconda.com/anaconda/install/

10.4.1 Parts of a Figure

A figure consists of the following elements (see also Fig. 10.2):

- The *title* of the figure.
- The *axes*, together with their *ticks*, *tick labels*, and *axis labels*.
- The *canvas* of the *plot*, which contains a drawing of your data in the form of dots (*scatter plot*), lines (*line plot*), bars (*bar plot*), surfaces, etc.
- The *legend*, which informs the observer about the different plots in the canvas.

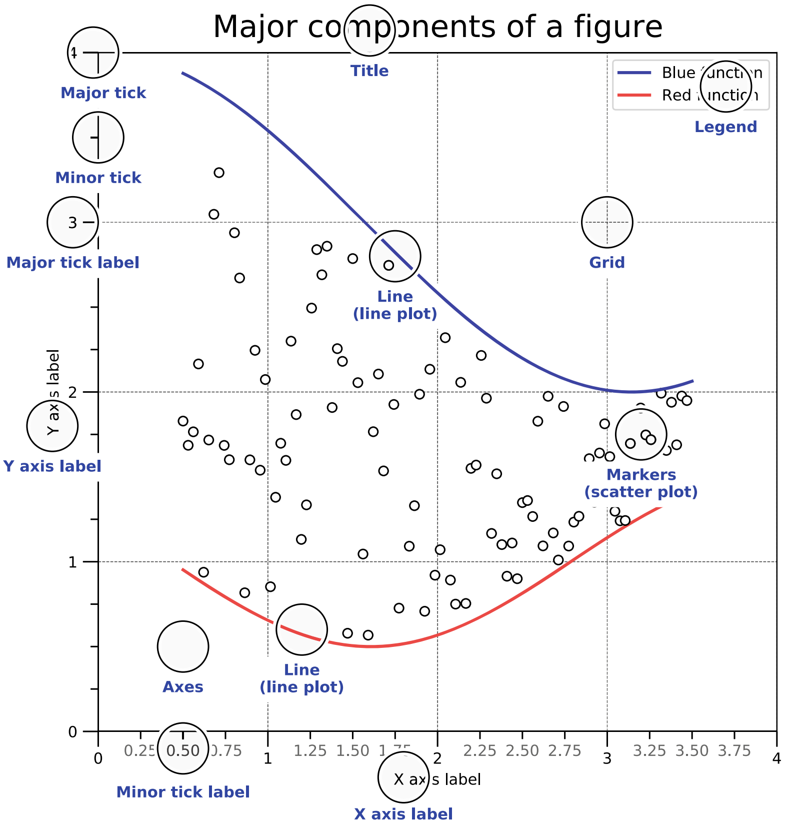


Fig. 10.2 A figure consists of several components all of which you can change in Matplotlib easily. [Produced by adapting the code at https://matplotlib.org/2.0.2/examples/showcase/anatomy.html]

10.4.2 Preparing Your Data for Plotting

Matplotlib expects NumPy arrays as input and therefore, if you have your data in a NumPy array, you can directly plot it without any data type conversion. With pandas `DataFrame`, the behavior is not guaranteed and therefore, it is recommended that the values in a `DataFrame` are first converted to a NumPy array, using, e.g.:

Hands-on Code 10.11

https://pp4e.online/c10s11

```
print(df)
age_array = df['Age'].values
print("The `Age` values in an array form are:", age_array)
print("The type of our new data is: ", type(age_array))
```

Output

```
        Grade  Age
Jack     40.2   20
Amanda   30.0   25
Mary     60.2   19
The Age values in an array form are: [20 25 19]
The type of our new data is:  <class 'numpy.ndarray'>
```

10.4.3 Drawing Single Plots

There are two ways to plot with Matplotlib: In an *object-oriented style* or the so-called *PyPlot style*:

1. Drawing in an Object-Oriented Style

In this style, we create a figure object and an axes object and work with those to create our plots. How this is done is illustrated in the following example:

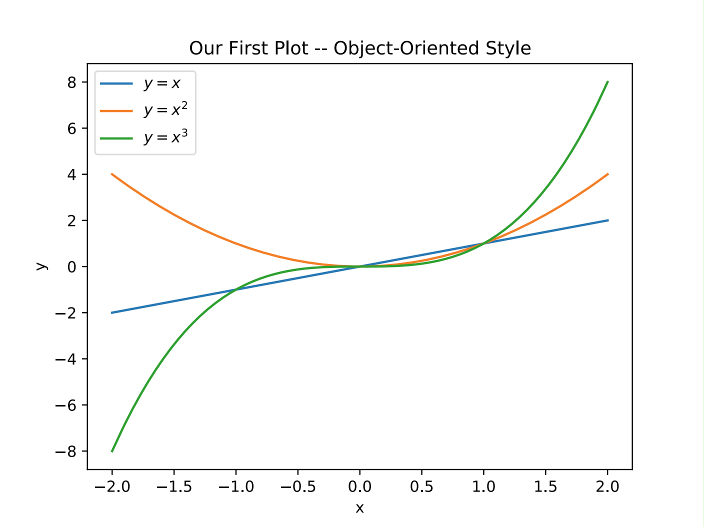
Hands-on Code 10.12

https://pp4e.online/c10s12

```
import matplotlib.pyplot as plt
import numpy as np

# Uniformly sample 50 x values between -2 and 2:
x = np.linspace(-2, 2, 50)

# Create an empty figure
fig, ax = plt.subplots()


# Plot y = x
ax.plot(x, x, label='$y=x$')

# Plot y = x^2
ax.plot(x, x**2, label='$y=x^2$')

# Plot y = x^3
ax.plot(x, x**3, label='$y=x^3$')

# Set the labels for x and y axes:
ax.set_xlabel('x')
ax.set_ylabel('y')

# Set the title of the figure
ax.set_title("Our First Plot -- Object-Oriented Style")

# Create a legend
ax.legend()

# Show the plot
# fig.show() # Uncomment if not using Jupyter
```

Output

```
<matplotlib.legend.Legend at 0x125532250>
```



2. Drawing in a Pyplot Style

In the Pyplot style, we directly call functions in the Pyplot module (`matplotlib.pyplot`) to create a figure and draw our plots. This style does not work with an explicit figure or axes objects. This is illustrated in the following example (assuming Matplotlib and NumPy libraries are already imported as in the previous example):

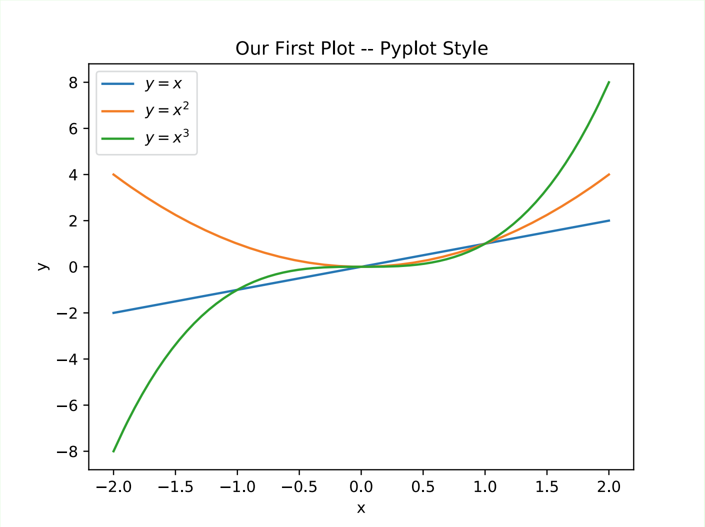
Hands-on Code 10.13

https://pp4e.online/c10s13

```
# Uniformly sample 50 x values between -2 and 2:
x = np.linspace(-2, 2, 50)

# Plot y = x
plt.plot(x, x, label='$y=x$')

# Plot y = x^2
plt.plot(x, x**2, label='$y=x^2$')

# Plot y = x^3
plt.plot(x, x**3, label='$y=x^3$')

# Set the labels for x and y axes:
plt.xlabel('x')
plt.ylabel('y')

# Set the title of the figure
plt.title("Our First Plot -- Pyplot Style”)

# Create a legend
plt.legend()

# Show the plot
#plt.show() # Uncomment if not using Jupyter notebook
```

Output

```
<matplotlib.legend.Legend at 0x1256ab9a0>
```



10.4.4 Drawing Multiple Plots in a Figure

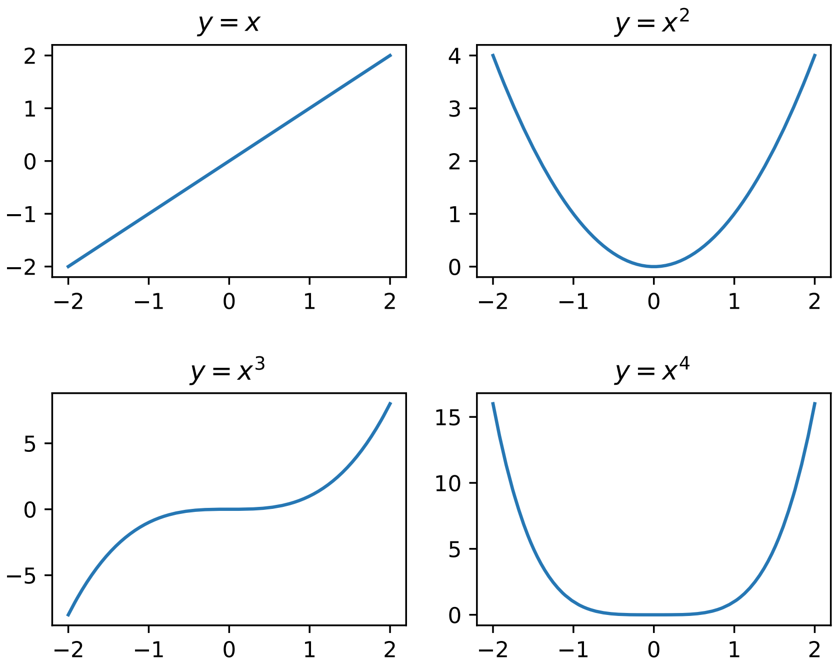
In many situations, you will need to draw multiple plots side by side in a single figure. This can be performed in the object-oriented style using the `subplots()` function to create a grid and then use the created subplots and axes to draw the plots. This is illustrated with an example below (assuming the libraries are already imported and x has been defined as before):

Hands-on Code 10.14

https://pp4e.online/c10s14

```
# Create a 2x2 grid of plots
fig, axes = plt.subplots(2, 2)

# Plot (1,1)
axes[0,0].plot(x, x)
axes[0,0].set_title("$y=x$")

# Plot (1,2)
axes[0,1].plot(x, x**2)
axes[0,1].set_title("$y=x^2$")

# Plot (2,1)
axes[1,0].plot(x, x**3)
axes[1,0].set_title("$y=x^3$")

# Plot (2,2)
axes[1,1].plot(x, x**4)
axes[1,1].set_title("$y=x^4$")

# Adjust vertical space between rows
plt.subplots_adjust(hspace=0.5)

# Show the plot
#fig.show() # Uncomment if not using Jupyter
```

Output



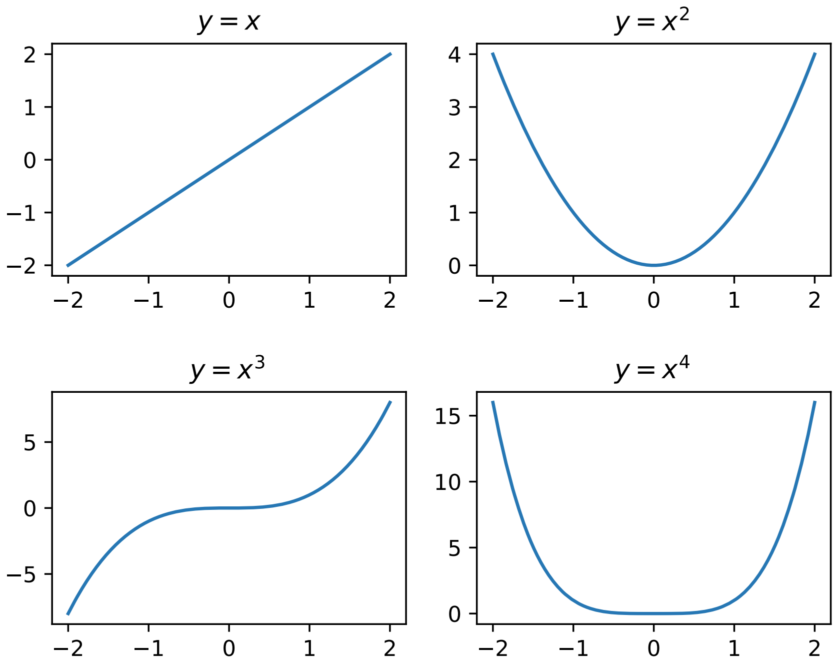
It is possible to do this in the PyPlot style as well, as illustrated below:

Hands-on Code 10.15

https://pp4e.online/c10s15

```
# Plot (1,1)
plt.subplot(2, 2, 1)
plt.plot(x, x)
plt.title('$y=x$')

# Plot (1,2)
plt.subplot(2, 2, 2)
plt.plot(x, x**2)
plt.title('$y=x^2$')

# Plot (2,1)
plt.subplot(2, 2, 3)
plt.plot(x, x**3)
plt.title('$y=x^3$')

# Plot (2,2)
plt.subplot(2, 2, 4)
plt.plot(x, x**4)
plt.title('$y=x^4$')

# Adjust vertical space between rows
plt.subplots_adjust(hspace=0.5)

# Show the plot
#plt.show() # Uncomment if not using Jupyter Notebook
```

Output



10.4.5 Changing the Elements of a Plot

All elements that are visualized in Fig. 10.2 can be changed in Matplotlib. We will skip these details to keep our focus in the book on the practical uses of these libraries. The interested reader can look up the extensive documentation at https://matplotlib.org/2.1.1/contents.html or look at the help page of a function (e.g., `help(plt.plot)`) to see how to modify all elements of a figure.

10.5 Important Concepts

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

- NumPy arrays and their properties: array shape, dimensions, sizes, elements.
- Accessing and modifying elements of a NumPy array.
- Simple algebraic functions on NumPy arrays.
- SciPy and its basic capabilities.
- Pandas, DataFrame, loading files with Pandas.
- Accessing and modifying content in DataFrames.
- Analyzing and presenting data in DataFrames.
- Matplotlib and different ways to make plots.
- Drawing single and multiple plots. Changing elements of a plot.

10.6 Further Reading

- NumPy User Guide: https://numpy.org/doc/stable/user/index.html [2].
- SciPy User Guide: https://docs.scipy.org/doc/scipy/tutorial/index.html [5].
- Pandas User Guide: https://pandas.pydata.org/docs/user_guide/index.html [4].
- Matplotlib User Guide: https://matplotlib.org/2.1.1/users/index.html [1].
- Introduction to Algebra with Python by Pablo Caceres: https://pabloinsente.github.io/intro-linear-algebra [3].

10.7 Exercises

1. Define functions that work like the `sum()`, `mean()`, `min()`, and `max()` operations provided by NumPy. These functions should take a single 2D array and return the result as a number. You can assume that the operation applies to the whole array and not to a single axis.
2. Create a simple CSV file using your favorite spreadsheet editor (e.g., Microsoft Excel or Google Spreadsheets) and create a file with your exams and their grades as two separate columns. Save the file, upload it to the Jupyter Notebook viewer and do the following:

    - Load the file using Pandas.
    - Calculate the mean of your exam grades.
    - Calculate the standard deviation of your grades.

3. Using Matplotlib, generate the following plots with suitable names for the axes and the titles.

- Draw the following four functions in separate single plots: $\sin(x)$, $\cos(x)$, $\tan(x)$, $\cot(x)$.
- Draw these four functions in a single plot.
- Draw a multiple 2x2 plot where each subplot is one of the four functions.

References

[1] Matplotlib Documentation: Matplotlib user guide (2023). https://matplotlib.org/2.1.1/users/index.html. Accessed 5-September-2023

[2] NumPy Documentation: Numpy user guide (2023). https://numpy.org/doc/stable/user/index.html. Accessed 5-September-2023

[3] Pablo Caceres: Introduction to linear algebra for applied machine learning with python (2023). https://pabloinsente.github.io/intro-linear-algebra. Accessed 5-September-2023

[4] Pandas Documentation: Pandas user guide (20223). https://pandas.pydata.org/docs/user_guide/index.html. Accessed 5-September-2023

[5] SciPy Documentation: Scipy user guide (2023). https://docs.scipy.org/doc/scipy/tutorial/index.html. Accessed 5-September-2023

11 An Application: Approximation and Optimization

In this chapter, as an application for programming with Python, we cover some mathematical methods widely used in many disciplines. We start with the *Taylor series* for approximating a function. We then look at *Newton's method* for finding the roots of a function $f(x)$, which can also be considered as an application of the Taylor series. Finally, we extend Newton's method to find the minimum of a function.

11.1 Approximating Functions with Taylor Series

In some problems, we have limited access to a function: We might be provided with only the value of the function or its derivatives at certain input values, and we might be required to make estimations (approximations) about the values of the function at other input values. The Taylor series is one method that provides us with a very elegant solution for approximating a function.

The Taylor series of a function $f(\cdot)$ is essentially an infinite sum of terms that converges to the value of the function at x, i.e., $f(x)$. In more formal terms:

$$f(x) = f(a) + \frac{f'(a)}{1!}(x-a) + \frac{f''(a)}{2!}(x-a)^2 + \frac{f'''(a)}{3!}(x-a)^3 \ldots \tag{11.1}$$

which can be re-written in a more compact form:

$$f(x) = \sum_{n=0}^{\infty} \frac{f^{(n)}(a)}{n!}(x-a)^n. \tag{11.2}$$

In the limit ($n \to \infty$), $f(x)$ will be equal to its Taylor series. However, in realistic applications, it is impractical to take $n \to \infty$. In practice, we take the first m terms of the series and approximate the function with them:

$$f(x) \approx f(a) + \frac{f'(a)}{1!}(x-a) + \frac{f''(a)}{2!}(x-a)^2 + \frac{f'''(a)}{3!}(x-a)^3 \ldots + \frac{f^{(m)}(a)}{m!}(x-a)^m, \tag{11.3}$$

ignoring the terms for $n > m$.

S. Kalkan et al., *Programming with Python for Engineers*,
https://doi.org/10.1007/978-3-031-57148-0_11

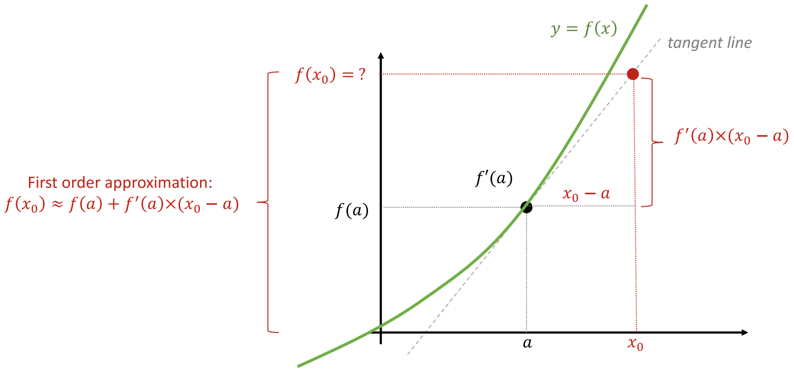


Fig. 11.1 The Taylor Series approximates a function by using its derivatives. The drawing illustrates $f(x_0)$ being approximated (with the first derivative of the Taylor Series only) by the known value of the function $f(a)$ and the derivative $f'(a)$ at that point. The gray line denotes the tangent line at $f(a)$ whose slope is $f'(a)$

Note that the denominator ($n!$) grows very large very quickly when n grows large. Therefore, the higher-order terms in the Taylor series are likely to be very small. For this reason, taking the first m terms in the series provide sufficient approximations to $f(x)$ in practice.

We can provide some intuition for the Taylor series as follows: If we know the value of the function f at $x = a$, then we can calculate the value of the function at any other value of x as long as we know all (or the first m) derivatives of the function at $x = a$, i.e., $f^{(n)}(a)$ for $n = 1, ..., \infty$ (or m).

To see this better, consider the Taylor series approximation with the first two terms: $f(x) \approx f(a) + \frac{f'(a)}{1!}(x - a)$, which is illustrated in Fig. 11.1. With this *first-order* approximation, we can use the first derivative of the function to calculate the change in the function's value and anticipate the value for a new x (x_0 in the figure).

If the function changes with higher-order nonzero derivatives, there will be a difference between the correct value of the function ($f(x_0)$) and our approximation. This is called the *approximation error*.

11.1.1 Taylor Series Example in Python

Let us take a "simple" function, namely $f(x) = x^3$, and approximate it with the first few terms of the Taylor Series:

Hands-on Code 11.1

https://pp4e.online/c11s1

```
# x^3 function
def xcube_a(a): return a*a*a

# Function for the derivatives of x^3
#   Inputs: n (integer) => derivative degree
#           a (real number) => constant at which derivative is calculated
def xcube_nth_deriv_at_a(n, a):
  if n == 1: return 3*a*a # 1st derivative: 3x*x
  if n == 2: return 6*a   # 2nd derivative: 6x
  if n == 3: return 6     # 3rd derivative: 6
  if n > 3:  return 0     # nth derivative (n>3): 0

# Factorial function
def fact(n): return 1 if n < 1 else n*fact(n-1)

# Function approximating x^3 with m terms of the Taylor series
def xcube_approx(x, m):
  a = 1     # You can try different a values
```

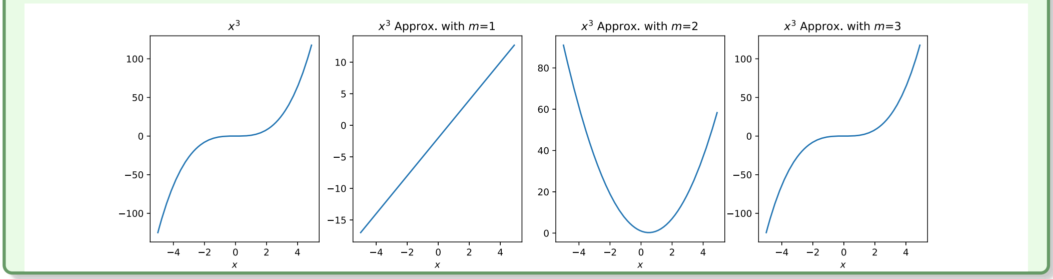
continues on next page

continued from previous page

```
 result = xcube_a(a)   # f(a)
 for n in range(1, m+1):
  result += xcube_nth_deriv_at_a(n, a)/fact(n)*pow(x-a, n)

 return result
# Now let us evaluate how close our approximation is for various values of m.
import numpy as np
import matplotlib.pyplot as plt

# Let us compare the approximations for a set of x values:
x_vector=np.arange(-5, 5, 0.1)

# Find the approximations for our x values:
y_m_1=[xcube_approx(x,1) for x in x_vector]
y_m_2=[xcube_approx(x,2) for x in x_vector]
y_m_3=[xcube_approx(x,3) for x in x_vector]

# We have our ground truth (correct) values:
y_correct = [x*x*x for x in x_vector]

# Let's plot the results and the correct function (x**3)
# We have a multi-plot plot, so we will use subplot() function
plt.rcParams['figure.figsize'] = 15, 4 # Sets the figure size

# First subplot: The correct (original) function that we are approximating
plt.subplot(1, 4, 1)
plt.plot(x_vector,y_correct, label='x**3')
plt.title('$x^3$')
plt.xlabel('$x$')

# Second subplot: Approximation with one term
plt.subplot(1, 4, 2)
plt.plot(x_vector,y_m_1)
plt.title("$x^3$ Approx. with $m$=1")
plt.xlabel('$x$')

# Third subplot: Approximation with two terms
plt.subplot(1, 4, 3)
plt.plot(x_vector,y_m_2)
plt.title("$x^3$ Approx. with $m$=2")
plt.xlabel('$x$')

# Fourth subplot: Approximation with three terms
plt.subplot(1, 4, 4)
plt.plot(x_vector,y_m_3)
plt.title("$x^3$ Approx. with $m$=3")
plt.xlabel('$x$')
```

Output

```
Text(0.5, 0, '$x$')
```



11.2 Finding the Roots of a Function

Let us assume that we have a non-linear, continuous function $f(x)$ that intersects the horizontal axis as in Fig. 11.2. We are interested in finding r_0, ..., r_n such that $f(r_i) = 0$. We call these intersections (r_0, r_1, r_2 in Fig. 11.2) the *roots* of function $f()$.

For many problems, finding the values of these intersections is a fundamental step. For example, many problems requiring solutions to equations like $g(x) = h(x)$ can be reformulated as a root-finding problem for $g(x) - h(x) = 0$.

11.2.1 Newton's Method for Finding the Roots

Newton's method is one of the many for finding the roots of a function. It is a very common method that randomly starts at an initial, x_0, value and iteratively takes one step at a time towards a root. The method can be described with the following main iterative steps:

Step 1: Set an iteration variable, i, to 0. Initialize x_i randomly; however, if you have a good guess, it is always better to initialize x_i with that. For our example function $f(x)$ in Fig. 11.2, see the selected x_i in Fig. 11.3.

Step 2: Find the intersection of the tangent line at x_i with the horizontal axis. The tangent line to the function at x_i is illustrated as the gray dashed line in Fig. 11.3. This line can be easily obtained using x_i and $f'(x_i)$.

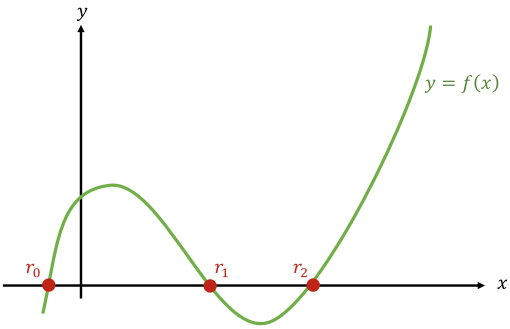


Fig. 11.2 Finding the roots of a function $f(x)$ means finding r_0, .., r_n for which the function is zero, $f(r_i) = 0$

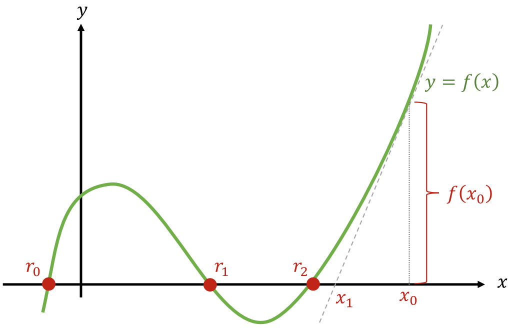


Fig. 11.3 One iteration of Newton's method. The tangent line at $f(x_0)$ is used to calculate the next value of x_i, getting us closer to a root

Let us use x_{i+1} to denote the intersection of the tangent line with the horizontal axis. Using the definition of the derivative and simple geometric rules, we can easily show that x_{i+1} satisfies the following:

$$\frac{x_i - x_{i+1}}{f(x_i)} = \frac{1}{f'(x_i)}, \tag{11.4}$$

with which we can formulate how we can find x_{i+1} as follows:

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}. \tag{11.5}$$

Step 3: Repeat Step 2 by replacing $x_i \leftarrow x_{i+1}$ until "convergence".

The intuition behind Newton's method is simple: For a linear function such as $y = ax + b$, the root is $r = -b/a$, which can be easily derived from the slope and a known y value. For a non-linear function $y = f(x)$, we make an approximation by assuming that $f()$ is linear at the point x_0 and use the slope of this linear approximation to find a new x_0 that is closer to the root.

11.2.2 Misc Details on Newton's Method for the Curious

It is not possible to cover all aspects of these elegant methods in detail. However, it would be a shame to skip the following details and hide them from the curious mind.

The Taylor Series Perspective for Newton's Method

Newton's method for finding a root of a function $f()$ works iteratively starting from an initial value x_0 using the first-order expansion of $f()$ around x_i:

$$f(x_{i+1}) \approx f(x_i) + \frac{f'(x_i)}{1!}(x_{i+1} - x_i), \tag{11.6}$$

which can be simplified by rewriting $x_{i+1} = x_i + \delta$:

$$f(x_i + \delta) \approx f(x_i) + \frac{f'(x_i)}{1!}\delta. \tag{11.7}$$

We are interested in finding δ such that the approximation is equal to zero, in other words:

$$0 = f(x_i) + \frac{f'(x_i)}{1!}\delta, \tag{11.8}$$

which yields the delta value for which the approximation is zero:

$$\delta = -\frac{f(x_i)}{f'(x_i)}. \tag{11.9}$$

Plugging this into $x_{i+1} = x_i + \delta$, we obtain the equation to calculate the next value in the sequence towards a minimum of the function $f()$:

$$x_{i+1} \leftarrow x_i - \frac{f(x_i)}{f'(x_i)}. \tag{11.10}$$

Notes on Convergence

A careful reader should have identified two important issues with Newton's method:

1- How can we be sure that this method converges to a solution, i.e., a root?

If the function $f()$ satisfies certain conditions, then we can guarantee that Newton's method converges to a root. The details of these conditions and the proofs are beyond the scope of the book and the interested reader can check [4, 5].

To be on the safe side, we can limit the maximum number of steps that we can take and use the calculated value at the end.

2- If convergence can be guaranteed, how can we know that we have converged to a solution?

A common technique in such iterative algorithms is to check whether the difference between the consecutive values is tiny. In our case, if $|x_{i+1} - x_i| < \epsilon$, where ϵ is a very small number, then we can assume that the consecutive steps do not lead to much change in x_i and, therefore, we can stop our algorithm. Of course, the value of ϵ will be critical in determining how many iterations are needed to "converge" to a result.

Notes on Multiple Roots

Newton's method converges only to a single root based on the initial value x_0. To find other roots, the algorithm needs to be executed with a different value of x_0, which will hopefully lead to a different root than the ones we have identified before.

Since our knowledge about $f()$ or its roots is limited, running the algorithm for N times for different values of x_0 is the only option. If, on the other hand, we knew roughly the ranges of x where the function $f()$ might have roots, then we could use this knowledge to choose better values as x_0.

11.2.3 Newton's Method in Python

Let us implement Newton's method in Python and try to find the roots of an example function.

Hands-on Code 11.2

https://pp4e.online/c11s2

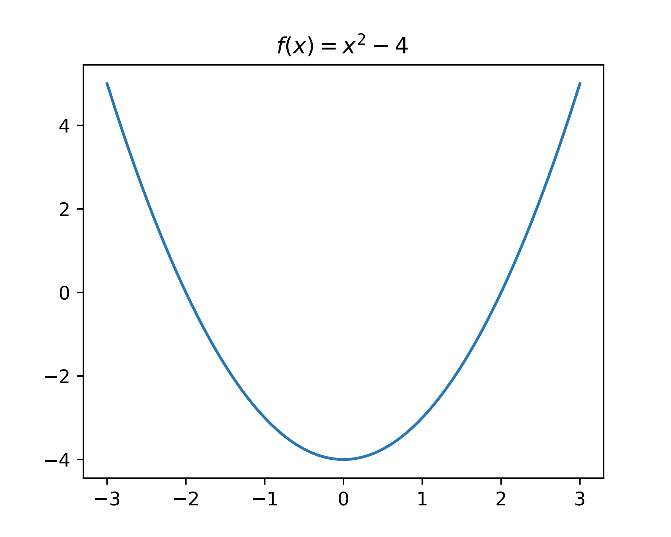
```
def find_root(f, f_deriv, x_0, max_steps=10, delta=0.001, verbose=True):
  """
    Use Newton's method to find a root of function f, given its derivative f_deriv.
    Inputs: f => function
```

continues on next page

```
      f_deriv => derivative of function f
      x_0 => initial x value
      max_steps => maximum number of iterative steps (default: 10)
      delta => if |x_i+1 - x_i| < delta, then assume convergence
 """
 x_old = x_0
 for i in range(max_steps):
  # Step 2
  try:
   x_new = x_old - f(x_old) / f_deriv(x_old)
  except ZeroDivisionError:
   # Encountered division-by-zero, return None
   return None

  if verbose: print("Iteration ", i, ". (x_old, x_new): ", (x_old, x_new), " |x_new-x_old|: ",
  ↪  abs(x_new-x_old))
  if abs(x_new-x_old) < delta: break

  # Step 3
  x_old = x_new

 return x_new
# Let us pick a function and test our implementation

# Assume that our function is: f(x) = x^2 - 4
def f(x_i): return x_i**2-4

# This is the derivative, f'(x_i)
def f_deriv(x_i): return 2*x_i

# Let us first draw our function
import matplotlib.pyplot as plt
import numpy as np

def draw_f():
 # Uniformly sample 50 x values between -5 and 5:
 x = np.linspace(-3, 3, 50)

 plt.rcParams['figure.figsize'] = 5, 4

 plt.plot(x, f(x))
 plt.title('$f(x)=x^2 - 4$')

draw_f()
```

Output

continued from previous page



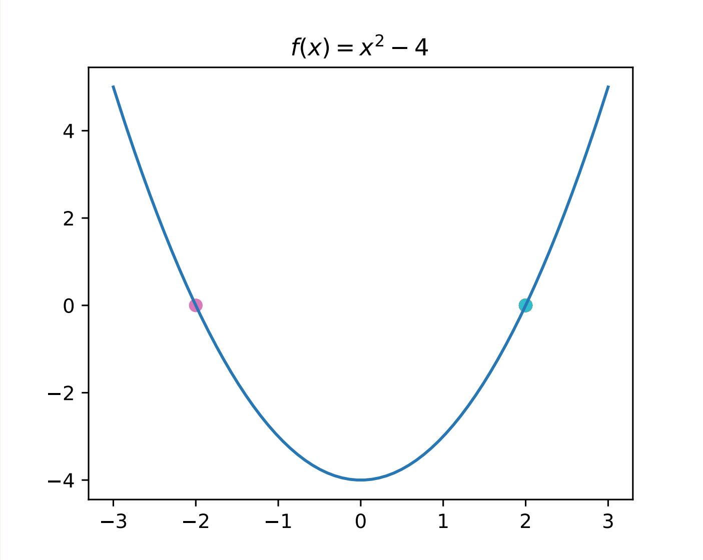
Hands-on Code 11.3

https://pp4e.online/c11s3

```python
# Let us test our solution with 10 random initial values
import random

draw_f()
for i in range(10):
  x_0 = random.randint(-5, 5) # A random integer in [-5, 5]
  r = find_root(f, f_deriv, x_0, max_steps=50, delta=0.0001, verbose=False)
  if (not r) or (abs(f(r)) > 0.001): continue # Diverged or divison-by-zero, ignore
  print("trial", i, " ended with root: ", r, " value at root, f(r): ", f(r))
  plt.scatter(r, f(r))
```

Output

```
trial 0  ended with root:  2.000000000000002  value at root, f(r):  8.881784197001252e-15
trial 1  ended with root:  -2.000000000000002  value at root, f(r):  8.881784197001252e-15
trial 2  ended with root:  2.0  value at root, f(r):  0.0
trial 3  ended with root:  2.000000000026214  value at root, f(r):  1.0485656787295738e-10
trial 4  ended with root:  2.000000000026214  value at root, f(r):  1.0485656787295738e-10
trial 5  ended with root:  -2.000000000000002  value at root, f(r):  8.881784197001252e-15
trial 6  ended with root:  -2.000000000000002  value at root, f(r):  8.881784197001252e-15
trial 7  ended with root:  2.000000000026214  value at root, f(r):  1.0485656787295738e-10
trial 8  ended with root:  2.000000000000002  value at root, f(r):  8.881784197001252e-15
trial 9  ended with root:  2.0  value at root, f(r):  0.0
```



11.2.4 Newton's Method in SciPy

Let us see how we can use Newton's method from the SciPy library (from Sect. 10.2) to find the roots of a function. Note that we do not need to provide the derivative of the function to SciPy. It uses "numerical differentiation" to calculate the derivative for us.

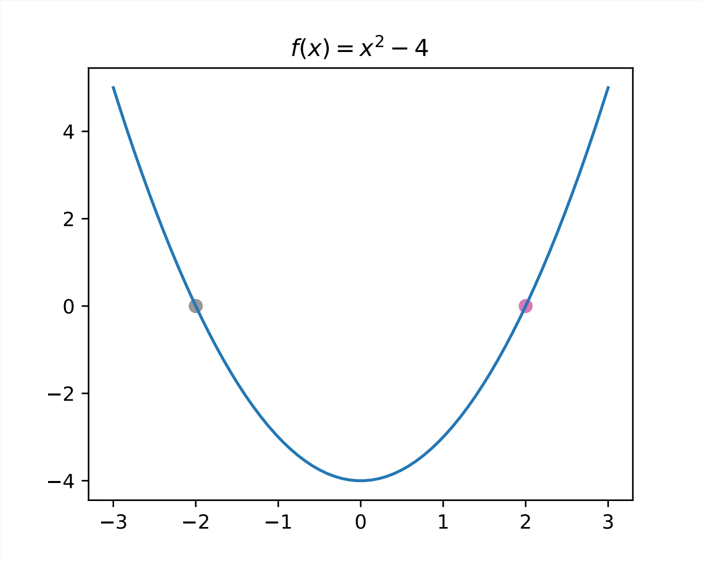
Hands-on Code 11.4

https://pp4e.online/c11s4

```
from scipy.optimize import newton
import random

draw_f() # Defined above
for i in range(10):
  x_0 = random.randint(-5, 5) # A random integer in [-5, 5]
  r = newton(f, x_0, fprime=None, args=(), tol=0.0001, maxiter=50)
  if (not r) or (abs(f(r)) > 0.001): continue # Diverged or divison-by-zero, ignore
  print("trial", i, " ended with root: ", r, " value at root, f(r): ", f(r))
  plt.scatter(r, f(r))
```

Output

```
trial 0  ended with root:  2.000000075022694  value at root, f(r):  3.000907806693931e-07
trial 1  ended with root:  -2.0000000032851974  value at root, f(r):  1.3140789789645169e-08
trial 2  ended with root:  2.000000075022694  value at root, f(r):  3.000907806693931e-07
trial 3  ended with root:  2.0000000032851974  value at root, f(r):  1.3140789789645169e-08
trial 5  ended with root:  -2.0  value at root, f(r):  0.0
trial 6  ended with root:  -2.0000000003832255  value at root, f(r):  1.532901805489928e-09
trial 7  ended with root:  2.000000075022694  value at root, f(r):  3.000907806693931e-07
trial 8  ended with root:  -2.0000000003832255  value at root, f(r):  1.532901805489928e-09
trial 9  ended with root:  -2.0000000032851974  value at root, f(r):  1.3140789789645169e-08
```



11.3 Finding a Minimum of Functions

An important problem frequently encountered in many disciplines is *optimization*. In such problems, as illustrated in Fig. 11.4, we have a function $f(x)$ and we wish to find its minimum value (f^*):

$$f^* \leftarrow \min_{x \in \mathbb{R}} f(x), \tag{11.11}$$

or the value (x^*) that minimizes it:

$$x^* \leftarrow \arg\min_{x \in \mathbb{R}} f(x). \tag{11.12}$$

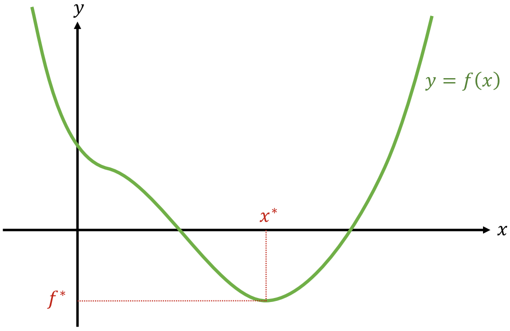


Fig. 11.4 Finding a minimum of a function $f(x)$ means finding x^* for which $f(x^*)$ is the lowest value (or lower than all others in a local neighborhood) that function $f()$ can take

In many practical settings, the functions with which we work may have multiple minima. In such a case, the minimum point, x^*, can be either (i) a *global minimum* such that there may *not* be any other point $\hat{x}$ with $f(\hat{x}) < f(x^*)$, (ii) a *local minimum* such that x^* is the minimum of a local neighborhood.

The methods generally exploit two important cues: (i) At a minimum, the sign of the first-order derivative changes (on the sides of the minimum). (ii) At a minimum, the first derivative is zero.

11.3.1 Newton's Method for Finding the Minimum of a Function

Similar to find the root of a function explained in Sect. 11.1.1, Newton's method for finding a local minimum of a function $f()$ works iteratively starting from an initial value x_0 by using the *second-order expansion* of $f()$ around x_i this time:

$$f(x_{i+1}) \approx f(x_i) + \frac{f'(x_i)}{1!}(x_{i+1} - x_i) + \frac{f''(x_i)}{2!}(x_{i+1} - x_i)^2, \tag{11.13}$$

which can be simplified by rewriting $x_{i+1} = x_i + \delta$:

$$f(x_i + \delta) \approx f(x_i) + \frac{f'(x_i)}{1!}\delta + \frac{f''(x_i)}{2!}\delta^2. \tag{11.14}$$

We are interested in finding δ that minimizes the approximation. This can be obtained by setting the first derivative to zero:

$$\frac{d}{d\delta}\left(f(x_i) + \frac{f'(x_i)}{1!}\delta + \frac{f''(x_i)}{2!}\delta^2 \right) = f'(x_i) + f''(x_i)\delta = 0, \tag{11.15}$$

which yields the delta value minimizing the approximation:

$$\delta = -\frac{f'(x_i)}{f''(x_i)}. \tag{11.16}$$

Plugging this into $x_{i+1} = x_i + \delta$, we obtain the equation for calculating the next value in the sequence towards a minimum of the function $f()$:

$$x_{i+1} \leftarrow x_i - \frac{f'(x_i)}{f''(x_i)}. \tag{11.17}$$

11.3.2 Misc Details for the Curious

Notes on Convergence

Newton's method can converge to a local minimum if (i) f is a strongly convex function with "Lipschitz Hessian" and (ii) x_0 is close enough to $x^* \leftarrow \arg\min_{x\in\mathbb{R}} f(x)$. More formally when:

$$\|x_{i+1} - x^*\| \le \frac{1}{2}\|x_i - x_*\|^2. \qquad \forall i \ge 0. \tag{11.18}$$

In plain English, (i) we have a "well-shaped" local neighborhood in which we can talk about the concept of a minimum, (ii) our initial value, x_0, is within such a neighborhood.

Notes on Multiple Minima

Similar to the case for root finding, Newton's method converges only to a single minimum based on the initial value x_0. In order to find other minima, the algorithm must be executed with a different value of x_0, which will hopefully lead to a different minimum than the ones that we have identified before.

11.3.3 Newton's Method in Python

Now let us implement Newton's method for finding a minimum of a function from scratch, in Python.

Hands-on Code 11.5

https://pp4e.online/c11s5

```
def find_local_min(f_deriv, f_second_deriv, x_0, max_steps=10, delta=0.001, verbose=True):
  """
    Use Newton's method to find a local minimum of function f, given its derivative f_deriv.
    Inputs: f => function
            f_deriv => derivative of function f
            x_0 => initial x value
            max_steps => maximum number of iterative steps (default: 10)
            delta => if |x_i+1 - x_i| < delta, then assume convergence
  """
  x_old = x_0
  for i in range(max_steps):
    try:
      x_new = x_old - f_deriv(x_old) / f_second_deriv(x_old)
    except ZeroDivisionError:
      # Encountered division-by-zero, return None
      return None

    if verbose: print("Iteration ", i, ". (x_old, x_new): ", (x_old, x_new), " |x_new-x_old|: ",
    ↪   abs(x_new-x_old))
    if abs(x_new-x_old) < delta: break

    # Step 4
    x_old = x_new

  return x_new

# Assume that our function is: f(x) = x^2 - 4
def f_deriv(x_i): return 2*x_i
def f_second_deriv(x_i): return 2
# Let us test our solution with 10 random initial values
import random
```

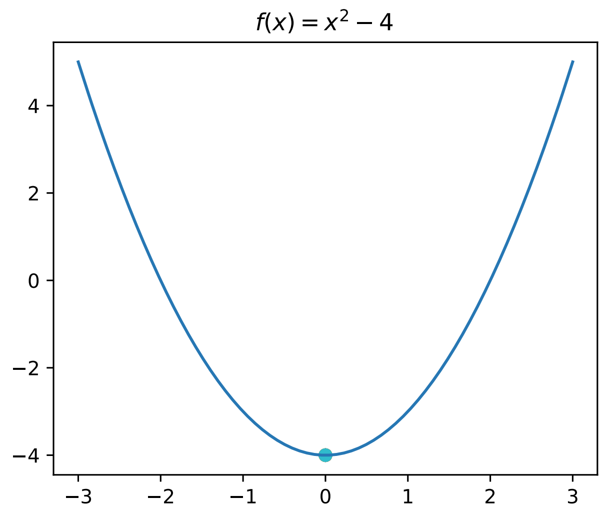
continues on next page

continued from previous page

```
draw_f() # defined in Sect. 11.2
for i in range(10):
  x_0 = random.randint(-5, 5) # A random integer in [-5, 5]
  x_min = find_local_min(f_deriv, f_second_deriv, x_0, max_steps=50, delta=0.0001, verbose=False)
  if (x_min == None) or (abs(f_deriv(x_min)) > 0.001): continue # Diverged or divison-by-zero, ignore
  print("trial", i, "led to x_min: ", x_min, ", f'(x_min): ", f_deriv(x_min))
  plt.scatter(x_min, f(x_min))
```

Output

```
trial 0 led to x_min:  0.0 , f'(x_min):  0.0
trial 1 led to x_min:  0.0 , f'(x_min):  0.0
trial 2 led to x_min:  0.0 , f'(x_min):  0.0
trial 3 led to x_min:  0.0 , f'(x_min):  0.0
trial 4 led to x_min:  0.0 , f'(x_min):  0.0
trial 5 led to x_min:  0.0 , f'(x_min):  0.0
trial 6 led to x_min:  0.0 , f'(x_min):  0.0
trial 7 led to x_min:  0.0 , f'(x_min):  0.0
trial 8 led to x_min:  0.0 , f'(x_min):  0.0
trial 9 led to x_min:  0.0 , f'(x_min):  0.0
```



11.3.4 Newton’s Method for Finding Minima in SciPy

Unfortunately, Newton’s method as explained in Sect. 11.3.1 does not exist in SciPy. However, there are many alternatives, and we will just illustrate one of them here.

Hands-on Code 11.6

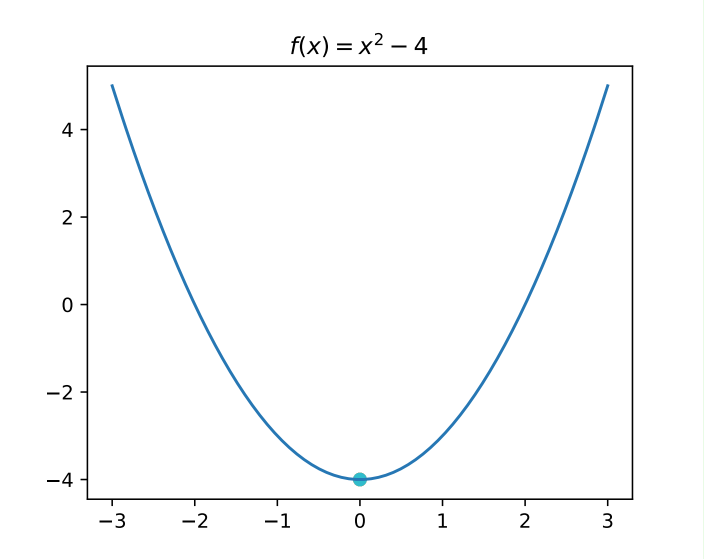
https://pp4e.online/c11s6

```
from scipy.optimize import minimize_scalar
import random

draw_f()
for i in range(10):
  x_0 = random.randint(-5, 5) # A random integer in [-5, 5]
  result = minimize_scalar(f, method='golden', tol=0.001, options={'maxiter': 50})
```

```
    x_min = result['x']
    print("trial", i, "led to x_min: ", x_min, ", f'(x_min): ", f_deriv(x_min))
    plt.scatter(x_min, f(x_min))
```

Output

```
trial 0 led to x_min:  1.4872654968309704e-08 , f'(x_min):  2.974530993661941e-08
trial 1 led to x_min:  1.4872654968309704e-08 , f'(x_min):  2.974530993661941e-08
trial 2 led to x_min:  1.4872654968309704e-08 , f'(x_min):  2.974530993661941e-08
trial 3 led to x_min:  1.4872654968309704e-08 , f'(x_min):  2.974530993661941e-08
trial 4 led to x_min:  1.4872654968309704e-08 , f'(x_min):  2.974530993661941e-08
trial 5 led to x_min:  1.4872654968309704e-08 , f'(x_min):  2.974530993661941e-08
trial 6 led to x_min:  1.4872654968309704e-08 , f'(x_min):  2.974530993661941e-08
trial 7 led to x_min:  1.4872654968309704e-08 , f'(x_min):  2.974530993661941e-08
trial 8 led to x_min:  1.4872654968309704e-08 , f'(x_min):  2.974530993661941e-08
trial 9 led to x_min:  1.4872654968309704e-08 , f'(x_min):  2.974530993661941e-08
```



11.4 Important Concepts

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

- Taylor series for approximation of functions.
- Newton's method for finding the roots and the minima of functions.
- Local minimum versus global minimum.

11.5 Further Reading

- Taylor series chapter of "Calculus Volume 2" available at https://openstax.org/books/calculus-volume-2/pages/6-3-taylor-and-maclaurin-series [3].
- Newton's method section of "Calculus Volume 1" available at https://openstax.org/books/calculus-volume-1/pages/4-9-newtons-method [2] and Sect. 2.7 of "Contemporary Calculus" available at https://www.contemporarycalculus.com/ [1].

11.6 Exercises

1. For the Taylor series approximation of x^3 in Sect. 11.1.1:

   a. Discuss how and why the approximation changes when we add more terms.
   b. Discuss why having three terms in the approximation yields a function that is very close to the original function.
   c. Write a Python code that checks whether an approximation is exactly the same as the function being approximated. Use this code to check whether using three terms in Taylor series expansion was sufficient to approximate x^3.

2. Approximate the $\sin(\cdot)$ function:

   a. Approximate the $\sin(\cdot)$ function with the Taylor series by taking $a = 0$. You will need the nth derivative of sin, which turns out to be simply $\frac{d^n}{dx^n}\sin(x) = \sin(x + n\pi/2)$.
   b. Plot the approximation for a different number of terms (m): 1, 2, 3, and 4. Observe what happens when more terms are added to the approximation.
   c. Compare your approximation results with those we obtained for x^3. Discuss the reasons for similarities and differences.

3. Find the roots of the following functions using Newton's method:

   a. $f(x) = x^2 - 5x + 6$ for $x \in [0, 5]$.
   b. $f(x) = \sin(x)$ for $x \in [0, \pi]$.
   c. $f(x) = \log(x)$ for $x \in [0.5, 2.5]$.

4. Consider a second-order function, e.g., $f(x) = x^2 - 5x + 6$, which has a minimum at $x = 2.5$.

   a. Plot $f()$ and its derivative $f'()$ using matplotlib.
   b. Do you see a link between the root of $f'()$ and the minimum of $f()$?
   c. Using `scipy.optimize.newton()` function, which is for finding the roots of a function, find the *minimum* of function $f()$. Hint: A minimum is a root of the first-order derivative.

References

[1] D. Hoffman, *Contemporary Calculus*. Open Book at https://www.contemporarycalculus.com/ (2013)

[2] G. Strang, E.J. Herman, *Calculus Volume 1*. OpenStax (2016)

[3] G. Strang, E.J. Herman, *Calculus Volume 2*. OpenStax (2016)

[4] Wikipedia contributors: Newton's method, (Analysis Section), Wikipedia, The Free Encyclopedia, https://en.wikipedia.org/w/index.php?title=Newton (2023). Accessed 5-September-2023

[5] T. Yamamoto, Historical developments in convergence analysis for newton's and newton-like methods. J. Comput. Appl. Math. **124**(1–2), 1–23 (2000)

12 An Application: Solving a Simple Regression Problem

In this chapter, we cover another important problem frequently encountered in many disciplines: The problem of fitting (*regressing*) a function to a set of data, the so-called *regression* problem. We look at two versions, namely *linear regression* and *non-linear regression*, using SciPy. While doing so, we also use Pandas, NumPy, and Matplotlib, which we covered in Chap. 10.

12.1 Introduction

Let us first formulate the problem and clearly outline the notation. In a regression problem, we have a set of x, y values: $\{(x_0, y_0), (x_1, y_1), ..., (x_n, y_n)\}$, where x_i and y_i can be multidimensional variables (i.e., vectors) but for the sake of simplicity, they are assumed to one-dimensional in this chapter.

In a regression problem, we are interested in finding the function $f()$ that goes through those (x_i, y_i) values. In other words, we wish to find $f()$ that satisfies the following:

$$y_i = f(x_i). \tag{12.1}$$

At first, this might seem difficult, since we are trying to estimate a function from its input–output values and there can be infinitely many functions that can go through a finite set of (x_i, y_i) values. However, by restructuring the problem a little bit and making some assumptions about the general form of $f()$, we can easily solve such complicated regression problems.

Let us briefly describe how we can do so: We first re-write $f()$ as a *parametric* function and try to formulate the regression problem as the problem of finding the parameters of $f()$:

$$y = f(x) = f(x; \boldsymbol{\theta}) = \sum_{i=1}^{m} g_i(x; \theta_i), \tag{12.2}$$

where $\boldsymbol{\theta} = (\theta_1, ..., \theta_n)$ is the set of parameters that we need to find to solve the regression problem; and $g_1(), ..., g_m()$ are our "simpler" functions (compared to $f()$) whose parameters can be identified more easily compared to $f()$.

S. Kalkan et al., *Programming with Python for Engineers*,
https://doi.org/10.1007/978-3-031-57148-0_12

With this parameterized definition of $f()$, we can formally define regression as an optimization (minimization) problem (see also Chap. 11):

$$\theta^* \leftarrow \arg\min_{\theta \in \mathbb{R}^d} \sum_i (y_i - f(x_i; \theta))^2 , \tag{12.3}$$

where the summation runs over the (x_0, y_0), (x_1, y_1), ..., (x_n, y_n) values that we are trying to regress; $(y_i - f(x_i; \theta))^2$ is the error between the estimated (regressed) value (i.e., $f(x_i; \theta)$) and the correct y_i value with θ as the parameters.

We can describe this optimization formulation in words as follows: Find parameters θ that minimize the error between y_i and what the function predicts given x_i as input.

This is a minimization problem, and in Chap. 11, we have seen a simple solution and how SciPy can be used for such problems. In this chapter, we will spend more time on this and examine sample regression problems and solutions. That is,

- *Linear regression,* where $f()$ is assumed to be a line, $y = ax + b$, and θ is (a, b).
- *Non-linear regression*, where $f()$ has a non-linear form, e.g., a quadratic, exponential, or logarithmic function.

12.1.1 Why Is Regression Important?

In many disciplines, we observe an environment, an event, or an entity through sensors and obtain some information about what we observe in the form of a variable x. Each x generally has an associated outcome variable y. Or, x can represent an action that we are exerting in an environment and we are interested in finding how x affects the environment by observing a variable y.

Here are some examples for x and y for which finding a function $f()$ can be really useful:

- x: The current that we are providing to a motor. y: The speed of the motor.
- x: The day of a year. y: The average rainfall in a year.
- x: The changes in the stock value of a bond on the last day. y: The value of a bond in one hour.
- x: The number of COVID cases per day. y: The number of casualties due to COVID.

12.1.2 The Form of the Function

As discussed before, solving a regression problem without making any assumptions about the underlying function is very challenging. However, an incorrect assumption can produce an inaccurate regression model as illustrated in Fig. 12.1 (see also https://xkcd.com/2048/ for a simplified comic illustration):

If the form of the function is too simple (e.g., linear), the fitted function will not be able to capture the true nature of the data. However, if the assumed function form is too complex (e.g., a higher-order polynomial), the fitted function will try to go through every data point, which may not be ideal since the data can be noisy.

Choosing the right form of the function is therefore tricky and requires some prior idea about the function or the problem. If the relationship between x and y is known to be linear, then a linear function form should be chosen. If the relation between x and y is known to be non-linear, but the exact form is unknown, x and y can first be plotted to get a sense of the underlying form. This can give an idea of what kinds of function forms can be suitable.

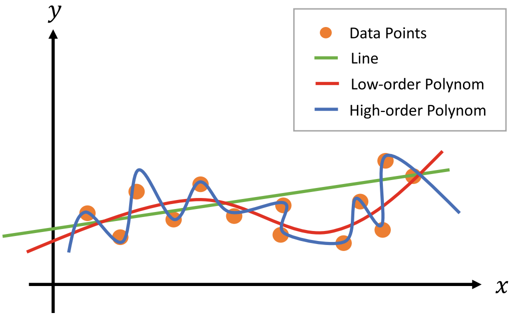


Fig. 12.1 An unsuitable function form may yield sub-optimal regression results

12.2 Least-Squares Regression

A commonly used method for regression is called *Least-Squares Regression* (LSE). In LSE, the minimization problem in Sect. 12.1 is solved by setting the *gradient of the objective* to zero:

$$\frac{\partial}{\partial \theta_j} \sum_i (y_i - f(x_i;\theta))^2 = 0, \quad \text{for each } j. \tag{12.4}$$

Linear Least-Squares Regression

If our function is linear, i.e., $f(x_i;\theta) = \theta_1 x_i + \theta_2$, setting the gradient to zero provides us with a set of equations for each θ_j. If the number of data points (x_i, y_i) is greater than the number of variables, these equations can be easily solved to obtain $\boldsymbol{\theta}$ that minimizes the objective in Sect. 12.1. Those interested in the derivation of the solution and its exact form can look up, e.g., the "Linear Regression: Least Squares Estimation" chapter of [1].

Non-linear Least-Squares Regression. If our function is not linear, a solution can be obtained iteratively by making a linear approximation/step at each iteration using the so-called *Gauss–Newton method* [2, 4].

12.3 Linear Regression with SciPy

SciPy has a function called `scipy.stats.linregress(x, y)` that we can use to regress a line passing through a set of points. Let us go through an example to see how we can do that. Moreover, we will use our regressed line to analyze some properties of our data.

12.3.1 Create Artificial Data

To keep things simple, let us generate some artificial data in Python and save it in a file called `ch12_linear_data_example.csv`. We have already run this code and saved this CSV file on the Web. In the following steps, we will continue with this file on the Web. However, you can change the following code to generate data in different forms and with different noise levels and observe how the different regression methods perform.

Hands-on Code 12.1

https://pp4e.online/c12s1

```python
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd

def plot_func(f, params, xs, legend):
  ys = f(xs, *params) # This is called parameter unpacking -- Google and learn this
  plt.plot(xs, ys, label=legend)

def line_func(x, a, b):
  """Line function."""
  return a*x + b

xs = np.linspace(-10, 10, 100)
ys = line_func(xs, 3, 0.5) + np.random.normal(0, 4, xs.size) # add random noise

df = pd.DataFrame({'x': xs, 'y': ys})
df.to_csv("ch12_linear_data_example.csv")

print("Created {0} many data points and saved them into CSV file.".format(xs.size))
```

Output

```
Created 100 many data points and saved them into CSV file.
```

12.3.2 Download and Visualize Data

You can download the `ch12_linear_data.csv` file from the interactive webpage, under the `input` directory. This way, you can experience the complete data analysis pipeline.

Hands-on Code 12.2

https://pp4e.online/c12s2

```python
# Import the necessary libraries
import pandas as pd

# Read the file named 'ch12_linear_data.csv'
df = pd.read_csv('ch12_linear_data.csv')

# Print the CSV file's contents:
print("The CSV file contains the following:\n", df, "\n")

# Check the types of each column
df.dtypes
```

Output

```
The CSV file contains the following:
     Unnamed: 0          x          y
0            0 -10.000000 -29.198799
1            1  -9.797980 -33.379918
2            2  -9.595960 -25.661456
3            3  -9.393939 -24.037193
4            4  -9.191919 -25.264474
..         ...        ...        ...
95          95   9.191919  27.905021
96          96   9.393939  24.494575
97          97   9.595960  32.212389
98          98   9.797980  28.511129
99          99  10.000000  33.593768

[100 rows x 3 columns]

Unnamed: 0      int64
x             float64
y             float64
dtype: object
```

We can quickly visualize the loaded data as follows:

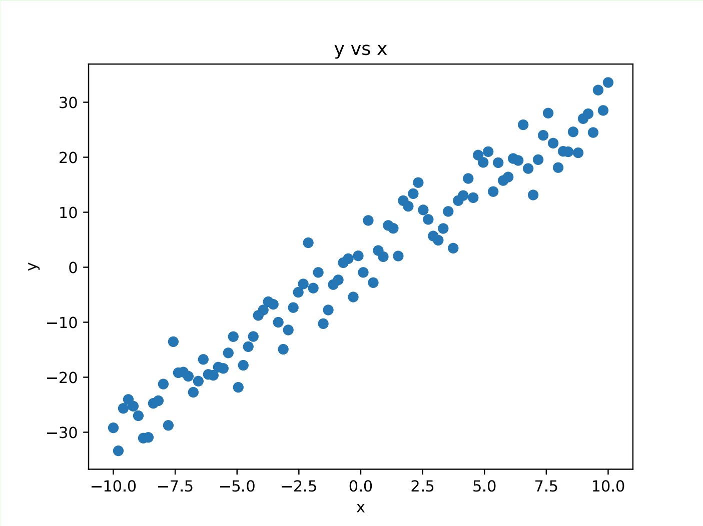
Hands-on Code 12.3

https://pp4e.online/c12s3

```
# Let us visualize the data
import matplotlib.pyplot as plt

xs = df['x'][:].values
ys = df['y'][:].values

# Plot y vs. x
plt.scatter(xs, ys)

# Set the labels for x and y axes:
plt.xlabel(’x’)
plt.ylabel(’y’)

# Set the title of the figure
plt.title(”y vs x”)
```

Output

```
Text(0.5, 1.0, ’y vs x’)
```



We see that our data is rather “noisy”, though it more or less follows a linear function. Let us see if we can capture that.

12.3.3 Fit a Linear Function with SciPy

Now, let us fit a line (linear function) to our data using `scipy.stats.linregress()`. The specific method used by this function is least-squares regression which we briefly explained in Sect. 12.2.

Hands-on Code 12.4

https://pp4e.online/c12s4

```
# Import stats module
from scipy import stats

# Fit a linear model:
solution = stats.linregress(xs, ys)

# Get the slope (a) and the intercept (b) in y = ax + b
a_estimated = solution.slope
b_estimated = solution.intercept

print(”The estimated solution is: y = %4f x + %4f” % (a_estimated, b_estimated))
```

Output

```
The estimated solution is: y = 2.963812 x + 0.442237
```

12.3.4 Analyze the Solution

We have obtained a solution. Now, let us analyze how good the solution is. We can do that *qualitatively* (by visual inspection) or *quantitatively* (by looking at some numerical measures of how good the fit is).

1- Qualitative Analysis

For regression, a qualitative analysis can be performed by plotting the data, the obtained solution, and the correct solution. If the obtained solution goes through the data well and is close to the original solution in the graph, then we can visually evaluate whether the solution looks acceptable.

Let us do this for our example:

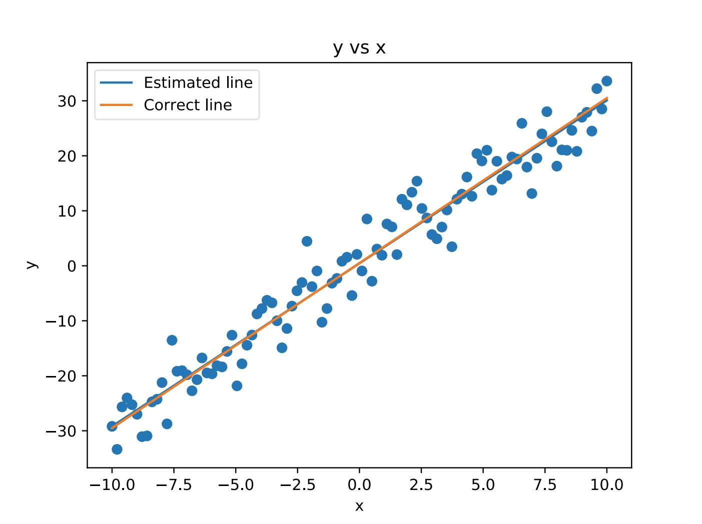
Hands-on Code 12.5

https://pp4e.online/c12s5

```
# Plot y vs. x
plt.scatter(xs, ys)

# Plot the estimated line
plot_func(line_func, (a_estimated, b_estimated), xs, "Estimated line")

# Plot the correct line
# The data was generated from y = 3x + 0.5
a_correct = 3
b_correct = 0.5
plot_func(line_func, (a_correct, b_correct), xs, "Correct line")

# Set the labels for x and y axes:
plt.xlabel('x')
plt.ylabel('y')

# Set the title of the figure
plt.title("y vs x")

# Legend
plt.legend()
```

Output

```
<matplotlib.legend.Legend at 0x12e6d8d30>
```



From this plot, we can visually confirm that the obtained solution (blue line) passes through the original data points well and is almost identical to the correct line, i.e., the line from which we generated the original (x, y) pairs. Therefore, our regression solution appears to be visually (qualitatively) accurate.

2- Quantitative Analysis

Qualitative analysis can sometimes be difficult and its judgment is subjective; i.e., different observers might draw different conclusions, especially in cases where the solution is different from the correct solution. Therefore, in addition to qualitative analysis, as we did above, it is very important to measure the quality of our solution with a number.

Depending on the problem, different measures can be used. For regression, we frequently use the following three:

1. *Mean Squared Error (MSE)*. For N data points, assuming that $\hat{y}_i$ denotes the predicted value and y_i the correct value for x_i, MSE can be defined as follows:

$$MSE = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2, \tag{12.5}$$

   which essentially averages the squared errors for each data point.
2. *Root Mean Squared Error (RMSE)*. RMSE is simply the square root of MSE:

$$RMSE = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2}. \tag{12.6}$$

3. R^2 *Error* (also called the coefficient of determination). R^2 is a normalized measure that also considers the error of the "correct" solution. Since the data can be noisy, this can be a better way to understand how good/bad our error value (MSE or RMSE) was. R^2 Error is defined as

$$R^2 = 1 - \frac{\text{MSE of estimates}}{\text{MSE wrt. the mean of y}} = 1 - \frac{1/N\sum_{i=1}^{N}(y_i - \hat{y}_i)^2}{1/N\sum_{i=1}^{N}(y_i - \overline{y}_i)^2} = 1 - \frac{\sum_{i=1}^{N}(y_i - \hat{y}_i)^2}{\sum_{i=1}^{N}(y_i - \overline{y})^2}, \tag{12.7}$$

   where $\overline{y} = 1/N\sum_i y_i$ is the mean of y values.
   If $R^2 = 1$, that means the MSE of the estimates is zero; i.e., we have obtained a perfect regression. If $R^2 = 0$, all $y_i = \overline{y}$; in other words, all estimates are the mean of the y values, therefore, the regression method did not find a good solution. If, on the other hand, $R^2 < 0$, that means the obtained solution is even worse than the mean y_i values.

Now, let us analyze our solution using these three metrics.

Hands-on Code 12.6

https://pp4e.online/c12s6

```python
def MSE(estimated_y, correct_y):
  """
    Mean Squared Error between the estimated_y and correct_y values.
    Inputs are both numpy arrays.
  """
  result = 0
  N = estimated_y.size
  for i in range(N):
    result += (estimated_y[i] - correct_y[i]) ** 2
  return result / N

def RMSE(estimated_y, correct_y):
  """
    Root Mean Squared Error between the estimated_y and correct_y values.
    Inputs are both numpy arrays.
  """
  return MSE(estimated_y, correct_y) ** 0.5

def R_squared(estimated_y, correct_y):
  """
    R^2 error between the estimated_y and correct_y values.
    Inputs are both numpy arrays.
  """
  MSE_of_estimates = MSE(estimated_y, correct_y)
  mean_y = correct_y.mean()
  mean_y_array = np.full(correct_y.size, mean_y)
  MSE_wrt_mean_of_y = MSE(estimated_y, mean_y_array) # an array filled with mean-y
  result = 1 - MSE_of_estimates / MSE_wrt_mean_of_y

  return result

estimated_y = xs * a_estimated + b_estimated
correct_y = ys

print("MSE Error: ", MSE(estimated_y, correct_y))
print("RMSE Error: ", RMSE(estimated_y, correct_y))
print("R-squared Error: ", R_squared(estimated_y, correct_y))
```

Output

```
MSE Error:  14.710406962785715
RMSE Error:  3.835414835814467
R-squared Error:  0.950755415581623
```

We see that we have obtained very low errors. Especially the R^2 value is very close to one, which suggests a good approximation of the regressed line to the underlying solution. Note that since our data was noisy, it was not possible to obtain $R^2 = 1$.

12.4 Non-linear Regression with SciPy

Let us now see how we can perform non-linear regression.

12.4.1 Create Artificial Data

To keep things simple, similar to the linear regression example, let us generate some artificial data in Python and save it in a file called `ch12_nonlinear_data_example.csv`. We have already run this code and saved this CSV file on the Web. In the following steps, we will continue with this file on the Web. However, you can change the following code to generate data in different forms and with different noise levels and observe how the different regression methods perform.

Hands-on Code 12.7

https://pp4e.online/c12s7

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd

def nonlinear_f(x, a, b):
    return  a * np.sin(x) + b

xs = np.linspace(-10, 10, 100)
ys = nonlinear_f(xs, a=3, b=0.5) + np.random.normal(0, 0.7, xs.size) # add random noise

df = pd.DataFrame({'x': xs, 'y': ys})
df.to_csv("ch12_nonlinear_data_example.csv")

print("Created {0} many data points and saved them into CSV file.".format(xs.size))
```

Output

```
Created 100 many data points and saved them into CSV file.
```

12.4.2 Download and Visualize Data

You can download the `ch12_nonlinear_data.csv` file from the interactive webpage so that we can provide you with the experience of a full data analysis pipeline.

Hands-on Code 12.8

https://pp4e.online/c12s8

```
# Import the necessary libraries
import pandas as pd

# Read the file named 'ch12_nonlinear_data.csv'
df = pd.read_csv('ch12_nonlinear_data.csv')

# Print the CSV file's contents:
print("The CSV file contains the following:\n", df, "\n")

# Check the types of each column
df.dtypes
```

Output

```
The CSV file contains the following:
     Unnamed: 0          x         y
0             0 -10.000000  3.911998
1             1  -9.797980  1.781296
2             2  -9.595960  1.125093
3             3  -9.393939  0.738523
4             4  -9.191919 -0.343199
..          ...        ...       ...
95           95   9.191919  0.667528
96           96   9.393939 -0.729794
97           97   9.595960  0.643281
98           98   9.797980 -1.533940
99           99  10.000000 -0.760156

[100 rows x 3 columns]

Unnamed: 0      int64
x             float64
y             float64
dtype: object
```

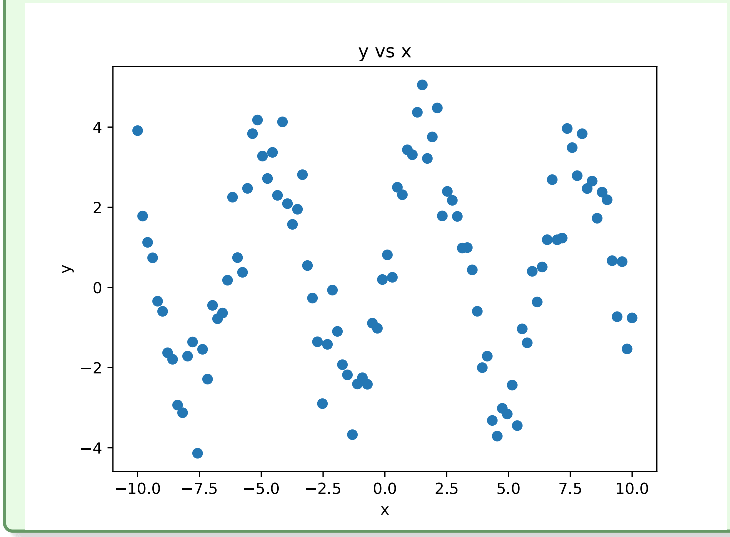
Hands-on Code 12.9

https://pp4e.online/c12s9

```
# Let us visualize the data
import matplotlib.pyplot as plt

xs = df['x'][:].values
ys = df['y'][:].values

# Plot y vs. x
plt.scatter(xs, ys)

# Set the labels for x and y axes:
plt.xlabel('x')
plt.ylabel('y')

# Set the title of the figure
plt.title("y vs x")
```

Output

```
Text(0.5, 1.0, 'y vs x')
```



12.4.3 Fitting a Non-linear Function with SciPy

For non-linear regression, we will use `scipy.optimize.curve_fit` which internally performs non-linear least-squares regression.

Hands-on Code 12.10

https://pp4e.online/c12s10

```
# Import optimize module
from scipy import optimize

def nonlinear_f(x, a, b):
    return  a * np.sin(x) + b

# Fit a linear model:
solution, _ = optimize.curve_fit(nonlinear_f, xs, ys, method='lm')

print("The estimated solution is: ", solution)
```

Output

```
The estimated solution is:  [3.11746172 0.48206522]
```

12.4.4 Analyzing the Solution

Similar to our analysis in Sect. 12.3.4, let us analyze our solution qualitatively and quantitatively.

1- Qualitative Analysis
Let us first analyze the found solution qualitatively.

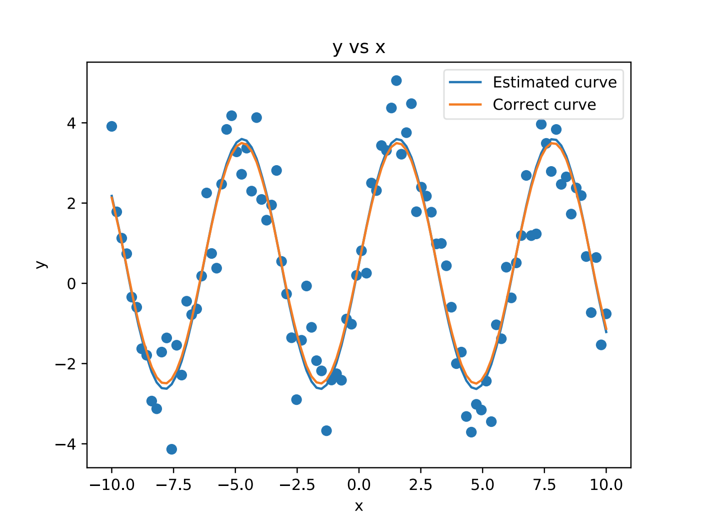
Hands-on Code 12.11

https://pp4e.online/c12s11

```
# Plot y vs. x
plt.scatter(xs, ys)

# Plot the estimated line
plot_func(nonlinear_f, solution, xs, "Estimated curve")

# Plot the correct curve
plot_func(nonlinear_f, (3, 0.5), xs, "Correct curve")

# Set the labels for x and y axes:
plt.xlabel('x')
plt.ylabel('y')

# Set the title of the figure
plt.title("y vs x")

# Legend
plt.legend()
```

Output

```
<matplotlib.legend.Legend at 0x12e7b9b80>
```



Visually, it appears that our estimated solution is a very good fit for the data, and it overlaps nicely with the correct function with which we had generated the data.

2- Quantitative Analysis
Now, let us look at some of the metrics we have defined in Sect. 12.3.4 to get a better feeling for how good our fit was:

Hands-on Code 12.12

https://pp4e.online/c12s12

```python
estimated_y = nonlinear_f(xs, *solution)
correct_y = ys

print("MSE Error: ", MSE(estimated_y, correct_y))
print("RMSE Error: ", RMSE(estimated_y, correct_y))
print("R-squared Error: ", R_squared(estimated_y, correct_y))
```

Output

```
MSE Error:  0.6748838046926273
RMSE Error:  0.8215131190021419
R-squared Error:  0.8540114575179684
```

We again see that we have obtained low error values, despite the noise. However, we could not obtain a perfect regression ($R^2 = 1$) since our data was very noisy.

12.5 Important Concepts

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

- Regression problem and its uses.
- Least-squares regression.
- Linear versus non-linear regression.
- Linear and non-linear regression with SciPy.
- Qualitative and quantitative analyses of a regression solution.

12.6 Further Reading

- A good resource which covers linear and non-linear regression is the book by T. Hastie, R. Tibshirani, and J. Friedman, titled “The Elements of Statistical Learning: Data Mining, Inference, and Prediction” [3].

12.7 Exercises

1. For clarity, we implemented the error metrics in Sect. 12.3.4 using `for` loops. Re-implement them using NumPy functions without using explicit `for` or `while` loops.
2. Generate 100 random (x, y) values from a non-linear function, e.g., $y = x^2$, for $x \in [-10, 10]$, and use the linear regression steps in Sect. 12.3 to regress a linear solution. Qualitatively and quantitatively analyze the obtained solution.
3. Choose your favorite city and download its temperature readings for two consecutive days from Open-Meteo.[1] Download the readings as CSV files and load them using Pandas. Plot the data using Matplotlib and see the general shape of the function. Choose a suitable non-linear function and follow the steps in Sect. 12.4 to regress a function to the temperature data. Measure also how well the function represents the temperature values.

[1] https://open-meteo.com/en/docs/historical-weather-api.

References

[1] T. DelSole, M. Tippett, *Statistical Methods for Climate Scientists* (Cambridge University Press, 2022)

[2] P. Deuflhard, Least squares problems: Gauss-newton methods. *Newton Methods for Nonlinear Problems: Affine Invariance and Adaptive Algorithms* (2011), pp. 173–231

[3] T. Hastie, R. Tibshirani, J.H. Friedman, J.H. Friedman, *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*, vol. 2. (Springer, 2009)

[4] Wikipedia contributors, Gauss–newton algorithm, Wikipedia, The Free Encyclopedia (2023). https://en.wikipedia.org/w/index.php?title=Gauss%E2%80%93Newton_algorithm&oldid=1171884066. Accessed 5 Sept 2023

Glossary

a

address An integer that uniquely identifies a location (byte) in the memory.

algorithm A step-by-step procedure that, when executed, leads to an output for the input we provided.

aliasing A situation where more than one variable is the name of (or point to) the same location in memory.

assembler A machine code program that serves as a translator from some relatively more readable text, the assembly program, into machine code.

assembly A human-readable low-level programming language that is very close to binary machine code that is understandable by the CPU.

Arithmetic Logic Unit (ALU) One of the important units in the CPU responsible for performing arithmetic (addition, subtraction, multiplication, division) and logic (less-than, greater-than, equal-to, etc.) operations.

b

basic data types The data types that represent single information items, i.e., integers, floating-point numbers, complex numbers, and Booleans.

Basic Input Output System (BIOS) A machine code responsible for initializing the critical computer hardware and the main operating system in the computer. It resides in a memory which, even when the power is off, does not lose its content, much like a flash drive. It is electronically located exactly at the address where the CPU looks for its first instruction.

bit The smallest unit of information in digital computers. A bit can be either 0 or 1.

Boolean A data type which can take only two values, `True` or `False`.

byte A sequence of 8 bits.

S. Kalkan et al., *Programming with Python for Engineers*,
https://doi.org/10.1007/978-3-031-57148-0

c

Central Processing Unit (CPU) One of the main components of the von Neumann Architecture responsible for executing programs. It has three main units: Control Unit, Arithmetic Logic Unit, and Registers.

class A prescription that defines a particular object. The blueprint of an object.

compiler A program that can translate code written in a high-level human-readable programming language into binary machine code. The binary code produced by a compiler is generally stored in a file that can be loaded into the memory and executed by the CPU when requested.

complexity (of algorithms) A measure for comparing different algorithms, in general, with respect to their running times.

computer An electronic device that has a “microprocessor” in it. In a broader sense, any physical entity that can do “computation” can also be called a computer.

computer program Instructions and data that perform a certain task.

computing The process of inferring data from other data.

constructor A special member function of an object that is responsible for initializing the object, e.g., its member variables.

container data types Data types that represent multiple information items. Commonly used container data types are tuple, list, str, dict, and set data types.

Control Unit (CU) A special unit in the CPU 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.

d

debugging An act of looking for errors in a code, finding their causes, and correcting them.

dictionary (dict) A mutable container data type in Python for storing pairs of (key, value).

directory A file system structure that contains files and other directories. Directories are used by operating systems to provide an easy-to-use (file system) hierarchical access to information in storage devices.

e

electronics Circuits or devices constructed using transistors, microchips, and other hardware components.

encapsulation One of the important features of object-oriented programming. It describes that data and actions are glued together in a data–action structure called “object” that conforms to a rule of “need-to-know” and “maximal-privacy”.

exception An event that occurs due to a specific error (e.g., division by zero) during the execution of a program.

expression Specification of a calculation (e.g., 3 + 4 * 5), which, when evaluated (performed), yields some data as a result.

f

Fetch–Decode–Execute Cycle The cycle followed by a CPU while running a program. A computer typically follows this cycle from the moment it is turned on until it is turned off.

file A sequence of bytes stored on a secondary storage, typically a hard drive (alternative secondary storage devices include CD, DVD, USB disk, tape drive).

file system A hierarchical organization of data into files. It usually consists of files separated into directories where directories can contain files or other directories.

floating point The data type used to represent real numbers.

flowchart A graphical representation of an algorithm. A flowchart uses boxes with specific shapes to denote the steps and arrows to describe the flow of an algorithm. See also pseudo-code.

folder —see directory.

function A grouping of actions under a given name that performs a specific task.

i

immutable An immutable data type cannot be changed after being initialized. Common examples are the tuple, str, and frozenset data types.

index An integer representing the position of an item in a container. It can be positive or negative. If it is positive, the index of the first element is zero. If it is negative, the index of the last item is −1.

inheritance One of the important features of object-oriented programming. It refers to defining ‘descendant’ objects from ‘parent’ objects so that the descendant objects inherit all functions and data of their ancestors.

instruction A machine-understandable specification of an operation that can be understood and executed by a CPU. It is often used to refer to binary machine instruction.

Instruction Register (IR) A special-purpose register in the CPU to hold the current instruction being executed.

interpreter An interactive program that translates code written in a high-level human-readable programming language into binary machine code. An interpreter does not provide binary machine code to the programmer but, instead, it always acts as an intermediary between the programmer (i.e., the programmer’s code) and the binary machine code.

interrupt A low-level mechanism that facilitates communication between peripheral devices and the CPU.

iteration A fundamental programming ingredient that facilitates repeating a sequence of instructions over and over.

iterator An object that provides a countable number of values on demand.

l

library A package that contains a wide spectrum of predefined and organized actions and data.

list comprehension A compact mechanism for specifying the elements of a list.

loop —see iteration.

m

Master Boot Record (MBR) A small table at the beginning of a storage device, which contains a short machine code program to load the operating system if there is one.

memory An electronical table composed of a set of rows with a fixed set of cells each. Each cell is responsible for representing a bit. Each consecutive 8 bits (bytes) has a unique address, and reading or modifying a byte is possible by providing its address.

mutable A mutable data type may be changed after being initialized. Common examples are the list and dict data types.

n

nesting The mechanism with which certain compound statements (or data structures) can be defined within other compound statements (or data structures). For example, iterative statements (while and for) and function definitions can be nested.

o

object A computational structure that has functions and data fields built according to the blueprint, namely the class.

Operating System (OS) A program that, after being loaded into the memory, manages resources and services such as the 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.

operation code, or opcode A fixed-length binary sequence that identifies the type of the operation in an instruction.

p

package —see library.

paradigm —see programming paradigm.

peripheral Any device outside of the von Neumann structure, namely the CPU and the memory. Examples include the keyboard, the storage devices, the monitor, and the printer.

polling A mechanism where the CPU regularly asks the peripheral devices if there is anything the CPU needs to take care of. See also interrupt.

polymorphism A mechanism allowing a descendant object to appear and function like its ancestor object when necessary.

program —see computer program.

programming paradigm A style or an approach to programming that outlines rules, concepts, and practices for developing computer programs.

pseudo-code A natural language description of the steps that are followed in an 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. See also flowchart.

Program Counter (PC) A special register in the CPU that holds the address of the next instruction in the memory.

r

Random Access Memory (RAM) An addressable memory which provides constant-time access to data for reading or writing purposes given the address of the data.

recursion A fundamental tool of the functional paradigm for defining a function such that the function being defined is called while defining the function.

register A small memory unit in the CPU that is typically addressed with a special name, e.g., Program Counter, Instruction Register, and Data Register.

s

script A human-readable program code that can be executed by an interpreter.

sign-magnitude notation An intuitive method for representing integers in binary such that the first bit is reserved for the sign of the number and the remaining bits for the magnitude. See also Two's complement notation.

set comprehension A compact mechanism for specifying the elements of a set.

slicing An indexing mechanism for describing the indices of a range of elements.

string A data type for representing a sequence of characters, i.e., text.

t

two's complement notation An ingenious method for representing integers in binary such that a negative number is represented by flipping the bits of its magnitude (in binary) and adding 1 to it. See also sign-magnitude notation.

type conversion Transforming data from one data type to another; e.g., from a floating point number to an integer.

v

variable A name or reference to a memory location that holds data.

von Neumann Architecture An architecture that describes the organization of the main hardware components and their interaction.

Index

S. Kalkan et al., *Programming with Python for Engineers*,
https://doi.org/10.1007/978-3-031-57148-0