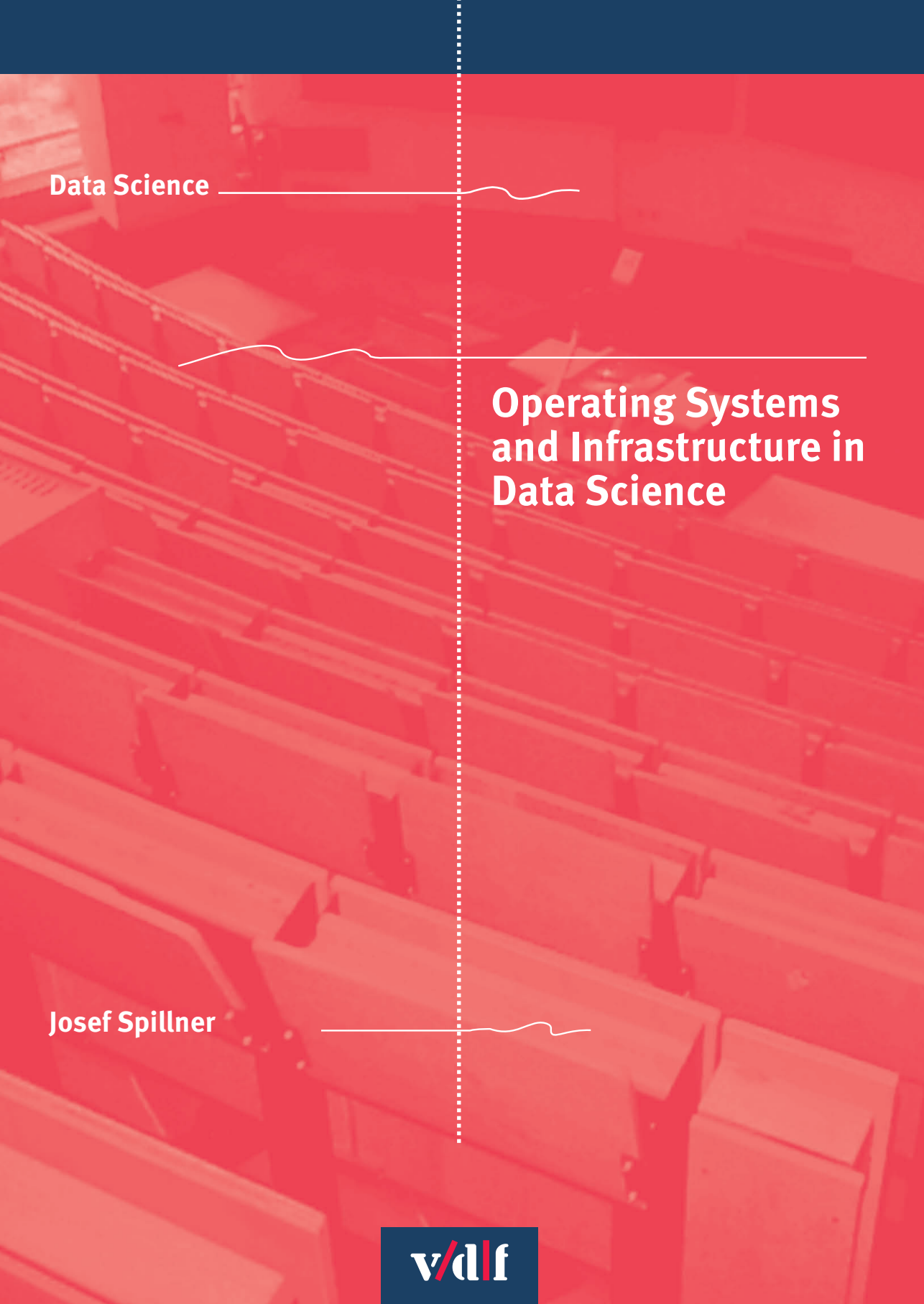
Data Science
Operating Systems and Infrastructure in Data Science
Josef Spillner
v/d/f

Operating Systems and Infrastructure in Data Science

1. Auflage

vdf Hochschulverlag AG
an der ETH Zürich

Josef Spillner

Operating Systems and Infrastructure in Data Science

1. Auflage

Financial support for the publication of this book was provided by the ZHAW University Library (HSB).

The author is affiliated with ZHAW.

Bibliografische Information Der Deutschen Bibliothek
Die Deutsche Bibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.dnb.de abrufbar.

ISBN 978-3-7281-4167-5 (Printausgabe)

Download open access:
ISBN 978-3-7281-4168-2 / DOI 10.3218/4168-2

1. Auflage 2023

www.vdf.ethz.ch
verlag@vdf.ethz.ch

Contents

Chapter 1

Introduction

The advent of the Stone Age around 2.6 million years ago marked a small but significant jump in human civilisation. In the Stone Age, humans started to become more productive through long-lasting and re-usable stone tools, first in the form of pebble tools and (much) later in the form of hand axes.

Fast-forward to several industrial revolutions that have defined and shaped modern highly productive civilisations over the past centuries. From agriculture over production to automated services and finally into the digital work domain, mastering tools has become increasingly essential skills to be productive. An interesting quote, *A man is only as good as his tools*, is attributed to Emmert Wolf, widely considered to have lived over a hundred years ago but without any further trace of existence. The statement still qualifies as valid, although with notable changes. First, while in traditional industries biological gender differences still play a role today, in digital work such as data science they do not. Second, individual work still remains important, but certain efforts require teamwork with various flavours of collaboration and coordination. Hence, the newer quote *A tool is only as good as the team using it* by Matthew Stublefield in 2019 gives the right educational context for this book.

In data science, mastering a system environment with its tools and processes is essential to achieve minimum productivity. Feeling alien to an environment, using the wrong tools or combining the right tools in the wrong order can lead not only to effectivity limitations but also yield wrong results. Hence, in this book, besides basic computer knowledge and programming skills, students on data science are empowered to assemble a battery of useful tools to employ in the right situation, ranging from small, versatile command-line tools to powerful online platforms. Compared to mastering a single programming language and thus controlling an application logic *in the small*, something that can be fitted into a few functions on the screen, this book advances the skills to *programming in the large*, beyond the boundaries of individual processes or

machines. Programming in the large means defining and orchestrating complex data-centric processes involving multiple tools, platforms and resources. The eventual goal for the reader is thus to be able to define data, model and code that should be provisioned and monitored as services in appropriate distributed infrastructures – from hosting data and models to running software in the cloud. As studying is only the first step towards practical application of skills in a professional setting, this book should therefore be a good starting point for students of data science and computer science, digital life sciences, digital mobility and similar curricula.

1.1 Prerequisites

The reader is expected to bring basic programming skills in an imperative language. This encompasses the definition of variables and control loops as well as the declaration and invocation of functions, including constructor methods to instantiate objects of predefined classes. To keep matters simple, this book assumes knowledge of the Python programming language and occasionally considers the Python interfaces to operating systems functionality. However, readers with knowledge in another language can also find their way around the explanations and exercises.

The reader also needs the ability to fully exploit the keyboard. If typing special characters beyond alphanumeric signs, such as `(@):$^`, poses any challenge, the advice is to train that before reading on.

1.2 Target competences and anticipated skillset

By ingesting the content in this book, including active participation in the repetition parts, the reader can expect to achieve learning goals on the three lower taxonomy levels: knowledge, understanding and application. Moreover, the avid reader is put in the position to decompose larger problems into smaller ones and thereby compose smaller tools into pipelines, workflows and other relationship models to address the larger problem as a whole, either fully or in part.

Eventually, the reader shall be brought into a position to fully control a computer, and even a distributed computer network, and make it work towards the processing of data. Such skills are indispensable to maintain the digital souvereignty on a personal level and in a business context. Instead of having to pay for data hosting and analytics services, it is certainly benefitial to at least maintain the option to do all of this oneself, and moreover, even when using such services, to fully understand what is happening and what could be improved.

1.3 Book structure

The book is structured into the following main parts, reflected as chapters with either a conceptual or practical focus. First, advanced programming concepts such as workflows with pervasive use across operating systems and data science infrastructures are introduced along with data concepts. On a technical-conceptual level, operating systems are then explained in greater detail for their influence on how users operate tools and how tools access data and interact. Next, applications and tools are introduced for use in local and networked environments, with emphasis on their practical use to solve smaller tasks. This chapter is followed by one on middleware as well as one more on collaborative platforms that combine middleware and tools. More complex data science infrastructures that combine cloud facilities with the aforementioned tools, middleware, platforms and workflows are then explained. Finally, non-commercial global platforms that make sense to be simply used at least in the exploration and prototyping stage of data science projects are described.

A glossary is not provided. Nowadays, there is no shortage of online resources to dive deeper into specific terms and topics whenever necessary. And yet, technology terminology is prone to ambiguity at times. Consider the word *key*. In hardware, it refers to a plastic key on the keyboard. In security discussions, it refers to a unique value or a unique sequence of bytes. In programming, it refers to the index that is uniquely associated to a value. All three semantic interpretations are present in the book in multiple occurrences. The book takes some effort to make these distinctions clear, but terms might be used colloquially at times, so that careful and conscious reading is recommended.

Command syntax that can be typed for execution is set in `teletype` font. For technical reasons, longer commands are hardcut on the right column edge.

1.4 Dedication

Operating systems and infrastructures have developed over time with millions of lines of code. None of that would be possible without the countless hours invested by free and open source software developers, community activists, researchers and skilled engineers. There are people behind the software tools introduced in this book, and this becomes apparent when they leave us. Bram Moolenaar, Sven Guckes and Dan Kohn are three shining examples whom the author had the pleasure to meet and to discuss with. And they would certainly like it if more people became proficient and self-determined on the command line to build solutions for all data science challenges.

Chapter 2

Concepts: Programming, Data Representation and DataOps

Programming in the large is conceptually not much different from programming in the small, but the terms and technologies differ, as do the implications of unsuitable algorithms, incorrect code or wrong administrative processes. In this section, a few terms are briefly explained, so that the references to them in tool and platform explanations later become clear: graphs and streams, data pipelines, workflows, shards, event processing, microservices and several others; and eventually, administrative concerns around the term DataOps.

2.1 Data Structures: Graphs, Streams and Units

In programming languages, native data types encompass scalar types (bytes, integers, floating point numbers, boolean values), vector types (lists, tuples, strings, sets, byte arrays), associative types (dictionaries) and geospatial types (dates, coordinates). Data represented in these types are always finite and lack arbitrary references. In practice, more types of data exist. Two important data structures indispensible for expressing data-driven activities are introduced here: graphs and streams. While there are niche languages for them such as Streams Processing Language (SPL), they are handled through libraries in mainstream languages. Their introduction here is therefore given broadly to understand them in every context. The section concludes with a summary view on physical units to express larger quantities in data and resources.

2.1.1 Graphs

Graphs consist of nodes (vertices) and edges – $G = (V, E)$ and are often expressed with graphical notation for human users. The edges can be undirected, unidirectional or bidirectional. As graph-processing algorithms take a predefined or arbitrary starting point and then descend into the graph iteratively or recursively, endless cycles might result from traversing the edges. The most useful graphs are, however, constrained to be cycle-free and (single-)directed, making them elements of the practical subclass of Directed Acyclic Graphs (DAGs). In DAGs, each node may be a splitting node with multiple outgoing edges or a joining node with multiple incoming edges. Moreover, both edges and nodes might be weighted, coloured and otherwise given attributes that are then taken into account in graph-processing applications.

Standard algorithms for graph structures encompass the calculation of the shortest path, or the path with lowest or highest sum of weights, between two nodes and the determination of all nodes reachable from a given node with less than a defined number of hops. Graph rewriting is the process of optimising a given graph structure by eliminating redundancies or introducing controlled parallelism or encapsulated subgraphs.

An example for a graph data structure and its application in the business data domain is the list of stations and the connectivity links between them in General Transit Feed Specification (GTFS), a tabular open format specification for public transport networks.[1] Transitive connections with stopovers can be trivially calculated based on the graph structure. Another example in agriculture is given by the relationships between producers and consumers of manure. Some farms produce a lot and have many associated consumers, with the volume of manure expressed as edge weights, while others may self-consume everything and thus remain isolated nodes instead of being on the graph.

Trees are subsets of graphs with only splitting and no joining nodes, i.e. for each node there can be multiple outgoing edges but only one incoming edge. Nested trees are therefore useful to represent arbitrarily nested hierarchies. Sequences are in turn subsets of graphs without any splitting nodes, and often with implied chronological order. Fig. 2.1 compares these graph structures with representative examples.

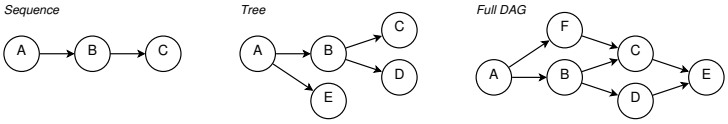

Figure 2.1: Comparison of graph structures

[1] GTFS example: `https://opendata.swiss/de/dataset/timetable-2023-gtfs2020`

2.1.2 Streams

Streams are (potentially) endless data structures that allow for incremental processing. Streams may be represented as characters, lists, trees or other structures, especially those permitting partial representation, which is the case for lists. A stream can be read or written sequentially but not at random locations. Furthermore, while already read data can be buffered to some extent, older data may be completely lost. Hence, in simple stream processing, data records are processed individually or in microbatches. In complex stream processing, operators aggregate information with modest size, which can be kept as opposed to the raw underlying data, which are transient.

2.1.3 Units for Data and Resources

The atomic unit for data is a byte, divided into 8 bits, each assuming the value 0 or 1. Hence, a byte can represent $2^8 = 256$ bits. This is tiny amount of data for today's system, hence more human-friendly prefixes have been introduced for larger amounts of data. Due to historic development, there are both the *Système international d'unités* (SI) prefixes known from physics on the basis of 1000 and the prefixes based on the binary system, more accurately on the basis of 1024. What causes additional confusion is that, in colloquial communication, sometimes one is meant but the other is referred to.

Modern 64-bit processors can hold up data with a volume of up to 8 bytes (64 bits) natively in their registers, corresponding to a `long` value and twice the size of an `integer` value. Anything larger than that, especially arrays such as strings, can only be natively represented in main memory and on disk, and this is where the prefixed units become relevant.

On the SI scale, $10^3 = 1000$ bytes are one kilobyte (KB), $10^6 = 1000000$ bytes are one megabyte (MB), $10^9 = 1000000000$ bytes are one gigabyte (GB), and $10^{12} = 1000000000000$ bytes are one terabyte (TB). The largest consumer-level hard disks nowadays have a capacity of few TBs, making this the largest prefix with common practical use. On the binary scale, $1024^1 = 1024$ bytes are one kibibyte (KiB), $1024^2 = 1048576$ bytes are one mebibyte (MiB), $1024^3 = 1073741824$ bytes are one gibibyte (GiB), and $1024^4 = 1099511627776$ bytes are one tebibyte (TiB). As can be noticed from the numbers, the deviation to the SI units increases with the order of magnitude with approximately 2.4%, 4.9%, 7.3% and 10.0%, which is why the distinction has become more important over time.

Storage resources such as HDDs and SSDs on a computer are typically measured in SI units, whereas main memory is measured on the binary scale. Compute resources are measured by fractions of their capacity, for example, a tenth of a CPU core (or 100 milli-CPUs), over time.

2.2 Data Formats

Within program code, native datatypes and their compositions (for instance, linked lists or graphs) are suitable to represent the structure and content of information. However, as soon as data need to be transferred to other programs or to the outside world, such structures need to be serialised in appropriate formats. The main structures for larger quantities of data are tabular and tree/graph data. Most programming languages have their own specific serialisation formats, such as Pickle in Python, that promise high performance but reduce the ability to exchange such data with software written in different languages. To increase interoperability, standardised textual representations exist for both tabular and tree structures, whereas there are only application-specific formats for most graph data.

Tabular data can be represent in text as CSV (Comma-Separated Values). All rows correspond to a line of text, whereas all columns correspond to tokens per line. These tokens are either unquoted (e.g. `house`) or quoted (`"green house"`), and separated by a comma or by another typical separator sign (`,;|`) or a tabulator (⇄). CSV files may have a header line with column names, but sometimes these names are application-defined and not represented in the data. Hence, overall CSV is a loosely defined format, leading to many interoperability challenges but with good support for line-based stream processing.

XML (eXtensible Markup Language), JSON (JavaScript Object Notation) and YAML (Yet Another Markup Language) are all capable of expressing tree-structured data beyond tables. To represent the hierarchical structures, these formats use different syntax, such as angle brackets (`<>`) in XML, lists and dictionaries in JSON (`[]`, `{}`) and indentation in YAML. Several derived formats exist and are standardised, such as JSON Patch to represent differences between two JSON files again as JSON document. While stream processing exists for these formats, this is not the norm for many applications. Rather, a full file is typically deserialised into memory.

The following listing compares three out of these four structured data formats: CSV, JSON and XML. Typing information is more explicit in JSON but across all formats requires an external schema to avoid heuristic determination.

```
CSV: person;age     JSON: [{"person": "Hans", "age": 22}]
     Hans;21        XML:  <persons><person name="Hans"><age>
     Heidi;22              22</age></person></persons>
```

To inform about the data format of any given document without having to inspect it, media type specifications such as Multipurpose Internet Mail Extensions (MIME) introduced a classification system. Accordingly, the textual representations of tabular and tree-structured data can be expressed as `text/csv`, `text/xml`, and (not fully consistent) `application/json` and `application/yaml`.

Schemas may be defined especially for JSON and XML, unsurprisingly called JSON Schema and XML Schema, respectively. A data schema informs about mandatory and optional fields, their names and data types, permissible value ranges and further constraints. For instance, the regular reporting about COVID-19 case numbers follows a strictly defined format.[2]

The concrete byte-level format of the data depends on its machine representation, which may be subject to further transformations. These include encoding, compression and encryption. Encoding specifies how human-interpretable characters map to byte sequences. In the context of internationalisation, Unicode has emerged as the standard vocabulary with the eight-bit Unicode Transformation Format (UTF-8) being the dominant encoding. Still, a lot of data exists with legacy encodings such as Latin1 (ISO-8859-1). The compression packs similar data segments together to save space. For structured data, typically lossless compression is used, whereas for unstructured data (large corpora of text, images) also lossy compression schemes may be used for even higher savings. Examples of compressed file formats include ZIP (`.zip`), GZip (e.g. `.tar.gz`) and BZip2 (e.g. `.tar.bz2`), each related to custom compression algorithms and corresponding tools (packers, unpackers), with Tar being an intermediate format combining all files in one archive without compression. In data science, encoding and compression are major concerns whereas encryption is only used in specific circumstances, in particular due to the still emerging techniques to compute over encrypted data.

2.3 Compute-Centric Processing: Pipelines and Workflows

A pipeline is a sequential execution of instructions or programs. Pipelines can be represented as purely sequential DAGs, with edges referring to transitions between the individual instructions. The transitions therefore define a temporal and causal order between the instructions. The pipeline commences with input data received by the first instruction. The output of the instruction can, fully or partially, and optionally in conjunction with the original input, be forwarded as input to the next instruction. The output of the last instruction determines the result of the pipeline. Any instruction failure is either ignored or leads to the failure of the entire pipeline.

Fig. 2.2 visualises an exemplary pipeline. Especially for pipelines containing many instructions, the raised abstraction level and uniform interface with input, output and error messages makes pipelines a suitable structure and mechanism for data processing.

[2]COVID-19 schema definition: `https://data.tg.ch/api/datasets/1.0/dfs-ga-1/attachments/variablenbeschreibung_covid19_tg_pdf/`

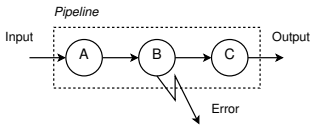

Figure 2.2: Pipeline structure for sequential processing

The instructions can be represented by tools, i.e. OS-level pipelines, or by microservices, i.e. pipelines as orchestrated service sequences. The handover of data (i.e. the edges) can happen as streams or through agreed-upon locations such as files. Moreover, when input and output interfaces do not fully match, the edges can be augmented with additional transformation and conversion nodes. Transformation typically refers to changes within one data format, whereas conversion may refer to changes in data format. Fig. 2.3 visualises an example of such an augmented pipeline.

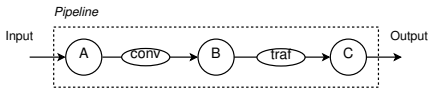

Figure 2.3: Augmented pipeline structure with transformation and conversion nodes

When sequential execution is no longer sufficient, branching and joining can be conducted. This leads to full workflows, essentially directed graphs with a single start-point and a single end-point. Each node of the graph represents one instruction. Depending on the result of each instruction, the branching can be controlled by either selectively, conditionally and exclusively entering one of the branches and leaving out the others or by parallel execution of all subsequent branches.

Textual notations exist for pipelines and workflows, although they are based on convention rather than standardisation and therefore often specific to the runtime environment. Two activities A and B may run sequentially without data handover, with or without activity failure causing pipeline failure, like this: `A && B` and `A; B`. Further variants are execution with data handover (`A | B` or `A >> B`) and execution in parallel (`A && B` or `[A, B]`).

Given that definition, pipelines represent a reduced form of workflows, and hence runtime support for both pipelines and workflows is broadly available from many workflow management systems, whereas basic pipeline support is also baked into operating systems. Each pipeline or workflow can be executed multiple times, with each instance indicating progress by a reference to the current activity.

2.4 Data-Centric Processing: Sharding and Map-Reduce

Pipelines and workflows emphasise the compute activities as first-class citizens, with secondary specification of what happens to the data. In contrast, other processing paradigms put data first and then specify how application logic is applied to it.

In this context, sharding and partitioning are terms that refer to splitting a large amount of homogeneous data records, for instance, lines in a text file or objects serialised as JSON, in a way that the resulting shards or partitions are equally sized or otherwise balanced. In case each shard can then be processed independently, for example, in a counting operation, the processing can be trivially parallelised. This speeds up the computation significantly, either to the number of available cores on a machine or in the case of distributed parallelisation even to all cores across a number of attached machines.

Sometimes, complete parallelisation is not possible, for example, in the operation to find the maximum of a large list of numbers. In that case, the algorithm is adjusted to parallelise as much as possible – for instance, finding the local maximum in each shard – and then in a second step to find the global maximum by working across the intermediate results from the first step.

The map-reduce programming paradigm combines both steps. First, functions are applied (mapped) to individual data records within a shard, yielding equally sized result shards, except for filter functions, which can also yield smaller result shards. Then, the resulting shards are combined in a reduce step. The following example shows map-reduce processing over a list of text lines in Python, by first converting all text to uppercase, and then counting and summing the occurrences of the letter A. It is apparent that there are no instructions to parallelise. Whether or not each map activity runs in parallel is decided by the implementation, whereas the programming can focus on working with the data.

```
import functools
text = ["Heavy snow", ...]
result = map(str.upper, text)
result = map(lambda line: line.count("A"), result)
result = functools.reduce(lambda x, y: x + y, result)
print(result)
```

2.5 Event Processing, Handlers and Hooks

There are multiple ways to start the execution of an instruction or a complex program. The first one is the conscious interactive invocation, for instance, by

the data scientist during development or by the user in production. The second way is programmatic invocation. An application may call another application to delegate processing tasks. The third way is time-triggered execution. A specific time pattern such as 'at the full hour' or 'every 10 minutes' is given and translated into action by a time trigger system such as an implementation of Cron[3]. Time triggers are predictable and can be anticipated. The fourth way is an unpredictable event trigger. A program is to be executed whenever something happens, such as changes in a data structure. The advantage is that, if the event rarely occurs, there is no unnecessary invocation and therefore no overhead cost. Within the processing logic, it is often less relevant what exactly was the trigger compared to what context information is available, i.e. how the trigger was parameterised.

Handlers and hooks provide the glue between source events and program invocation. For instance, data modification can be monitored over a long time, and whenever a modification is detected by a helper program, a specified program is invoked. This also works over the network, where appropriate network messages can be sent upon detection of source events, in the form of network hooks or web hooks. Hence, these handlers perform a similar role as transformers and converters in workflows.

The term handler is furthermore used for handling internal events in a program. This may be an exception, an interruption or some other signal from the environment. Operating systems provide handler support on an OS programming level, whereas workflow management systems also translate these to higher-level handlers to specify what should happen to a pipeline or workflow in the event of a fault or signal.

2.6 Encapsulation: Functions, Tools, Containers and Services

Encapsulation refers to the separation of the interface from the implementation. It is an important concept to hide implementation complexity by raising the abstraction level. Pipelines and workflows have already been introduced as abstraction layers and encapsulation mechanisms. This section takes a broader look at encapsulation.

Within a program, application logic might be encapsulated as a function with a well-defined signature, consisting of the function name, its mandatory and optional parameters, and its return values. Beyond functions, logic can be encapsulated as a class with multiple functions (methods) or as a module or library potentially encompassing multiple classes. The limitation to all of

[3]Cron syntax: `http://www.quartz-scheduler.org/documentation/quartz-2.3.0/tutorials/crontrigger.html`

these encapsulations is that they are only accessible within the application, demanding all programming be done with the same programming language.

It is possible to go beyond this limitation. The program itself then represents another form of encapsulation with the ability to parameterise the execution, capture intermediate information about its activity and eventually determine the success of the execution based on mostly self-declared information and conventions. The level of encapsulation is stronger if idempotency is guaranteed, such that multiple repeated invocations do not produce results different from a single invocation. This is typically achieved through statelessness, a property helpful also for the automated parallelisation. For data-centric programs, statelessness results from read-only operations that do not modify data, for instance, searching, whereas modifications lead to statefulness and then only rarely combine with idempotency. For instance, updating an entry to the current invocation time is not an idempotent operation, whereas nulling a fixed field is.

Smaller programs with well-defined input and output interfaces are conventionally called *tools* on the OS level. They might read data from files and over the network, as well as on the command line and through interactive input, and they might return results via files, network and standard output, in addition to an execution status code or exit code. Their invocation context encompasses explicitly given parameters in addition to configuration taken from configuration files and environment variables. More complex application functionality can be encapsulated as containers, with similar parameterisation.

Another form of encapsulation is network invocation of these programs via a well-defined service interface, which abstracts from implementation details such as the programming language. Such services have the advantage of being deployable and remote-invokable across machines. Often, the services take the form of microservices with specific forms of packaging and lifecycle management. Either they run continuously and directly accept requests, or a proxy takes requests on their behalf and invokes them only on demand. This latter form has become popular under the name cloud functions, conveying the conceptual similarity to programming-in-the-small function encapsulation.

2.7 Data Management, Engineering and Operations

Concepts such as data formats, workflows and event processing are generic and not necessarily bound to the activities of a data scientist. In this section, these concepts will be connected and explained from a data-centric angle. They encompass data management, engineering and integration, machine learning, operations (coined DataOps) and reproducibility.

2.7.1 Data Engineering

Data processing requires two main ingredients – input data and software that performs the processing. Building software in the form of applications and services, but also curating the input data, often starts with a repository managed by a version control system (VCS) to manage all ingredients and to track their evolution. Additionally, data may be managed in a relational or non-relational database (DB), in storage services, or in a data warehouse. To reduce the effort, when maintaining both modest amounts of schemaless or schema-flexible data and software code, a VCS with file-based data representation is a pragmatic choice.

A notable feature of VCS and DBs are hooks triggered whenever the repository or database contents change. For instance, whenever a VCS-managed file is modified, a user-defined script is executed that validates the repository's data files or submits the updated files to some online service. Similarly, User-Defined Functions (UDFs) may execute within a DB upon the insertion of records. Making use of such triggers for further processing leads to continuous integration processes, with reference to data integration explained in the next paragraph, or continuous deployment of data on a provisioning system, collectively referred to as CI/CD.

The processing itself can then take various forms. Apart from integration or fusion, data might be aggregated, augmented, analysed, filtered or used as basis for decision-making. In recent years, machine learning (ML) has gained significance, with large volumes of complete data used to train a mathematical model used for inferring characteristics of incomplete data records. For instance, a regression test on given X/Y coordinates may infer the associated Z coordinate by interpolating from known Z values of nearby X/Y pairs. Many ML algorithms work on vectors or tensors and benefit from specialised hardware to support concurrent vector processing. ML algorithms can be used to predict such missing values, but also to categorise objects described by data, and to recombine information, always based on statistics and heuristics with an imperfect accuracy. Recombination works on large language and media models (LLM, LMM) and is able to produce text and multimedia content with defined characteristics, although often not perfect due to the mentioned heuristics. Especially when generating results for human consumption, these techniques are also referred to as Artificial Intelligence (AI).

Moreover, data engineering is concerned with the right design of data-processing software, in particular with its non-functional runtime characteristics such as scalability, performance, reliability and cost. These characteristics take effect when the software is operated, especially under the DataOps paradigm. The design also involves the decomposition of software functionality into internal structures such as microservices and workflows and the preparation of automated deployment of the corresponding management systems.

2.7.2 Data Integration

Data integration refers to the ability to load and interpret data from any source and in any format. It requires a unification of formats through transformation and conversion. Transformation modifies structured data within one homogeneous format, for instance, by adding, removing or renaming fields. Conversion refers to the change between heterogeneous formats, for instance, from JSON to XML. Assuming there are N source formats and M target formats, this would require the implementation of $N \times M$ converters. Instead, the conversion can be conducted to and from a meta-format at the expense of slightly slower processing due to two conversion steps. The benefit of such an approach is that at a maximum, $N + M$ converters need to be implemented and maintained.

2.7.3 DataOps

When one combines processing paradigms with input data in various formats and activities in various encapsulations, including storage, transmissions and triggers across machine boundaries, the question arises whether this sort of programming in the large can be summarised under one term. There are several contenders, but, for pragmatic reasons, the term DataOps is used here to conclude the first concept chapter.

DataOps is not a rigorously well-defined term. Rather, based on the industry-invented term DevOps and related to GitOps and MLOps, it describes a rather wide set of activites around providing code and data as managed services to other users or applications, primarily based on network interfaces and microservice encapsulations, based on CI/CD. This implies attending to repository design, powerful data processing pipelines development and deployment, integration and inference, sufficient resource allocation, fair scheduling, protection, monitoring, cleanups and troubleshooting to provide flawless services. From the data science engineer, it requires a fundamental understanding of concepts and tools at the operating system level but also concerning data engineering and data science platforms and distributed infrastructure. The engineer needs to understand the contribution of any technology to the resulting data product, including business-affecting terms such as technical debt and dependency risks. Moreover, the engineer needs to be able to resolve emergent, sporadically occurring unexpected problems, apart from investing time and effort into longer-term improvements that may be requested by users of a data-centric application. Hence, from a data product engineering perspective, it may be more cost-effective and sustainable in the long term to build on proven portable technology stacks rather than the latest offerings, which may not exist anymore or become less economical within few years time. For the data scientist to be productive in the operations domain, there are also

certain requirements on powerful zero-configuration management tools, dashboards and actionable advice when problems occur. These requirements are only partially fulfilled by today's software, even when taking the latest data warehouse or data lake offerings into account, leading to even more emphasis on being able to master basic tools and foundational processes in order to build custom DataOps solutions in any business context.

2.7.4 Reproducibility

Data management requires a thorough documentation of technical and legal aspects concerning the origin of data, any modifications performed and long-term archiving. These processes are often subject to regulatory constraints, for instance, related to data protection. Yet having good documentation, especially an automatically generated one, also helps identifying improvements and regressions in data quality and data-driven implementation and process quality. A number of terms relate to the ability to reproduce results from the same input data. The goal is always 100% reproducibility, implying that the results do not change when there is also no change in the input data. The following four terms are specifically of relevance in this context:

1. Lineage. Data lineage is the process of maintaining the journey of data from its originating sources to its ultimate destination. It refers to the automation and exact documentation of all transformations and conversions on that journey.

2. Provenance. Data provenance is the historical tracing of data from its originating source to its final stage. Emphasis is on the source, i.e. is the data source credible, legal, of sufficient quality and so forth.

3. Reproducibility. In general, this represents the degree of agreement of a measurement by different individuals, locations and instruments. Specifically related to data science, it specifies the equality of results of data processing and analytics no matter what code or environment is used.

4. Repeatability. This denotes the variability of a measurement or of a pipeline or workflow across iterations under the same conditions. Typically, measurements are conducted multiple times and evaluated statistically. If the arithmetic mean average, median and spread are not far apart, the data processing is repeatable due to producing comparable results in repetitive iterations.

In data science practice, all four terms are important, and all require specialised tools and platforms to assist the data scientist.

Repetition

All repetition solutions can be found at the end of the book.

1. What relations may exist between two tasks of a workflow?
2. Why is the map function a suitable primitive for highly scalable applications?
3. A VC-funded startup offers analytics as a service. Should one consider a subscription?

Chapter 3

Concepts: Operating Systems

This section conveys background knowledge on the inner workings of a computer system, with emphasis on contemporary operating systems. The intention is not to give a formal education on all aspects, but rather to provide a practical-oriented jumping board to understanding any later interaction with the system. Most topics are touched on only briefly and require additional literature for full understanding. The chosen method is to touch on topics of further necessity in this book, and to give a technological foundation to the programming concepts presented beforehand. This section presents fundamentals including the boot process, current operating systems, technical building blocks such as process and resource management, isolation concepts, file system interaction and networking, and user management.

3.1 Fundamentals

An operating system (OS) is a complex piece of software that runs all the time on a given hardware to manage that hardware and facilitate the concurrent execution of multiple further software applications. Hardware can relate to interactive work devices such as notebooks, personal computers and mobile phones, but also to servers, as well as embedded applicances (such as programmable machines) and virtualised hardware. On a typical data scientist workstation, the core computer hardware could, for instance, be a set of n processors (central processing unit – CPU), an accelerator (graphics processing unit – GPU), x GiB of main memory, y GiB of storage on a local SSD, a wireless network adapter and a 10G wired network connection, along with a mainboard to tie these components together and peripherals for human input and output. The peripherals may encompass a monitor, keyboard and mouse, printer and scanner as well as loudspeakers. In an application scenario, a hypothetic dataset

might be first read from the SSD. The application software processes it according to program logic as a set of instructions similarly read from the same SSD, and the results are written to main memory. The OS must coordinate this workflow including permissions and auxiliary program execution. Hence, the OS performs tasks such as hardware initialisation, coordinated access to hardware resources, memory and file management, scheduling of tasks that are part of such workflows, permission checks and usage accounting, among many others. In terms of complexity, operating systems of practical relevance encompass millions of lines of source code.

Operating systems are activated by a *boot process*, taking over from the hardware's hard-wired internal initialisation routines such as Basic Input/Output System (BIOS) or Unified Extensible Firmware Interface (UEFI) that are provided with the mainboard. They then activate the peripheral hardware, load initialisation code from the storage media such as disks, start system-wide background processes and prepare the system for human and automated usage. The usage is facilitated by offering local and networked interfaces to log in and run commands, primarily in the form of textual or graphical login screens or remote login services, again either textual or graphical. Nowadays, the boot process in both physical and virtual machines is often complicated by the presence of mandatory Trusted Computing Modules (TPMs).

The hardware initialisation depends on the degree of standardisation. Some hardware models are easy to integrate from the OS perspective, whereas most others ship with their own firmware requirements that need the firmware to be uploaded to the hardware first. This is often the case with wireless networking adapters, without which a network-based installation or maintenance becomes impossible. Hardware unable to work without firmware updates is another cause of complicated boot processes. Modern operating systems cover a broad range of hardware but may lack firmware or driver support for exotic models or very new models.

The entire boot process usually takes a few seconds. To avoid costly reboots, hardware can also be suspended or put into standby when it is not in use, for instance, by instructing the OS to do so or by provoking that with predefined actions such as closing the lid of a notebook. Correspondingly, hardware can be woken up again either by user interaction (opening the notebook lid, pressing a key, touching the screen) or by a network signal (wake-on-LAN). However, most system suspensions still consume a certain amount of standby power, which has not only the implication that more electricity needs to be generated, but also that the suspension does not survive a longer power cut.

On a new computer without pre-installed OS on permanent storage, apart from netboot in a prepared environment, the OS can only be loaded from removable boot media such as USB drives or CD/DVD, which would then provide an installation option. Alternatively, in case an OS is already installed, some

systems provide installers running as applications on other systems, which allows for complementing or replacing an existing OS. For that matter, the disks of a computer can be partitioned. An OS requires at least one such partition for booting, containing the boot loader initialisation code in the first sector. An OS can, however, also span across multiple partitions to increase the storage capacity.

3.2 Current Operating Systems

The scope of operating systems has changed considerably over the decades. Today, despite an unprecedented variety of available operating systems (to get an idea, occasionally read up on OS-related news websites[1]), only very few have practical and commercial relevance in the market and satisfactory support for applications, especially those that integrate, process and visualise data. Some systems like Minix are suitable only from a didactic perspective to understand the inner structures without broadly supporting today's hardware and applications ecosystem. Others like the Robotic Operating System (ROS) refer to industry-specific middleware on top of an existing OS kernel.

The three dominating contemporary operating systems for data science interaction and DataOps, both vendor-neutral and vendor-specific offerings, are those that are used on workstations. They thus define the immediate environment for the data scientist, allowing for rapid interaction between programming in the large and verifying the results of program execution:

1. Linux[2], with several different representations including flavours of GNU/Linux (e.g. Debian, Ubuntu, CentOS), and similar open systems with similar userland such as various Berkeley Software Distribution (BSD) flavours;
2. Apple Mac OS X, similar to Linux, being of the Unix family of systems; and
3. Microsoft Windows.

From a consumer perspective, these operating systems are often known from mobile devices such as the Linux-derived Android or the OS X-derived iOS. However, apart from mobile data acquisition and visualisation, such devices, including mobile phones and smart watches, do not play a major role in defining infrastructure for application programming and data processing.

Linux, the BSDs and Mac OS X both follow a similar technical interface design as the original Unix systems, and are therefore also referred to as Unix-like systems. Unix systems date back to the 1970s, and some elements of that

[1] OSNews: `https://www.osnews.com/`
[2] Linux Kernel: `https://kernel.org/`

era are still reflected in modern systems, including conventions on timestamps and programming languages. Each operating system is nevertheless characterised not only by the actual up-to-date OS kernel but also by the supported subsystems such as file systems, device drivers, input modalities and finally the supported and still growing ecosystem in terms of libraries and applications. The collaboration models facilitated by the Internet have led to large global communities contributing to the software development from the OS level to the applications. As the ecosystem is therefore neither static nor centrally coordinated, new software is constantly being published, and existing software might need to be updated to fix stability and security issues. Consequently, concepts of trust and reputation regarding software found online are important. To reduce the setup effort, curated distributions, package managers, software stores and similar consumer interfaces to the ecosystem, along with corresponding producer interfaces, exist and are central for an effective usage for setting up data science infrastructures. There are OS-specific and cross-OS (or cross-platform) applications and libraries within each ecosystem, with further differences in the hardware architecture. Fig. 3.1 conveys typical software-architectural layers found in current operating systems, running within a privileged OS scope and an unprivileged scope defined by the respective OS distribution and managed by the OS itself at runtime.

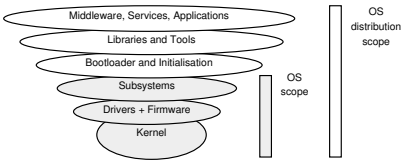

Figure 3.1: Typical layered system architecture

For reasons of practicality related to condensed explanation of commands, and reflection of the high pace of the ecosystem, although the operating systems theory are presented in abstract form, concrete usage examples in most cases assume a running Linux system. Students running different operating systems are either able to relate the commands to equivalents on their systems or use virtualisation or network access to get their hands on a native Linux environment.

3.3 Building Blocks: Executables, Processes and Resource Management

Most users know computers in terms of applications. For instance, a data scientist may create a Jupyter notebook (detailed later) and use the Jupyter

application for that purpose. The visible part is the notebook shown on the screen, with cells to input code and other cells to display results; but of interest from an OS perspective is the underlying structure. They determine how the Jupyter notebook got found, started and delivered to the user.

Applications within the ecosystem of an operating system are shipped in the form of bundles or *packages* consisting of files and, sometimes, online services. The central parts of any application are called *executables*, referring to single files that can be executed on a CPU under the control of the operating system. They contain processor-specific binary code in an umbrella file format understood by the OS, such as *ELF* or *EXE*, whose headers give information on how to load and execute the file. The production of such executables is the task of a compiler that takes a human-readable language, either Assembler or a higher-level language such as C, C++ or Rust, and optimises the set of instructions for the processor.

Executables can be statically or dynamically linked to essential library code. Such *libraries* extend the functionality of executables with re-usable program logic, including functions to perform basic interaction with the operating system. Based on the historic dominance of the C programming language in operating systems, the main library is the standard C library, or `libc`.

In the case of dynamic linking, especially for highly dynamic *plugins*, the code is resolved upon execution by the OS *loader*. The advantage of dynamic linking is that the code size of the executables themselves can be kept modest, at the disadvantage of increased complexity during loading. In practice, the executable does not run unless all library dependencies are properly resolved, including the determined versions.

If the application is authored in a scripting language, such as Python, then, from an OS perspective the application itself is just data, and the Python interpreter is the actual executable. To make matters complicated, there are multiple Python interpreters available, such as the classic cPython, iPython and PyPy. Multiple applications can also run at the same time with coupling between them through communication. The Jupyter notebook environment would be one such case, which is itself developed in Python, typically executing on cPython and running an iPython interpreter instance to execute the content of its cells.

Before an executable can run, the OS also needs to prepare its internal structures. It allocates a new *process* that refers to the binary code in the executable but also contains management information such as process identifier, parent process identifier, current working directory, priority, owner, access privileges, resource limits and statistical information about the ongoing execution. In practice, a typical data scientist device runs around 300–500 processes concurrently, many of them referring to invisible background activities. To give a more reasonable example, the aforementioned execution of Python code in a

Jupyter notebook is explained in Fig. 3.2. It highlights the uniquely separated aspects of processes such as identifiers, owners and memory areas.

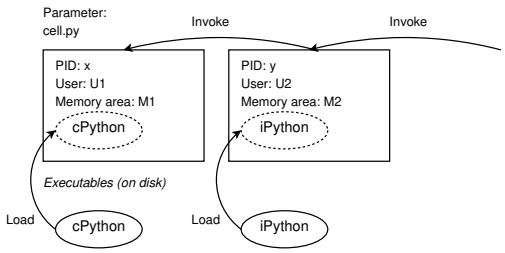

Figure 3.2: Processes, executables and data in an operating system context

To run correctly, a process needs a certain amount of resources. This refers primarily to CPU and main memory, but depending on the program logic may also refer to disk space and network throughput.

By default, processes can use all virtual memory made available by the operating system. This value often exceeds the available physical memory due to concurrent process execution. Hence, processes may enter into situations where no more main memory is available. When other forms of memory (e.g. swap space) are also exhausted, the OS decides on the termination of the offending process or another process depending on the memory management policy. Eventually, one process must be terminated with an out-of-memory (OOM) cause. Then, more memory becomes available again, although not necessarily in sufficient quantity to fully accomodate the original request. Unless it runs into such critical situations of resource exhaustion, each process nevertheless requires different amounts of memory over time. The amount is hard to predict, although often similar if the process is run with comparable parameters. One task of the operating system is to shuffle with memory pages behind the scenes, reallocating them as needed to the active processes.

Processes can also terminate irregularly for reasons other than out-of-memory conditions. The primary reason may be an internal programming mistake such as division by zero or access to a file that does not exist. Programming languages allow for capturing such mistakes as exceptions, but if that was forgotten, the process is terminated by the operating system.

Processes run with configurable priorities. If a batch process, such as data cleansing, runs quietly in the background and does not disturb interactive work, it can be explicitly de-prioritised. On the other end of the spectrum are processes that require a minimum allocation of CPU slices, for instance, to be able to perform guaranteed real-time processing of a large and fast data stream. Most operating systems have no support for hard real time processing and instead work on a best-effort basis.

At any point in time, an OS thus runs a number of processes, from which more processes may be spawned. Each process possesses a priority and an associated memory allocation apart from the other mentioned characteristics. All processes form a tree, with the initialisation process as its root. This process is typically instantiated from a special application program that gets a list of other processes to start at system boot and supervises them to be able to conduct restart in the case of crashes. On modern Linux systems, this task is performed by OpenRC or SystemD.

3.4 Isolation, Virtualisation and Containerisation

As outlined in the programming concepts, encapsulation is an important principle to raise the abstraction level and shift from programming in the small to programming in the large. In operating systems, the main encapsulation techniques relate to process-level isolation, virtualisation and containerisation.

Processes that run concurrently have a unique identifier within a single namespace and are protected from mutual access to main memory. Only the owner of a process, or a privileged superuser can change process metadata such as its priority. The system may also enforce a certain fairness between users by setting quotas that effect the number of processes that can run, the amoung of main memory allocatable by each process, the number of open files per process, the cumulative CPU time and similar metrics. However, in principle, the separated processes can see each other and might volitionally or inadvertantly share information through the file system or other less protected channels, including non-intuitive covert channels. In order to prevent information sharing and interference, specifically for shared user environments, different *isolation* concepts beyond regular OS process isolation have emerged.

A strong level of isolation can be achieved by *virtualisation*, i.e. running an application that represents a guest operating system including kernel and is thus able to execute applications within itself, without visibility of its internals to the underlying host operating system. The guest OS can be a different version of the host OS or even an entirely different OS, for instance, when running Windows atop Linux. In principle, virtualisation works even without specific support in the host OS but is then often too slow. Modern OSes thus support concepts such as para-virtualisation and hypervisors that provide near-native execution speed in conjunction with specific CPU instructions. Depending on the device configuration, virtualisation can even be nested, so that complex applications can be built in a portable way. The management of guest OSes is the responsibility of a *hypervisor*, which is usually built into host OSes nowadays and apart from the runtime functionality offers ways to

configure virtual machines (VMs) as well as manage VM images that contain the root filesystem of the guest OS. Typical examples for native hypervisors are KVM on Linux and HyperV on Windows. There are also third-party hypervisors such as VirtualBox for all maintream operating systems.

A weaker but in practice often sufficient level of isolation is provided by *containerisation*. In this model, the OS kernel is fully shared and the process namespaces are decoupled selectively. For instance, a guest process may still share the networking with the host but not the process table or the file system. In contrast to most virtualisation approaches, containerisation is more optimised for software development environments and thus also suitable for running most software of relevance to data scientists. The first widely used common cross-platform approach for containerisation was *Docker*, providing management capabilities for container images and runtime aspects. Tools like `Podman` make it easy to work with such container images.

Virtualisation and containerisation can be combined with networking (explained below) to achieve flexible distributed computing. In this model, each operating environment should indicate its name, as otherwise the concurrent work across several systems quickly becomes confusing for the user.

3.5 File System, Paths and File Access

Files are sequences of bytes – either representing human readable characters (text files, e.g. Python scripts or structured data) or arbitrary bytes (binary files, e.g. executables or unstructured multimedia data). The native representation of data could be modified in a file context, as outlined in the explanation about data formats. For instance, text data might be compressed to use less space and encrypted to be more confidential, resulting most likely in a binary-encoded file.

For permanent access, such files are stored on storage devices (e.g. SSD) and are written and read sequentially in both text and binary mode or with random position access in binary mode. Alternatively, they are memory-mapped for direct access, especially larger data files for which a sequential access might be too much of a bottleneck. The task of the OS, more specifically of the OS-supported file systems on any storage media, is then to organise all files, arranging them, allocating the physical storage locations, and regulating as well as journalising access.

File systems therefore arrange files in suitable structures, most of which are hierarchical or tree-structured. They emerged as integral parts of operating systems, and most optimised file systems remain OS-specific (e.g. ZFS, ext4), although many de-facto standards such as the exFAT file systems have emerged along with portable media such as USB drives. In general, file systems require a (physical) storage medium such as a hard disk, solid-state drive

or the mentioned USB drives; or alternatively, a partition on that medium. Multiple partitions can be integrated as additional file systems located as subdirectories or drives depending on the OS type. At least one partition is marked as bootable and contains the OS files along with the bootloader, reachable from the medium's master boot record. Being the central interface between humans and data management on the OS level, file systems are designed to balance user needs in terms of structuring data with operational concerns such as file creation and search speed as well as resilience.

The user perspective on a file system typically encompasses a directory tree view. Directories with arbitrarily nested subdirectories form tree structures that, along with file contents and metadata, assist users in data management. The top-most directory is called root directory (i.e. `/`) or drive (e.g. `C:\`), depending on the OS. The concatenation of directories, and optionally a file, is called a *path*. Paths can be absolute, starting from the top-most directory, or relative to the current working directory of a process. For instance, `/etc/passwd` is an absolute file path, and `..` is a relative directory path, referring to the root directory `/` when applied to the directory `/etc`. In addition to data and executable files as well as subdirectories, and depending on the file system and operating system, directories may also contain symbolic links (symlinks), device files, fifos and other special files.

Standard top-level directories on Unix-like systems encompass three hierarchies: `/`, `/usr` and `/usr/local`, with the semantics that larger partitions could be mounted later in the boot process and only `/` must unconditionally exist for booting and system repair activities. Under each of these hierarchies, certain second-level directories may exist. These include `bin` and `sbin` for unprivileged and privileged binaries, and `lib` for shared libraries and other architecture-dependent files. Further directories only exist in the top-level hierarchy, including `etc` for configuration, `home` for user accounts, `tmp` for temporary files, `opt` for optionally installed large applications, `root` for the super-user account, `mnt` for mounted filesystems, `srv` for services and `var` for generated state information. The `share` directory for architecture-independent files, effectively all data files read by applications, only exists within the `/usr` and `/usr/local` hierarchies, presumably because the core utilities under the root hierarchy work as executables without references to data files. These directories are further subdivided into purpose and application Hence, a typical application would consume data from `/usr/share/<app>`, write temporary data into `/tmp/<app>[/<user>]`, cache results into `/var/cache/<app>`, and persist precious data into `/var/lib/<app>`. Some directories contain OS-related support files and special files (`boot`, `dev`, `proc`, `sys`). These names and conventions have evolved over time and are now largely standardised due to the Filesystem Hierarchy Standard (FHS).

As outlined above, files are characterised by their contents, either textual or binary, as well as their metadata. While content is always defined by the

user or applications on the user's behalf, some of the metadata is automatically maintained by the file system. Typical text formats encompass unstructured plain text in various encodings, but primarily Unicode (i.e. UTF-8) as well as structured plain text to represent data such as CSV, XML, YAML and JSON formats. Such files are interpreted as sequence of lines, where each line is separated from another by an end-of-line symbol, typically either the newline symbol (`\n`) and/or the character return symbol (`\r`). Binary files, on the other hand, have application-specific structures that render them unfit for human reading. Many data science tasks require working with structured data formats and understanding their content. This implies dealing with potential quality issues, including misformatted files that would disturb the process of reading or *parsing* but also silent errors that would cause issues after the parsing.

Files are also characterised by their metadata. These include auditable timestamps of creation, modification and last access but also size, ownership and access permissions. The file size may not necessarily correspond to their content, for sparse files may be created conveying a large file size despite little content and correspondingly small storage space requirements.

From a programming perspective, files are opened in a certain access mode, typically for reading, (over)writing or appending. The OS checks the eligibility of the file open request and either signals success by returning a file descriptor or signals an error through appropriate OS-specific error codes. Typical errors include 'file not found' and 'permission denied'. Subsequent read and write operations may similarly result in errors such as 'no space left on device' or – specifically for reading – in application-generated parser errors. At the end of a file operation, the file is closed again and the OS-internal control structures are released.

A trivial way to read files is to read them line by line (for text files) or block-wise (for binary files). In order to cope with very large files and to avoid the input/output bottleneck, modern OSes allow the virtual mapping of files into memory, so-called mem-mapped files, on the assumption of sufficient main memory. Any modification in main memory then results in an optimised write access on the storage medium.

3.6 Networking

Network access across the boundaries of single computers is useful for a number of reasons. It allows fetching data from all over the world, along with code in the context of system maintenance, to backup data remotely and to let users collaborate on data science projects. In conjunction with file systems, networking also allows for remote file management through networked file systems.

On a practical level, networking is considered to consist of four layers. On the lowest level, a physical link between two computers is established using either wired or wireless communication media, such that a computer becomes reachable from another one by a network address. This link may be direct, but it might also involve a number of intermediate machines, physically represented as a sequence of links. The physical link determines the key networking characteristics: throughput as in data volume per time unit, latency as in time unit that needs to pass before an arbitrarily small message has reached the destination and stability, which in the case of wireless links is affected by the signal quality among other factors.

On the next level atop the physical link, a packet communication is established, dominated by the Internet Protocol (IP) for data transfers as well as several network management protocols. Each IP packet consists of a header and a payload of fixed maximum size, typically around 1500 bytes but in principle up to 64 kiB. The IP abstracts the underlying physical linkage away so that the endpoint computers would always assume a single logical link in between them. Endpoints are specified as a combination of IP address (IPv4 or IPv6) and a 16-bit port number, with all numbers below 1024 being in the reserved range. Services can occupy one or multiple of those ports, and clients specify them when attempting to communicate with a service. The `/etc/services` file on Unix-like systems or `C:\Windows\System32\drivers\etc\services` on Windows maps relevant port numbers to human-interpretable names, in most cases protocol acronyms, as defined by the Internet Assigned Numbers Authority (IANA).

On top of IP, encapsulated in its payload section, data transfer with session or stream semantics across several individual packets is the task of more specialised protocols such as the User Datagram Protocol (UDP) for low-latency transmission, Transmission Control Protocol (TCP) for reliable transfers and Stream Control Transmission Protocol (SCTP) with characteristics from both, each of which again consists of headers and payloads. To secure connections, especially TCP connections can be upgraded with Transport-Layer Security (TLS), allowing for mutual certificate checks to verify identities and set up encrypted transfers.

On the highest layer, application-specific protocols define the content of the payloads and the sequence of transmissions as well as the details of addressing. Many applications assume a client-server topology where a server listens for packets on a specific numeric port, often chosen from a fixed assignment such as port 80 for web applications using the Hypertext Transport Protocol (HTTP), but in many cases with flexibility especially in the non-reserved range of ports. A client then connects to that port, establishing on its side a sending port with a random port number. Either side may initiate the conversation and may terminate it. Similar to files, networking protocols might be text-based (e.g.

HTTP, SMTP, XMPP) or binary (e.g. SSH), and data representation on the network may be subject to modification such as compression and encryption.

In IP networks, the IP address is the native address, although memorisable alias names can be given. This includes the computer's own internal name, independent of the network configuration, as well as network-reachable names including the computer's own external name. Unless the configuration is local, all network-reachable names can be retrieved from the Domain Name System (DNS), which is one of the standard services supporting network administration and navigation. Local configuration can reside in the file `/etc/hosts` on Unix-like systems, mapping IP addresses to local hostnames per line, e.g. `192.168.0.5 mynas`. On Windows, this file is called `C:\Windows\System32\drivers\etc\hosts`.

From an OS perspective, *sockets* are used as internal memory structure to represent active network connections from clients to servers and listening connections on servers. Each port number may only be in use once per machine. Therefore in conjunction with virtualisation and containerisation, virtual networks are introduced to route requests, for instance, to port 80, to different physical ports of the respective virtual machines or containers.

Sockets together with files and named or unnamed (pipe) FIFOs permit narrow-band inter-process communication (IPC). For sharing access to large data volumes efficiently between local processes, shared memory areas may be defined, whereas for communication across nodes, sockets are the only option.

An application offering functionality over a network port is typically called a *service*. For managing and providing such services at scale, several *service platforms* and *cloud platforms* have emerged, with some also offered on a commercial basis. Similar to OSes, many different platforms are available. Global-scale providers such as Google Cloud Platform, Microsoft Azure, Amazon Web Services, IBM Cloud or Alibaba are among the well-known ones with the richest portfolio of platform functionality, although there are also many local providers offering at least basic services and service management over the network.

3.7 User Management, Authentication, Authorisation and Credentials

The concept of users, roles and identities in a complex, shared system environment, especially in a networked environment, is an important one. Once the identity of a user or application is known, permissions can be linked to it and codified into the file system and other OS structures. The key question is then who is allowed to access or modify which files under what circumstances. On the OS level, system users are registered along with a home location in the

filesystem and credentials, typically in the form of passwords, and appropriate privileges, typically in the form of being in a group that has wider access permissions. Specifically, super users or root users have all possible permissions, and users are able to obtain these permissions temporarily. Hence, each OS process has a user identification and a (short-term) effective user identification that governs the permitted actions such as access to files and devices. On a Unix-like system, the super user's home directory is `/root`, whereas all other users typically reside in a user-specific directory such as `/home/user`. On Windows, the equivalent path is `C:\Users\user`.

Obtaining the permissions is called *authentication*, whereas making use of the permissions for access purposes is called *authorisation*.

To enforce authorisation across computer boundaries over a network, *credentials* such as passwords, keys or tokens are registered in services and supplied from clients during protocol-specific remote authentication. For increased confidentiality and to protect against leaks, the credentials are often not stored in plain but instead in hashed format. Moreover, they are also encrypted during transmission, based on timestamps as part of a session-specific encryption.

Repetition

1. What happens to a process when it requests additional memory pages but all main memory has already been used up?
2. Does containerisation provide the maximum possible isolation level between processes?
3. What happens when a web browser is used to connect to the website at `http://my-little-website`?

Chapter 4

Concepts: Infrastructure

The section informs about important concepts and goals related to digital infrastructure in the wider senses of data science. Therefore, it covers computing infrastructure, data infrastructure and similar forms, all of which are built atop operating systems and made available in various appearances such as compute clusters, online services and integrated platforms.

Similar to civil infrastructure such as roads and bridges or energy infrastructure such as power lines and gas tanks, digital infrastructure is a necessity with high demands on reliability, security, performance, scalability and cost effectiveness. Digital infrastructure relates to computing resources for computation, communication and storage. Sometimes, the handling of these resources is tightly combined. For the purpose of introducing typical infrastructures, the following types are covered here: networks and Internet, networked computers, services and platforms, high-performance computing and cloud computing.

4.1 Networks and Internet

Access to computer networks was already explained in the networking section of the operating systems concepts chapter from the perspective of a single system. Here, a bird's view on entire networks is given. Computer networks can be represented as graphs, with each system being a node and the available connections being edges. Not all edges need to be available or activated at all times, and sometimes more than one connection (e.g. wired and wireless) exists between nodes, making the nodes multi-homed due to being in different networks with different IP addresses at the same time. Hence, even the basic topology of a computer network is dynamic. It is also heterogeneous due to different characteristics of the connections related to bandwidth, latency and connection quality. Wired connections typically show predictable bandwidth

despite relying on slightly heuristic protocols, whereas wireless connections vary wildly. More dynamic behaviour is introduced by traffic flows, caused by the communication between nodes. Such traffic can lead to congestions on the network or overload of the attached systems. Unreliable connections can also lead to a loss of data. Overall, computer networks must be assumed to be dynamic, heterogeneous and imperfect. Fig. 4.1 summarises these characteristics in a topology in which nodes can have different roles at the same time due to being connected to multiple other nodes and applications concurrently. A node may serve as peer for video storage and as client for obtaining weather data.

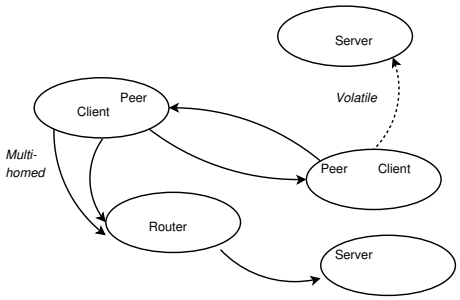

Figure 4.1: Exemplary computer networks topology and traffic flows

Computer networks may exist within physical premises as home network, corporate network or intranets, collectively called private networks. Access to services on these private networks from outside is then often physically impossible, adding an additional layer of protection on top of packet filters and application-level firewalls. The nodes within these networks have private IP addresses that are not routed on the public Internet. Any computer without such protection and with a publicly reachable IP address participates in the Internet. Virtual private networks can be used to extend the protection of private networks across the whole Internet.

IP addresses are either of type IPv4 or IPv6. In IPv4, addresses consist of four blocks of 8 bits, or 32 bits in total, in the form `a.b.c.d`, with each letter consisting of the values 0 to 255. Subnets are formed from right to left through a subnet mask. Hence, `a.b.c.0/8` refers to all addresses possible with d being 0 to 255, and `a.b.0.0/16` is a larger subnet with 64k addresses. In IPv6, there are eight blocks of 16 bits, or 128 bits in total. The global IPv4 address space is maintained by the Internet Assigned Numbers Authority (IANA), with Regional Internet Registries (RIRs) further allocating and assigning addresses to registered providers. The address space has become almost exhausted over recent years, in parallel with more widespread support for IPv6.

4.2 Networked Computers

A networked computer is a node on a network reachable by others on the same network or even globally, offering a certain number of services on one more multiple network interfaces. Three levels of access are typically distinguished: On the local interface (localhost), represented by the IPv4 address `127.0.0.1` and/or the IPv6 address `::1`; on a local network interface with locally addressable IP address such as `192.168.x.y`, `172.16.x.y` or `10.x.y.z`; and on a publicly routed IP address that, unless directly associated to a computer's interface, is network-translated by a router or gateway.

The quality of service on networked computers depends on the hardware resources available on that computer and on others attached to it as subordinates, i.e. slaves or workers, if applicable. Services may be offered primarily as storage and compute services. Two non-functional properties are of primary interest for compute services: performance and scalability. The performance indicates in which time an incoming service request or batch job is processed by the computer. Scalability is strongly related but indicates how many parallel requests or jobs can be processed without major regression in performance and without having to reject requests. For storage services, the performance (i.e. read and write access times) remains an important metric, but the capacity for storing data is also crucial, as is the integrity and further concerns. After all, computations can be restarted if temporarily lost (even at a cost), whereas loss of data may be irrevocable.

Attaching a computer to a network brings up security and trust issues. Especially on the Internet, access from other locations cannot be trusted per se, and appropriate security measures need to be taken. This refers primarily to keeping the system up to date (patched), running only the necessary services while shutting down others, introducing packet filtering and application-level firewalling to throttle requests and blacklist malicious origin sites, and performing offsite logging and intrusion detection to aid in post-mortem analysis.

From a data science workstation perspective, this means that complex services like those presented in the sections on middleware and platforms should be running only on localhost during the development phase without being exposed to access from outside, and that remote backups (potentially via version control) of at least all human-generated artefacts should be regularly conducted.

4.3 Services and Platforms

Services encapsulate application logic and further functionality behind a well-defined network interface. They are typically long-running processes exposing and expecting a higher-layer protocol on a specified port number. By connect-

ing to that port in a longer session or occasionally, an application becomes loosely distributed because some of the logic is executed in another process. Services in the wider sense encompass both programmatic and human interaction. Humans as service consumers interact with websites or virtual desktops among other human-computer interfaces, using specialised client applications such as web browsers or desktop widgets. Programmatic interaction on the other hand exposes Application Programming Interfaces (APIs) based on textual or binary network protocols. The interaction in terms of message contents and message exchange patterns is then controlled on the code level, either by directly composing the messages or by leveraging programming language-specific Software Development Kits (SDKs).

From a service consumption side, the service functionality is of primary interest. Services in the field of data science may offer the definition of projects or spaces, data storage and retrieval, data transformation and format conversion, queries, model serving, execution of predefined or supplied code and many other functions.

From the service provisioning side, notable configuration includes the network interface to bind to, the port number and permitted authentication methods, if any. As the service listens on the port number to handle incoming requests, it would have to interrupt the listening during the handling, leading to poor scalability. For this reason, multiple techniques exist to parallelise request handling. The first is the threading model, in which for each incoming request from a client, the service spawns a new thread that handles the communication with that client. This is very fast but also requires careful programming due to all threads sharing the same address space with a low degree of isolation. The second is the process model in which a full OS child process is created per request. This takes a bit longer (still in the milliseconds range though) but offers stronger isolation. The third is the proxy model, in which the actual processing is conducted on another machine and the service endpoint only performs input and output redirection, which saves CPU time. Various combinations and optimisations exist such as maintaining pools of prespawned threads or processes. Apart from high scalability, services can also be outfitted with high availability in terms of multiple instances. Requests may be redirected to one of those instances through load balancing at the client, DNS or proxy level, or they may be redirected to standby machine in case the primary machine is no longer available.

Services may be made discoverable with a well-defined declarative interface description or service description. Such descriptions first define permissable input and output messages including composite types on the interface level, but they can also cover messaging patterns, example requests and replies, and other aspects. Service descriptions with languages such as OpenAPI or the RESTful API Modelling Language (RAML) are common especially in the

field of web services that communicate over HTTP with pairs of requests and replies.

Service-oriented platforms encompass multiple complex multi-tenant software services that are typically accessed by users through an overarching web interface and connected to data processing services and backend services such as databases or file storage. Their architecture often follows a three-tier design with frontend, processing logic and persistence tiers, where each tier is not necessarily monolithic but can also be composed of multiple services, including microservice compositions. Platforms either derive users and permissions from the underlying operating system, or define their own model with the possibility to manage users, groups, roles, permissions, authentication methods and credentials. Many times, users can then generate credentials on their own, such as API keys, to facilitate the binding of clients to programmatic service interfaces.

Such platforms can be self-hosted or operated by a commercial entity. The former requires access to computing infrastructure as well as sufficient knowledge and skills of operating system administration. The latter types require registration for accessing most functionality. Both operational models may pose different risks from a privacy and security perspective. For proprietary commercial platforms, additional vendor lock-in risks exist.

Online services and platforms can be differentiated not only from a technical and operational perspective, but also on the business and legal terms of their operational model. Often, mixed models are used. A platform may require registration and filing a credit card, but gives a certain amount of free resources for first-time users before requiring the users to choose from regularly priced plans, either with fixed monthly price or with pay-per-use accounting and billing.

4.4 Parallel and High-Performance Computing

As explained for the request handling in services, scalability can be achieved through multi-threading or multi-processing, especially in conjunction with multi-core hardware. In general, multiple processors (CPUs/GPUs) can concurrently process partitioned data or perform other parallel activities. As a result to the denser use of computing resources, the overall processing time (wall-clock time) shrinks almost proportionally to the number of processors, although constrained by overheads. Those are not always predictable but are captured by a number of mathematical laws such as Amdahl's law, the law of Gustafson-Barsis, Sun-Ni and others leading to upper bounds for the possible speedups through parallelisation. Programming frameworks such as OpenMPI support the practical acceleration through parallelisation down to the code level, for instance, by detecting loops over data structures with no

causal dependencies between iterations. Message passing to synchronise parallel processes and tolerance for process failures are among the features that make parallel programming possible for average-skilled engineers. The support also happens at the data level with frameworks proposing suitable partitioning schemes so that the resource utilisation is maximised. From a cost and sustainability perspective, one should nevertheless be aware of the increased resource use. Slower batch processing may be an alternative to wall-clock time reduction for some scenarios.

Parallel computing with dedicated programming frameworks is sometimes offered as a distributed service for modest speed-ups beyond the number of CPU cores in a single system, i.e. for two- and low three-digits numbers of CPUs across systems. This evolves parallel computing into distributed parallel computing by offering capable high-level interfaces on top of the basic crunching. These interfaces allow for job submission and automatic distribution across machines along with the already mentioned functionalities known from parallel computing. Often, the distributed computing is then also subject to heterogeneity effects, with some CPUs finishing faster than others or some network links being slower than others, making predictable computing times a challenge.

High-Performance Computing (HPC) speeds up computation even more by massive parallelisation across a high number (hundreds or thousands) of compute nodes within a larger system or cluster, sometimes also within a supercomputer. Application programs are prepared for use on these clusters by internal parallelisation as well as primitives to synchronise the parallel activities such as independent loop iterations, often with message passing. The compute nodes are not freely accessible, and the compute time cannot be chosen freely. Instead, a compute job is defined, consisting of the program and necessary input data, and scheduled to be executed at the next possible time, within the allocation constraints. For instance, a user may be given a compute time of 1 hour across 20 nodes, and the program should be designed to maximise this allocation, ideally keeping partial results and being able to continue the computation in case of exceeding that allocation. A workload manager such as Slurm is taking care of the scheduling along with monitoring and production of execution statistics. Hence, HPC is suitable for batch processing of computationally intensive jobs such as number crunching and statistics, and often supported by specialised OSes such as Cray OS or Raise OS apart from tuned vanilla Linux setups.

HPC clusters are operated commercially but also within universities and national computing centres, covering scientific applications but also several data science-related tasks. In addition to the compute nodes, they often provide login nodes on which users can prepare the jobs and inspect the results. The global Top500 list informs about the peak performance achieved with HPC

machines every couple of months, but also in this community, sustainability concerns have become more important with the Green500 list, PUE and energy efficiency metrics and other indicators.

4.5 Cloud Computing

Cloud computing refers to the provisioning of applications, middleware and data through a set of infrastructure and platform services, collectively and colloquially called a cloud. These services are programmable and elastically scalable, they are provided on demand and, in a commercial context, they are metered and billed depending on the usage. The scope of the platform services relates to the platforms mentioned in the previous section. It encompasses the fully managed hosting of applications as virtual machines and containers, of data through various storage and database services along with other middleware, of network-specific functionality (DNS, HTTP gateways), of development services (Git, CI/CD), of large-scale data processing and machine learning interfaces including HPC and of many other functionalities.

Private cloud computing infrastructure is self-operated atop virtualisation and containerisation technology. Examples for these so-called cloud stacks are OpenStack, Kubernetes and OpenShift. For reliable storage, examples include Ceph and MinIO. Some of those stacks are also prepared to be run locally for development purposes, such as various flavours of Kubernetes including Minikube, Microk8s and K3s. Nevertheless, operating such basic infrastructure is typically outside the scope of a data scientist or engineer even with a faible for DataOps. A small misconfiguration can have serious and irreversible consequences such as data leak or data loss. On the positive side, a private cloud gives a maximum amount of freedom and flexibility, especially for trying out new technologies.

Many smaller commercial cloud providers exist on a national level in case a private cloud is not an option. Typically, these public providers operate their service portfolio based on existing cloud stacks. However, they do often not extend beyond simple application deployment and hosting, and their physical presence in many cases encompasses a single data centre or a pair of primary and secondary (failover) location. For most Small and Medium Enterprises (SMEs), this is more than sufficient, but a lack of support for diverse deployments (edge, serverless, accelerated computing) may become apparent soon even for them. A commercial but not-profit-oriented variant are institutional clouds operated, for instance, by national research and education networks for academic purposes.

Large multinational cloud providers, the so-called hyperscalers, all provide fully managed services for all target groups including a rich variety of offers for data scientists. Often these are provider-specific and sometimes even domain-

specific, such as analytics services for the health domain. These providers operate dozens of data centres in multiple regions. Additionally, the hyperscalers operate smaller edge location in order to reduce the network latency for time-critical messaging and computation, such as processing data from IoT devices. The global presence allows for operating responsive follow-the-sun services for a global audience, while it is less impactful on time-tolerant batch processing. A rich portfolio of turnkey services especially for data ingestion, processing and analytics is available at all hyperscalers. While often competitive in portfolio and pricing, failures in hyperscalers do occur frequently, often with devastating impact for large parts of the economy. Moreover, they are typically not well integrated into local research, innovation and supplier structures, leading to long-term issues with digital souvereignty.

Hence, a conscious and informed choice of the operational model with clouds, as a basis for DataOps, is crucial. More background information on three typical cloud scenarios is consequently given next.

4.5.1 Full application hosting

In this model, hypervisors, container engines and programming language-specific frameworks (e.g. web applications or function executors) are offered as managed services, ready to ingest custom program logic in the form of virtual machine images, container images or program code, respectively. This logic needs to adhere to certain conventions concerning port numbers, environment variables and lifecycle behaviour. Likewise, an application needs to be broken down into parts that fit into this environment, with appropriate glue in the form of triggers and messages in between. In a layered cloud environment, basic program execution is referred to as Infrastructure-as-a-Service (IaaS), whereas higher-level program management along with development and middleware services is referred to as Platform-as-a-Service (PaaS). The running application, serving multiple end users or tenants through a web interface or other interfaces, is then referred to as Software-as-a-Service (SaaS). However, all of these collective *aaS (or XaaS) terms are often used in a blurred manner in commercial practice and are merely a vague indicator of the service functionality and characteristics.

4.5.2 Partial hosting and on-demand offloading

Both in fully public cloud-hosted and in self-operated, private cloud-hosted scenarios, it might be useful to speed up program execution or achieve greater functionality by offloading crucial parts of the program and data to another cloud. Such offloading requires bundling the credentials of the secondary offloading cloud with the application code running either on the primary cloud or on cloud-attached devices such as mobile phones or edge machines. It

might also require workflow orchestration so that the offloading happens with the right context. Multi-cloud and cross-cloud frameworks such as Crossplane attempt to abstract away from the concrete cloud runtime locations. They facilitate programming in the large across execution technologies and providers, but are still emerging and neither required nor recommended for data scientists.

Similarly, data can be offloaded to storage services while performing the computation in a conventional form. This is especially useful for differential storage, in which data archiving happens across providers for maximised durability. The commonly used storage abstractions (FUSE, RClone) offer support for over 40 commercial cloud services and eliminate the network protocol differences between these services, presenting them all as unified storage resources.

4.5.3 Cloud backup

Almost the inverse to selected offloading of computation to the cloud is merely using the cloud as occasional backup for data. All primary data remains elsewhere, but sending encrypted data into a cloud storage service is cost-effective (ingress is usually for free), protective against data loss (although recovery egress will cost), and reasonably secure against data leaks. If infrastructure can be self-operated and hyperscaler service offerings are not needed, then this cloud usage pattern should be evaluated. Data can be synchronised to the cloud in intervals or based on events. Moreover, a trade-off between confidentiality and functionality can be achieved: all unencrypted data can still be processed by cloud services, for instance for analytics dashboards, and even some forms of encrypted data (using homomorphic, order-preserving and structure-preserving encryption) can still be put to constrained use without revealing too much. Lastly, anonymisation algorithms can be applied to maintain data characteristics while hiding the most concerning aspects, in particular Personally Identifiable Information (PII) from a privacy angle.

Repetition

1. Would a service offered on the IP address `127.0.0.1` be reachable from other computers?
2. A web service should be formally documented. Which language can be used for that purpose?
3. A latency-sensitive software function is to be deployed at massive scale to respond to vehicle movements. Would the deployment be more suitable in cloud computing or in high-performance computing environments?

Chapter 5

Applications and Tools

In this section, the foundational and conceptual knowledge conveyed in the first part of the book is put into practice in a local system context. This refers to a number of applications, primarily in the form of small composable tools and utilities that run locally on the interactive workstation of the data scientist or within the personal account on a remote server. These applications and tools support the exploration of the operating system and the juggling of data, models and code on that system as needed. Whenever appropriate, families of semantically identical or at least very similar tools are outlined first. To keep the text compact, at most one of the alternative implementations within each family is explained in detail. That does not imply that the alternatives are not valid choices depending on the scenario and surrounding conditions. Appropriate further reading pointers are given in selected cases to allow for making an informed decision about viable options.

5.1 Fundamentals

In order to work with local data processing and management tools efficiently, one has to understand a number of underlying concepts. Good tools are *optimised for one task* but also *configurable* and *composable* to be able to accomplish more complex tasks. Depending on the operating system, these concepts are more or less implemented on the operating system level or within third-party tools. Conventionally, Unix-like systems such as Linux and Mac OS X have had strong support with many pre-installed tools, whereas Windows has had limited support for many common tasks especially in networked environments. Nevertheless, all OSes evolve and, judging by their footprint of requiring multiple GB of storage space for installation nowadays, ship with an impressive number of tools out of the box. Generally, any functionality

already implemented in a tool and battle-tested in engineering scenarios saves time and effort to re-implement the same functionality, including in a higher-level programming language such as Python.

When interfacing with computers, the input modality or user interface is a primary concern. Tools are either *headless*, requiring no user interaction, or assume input and output in formats such as plain text (TUI, or Text User Interface), raster graphics (GUI, or Graphical User Interface), speech (VUI, or Voice User Interface) or, still rarely, gestures, eye movement or neuroelectronics. The history of computing has favoured text-mode tools, in alignment with textual programming languages, for automation and control tasks, which therefore form the focus of this chapter. Text refers to characters that are human-readable in the mastered languages and also human-writable, which comes with some challenges due to restricted keyboard layouts. The output of the text-mode interaction can be based on a raw line-based terminal where printing occurs sequentially but the cursor can be shifted as well as the terminal cleared. The interaction can also be done on the basis of a text-based navigation menu, with the background of text characters forming into rectangles that represent menus and input widgets. Foreground font attributes such as colours, underlining and bold face are also commonly supported by terminals to improve the text appearance. Limited support for mouse actions is provided by some text-mode applications, but typically the keyboard is the main input device. While the era of true text terminals (i.e. text-only combinations of screens and keyboards) has long been over in favour of high-resolution displays, the concept of terminals still exist today through terminal emulators, virtual terminals and virtual keyboards.

Such text tools, accessible through a terminal, work in a controlling operating system environment, typically a *shell*, that runs itself in a terminal as its main interaction point while putting itself into the background whenever a tool is in the focus. The shell completely supervises the tool lifecycle and facilitates composition.

A shell running as OS process permits navigating the filesystem and interactively entering the name of an executable on the search path. It then starts that executable as a subprocess, either in the foreground or in the background, with the working directory of the process set to the current directory of the shell. Tools might also be launched by other tools programmatically. While some tools strictly abide by the principle of being optimised for a single repetitive task, the behaviour of many tools can be adjusted to a certain degree. This can be accomplished statically before or during the invocation by parameterisation, through environment variables, or through configuration files. Moreover, some long-running tools allow for dynamic reconfiguration through OS signals, configuration file updates and other techniques. Command-line parameters, options and arguments are specified as a space-separated list after

the command name. They are then interpreted by the command depending on its implementation; for instance, a tool created in Python can refer to the system argument vector (`sys.argv`).

Once a tool is running, it can either remain quiet or provide text output to inform about progress and results. In addition to direct feedback from the execution in the form of standard output and error messages, many tools (especially those operating in headless mode) also produce log files that can be inspected after the execution to verify whether the invocation succeeded.

Most OSes support a superset of the shell and tools functionality defined in the 3rd volume of the IEEE Standard 1003.1-2017 "POSIX" (Portable Operating System Interface) that can be consulted for a more formal coverage.

5.2 Mastering Tools

5.2.1 Text-mode interaction

As text consists of individual characters, being able to read and type these characters in a fast way is a key skill in effective working with text tools. Reading and understanding character sets is a first step towards that. In Unicode, most characters have a single width, although some symbols from languages may have a double width, hence occupying twice the space as a single-width character. Text terminals are not always able to represent all characters depending on the chosen font (often replacing them by just an empty box □) and moreover may have problems with variable-width character sets. Legacy tools may also have problems with alphabetic characters beyond the Latin alphabet, although this issue has been reduced in recent years. Characters in Unicode are grouped into language-specific letters and symbols (A–Z, umlauts, accented letters and others), numbers, visible symbols including punctuation and mathematic operators (e.g. `<`, `;`) as well as invisible control characters such as backspace or enter.

With the keyboard, a limited set of characters can be entered directly by typing a key. More characters become available by combinations: ⇧(shift)+key, Alt+key, Alt+⇧+key, or multiple keys pressed in the right order. Multiple physical keyboard layouts exist to further complicate the input. For high productivity expressed by fast typing, it is important to understand these combinations, some of which are not shown on the physical keys themselves.

Care must also be applied when copying and pasting text from other sources such as web pages and PDF files. Often, they embed formatting and text modifications meant only for human consumption such as ligatures, ellipses and adjusted quotation marks (compare: „fi. . .“ and "fi..."). Search and replace patterns are then needed to make such text usable in the human-computer interaction. Pasting with Ctrl+Shift+V helps removing formatting.

Raw typing speed alone is often insufficient to be productive. Using macros or user-defined functions to avoid typing in the first place is often possible within shells, text editors and other tools. Moreover, interactive text input often allows for retrieval of the previously entered text for modification, which is another timesaver.

5.2.2 Types of tools

When choosing an appropriate tool, one should take a number of considerations into account. The first one is documentation. Good tools are documented with regard to what their purpose is, how they are configured and invoked (including examples), what error situations may occur and what standards they follow, if any. The second consideration is interface stability. Tools that have been around for a long time and are properly maintained by the OS distribution require less pre-configuration and are less likely to break in the future. This is important in the context of reducing technical debt when creating scripts for automating tasks.

In the next subsections, a number of representative tools from various categories are introduced. There are certainly valid alternatives to any of them; however, in order to get the job done, a data scientist is supposed to master at least one per category, and that one is introduced in sufficient detail. Several tools exist in multiple flavours, for instance, as stand-alone tools and as Python modules, enabling the re-use of functionality across working environments.

1. Operating system interaction: Shells, inspection and management of files, OS interaction, package and container management.

2. Data management: Synchronisation, version control.

3. Data processing: Text search, text processing and numeric processing, as well as visualisation.

5.3 Shells

Shells are command interpreters and process managers at the operating system level. In order to control and manage a system, but also to run and use complex infrastructure, it is essential to understand the basic role of local and remote shells. Interactive shells provide the command-line interface (CLI) as a specialisation of a textual user interface (TUI) to further tools and application programs. In contrast, shell command can also be invoked in batch mode. In this case, the commands are supposed to be written in a script file that is linearly interpreted, and interaction is only possible at the level of individual commands.

5.3.1 Overview on shells and terminals

Due to historic developments, there is no single shell. Rather, each OS has its own concept of a shell, and each shell has its own command set and language for automating the execution of tools. Windows has two main shells - Command Prompt (cmd) and PowerShell. Typing `cmd` in the OS start menu allows launching an instance of the former. These shells are both tightly coupled to the terminal window, whereas other operating systems maintain a separation. Mac OS X has the native Terminal.app that uses either Bash or Zsh as shells; although many users prefer iTerm that adds support for the Tcsh and Fish shells among other features. And Linux has a whole range of shells (e.g. Bash, Dash, Zsh, Fish, Tcsh, Ksh, Xonsh, Busybox-Ash, Yash), in addition to a plethora of graphical terminal emulators that make the shells accessible to their users. These emulators are often strongly related to the corresponding desktop environments, such as GNOME Terminal, XFCE Terminal, Konsole, ETerm and XTerm but can be used interchangeably. On a graphical Linux desktop, one either types the emulator's command name (`konsole`, `gnome-terminal`, `xfce4-terminal`, `xterm`, `Eterm`) or typically finds a menu entry for a terminal under System or Tools. Even on such systems, a text-mode shell terminal might be autostarted by default. Switching from the graphical environment to the text mode is then possible with a key combination such as Ctrl+Alt+F2. However, this should rarely be needed, especially with graphical terminal emulators that can be started in full-screen mode or switched to this mode via a menu entry or application-specific key combination, such as Alt+Enter in XTerm.

5.3.2 Local shell access with Bash

Bash[1] is one of the most powerful shells, acting as an interface between the user (or user applications), the virtual terminal and the operating system. It is one of the native shells in Linux and Mac OS X and is also available as an add-on package in Windows. In this section, Bash is explained in greater detail as one of the most capable and widely used interactive shells.

Bash commands can be given interactively at the *command prompt* or as interpreted files in the form of shell scripts. This is similar to Python, whose shell (i.e. interpreter) understands both interactive commands and scripts. However, the Bash language is different from Python in many ways. For example, built-in command names differ (`echo` instead of `print`), variable assignments must not use any spaces (`a=0`), and references to variables require a dollar sign (`echo $a`). Single-line comments, on the other hand, are given in similar form preceded with the hash or pound sign (`#`).

[1] Bash website: `https://www.gnu.org/software/bash/`

Attributed to the rich feature set of Bash is its startup behaviour, which is often too slow for non-interactive use, i.e. batch scripts. Therefore, alternative shells remain relevant, such as Dash for non-interactive shell scripts. The mix of shells might lead to confusion on the user side. A command executes well on the interactive terminal but does not run in a script. Often, this is caused by different shells executing the same command, with correspondingly different behaviour and outcome. As a rule of thumb, all shell scripts should be explicit about the interpreter in the form of a shebang line as explained below.

A brief introduction to the language and behaviour in addition to practical Bash handling is given in the work 'Bash Quick Start Guide' and reference usage documentation, such as the manual page for Bash (`man bash`), also hosted as an online copy.[2] This section only documents a few essential Bash commands and concepts to cover the occasional use, without claiming to be a full shell programming guide.

The canonical form of interactive usage is `bash`, leading to a new shell process being instantiated. Alternatively, the shell is invoked as a non-interactive wrapper around a command, i.e. `bash -c 'command to be executed'`, where commands may range from simple executables to compound commands with input-output redirection, boolean logic and other shell execution facilities. A special form is the login shell, which is invoked automatically upon text-mode login to the system, either at a local text-mode login prompt or over the network using a terminal emulator. Login shells are interactive but ready different configuration files upon invocation. The startup of the shell can thus be customised through invocation-specific configuration files that are themselves shell scripts. For login shells, this is primarily the system-wide file `/etc/profile`, complemented by the per-user file `~/.profile` and further Bash-specific files in the home directory (`.bash_profile`, `.bash_login`). For non-login interactive shells, notable configuration takes place in `/etc/bash.bashrc` and the `.bashrc` file in the user's home directory. This file can source other files, and this is supported by convention with `.bash_aliases`. In contrast, non-interactive shell processes do not read any default configuration.

Bash scripts marked as executable in their file metadata (explained below) can and should contain a first line pointing to the executable of the shell to execute these scripts with exactly that shell. This is called a shebang line. For instance, a script called `test.sh` might be marked with `#!/bin/bash` so that running `./test.sh` works and shows up in the process list despite the shell script not being an executable in the OS sense of the word. This first line is treated as a comment by the shell itself but instructs the OS loader to invoke the right shell.

Interactive Bash processes require a controlling terminal, either a native text-mode terminal of the operating system or a graphical terminal emulator.

[2] Bash manual page online: `https://man7.org/linux/man-pages/man1/bash.1.html`

These interactive shells keep track of the current working directory and permit navigation with built-in commands such as `cd` (change directory). The command prompt is configurable but in most cases shows the working directory for reference. The special character ~ (tilde) refers to the current user directory, e.g. `/home/user`, and `~otheruser` to the home directory of other users.

Commands to repeat in alphabetic order: bash, cd (built-in)

5.3.3 Bash variables

Variable names in Bash are case sensitive. They start with a letter (restricted to ASCII) or underscore and can contain further letters, underscores and digits. Variables are dynamically typed and created by assignment to a variable name by using the equal sign without surrounding spaces and without spaces in unquoted values. Exemplary assignments are `var=123`, `var=abc` and `var="abc def"`. Both single and double quotes can be used, with the difference that other variables referenced within double quotes are substituted by their values, whereas single quotes force the verbatim assignment. The only exception is the assignment of special characters in the form of `var=$'\n'`.

Assigned variables can be used with the dollar sign prepended, such as `$var`. The most common case shows their value interactively with the shell built-in command `echo`, as in `echo $var`. While the echo command tolerates an empty argument in case the variable is not set at all, other commands do not necessarily tolerate the same. To enforce passing an empty argument instead of no argument in this case, the recommended notion is enclosing the variable in quotation marks, resulting in the instruction `echo "$var"`. Another difficulty is that, in a string context, Bash would sometimes not know the boundaries of a variable name, as in `echo "$numberhouses"`. The boundaries can then be supplied explicitly, as in `echo "${number}houses"`.

Bash scripts and functions can be parameterised; while `$0` refers to the script or function name itself, `$1` and following arguments contain the parameters. They can be tested for being empty or not with an if-clause involving for instance `test -z "$1"`. In that case they should always be enclosed in double quotes, because the variable may not exist, leading to a syntactically incorrect test statement. In Bash, the test instruction can be replaced by square brackets in conditional clauses. For instance, `if [ ! -z "$1" ]; then ...; fi` only executes a block of code if the first parameter has been supplied.

Bash processes also keep track of the environment through *environment variables*. Any process spawned from the shell inherits these exported variables so that custom program behaviour can be triggered by setting up appropriate variables. This differs from regular local variables that are only valid within one shell process. A simple assignment of an environment variable may read

like `export a=0`, although exported variables by convention use uppercase names. All variables can be shown with the `env` command, or alternatively shown and filtered with `printenv`. Among the ones often referred to are `$USER`, `$HOME` and `$LANG`, representing the current user's identity, home directory and language setting, respectively, as well as `$PWD` and `$PATH` further explained below. Rarely, there are recommendations to set `LD_PRELOAD` to inject override functionality into compiled applications, such as `eatmydata` to avoid file system syncs, `fakeroot` to not let privileged operations fail or faking the system time; however, this dynamic reprogramming must be used with care. To temporarily set an environment variable for one particular process without affecting the shell or other processes, the variable assignment can precede the command, as in forcing a particular language: `LANG=en_US <command>`.

While environment variables are a portable concept and also work in shells other than Bash, there are additional Bash-internal variables predefined by the interpreter instance and partially affecting the programming behaviour of the shell itself. Examples include `OSTYPE` and `HOSTTYPE` giving information about the operating system, `EUID` containing the effective user id of the current process and `IFS`, the internal field separator used when reading from sequences of data. These variables are not shown in `env`. To nevertheless include these variables but also other internal definitions such as functions, and to see all entries in the entire namespace of the running shell, the `set` command can be used. Shell-internal variables are explained later in the section on shell programming.

Special variables exist in Bash to retrieve the current shell process identifier (`$$`) and its parent (`$PPID`) and to generate a random integer number in the range 0–32768 (`$RANDOM`).

Commands to repeat in alphabetic order: echo (built-in & executable), env (built-in), export (built-in), printenv (built-in), set (built-in), test (built-in)

Environment variables to repeat: $$, $HOME, $IFS, $LD_PRELOAD, $LANG, $PATH, $PPID, $PWD, $RANDOM, $USER

5.3.4 Bash commands

The imperative vocabulary of shells consists of both built-in and executable commands, where built-in commands are internal to the shell (part of the default vocabulary) or provided by the user within the shell programming environment, and executable commands refer to programs found in the search path. A number of executables are always installed by default on any system, whereas, sometimes, useful tools exist but must be regularly post-installed,

referred to as non-default or external commands. This is especially true for services (daemons) and other middleware.

The search path for executables, expressed as environment variable `$PATH`, is interpreted as a list of absolute and relative directories separated by colon (`:`). New entries can be prepended or appended to it as necessary. The order of evaluation is from left to right. As soon as an executable matching a command name not available as internal command is found, it is taken as a match. Hence, this paths list has similar semantics to the variable `sys.path` for searching modules to be included in Python. Notably, the default path does not contain the current working directory by default (`.`), and therefore executables not on the search path need to be addressed by relative or absolute path, such as: `./myprog` or `/opt/myprog`. Alternatively, the paths list can be extended to include the current directory (`export PATH=.:$PATH`). In that case, running just `myprog` works. Attention should be paid to this because it is at risk of changing behaviour when accidentally a globally installed program is shadowed by an executable produced in this directory. With the `which` command, the first matching path for an executable is informed.

The most trivial built-in is the colon (`:`), a no-op command that does not do anything but having command semantics, similar to `pass` in Python. POSIX defines around 20 built-ins while modern shells support a few more. Programmable shells also permit the creation of custom internal commands. Bash in particular allows two kinds of custom commands: simple aliases (with the built-in `alias` command) and arbitrarily complex functions. The resolution order matters so that sometimes external commands are shadowed by built-ins with slightly different behaviour (`echo`, but also `time` and `kill` introduced later). For instance, when talking about `echo`, this might refer to the shell built-in explained before or to the namesake executable on the default path, which is not much different (`/usr/bin/echo`), or potentially a self-built executable in the current working directory in case that is part of the search path. The `type` command, similar to the function with the same name in Python, tells about what type a command name is of (built-in command, user-defined function, alias, external command/executable or reserved name). A brief documentation for built-in commands may be shown with the `help` command.

Commands are entered on a per-line basis at the command prompt. Command names may be autocompleted with the Tab (tabulator) key (⇥). If the completion would be ambiguous because multiple commands with the same prefix exist, the Tab key needs to be pressed twice to show all of the candidates. For instance, typing `e<tab><tab>` shows `echo` among other command options. After completing a command with the Enter key, the command is executed and the result is shown before returning to the prompt. Previous commands may be retrieved in order by pressing the Arrow-up key. Search-

ing through previous commands is possible by pressing Ctrl+r followed by a search term. When composing the input at the prompt, further quick navigation is possible with, for instance, Ctrl+a and Ctrl+e to jump to the beginning and the end of the input line, respectively. With those few keyboard strokes, entering commands becomes efficient and the shell becomes an indispensable tool for automating tasks.

Complete commands in Bash consist of the case-sensitive command name, either built-in or an executable, and a number of options, parameters and arguments, all separated by spaces and otherwise defined in an application-dependent way. Values with spaces can be quoted with `'` (single quotes) or `"` (double quotes). The quotes may be combined, but if single quotes are used in the outer scope, then variables are not expanded within it. Arguments with placeholders (`?` for single characters, `*` for multiple characters) are expanded based on the content of matching files and directories in the underlying filesystem, through so-called globbing, but only if there is at least one such file or directory. Otherwise, the placeholders remain in place verbatim, which is often not the intended behaviour. There is also a limit to the size of the argument list. Hence, copying (`cp *.jpeg myfolder`) might fail if the number of files is too high; in that case, an iteration over all files with a `for` loop or with the `find` command must be used. Overall, shell globbing is a powerful feature but must be used with care due to the mentioned and often not obvious limitations.

Multiple independent commands may be chained for serial execution by `;` (semicolon), although that is not recommended as it impedes readability compared to placement across multiple lines. With the `parallel` wrapper command, multiple instances of the same command or other independent commands can also be run at the same time. Lastly, the pipe operator `|` is able to chain dependent commands such that there is a data stream flow from the first to the last while they execute in parallel.

An example invocation combining both piping and parallelisation is to shorten the execution time by converting multiple photos from the digital camera to a lower resolution. It could be achieved as follows: `ls P*.JPG | parallel convert -scale 1200 {} S{}`. As a result of involving the external conversion tools, each image (e.g. `P1020.JPG`) ends up downscaled and renamed (`SP1020.JPG`). Drawing from the way `parallel` works, one can concluded that external shell commands can be divided into two groups: regular commands, and wrapper commands that execute commands given as parameters. Among the more useful wrapper commands beyond the ones already mentioned are `stdbuf` to change especially the output buffering behaviour and `timeout` to run a command under a time barrier.

All commands are executed in the foreground, blocking the shell from proceeding, but they can be put into the background either at invocation time by appending `&` (ampersand) or after interactive invocation by pressing Ctrl+z

(suspend) followed by the built-in command `bg` (background). Likewise, processes can be brought to the foreground again with `fg`. Applications can be cancelled and terminated with Ctrl+c. In practice, this job control is only useful for batch processing, whereas some programs are meant to run interactively in the foreground as they take over the entire terminal and do not leave any output trace after termination. Batch output can also be redirected to a file, assuming the specified file location is writable for the current user. Writing and overwriting the file is achieved using the `>` operator for regular output and `2>` for error output, mirroring the ability to redirect input with `<`. Appending to the file without destroying previous content is also possible with the `>>` operator. If output should still be visible while at the same time be captured to a file, piping can be used as follows: `command | tee <file>` for overwriting, and using `tee -a <file>` for appending.

Subcommands can be executed and their output captured with the backtick character (‘), as in: `command ‘subcommand‘`. The subcommand is then executed first, and its output is placed in lieu of the backticked placeholder, effectively executing the first command with the subcommand output as argument.

Bash contains built-in facilities for integer arithmetics, for instance, `echo $(($a+1))` and basic support for advanced data types such as dictionaries. For more precise calculations and more versatile data structures, external tools must be used. Finally, Bash processes can be quit with the `exit` command, like Python but without the parentheses, or alternatively the keyboard combination Ctrl+d.

The following listing shows exemplary shell commands with an explanation.

```
echo "a b c" # output the string "a b c" to the terminal's
    standard output channel
sleep 5       # do nothing for five seconds
sleep 5 &     # do nothing in the background, while
    liberating the command prompt

ls -l /etc > ~/conffiles.txt # create file with list of
    system configuration files
```

Like the built-in help command in Python, it is often possible to obtain at least basic documentation on commands via manual pages. The representative command `man ls` documents all possible parameters for listing files and directories. For built-in Bash commands, the `help` command is used instead. Finally, when it becomes hard to keep the overview, the `clear` command can be invoked to clear the screen. In case the terminal is severely messed up and no longer produces linebreaks, the `reset` command can help to bring its state back to order.

Commands to repeat in alphabetic order: bg (built-in), clear, cp, fg (built-in), help (built-in), man, parallel, reset, sleep, tee, type (built-in), which

Environment variables to repeat: $PATH

5.3.5 Remote shell access with OpenSSH

Oftentimes, the data scientist's local workstation environment is considered unfit for a certain task due to general resource shortage (disk, memory, processors), lack of specialised resources (especially GPUs), outdated or unsuitable OS, or the need to collaborate among multiple people. In these situations, a dedicated physical computer might be set up or a virtual machine might be instantiated at a virtual machine provider. This machine then serves as a remote machine to which a connection can be established, linking it with the workstation by allowing cross-machine file access and tool execution.

An interactive and secure remote shell connection to any server can be established using the SSH (Secure Shell) protocol, most commonly in the form of *OpenSSH*. This stateful application creates a remote work session in which it takes control of keyboard input on the local (client) machine and relays it to the remote (server) machine, more specifically to the configured login shell of the chosen user, while relaying back the responses. The OpenSSH client has over time become a standard tool for remote operations on all major operating systems – Linux, Mac OS X and Windows. Apart from text-based sessions, it also permits the forwarding of graphical applications on pairs of systems supporting the X11 protocol, such as Linux.

The canonical form of usage for a text session is `ssh <user>@<server>`, using the default port number (22) and default negotiation of encryption parameters, the so-called ciphers. This command either asks for a password, or in case an SSH key is used for authentication, it may or may not ask for a passphrase, depending on how the key was created.

Graphical counterparts to the SSH CLI are available, for example, `putty` as utility on Windows formerly recommended over many years. However, they might not be suitable for automation in data science-related shell scripts. In case the standard OpenSSH CLI client is not available on a Windows installation, it can also be installed in Powershell easily with administrator privileges: `Add-WindowsCapability -Online -Name OpenSSH.Client~~~~0.0.1.0`.

Hence, no matter what shells are installed locally on machines, remote shell access permits standardising shell scripts and other shell-based automation across teams in a portable manner.

Authentication is performed using passwords or better using a public/private key pair that must be first generated on the client. The private key remains on the machine the user is working one, while the public key can be

deployed on the remote machine, including in several collaboration services beyond SSH itself. The usual command to generate a key on the local workstation is `ssh-keygen -f <filename.key>`. The command can be simplified to `ssh-keygen` upon first use, resulting in the key stored in the default location. On Unix-like systems, the private key path is `~/.ssh/id_rsa` whereas on Windows it is `C:\Users\<user>\.ssh\id_rsa`. The corresponding public key has the same path, but with `.pub` appended to the file name. A key can be protected with a passphrase, although that can also be omitted, especially when automated batch authentications are planned. Losing a password-less private key would, however, constitute a grave security risk, and the key needs to be handled with extra care to prevent that from ever happening.

With OpenSSH, the public key must then be placed into the file `~/.ssh/authorized_keys` on the remote account with locked down permissions by the account administrator or, if password-based logins are still enabled, by the user. The permissions require both the `.ssh` directory and the file to be only accessible by the user itself including automatically the superuser. Moreover, unless the default path is used, the key needs to be specified as identity upon each connection (`-i`) or the configuration file (`.ssh/config`) needs to have an `IdentityFile` setting for the affected `Host` entry.

Files are transferred via the SSH protocol with the `scp` command, meaning secure copy, in the form: `scp <filename> <user>@<server>:<path>`. The path can be omitted in case the user's home directory is the target. SSH also supports a secure variant of the File Transfer Protocol (FTP) with the `sftp` command. After logging into a system with `sftp <user>@<server>`, typical FTP commands can be used to copy files back and forth and to compare the presence of local and remote files.

Usually, the first file to transfer from the workstation to the remote machine is the SSH public key to prepare it for use. This procedure is explained in the following listing. All commands around copying files are explained in greater detail in a later section.

```
# First, copy the public key to the remote machine, into
    the user's home directory
scp id_rsa.pub user@machine:
# Then, log into the machine, and perform remaining steps
    there
ssh user@machine
# Create the .ssh directory with the right permissions; the
     mode 700 is explained further down
mkdir -m 700 .ssh
# Register the public key file
chmod 600 id_rsa
mv id_rsa.pub .ssh/authorized_keys
```

The commonly used flag `-r` is used for recursively copying entire directories. The secure copy commands are explained in greater detail in the section on file management. A remote shell can also be quit by terminating the shell with the `exit` command or by Ctrl+d.

More information on OpenSSH usage is available through the auxiliary article 'SSH: a Modern Lock for Your Server?' as well as through online documentation.[3]

> Commands to repeat in alphabetic order: exit (built-in), scp, ssh, ssh-keygen

5.3.6 Advanced shell management with Screen and TMux

Sometimes, the interactive use of applications, either shells or other applications spawned from them, stretches across long sessions beyond the login period. Due to occasional disconnects when using remote sessions caused by the SSH configuration or external network events, the login period can sometimes be quite short, and any tool running for more than a few seconds is at risk of either continuing its execution in the background or even of termination when no special protection is applied. Screen is a tool which extends remote shells with the possibility to reconnect to them later without losing the session. It achieves that by running as invisible layer in between the application and the shell. The shell runs Screen which in turn spawns a long-running background process, and that Screen process runs the tool command. It should be noted that Screen thus protects against disconnects, but not against restarts of the server itself. Evidently, in such cases the tool must be further protected with automated restart and internal checkpointing capabilities.

A basic invocation is `screen <command>`. Screen runs as long as the command itself, but a preliminary detachment is possible with the keycode sequence Ctrl+a;d (i.e. first pressing the control key (Ctrl) and the letter A without shift (a) simultaneously, then releasing the keys, then pressing d). The detachment drops the user back into the shell Screen was started from. Closing this shell has then no effect on the screened command execution. To re-attach, all running Screen sessions can be shown with `screen -list` and a particular session can be chosen and restored with `screen -r <session>`. Default session names include the terminal number and the local host name. Screen sessions can be invoked in the background in the first place, even with human-recognisable names, by adding the parameters `-d -m -S <name>`. Many additional invocation options and key codes for managing sessions exist. The authoritative documentation for Screen is available online[4],

[3] OpenSSH manual: `https://www.openssh.com/manual.html`

[4] Screen documentation: `https://www.gnu.org/software/screen/manual/screen.html`

and the manual page is also sufficiently useful for further customising the use of the tool.

On larger displays, it may also be desired to split up the terminal estate into two or more shells. That way, the output of two commands can be compared, or one command can be inspected while a shell is provided at the same time to further control that command, among other scenarios.

While Screen has some support for splitting sessions, a small tool not necessarily bound to background sessions might be useful in this case. TMux, the terminal multiplexer, can do just that. Its canonical invocation is just `tmux`, launching a subshell that behaves like the parent shell but accepts special key codes to manage screen splitting. Commands to a running tmux session are given by key sequences starting with Ctrl+b and followed by another key or combination of keys. Most importantly, Ctrl+b;% creates a new horizontally split pane (" for vertical split), and Ctrl+b;Ctrl+o rotates through the panes. Each pane runs a shell by default and can simply be closed by the command `exit` or Ctrl+d.

Commands to repeat in alphabetic order: screen, tmux

Repetition

1. How can a user log into the server X with account name Y?
2. How can the same user transfer a file Z into his or her home directory on the server?
3. The user would like to work on a text over many hours. Which command sets up a suitable long-running shell session?
4. Which wrapper commands to launch other commands exist?

5.4 Useful shell tools

Data scientists who are familiar with the shell as a working environment soon understand why it is called a shell: It is just a shell for a lot of tools that permit interacting with the system and with data. Like a precious pearl, the interest soon shifts to those tools. In this section, a lot of standard tools are introduced. The ambition should not be to remember all of them, at least not at once. Rather, the section makes an attempt to group similar tools together and explain their relationships, so that looking them up later becomes a systematic and efficient process. For that matter, seven groups of tools are distinguished.

The first two sections explain basic overview and exploration tools for the underlying hardware resources and the operating system, respectively. This is

followed by three groups of tools related to time and event handling, to data organisation through files and directories, and to data creation and modification. Two more sections then dive deeper into issues around networking and system administration, both of which form the basis of distributed DataOps in practice.

5.4.1 Hardware resources exploration

In a dynamic world where users roam across different local and remote systems, getting an overview of the current capabilities on the operating system and underlying hardware resource levels is often the first step. The commands `free -h`, `df -Th` and `lscpu` (or more verbose, `cat /proc/cpuinfo`) are used to convey information about the available main memory, free disk space per file system and CPU resources, respectively. They are usually called first on a new system to verify the ability to work. In greater detail, the `free` command differentiates between main memory and, if configured, disk-based swap space as resources to store volatile process data, and indicates the amount of available, currently used, and free memory. Memory usage occurs as the sum of all usage per process, with some being more memory-hungry than others. The free value often tends to be quite low, almost going towards the zero line, due to the OS keeping used memory pages (blocks of memory) around as cache and buffer memory and using some pages as shared area between processes. Therefore, the tool also informs about the net free value, which is often higher and the value of interest for checking if sufficient memory would be available when freeing up all disposable pages. While the default unit of the output numbers is KiB and the output could be processed by automated scripts, the parameter `-h` is meant for human consumption and adapts the unit to what makes sense for a number, such as MiB or GiB. The option `-h` is not universal and often implies the invocation of help, but a number of system-related tools use it to express human-friendly output.

The `df` (disk free) tool is such an example. It is the equivalent of `free` for persistent data stored across multiple storage media and storage-related block devices, potentially across an even higher number of partitions. Its often-used option are the mentioned `-h` for human-readable output and `-T` for displaying the partition types, effectively the file systems in use in each partition. This tool shows not only available and used disk space for each partition but also where that partition is mounted in the file system hierarchy. A system consists of at least a persistent partition at the root directory `/`, using file systems such as *ext4* or *xfs*. Typical systems furthermore include a safety-critical and therefore separated persistent boot partition `/boot` as well as OS-specific internal volatile partitions outlined below.

The command `lscpu` (list CPU specifications) informs about the processor architecture and performance, the number of processor cores, cache sizes,

hardware virtualisation support and similar CPU information. In contrast to the other two tools, its output is human-readable by default. For automation use, `/proc/cpuinfo` is consulted or `lscpu -b` is invoked to show all sizes in bytes, or even `lscpu -j` to output JSON-formatted data.

To get further information about system resources, a family of `ls`-derived list commands similar to `lscpu` exist. For instance, information about peripherals can be obtained with `lspci` for PCI-connected internal peripherals such as graphics adapters, disk and network controllers, audio chips and non-volatile memory. Likewise, especially for workstations, `lsusb` informs about the USB hub, devices connected to it, often including the notebook webcam by default, and plugged in storage media. Not all block devices may be mounted. The `lsblk` command shows the hierarchies of volumes and partitions and correlates them to active mountpoints.

Basic information about available network adapters can be retrieved by reading the file `cat /proc/net/dev`. A more comfortable management of network interfaces is explained later. With the covered commands, the capabilities of all important computing resources (CPU, RAM, disk, network, peripherals) should be known on any system, and the exploration can proceed further into the OS-specific processes and structures.

Commands to repeat in alphabetic order: df, free, lsblk, lscpu, lspci, lsusb

5.4.2 Operating system exploration

At the top of the hardware resources, `uname -a` (Unix name with all information) gives basic information about the running operating system kernel. A more challenging question could be which OS flavour or distribution is being used in case additional software needs to be installed. This is explained in the system administration section below. The name of a system is not sufficient to see what it does. Revisiting the `df` output, several internal partitions mounted in-memory without relation to physical block storage become evident. Some of them are using *tmpfs* as in-memory file system, such as `/run` for temporary data storage of services and `/dev/shm` for named shared memory areas. While the `df` output is human-friendly, it may omit information about some more internally mounted file systems, and hence the `mount` command gives a complete but less readable information about all file systems in use, all with highly exotic and system-dependent virtual file systems. This includes `/sys` for information on the hardware, `/proc` for running processes and `/sys/fs/cgroup` for process isolation among many others. Additional file systems may be mounted into the hierarchy from block devices with `mount -t <type> <device> <mountpoint>`, alternatively from images as loop mounts with `mount -o loop`, or from a directory with `mount -bind`.

More commands similar to `lscpu` exist for revealing OS structures. For instance, `lsipc` gives a statistical account of inter-process communication, with `lsipc -m` drilling down into shared memory areas. Information about open files and network connections is given with `lsof`. The more active a system is with running processes and services, the longer the output of these commands gets. They are useful especially for debugging, to find out which process has opened which file among other questions.

The main memory and disk-free commands in the previous section include information about the current resource utilisation, whereas the CPU-related statistics only show static capacity information. Going deeper into knowing the running processes and their computational needs requires process monitoring. There are tools to monitor all resources combined over time, such as `vmstat 1` showing memory and processor utilisation on a per-second basis, but their output is not easy to interpret and moreover does not reveal the troublesome processes that might cause a CPU to be overloaded.

Therefore, process-level monitoring is important to find out about the CPU utilisation and about the activities managed by an operating system in general. The command `ps` and the related `pstree` show the already running processes along with basic statistical information. Typical invocations are `ps wxf` to list all of the user's processes, and `ps wwauxf` to list all processes of all users with all arguments. Evidently, this comes with drawbacks such as verbose output and lack of highlighting the CPU-intensive processes. A live view can be performed with `top` or its fancier cousin `htop`. Both tools take over the terminal and need to be terminated with the key Q. In `top`, CPU-intensive processes are listed first, yet with the M key this can be changed to memory-intensive processes. In `htop`, process management works in a menu-based way. One or multiple processes can be marked with the space key (␣). To terminate selected processes, the F9 key (i.e. Fn+F9 on many notebooks) is pressed, offering the selection of a signal to be sent to these processes, by default the TERM signal that can be confirmed by pressing the Enter key.

In case a stray process is detected and must be terminated before re-invoking a command, the `kill` command outside of `top`/`htop` can be used for that purpose, taking the PID (process id) as parameter. This command sends a signal to the OS asking it to terminate the process, which works if the process is owned by the user sending the request or of course when the super-ruser sends this instruction. The default signal is *TERM*, asking for graceful termination while giving the affected process the ability to perform some last work or even block the request. A forced invocation would happen with the *KILL* signal, as that can not be blocked, via `kill -KILL <pid>`. Further useful signals from a user perspective are *STOP* to pause a process and *CONT* to continue its execution, while many other signals are sent by the OS itself to be handled by the process. Applications can implement handlers for interceptable user-defined signals (*USR1*, *USR2*), for instance, as Python methods.

If multiple processes are running with the same name, their PIDs can be obtained with the command `pidof <program>`. For example, `pidof bash` shows the identifiers of all active Bash processes, including both executed scripts and interactive sessions. With options such as `-s` (single shot) or `-q` (quiet, only report existence of at least one process via exit status), this tool is useful in shell scripts to detect for instance stray instances of a program, especially in conjunction with PID files, and to force termination of all instances in upgrade scenarios. Combining the functionality of `pidof` and `kill`, the `killall` command can send signals including soft and hard termination to a group of processes. Detailed statistics for a single process may be shown with the `prtstat <pid>` command.

The command `hostname` conveys information about the host's own internal name, which is typically reflected in the output of `uname -a` and in the command prompt appearance as well. As outlined in the description of networking concepts, the file `/etc/hosts` can map IP addresses to names for other hosts in the format `<ip> <hostname>`. On Windows, the corresponding file also exists and is located in `C:\Windows\System32\drivers\etc\hosts`, and on Mac OS X in `/private/etc/hosts`. Typically, entries in the hosts file override DNS queries. Often, though, the local IP address `127.0.0.1` resolves to `localhost` whereas the name given by `hostname` is unknown due to a missing entry in the mapping file. This might cause problems later working with services that report unresolved hostnames. The solution is to add both forward and reverse mappings to that file and update them in case the host gets renamed. To work further with the mappings, the `host <ip>|<name>` command can resolve names to IP addresses and vice versa but usually does not consult the hosts file unless this is configured in the file `/etc/resolv.conf`; the command `getent hosts` is more appropriate in that case. The `ping <ip>|<hostname>` command may be used to check for reachability of a host. It always consults the mapping file, although its communication via ICMP might be blocked by that host. No tool is perfect, and the necessary trade-offs show especially in attempts of debugging network issues..

A related command to convey OS identitity information is `whoami`, showing the current user that is likewise reflected in the prompt, along with `id` showing more information about that user, in particular group memberships. This encompasses the primary group by name and numeric identifier (*GID*), but also secondary groups. Moreover, the `uptime` command shows how long the system has been running and under which load. One may even maintain the history of all uptimes with `uprecords`. The related command `who` shows all users currently logged into the system, complemented by `w` giving more details and combining it with the uptime functionality, `last` showing the full history of previous login sessions, and `lastlog` informing about the most recent login of each user on the system. This typically includes special-purpose system users created to isolate more complex applications who obviously never log in.

With the additionally covered commands, the exploration on the OS level is complete. All essential configuration settings and actively running processes are known, and further tools can be used to perform productive work related to data, time and space (locations within the file system).

> Commands to repeat in alphabetic order: host, hostname, id, kill, killall, last, lastlog, lsipc, lsof, mount, pidof, ping, prtstat, ps, pstree, top/htop, uname, uptime, vmstat, w, who, whoami

5.4.3 Time- and event-related commands

The command `date` outputs the local date and time down to second precision according to the local conventions, consisting of both language and timezone. The language and associated culture-specific date format can be adjusted without affecting the time, as in: `LANG=en_US date`. The value of the environment variable `LANG` in this context is either just a language code (e.g. `en` according to the international norm ISO 639-1) or a language code combined with a country code (e.g. `US` according to ISO 3166-1 Alpha 2). The timezone can be set either to the canonical universal coordinated time (UTC) with `date --utc`/`-u` or with a timezone environment variable such as `TZ=Europe/Lisbon date`. The file `/etc/timezone` contains the system-wide timezone setting. Human-readable times can be converted into numerical timestamps with `date +%s`, and converted back from those timestamps with `date -d @1700000000` (short for for `--date`). These 32-bit integer timestamps refer to the number of seconds elapsed since the epoch on January 1, 1970. In other contexts, such as in Python's `time.time()` functions, the timestamps also appear with sub-second precision, indicating a microseconds fraction although the actual precision depends on the CPU quartz and is likely in the milliseconds range. In the shell, the closest equivalent is `date %s.%N`, indicating nanoseconds with the same caveat. Other standard notations of dates can be produced in the sorting-friendly ISO 8601 YYYY-MM-DD format with the shortcut `date -I` and in the human-friendly DD.MM.YYYY format with the string formatting `date +%d.%m.%Y`. Several more placeholders exist in the `+%` syntax, making custom formatting trivial.

To measure the relative execution time of commands as delta between absolute start and finish times, two timing facilities are available. The first is the Bash wrapper built-in `time` showing real elapsed time of a command with sub-second precision such as `time sleep 2`. The second is a more portable and sophisticated wrapper command with the same name that is shadowed by the built-in, and thus must be referenced explicitly, such as: `/bin/time sleep 2`. When supplied the option `-p`, its less readable output matches that of the built-in, providing portability also in that sense.

The performance of program execution as time-based metric is inherently resource-dependent. It is not trivially possible to slow down a processor or a disk in case certain situations should be simulated in a controlled environment. Per process, the priority can be influenced with the niceness level through the wrapper command `nice -N` with N between -20 (highest priority) to 19 (lowest priority). However, this only influences the relative priority and depends on the chosen scheduler, by defaut the Completely Fair Scheduler (CFS) in Linux. The `cpulimit` command (that needs to be installed separately) is helpful in this regard. It can act both as a wrapper command for new processes and to reconfigure running processes identified by a PID. With `cpulimit -l 5 <program>`, it would execute a program with 5% allocated CPU share.

On the networking level, throttling data transfer speeds is possible by introducing Sluice into the pipeline, which is another program to be installed separately beyond the typical base installation. For example, `cat <file.txt> | sluice -r 100 | ...` ensures that all data arrives at the end of the pipeline with a constant rate of 100 bytes per second. The throttling mechanism works with dynamically sized buffers, hence high speeds may lead to not all input data eventually arriving.

To run a command in regular intervals and compare the output in an interactive session, it can be watched with another wrapper: `watch -n 1 ls -l`. Complex commands should be quoted, as in: `watch -d -n 1 "pstree | grep x"`.

Non-interactive scheduling of commands based on absolute days and times can be achieved with Cron, or based on relative times pragmatically by a prepended sleep. Interacting with Cron primarily works by editing its plan (per user or superuser) with the `crontab -e` command, using the default text editor as described in the section on editing files. This requires discipline to produce a file in the right format, consisting of lines that are either comments (starting with `#`) or scheduled commands consisting of a time pattern and the full command line or environment variable settings. Viewing (listing) an existing plan is possible with the corresponding `crontab -l` command. A rarely used and rather destructive option is `crontab -r` to remove the plan for a user entirely.

The use of Cron requires understanding that as it runs as a central daemon, its time zone and path settings might be different from the user environment. To mitigate these issues, the standard Cron implementation supports setting and overriding environment variables (`PATH=...`) in addition to scheduled commands (e.g. `* * * * * <per-minute-command>`). The five time specification fields refer to the minute, the hour, the day of month, the month (counting from 1) and the day of week (starting with 1 for Monday but also accepting 0 for Sunday). An asterisk signals an unconditional invocation. Cron invocations are logged with a *CRON* tag into the system log file `/var/log/syslog`, which is useful to consult if a command did not run although it should have.

The behaviour of Cron is that missed schedules, often due to system downtime, are not repeated, and neither are commands that temporarily failed, for instance, due to a network transmission problem. Another restriction is that fine-grained scheduling on a per-second basis is not possible, unless with the workaround of using multiple invocation with sleep prepended if the interval is a divider of the 60 seconds of a minute. There are more advanced but not yet as widely installed alternative implementations such as `fcron`, `xcron` or `anacron` that overcome at least some of those limitations, although sometimes by introducing additional restrictions. If SystemD is available, then timer units can also start service units on specified times including with second resolution. SystemD is described later on in the system administration section.

When, instead of being triggered by time, events should rather run in response to certain general events, it is suitable to have a monitoring command that blocks until the event happens and then lets the next command run. Specifically for file system events, running `fsnotifywait -e modify /tmp/myfile` would block until a content modification happens with the specified file. This functionality can be used to trigger data processing after data arrival in an event-driven way.

In summary, the shell offers a plethora of tools related to retrieve absolute and relative (delta) time information and scheduling. Some of those might not be enough for production scenarios, and therefore especially for complex workflow scheduling there are alternatives that are discussed in later chapters of this book.

> Commands to repeat in alphabetic order: crontab, cpulimit (external), date, fsnotifywait, nice, sleep, sluice (external), time (built-in & executable), watch
>
> Environment variables to repeat: $LANG, $PATH, $TZ

5.4.4 Managing data in files and directories

The previous groups of tools were either command-centric or produced minimal amounts of transient data. Retaining the data and organising structures to maintain and use the data become important complementary activities that require a thorough understanding of the file system layout in operating systems including data and metadata. The subsequent paragraphs thus provide a guide to working with nested directories and files.

Directories are organised in trees of arbitrary depth, with files being leaf nodes. Each directory contains two pseudo-entries: a reference to itself (`.`) and a reference to its parent directory (`..`). This is even true for the topmost directory (`/`) for which both references are identical. The path to a leaf node flattens this tree structure. The leaf node name itself is referred to as

basename, whereas the remainder of the path leading to it is called dirname. Hence, the namesake commands `dirname a/b/c` and `basename a/b/c` would display `a/b` and `c`, respectively.

On most file systems, metadata on both files and directories contains of three sets of dates (creation, last modification and last access). This means that reading files causes write modifications on the file system, which might affect the underlying device's wear level. Apart from that association, reading and writing are thoroughly separated through permissions, which is the next information kept in the metadata. Finally, the ownership, the size and the name of a file or a directory as well as extended attributes complete the metadata.

The size of a directory represents the number of files and subdirectories contained in it. The size might not be reported accurately depending on the file system. For example, on the common *ext4* file system, an empty directory is 4 kB in size. This number remains constant until around 340 entries, when it grows to 12 kB, and then further in 4 kB increments for further hundreds of entries. The reported size of files corresponds to the volume of data stored in them, although files with explicit sparse data (i.e. with gaps in between significant data portions) may occupy less space on the storage medium.

The permissions are represented as a triple referring to the assigned owner of the file, users of the assigned group, and all other system users. Each set of permissions in turn consists of the read permission (`r`), write permission (`r`) and execute/search permission (`x`). The latter one refers to the permission to execute a file or to search through a directory and may be replaced by `t` for directories, which implies searching but with the additional constraint that only owners of the file in that directory (and the super-user) may delete it. The directory for storing temporary information (`/tmp`) is a case of `rwt`. Permissions can also be represented numerically as sums of permission bits, with the formula `x=1,w=2,r=4`. Hence, a permission of `755` (owner: `rwx`, group/others: `rx`) refers to normal directories and `644` (owner: `rw`, group/others: `r`) to normal files.

Typing `pwd` reveals the current working directory of the shell, thus providing an entry point to the file system. The output should be consistent with the environment variable `$PWD`. To find out the effective location, even in the presence of symbolic links, `pwd -P` can be used with confidence. Directly after login, the user's home directory is set as working directory. After logging in, the user can traverse the file system and therefore the working directory can be changed anytime with the command `cd <dir>` (change directory). The command without arguments (just `cd`) resets the working directory to the home directory. Special shortcuts are `cd ..`, changing to the parent directory, and `cd -`, changing back to the previous directory. A user can only change into directories with appropriate `rx` permissions. Moreover, shell sessions might be

restricted to prohibit directory traversal at all, especially in case the shell is launched through `bash -r`, although this is rarely seen in practice for real system users. Such facilities are rather used to lock down user accounts assigned to automatic program execution.

With the `ls` command, all files and directories within the current working directory are displayed along with their most important metadata, such as the last modification date. Common invocations include the options `-l` for the long view with all properties and `-a` to show all files including hidden ones, i.e. starting with a dot like `.hidden`. The physical occupation of blocks on the storage medium is shown with `-s`, and the reported sizes can be made human-friendly with `-h` similar to the same option in `df`/`free`. The output can be furthermore sorted by size (`-S`) or by modification date (`-t`).

With `stat`, all relevant metadata of files and directories such as size and timestamps are summarised. To get a tree-like view or calculate the disk usage of all subdirectories and files contained within, the `tree` and `du` commands complement its functionality. The latter command is typically invoked as `du -sh` to summarise disk usage in human-readable form.

Files and directories can be copied with the `cp [-r]` command and removed with the `rm [-r][-f]` commands, requiring recursive operation for directories and optionally forced mode to avoid confirmation prompts in batch scripts. The behaviour of the copy command differs depending on whether the target is an existing directory or not. For instance, `cp file1 dir/` assuming the directory exists copies the file into the directory under its original name (i.e. `dir/file1`), whereas `cp file1 file2` create a copy of the file under a different name in the same directory. If multiple source files are specified, the target is required to be a directory in any case (`cp file1 file2 dir/`).

While `cp` duplicates the content of files and directories on the storage medium, symbolic links (*symlinks*) can be used to create only references without requiring additional space apart from a single file system entry. The canonical invocation is `ln -s <source> <target>`, with source being an absolute path or relative to the target. The output of `ls -l` then shows an arrow in the form of `<target> -> <source>`. Apart from that, the link can be used normally just like the source file or directory, and removing it does not affect the source.

Renaming and moving works with `mv` for files and directories, and with the regular expression-capable `rename` for larger collections. The same behavioural switch as with file copies applies. For instance, `mv file1 file2` renames a file, whereas `mv file1 dir/` moves the file into the specified directory assuming it exists. New directories would be created with `mkdir`. The trailing slash shown at the end of directory names can be omitted. They are shown to clarify the status of a name as directory, and it is considered good practice to include such slashes in own commands.

Working with files across computers can be facilitated over the secure copy (`scp`) command SSH as previously mentioned. It is typically invoked with a local and a remote component. For retrieving a remote file to the current working directory, the syntax is: `scp <user>@<server>:/path/to/file .` including the significant final dot. As the default path refers to the home directory, any remote file located in there can be specified without absolute path, as in: `scp <user>@server:fileinhomedir .` again with the final dot. For the inverse direction, pushing a local file into the remote location, the syntax is `scp localfile <user>@<server>:` with the significant final colon. Called with the flag `scp -r`, entire directories can be copied in either direction. If a directory is specified but the flag is forgotten, the command complains accordingly.

File attributes in terms of user, group and other permissions are modified with the `chmod` command, using either symbolic or numeric syntax. Typical examples include `chmod 755 <dir>` and `chmod u+rw,g+r,o+r <file>`. The default permissions for newly created files are typically set to the difference 022, meaning that write access from group members or other users is not possible, or whatever is configured with the tool `umask` per shell session. Advanced attributes such as undeletable or append-only, if supported by the underlying file system, can be set with `chattr` and listed with `lsattr`. Checksums of files can be created with the simple `sum`/`cksum` tools or with their more robust alternatives `md5sum` and `sha1sum`. These checksums can be used to determine whether the file content has changed including after a corrupted or interrupted transfer. Differences between two files can be produced with `diff` (contextual) as well as with `cmp` (byte for byte).

As outlined in the concepts, files can be compressed to save space on the storage medium or during transmission. Commands such as `unzip` or `tar xvf` are useful to extract their contents, while packing is achieved with `zip -r <*.zip> <files/folder>` for ZIP files and `tar czvf`/`tar cjvf` for archives with the common GZip and BZip2 compression schemes, respectively. The `a` flag selects the compression scheme automatically based on the file extensions, which is useful when switching schemes often, involving also the supported LZMA, LZIP, LZOP, XZ and ZStd. Uncompressing works similarly with the commands `unzip` and, again automating based on the file extension, `tar xvf`. Many applications can work with compressed archives transparently at least in read-only mode; this applies in particular to text viewers and editors.

Commands to repeat in alphabetic order: cd, chattr, chmod, cmp, cp, diff, du, ln, ls, lsattr, md5sum, mkdir, mv, pwd, rename, rm, scp, sha1sum, stat, tar, tree, zip/unzip

Environment variables to repeat: $PWD

5.4.5 Creating, viewing and editing files

The `cat` and `tac` commands output the contents of a text file or multiple files (concatenated, hence the name) in forward and backward order to the standard output. The command pair `head` and `tail` are also aimed at line-based formats and display only a portion at the beginning or the end, respectively. A useful option for `cat` is `-n` for numbering the output lines. One might argue that `tac` should have the same option symetrically, but while it does not, it is easy to emulate, for instance, with the command combination `tac <file> | cat -n`. The most common option for `head` and `tail` is `-`*N*, with *N* being the number of lines to reproduce, by default 10. This technique of dynamic options has grown historically and can also be expressed more according to conventions by `-n` *N*.

With binary files, all of these text-assuming output commands might mess up the screen unless proper options are given such as `cat -v`. Hence, the `less` command provides a more convenient view of portions of a file including a binary-safe mode, and `hexdump -C` gives a detailed account of binary contents on the byte level. All file-related commands take one or multiple filenames as command-line argument, in addition to further options, as in `cat -n <filename>` (short for `--number`).

The `dd` command is useful for working with binary files. It allows for creating files with null content or random content as well as partial copies, such as: `dd if=/dev/zero of=/tmp/mynull bs=1 count=100`. For that tool, the options do not take leading dashes. They refer to the input file, output file, blocksize and number of blocks, respectively. The special device files usable as data sources are `zero` and `random`/`urandom`, and the one usable as blackhole data sink is `null`. Custom FIFOs (first in first out queues) as temporary buffers allowing for both writing and once reading the same content may be created with `mkfifo`. Files can be created by output redirection, such as `echo "text" > <file>`. Empty regular files can also be created with the `touch` command.

For interactive text file content modification, several text-mode editors exist. While working with them might not be easy in the beginning for people not used to text-mode tools, they are inherently powerful and can be found ubiquitously across systems so that it pays off learning them, especially as an engineer, to avoid time-costly roundtrips between a local editor and file copy commands. On the other hand, the file synchronisation might be automated, avoiding the need for editing in such a way. In practice, constraints differ and thus a basic mastering of these editors is still recommended. Common editors are `vi` (and its modern cousin `vim`[5]) as well as `nano` and `joe`. If the choice of editor does not matter, the `editor` command always works and brings up one that is installed. On the other hand, this alias is sometimes

[5]Vim website: `https://www.vim.org/`

invoked automatically, and it might be necessary to understand at least the basic commands in these three editors to not be surprised by the appearance while being completely lost in them.

With `nano`, the user gets to see a title line containing the name of the editor, and two footer lines containing hints for the most important keyboard combinations. Specifically, Ctrl+x quits (and asks whether changes should be saved if any), whereas Ctrl+o saves changes without quitting. With `joe`, or Joe's Own Editor, as it announced in the footer line on startup, the corresponding combinations are Ctrl+k;q for quitting with asking for saving changes, Ctrl+k;x for quitting with unconditional saving, Ctrl+k;d for saving without quitting, and Ctrl+c for an attempted quit without saving with asking in case anything has changed.

In `vim <filename>`, there is a concept of editor states (modes). As a result, text editing works within bounded sessions. An editing session starts by pressing the i key, and the escape key Esc ends it again. Direct character-by-character text modification is only possible within the editing session, whereas control commands work in between sessions, including after startup of the editor. These control commands allow for rule-based editing especially of larger portions of text.

Among the often-used control commands are copy (v/V) and paste (p/P). Similarly, this applied to navigation commands: gg jumps to the top, G to the bottom. Pressing a number followed by the arrow key jumps in the indicated direction by that number of rows or lines, for instance, 3↑ would place the cursor three lines above the current line. Numbers can also appear in other contexts as multipliers. Search works by pressing the slash key / followed by the search term (that can be a regular expression) followed by the Enter key. Working outside of the editing session also applies to further modification commands beyond paste. The command x deletes one character; multiple characters can be deleted by indicating the distance with a number and the direction with the cursor (d;number;←), and dd cuts a whole line. An often-used advanced command is the visual block selection (Ctrl+v and pressing cursor up/down ↑↓) along with insertion of characters at the beginning of the block (I; characters; Esc). This can be used to comment out a whole block of lines with the comment characters defined in a programming language or data format (`#`, `//`, `%` among others). For more rule-based editing, custom regular expression may be applied, for instance, with `:%s/before/after/`.

A whole range of such commands starting with colon (:) are available, including the program execution with `:!<program with arguments>`, which temporarily pauses the editor until the command returns. Entire files can be pasted with `:r <filename>`, and as the logical combination of the two commands, `:r! <program with arguments>`, executes a program and pastes its output into the currently edited file. For more productive editing with mul-

tiple views on the same file or different files, there is also `:split`/`:vsplit` to create new panes between which the user can navigate with the sequence Ctrl+w;w. The command without parameters creates another view on the same file, whereas `:(v)split <filename>` directly loads another file for concurrent editing.

The colon commands also govern how to save changes and leave the Vim editor, similar to what was explained for Nano and Joe. A `:w` command saves all edits to a file. A new file can be created by starting `vim` without arguments and the command `:w <filename>`. The command `:x` unconditionally saves and quits, `:q` quits a pane the editor if being in the last pane but asks if changes need to be saved, and `:q!` unconditionally quits without saving, potentially losing changes.

Encodings are an important concern when working with data files, especially those downloaded from various Internet sources. The canonical encoding should be Unicode, with its standard representation UTF-8. Sometimes, legacy encodings appear and are either handled on a case by case basis by specifying the corresponding encoding at each file open operation in Python, or the file is converted before use. With Vim, differently encoded files are shown as `[converted]` upon startup, and details about the file encoding can be shown with `:set` watching for the `fileencoding` setting. The appropriate command to fix the encoding is `:set fileencoding=utf-8` followed by a save `:w`.

> Commands to repeat in alphabetic order: cat, dd, editor, head, hexdump, joe, less, mkfifo, nano, tac, tail, touch, vi/vim

5.4.6 Networking

A number of commands are very useful to know and to master in networked environments beyond SSH and ping: `netstat`/`ss` and `netcat` for basic networking interaction, and `wget`/`curl`[6] to fetch files from the web and interact with web services.

An invocation of `netstat -ltp` shows all port numbers occupied by TCP-listening services, including the service names of those that run under the same user account. The parameters relate to showing listening services (`-l`), restricting to TCP connections (`-t`) and showing the program names (`-p`). Program names are only shown for accessible processes, i.e. own processes of the calling user or all processes when the tool is invoked by the super user (root). Even without this option, the PID is shown, allowing stray processes to be terminated to liberate port numbers. Depending on the service implementation, the port may still be occupied for a number of seconds before it is finally liberated. Another useful piece of information is the bind address.

[6] Curl: `https://curl.se/`

If it says `::1` or `localhost`, the service is only accessible from localhost; or otherwise, from outside as well. Bind hosts are resolved unless numeric display (`-n`) is requested.

The command `netcat` can be used to test simple client/server connections. First, `netcat -l -p <port>` starts listening on a certain port number. A `netstat` on a second terminal can verify that the service works. Then, `netcat <host> <port>` connects to this service, allowing for bidirectional communication. The host can be set to `localhost` if running on the same host as the service, and it can also be set to an IP address such as `127.0.0.1`. Of course, the client can also be used to communicate with existing services, as long as they accept a text-based protocol, by specifying a different hostname and port number.

With `wget <url>` and `curl <url>`, it is possible to download files from HTTP servers. The Wget command by default saves the file, whereas Curl pastes its contents on the terminal, making it suitable for rather small files unless additional parameters are set. Additionally, Curl makes it trivial to interact with web services, including the ability to post data. For instance, the command `curl -X POST "https://httpbin.org/post?a=b" -H "accept: application/json"` submits a JSON request to a web service that parses the URL arguments and returns an informational JSON structure about them and other arguments not used in this example. Some web services also require the content type to be set in order to recognise posted content. Setting another header solves this problem, as in `-H "content-type: text/csv"`.

For security reasons, it should be pointed out that the often-seen combination of curling and executing a script from untrusted sources (e.g. `curl https://... | sudo sh`) on trusted systems is discouraged. Such commands should be decomposed, executed manually and complemented with a brief validation of the file contents. This also applies to any advice given on the Internet on running certain scripts and commands.

On the other hand, too tight security often gets in the way of learning commands. In case a certificate cannot be validated over an HTTPS connection, both commands allow for switching off the check, at the expense of reduced security: `wget --no-check-certificate` and `curl --insecure` (alias `curl -k`).

Commands to repeat in alphabetic order: curl, netcat, netstat, ss, wget

5.4.7 System administration

Administering a system means being responsible for it. The system in question can be a data scientist's workstation or notebook, a physical server hosted at the company premises or a dedicated data centre, a virtual machine obtained

from a cloud provider offering computing infrastructure as a service or a container running on any of those. Typical administration tasks include regular housecleaning tasks, i.e. removal of cruft such as old files and stray processes to reduce the resource utilisation, keeping the system up to date and secure, the installation of additional software needed for being productive, and ensuring automation and auditability. Sometimes, unplanned maintenance needs to be performed, for instance, restarting a service that crashed, evidently after having found out about this situation.

After having obtained information about the system resources, the first step on any system is usually awareness about what flavour of the operating system is running. This knowledge is helpful when consulting documentation on any problem or when looking for software packages to install. If the operating system is self-installed, this knowledge is usually given, but otherwise it needs to be obtained. Due to many evolving flavours, there is no systematic way to learn the differences in operation and administration. Rather, using different systems once in a while helps in building up experience.

On Linux systems, there is also no systematic way but rather a heuristic approach. If the configuration file `/etc/apt/sources.list` exists, it hints at the presence of a Debian-based system, e.g. plain Debian[7] (indicated by the existence of `/etc/debian_version`) or Debian derivatives such as Ubuntu. These flavours or more specifically these distributions have emerged as most popular ones over time. They have existed since the mid-1990s and since the mid-2000s, respectively, and are largely maintained by volunteers around the world, with plenty of forums and mailing lists available for help. If the file `/etc/lsb-release` exists, it may give further information about the operating system distribution. Other popular distributions include Arch Linux, indicated by `/etc/pacman.conf`, Red Hat/CentOS and SUSE. Once a flavour or distribution is known, it can be put to work.

Providing services on a computer or performing complex data analysis tasks inevitably leads to long-running processes. Unless a computer is rebooted or the process crashes, the execution time of a long-running process may be in the order of hours, days or even months.

Long-running processes ideally provide their own means for inspection, such as status and progress messages, log files or dashboards. In absence of these means, to find out what a process is doing, the generic `strace` (system call trace) wrapper command provides insights into running processes on the operating system call (*syscall*) level. Although not all activities - such as complex numerical calculations and other internal logic - show up this way, it is possible to understand much of the OS-interfacing behaviour of the processes monitored with the command prepended to the usual command name and arguments. For instance, `strace ls -l >/dev/null` reveals how the list of files and subdirec-

[7] Debian universal operating system website: `https://www.debian.org/`

tories is read from the current working directory, potentially following child processes (`-f`) or attaching to a running process (`-p <pid>`). The wrapper also shows the wrapped command's writes to the terminal on standard output. The tracing information itself is written to the standard error channel, and due to output redirection, the tool's output is suppressed so that both do not mix and all terminal output comes from the tracing. Somewhere in the middle of much output, the experienced user can then spot an internal operating system call called `getdents64()`, the purpose of which is to get directory entries. These system calls are functions implemented in the programming language C within the Linux kernel. Even when programming in C close to the system, they are usually not invoked directly but rather through portable wrapper functions as part of the libc system library, in this case: `readdir()`, and those in turn are wrapped by the various Python modules close to the system, such as `os`, in this case by: `os.listdir()`. In the reverse direction, this allows for finding out what exactly a Python program is doing on the OS level.

Most operating systems do not have a concept of persistent sessions. All processes are lost when a reboot occurs. Moreover, when starting processes through SSH, they are often terminated along with the session, whereas it would be desired to keep them running automatically without having to invoke them through a screen wrapper. In case they crash, perhaps they should be even restarted automatically. Supervising and autostarting applications represent an important concept to achieve that and to ensure continuous service delivery. This applies to both user-level and system-level services. Technically, it involves setting up appropriate launch scripts and unit files.

With SystemD as supervisor, a system user first needs to be given the permissions by the super user to host long-running services with the command `sudo loginctl enable-linger <username>`. The user can then deploy and start a unit file with the unprivileged commands sequence: `cp my.service ~/.config/systemd/user` to register the unit file in the default location, followed by `systemctl --user enable my.service` to activate it upon next boot, and finally `systemctl --user start my.service` to also start it right away. Whether the service works correctly can be verified with the command `systemctl --user status my.service` and ultimately by checking the listening port with `netstat`. The service unit file content may be as follows, not taking automated restarts into account:

```
[Unit]
Description=My service
[Service]
ExecStart=python3 ...
WorkingDirectory=/home/username/mydir
[Install]
WantedBy=default.target
```

If the command is not a long-running service but rather something that should run in defined intervals, a Cron-like regular command invocation is possible by setting up an additional timer unit that references the sservice unit by name, either explicitly with the Unit specification, or implicitly by having the same name minus the suffix (i.e. `my.timer` in the example).

```
[Unit]
Description=My timer
[Timer]
OnUnitActiveSec=30 # or OnCalendar
AccuracySec=1
Unit=my.service
[Install]
WantedBy=timers.target
```

Several commands are related to the management of such system-wide services under the responsibility of the super user, including privileged services running as root. For all of those, the effective process user is set to root, expressed by the environment variable `$USER`. The command `sudo service --status-all` gives an overview about the registered services and indicates active ones with a + sign. A service (for instance an Apache web server listening on port 80 and thus required to be privileged) might be hanging and requires a restart. This requires knowing the service name and furthermore privileged execution through the `sudo` wrapper command, as follows: `sudo service apache restart`. The command informs the supervisor service to perform a lifecycle operation (start, stop, restart, reload configuration, status check) on an application service.

The same wrapper command Sudo can be used to edit global configuration files, for instance in the `/etc` folder that is write-protected to ordinary users, by carefully running `sudo editor <global-file>` or variations thereof. While incorrect use of an editor may always lead to a loss of data, privileged execution may furthermore lead to an unusable system, and therefore extra care needs to be applied to ensure the file is backed up beforehand. This can be achived with a simple `sudo cp <global-file> <global-file>.backup<date>` before editing, although more sophisticated version control and backup strategies exist and are explained later.

The way Sudo authenticates users is by asking for their system password. This makes it tricky to use Sudo in non-interactive scripts because the asking is interactive. As first precondition to gain privileged access, this password must be correct, but as second precondition, the user or a group the user is member of needs to be registered for being able to run Sudo in the file `/etc/sudoers.conf`.

Privileged command execution may also be necessary for other tasks. Among them is administering users, primarily the addition of new users and groups

with `adduser <username>` and `addgroup <groupname>` on Debian-based systems, and the similarly-named `useradd` and `groupadd` on other systems. For testing a new application, isolating the test by creating a custom user and group may be useful. To achieve the isolation, one would first create a new system account with `sudo adduser [--shell /bin/bash] [--home /home/testuser] --comment "" testuser`, with explicitly chosen login shell and home directory in case the default values are not applicable. This command first asks for the Sudo authentication, then twice for the new user's password, and with that information it creates the user, a corresponding group, and a home directory with default (skeleton) content. (It should be noted that the comment field may still appear as `gecos` on older systems, derived from human-readable information attached to the user account originally appearing in the General Comprehensive Operating System (GECOS).) The new user does not have Sudo privilege, and therefore an application run under that account can only cause minimal damage. To proceed, the interactive shell session switches to that new account (`sudo su - testuser`), testing can occur, and the test session can be closed with `exit`. Finally, the custom user can be removed along with the generated group and all files with `sudo deluser --remove-home testuser`.

Privileged commands are also required to change modes and ownership of files owned by other system users (`sudo chgrp`, `sudo chown`) and to send messages to all local users with active shell sessions (`sudo wall`) on a shared login system. Executables can be configured to gain elevated privileges automatically upon execution with the set-user-id and set-group-id bits (SUID/SGID), as in: `chmod u+s <executable>`. Careful application design involves dropping these privileges as soon as the privilege-requiring operation has finished.

Lastly, system administration also involves checking log files occasionally. System-wide logs are stored in `/var/log`. Many of them are rotated as they can grow big, marked by a number after the log file name. Of primary interest is the file `syslog` that contains kernel and application messages. Appended lines to that file may be followed with `sudo tail -f /var/log/syslog`. The log directory also contains the information base for commands such as `last`, with binary files such as `lastlog` and `wtmp`, but primarily contains text files, sometimes in subdirectories for more complex applications. In addition, hardware-related issues might be found out with `sudo dmesg` (diagnostic messages).

Commands to repeat in alphabetic order: addgroup/groupadd, adduser/useradd, chown, chgrp, dmesg, loginctl, service, strace, sudo, systemctl, wall

Environment variables to repeat: $USER

Repetition

1. What is the difference between a *TERM* signal and a *KILL* signal?
2. When grepping for a list of processes, the `grep` command itself appears in the list. How can this be avoided?
3. How to count the number of current and past login sessions to a system?

5.5 Shell programming

The previously given information on shell variables, shell commands and useful tools is sufficient for occasional use of a system. For automation, a higher-level programming language such as Python may be used. Sometimes, though, data scientists are confronted with complex shell scripts containing all sorts of programming constructs. Bash in particular is therefore not only a command prompt but also a programming language on its own. Learning yet another language might not be favoured, but being able to understand this language in order to customise or extend scripts is nevertheless a useful skill. In this section, basic shell programming concepts and constructs are conveyed.

5.5.1 Vocabulary and interaction with scripts

Bash contains a number of built-in commands that, together with the executables on a system and user-defined functions, form its vocabulary. An overview can be obtained with `help` as well as more comprehensively (but missing some commands unfortunately) with `man builtins`. The vocabulary can be divided into flow control, job management, directory bookmarking, arguments parsing and other groups.

One characteristic behaviour of shell scripts is that, while syntax errors are fatal, execution errors in commands signalled with non-zero exit status are ignored by default. The offending lines are then simply skipped, and the errors in these commands are ignored. In many cases, a stricter behaviour closer to that of most programming languages is desired. Shell scripts therefore often start with the command `set -e`, which makes the shell exit immediately whenever a command returns a non-zero status.

In turn, this behaviour has led to the convention that tools as well as user-defined functions return a zero exit code upon successful termination, and a documented non-zero exit code otherwise. In Bash, `exit 1` would adhere to this convention. In Python, using `exit(1)` or, to include an error message, `exit("message")` achieves the intended behaviour. In other languages, similar constructs are available to signal a non-successful termination to the shell. The special variable `$?` reports about the exit status of synchronously executed child processes.