There are also conventions on how to parameterise tools and functions. Historically, four main flavours developed. These are short dashless parameters (`<program> x`), long dashless parameters (`<program> extract`), short dashed parameters (`<program> -x`) and, finally, more self-explaining but requiring more typing effort, also long double-dashed parameters (`<program> --extract`). The first flavour is still in use with some historic tools but has largely fallen out of favour, such as `ar x <archive.ar>` for uncompressing a compressed file. The second flavour is typically used for subcommands offered by a complex command, such as `git clone`. The third and fourth flavour are very common and often go hand in hand as equivalents, such as `vim -h` equal to `vim --help`. Short options can also be combined, such that `-x -y` is equivalent to `-xy`. Due to the limited set of characters in the Latin alphabet that are traditionally used for the third flavour, complex programs only assign such short parameters to the most important options. Sometimes, they match with the first letter with the long parameter but sometimes they do not, something to be aware of. For those last two flavours, there are parameters with and without argument values (e.g. `-s` or `--strict`, either binary or with a default value, versus `-s high` or `--strictness high`). The argument value may often also be appended with equal sign, as in `--strictness=high`, and for short-style parameters also without space (`-shigh`). Its type (string, numeral, boolean, filename or URL) is defined by the application. In addition, commands may take a variable number of arguments, often files or URLs, given at the end of the invocation line. Hence, it is not unusual to see a command with mixed parameters and arguments in the following form: `prog -xy --strict a.py b.py`. In this book, for brevity and rapid practice, the short-style parameters are used in most places, sometimes complemented by their longer equivalents.

Shell scripts can thus be parameterised with arguments. Each parameter is given a numerical variable, with `$0` referring to the script itself and `$1` to the first parameter. If the parameter is not supplied, the variable remains unset and empty. The special variables `$@` and `$*` refer to the whole argument vector containing all parameters as space-separated list. The difference is in further passing the arguments internally to functions or commands. The variant `"$*"` joins the arguments into one (i.e. becomes `$1` in the called function or command), whereas `"$@"` keeps the list intact and passes each argument separately.

More sophisticated parameter parsing is possible with the `getopts` instruction. This command is called repetitively on an arguments list and is able to process short-style parameters with and without argument values. For instance, the following script can be called as `script -a -b -c x` or in various combinations thereof and allows its subsequent logic to be adapted to the parameters. The variable `$var` contains one of the valid parameters in each

iteration, and `$OPTARG` the argument value if indicated with a colon as is the case for c (`c:`). The loop and condition contained in this script are explained in detail further below.

```
while true
do
  getopts "abc:" var "$@"

  if [ "$var" = "?" ]
  then
    break
  fi
  echo "** $var [$OPTARG]"
done
```

Testing for empty or unset variables can be done with `test -z "$1"` or alternatively `[ -z "$1" ]`, a condition that returns 0 if the variable is empty, and 1 otherwise.

Text including variables is output by `echo`. The counterpart to assign variables with user-defined input is `read`. A typical invocation is by including a prompt to tell the user what the input should be, as in `read -p "Prompt? " var`. The variable `$var` can then be used for further processing.

Dynamic command evaluation can be performed with `eval`, a command to be used sparsely and with extreme care concerning user input. For example, `eval "ls -l"` interprets the provided string argument as a shell command to execute, and `a=9; eval "sleep $a"` builds and runs a variable-dependent command. Many more built-in commands exist, but three groups are now explained in greater detail: job management, control flow programming, and function definition.

> Commands to repeat in alphabetic order: eval (built-in), getopts (built-in), read (built-in), set (built-in), test (built-in)
>
> Environment variables to repeat: $*, $@, $0..$n$

5.5.2 Job management

Jobs in the context of a shell refers to active processes on the OS level that have been spawned from that shell. In addition to global process management tools (`pidof`, `kill` etc.), the shell offers additional management functionality for the commands directly under its control.

The command `jobs` displays all jobs. For instance, running `sleep 10 &` in the background immediately followed by `jobs -l` shows the still running sleep command along with its PID and its shell-internal job number in long

format. In case the process terminates, it appears one more time with the status finished before disappearing from that list. Any backgrounded job can be foregrounded, taking over the terminal, with `fg <job>`. A command run in the foreground can be suspended by pressing Ctrl+Z, assigning it a job number. The job can then continue in the background with the command `bg <job>`. A foreground job can also be terminated with Ctrl+C. Hence, any job is in the status of running, suspended, or finished.

The running shell itself may be terminated with `exit` or suspended with `suspend`. The suspension blocks the shell entirely unless it receives a continuation signal (`kill -CONT <pid>|<job>`). With many active shells, finding out the PID of a just suspended shell is not trivial but becomes easier knowing that the output of `ps` shows the status `Ts+`. In a running shell, the special variable `$$` informs about the PID. Shell suspension is rarely needed in practice. It should be noted that the shell command `kill`, similar to `time` and `echo`, shadows the identically named executable and is therefore able to control both processes and jobs under the control of the shell.

The command `history` shows all shell commands executed in the current session as well as in previous sessions based on session recording.

> Commands to repeat in alphabetic order: bg (built-in), fg (built-in), history (built-in), jobs (built-in), kill (built-in & executable), suspend (built-in)
>
> Environment variables to repeat: $$

5.5.3 Control flow programming

Bash is an imperative programming language based on conditional branching and looping. Such constructs can be written across multiple lines but can also be shortcut by using the semicolon (`;`) in place of line breaks. *While* loops follow the form `while <condition>; do <command>; done`. *For* loops iterate over a sequence of values or a globbed list of files, as follows: `for <var> in <list>; do <command>; done`. Loops can be controlled with `continue`, jumping back to the first line of the loop body, and `break`, jumping out of the loop.

Sequences for iteration are parsed as strings divided by a separator, the internal field separator (`IFS`), which by default is set to the space character. This may collide with spaces in file names, not when iterating over files directly but for instance when reading a list of files from another file. Hence, the IFS can temporarily be set to something else which is unlikely to occur as character in filenames, such as line breaks. The following code achieves that: `IFS=$'\n'; for fn in ‘cat myfileslist‘; do echo "* $fn!"; done`. However, this

alters the field separator beyond the end of the command, so that it should be saved and restored (`ORIGIFS=$IFS; ...; IFS=$ORIGIFS`).

Branching and conditional execution may be based on `if` or `case`. The canonical form of a choice between two branches is then either `if <condition>; then <command>; else <command>; done` or `case <var> in <pattern1>) <command1>;; <pattern2>) <command2>;; esac`.

> Commands to repeat in alphabetic order: break (built-in), case/esac (built-in), continue (built-in), for/do/done (built-in), if/then/fi (built-in), while/do/done (built-in)
>
> Environment variables to repeat: $IFS

5.5.4 Shell functions definition

Functions are declared by simply typing the function name followed by a pair of parentheses, and another pair of curly braces containing commands with significant spaces in between. Similar to a shell script, a function may receive arguments, which are numbered starting from 1. Hence, the function declaration `func(){ echo "$1"; }` leads to a new command `func <argument>` in the running shell.

Functions are either invoked explicitly or based on events such as OS signals. The `trap` command can be used to intercept system signals such as SIGTERM and SIGCONT in addition to the two user-defined signals SIGUSR1 and SIGUSR2. For instance, `trap func USR1` runs the previously defined function whenever a signal is received, which can be verified with `kill -USR1 $$`. All active traps can be printed with `trap -p`. In practice, trapping is in demand for cleanup actions that need to be performed no matter whether a script is terminated regularly or irregularly.

> Commands to repeat in alphabetic order: kill (built-in), trap (built-in)

Repetition

1. What may happen if a user types `xyz` into a shell?
2. What happens when the command `kill $$` is invoked?

5.6 Python modules for OS interaction

Although Bash programming is great for automation at the operating system level, it may be in itself tedious especially for higher-level applications that

require structure, modularity and more powerful built-in functions. For those, writing the corresponding code in Python, invoking Python scripts from the shell or combining Python code with OS-level tools might be the better application design. Hence, in the following, six additional Python modules (out of the larger built-in module list[8]) of high relevance in the interaction between Python code and the OS are introduced, and complementary information of interacting with Python on the shell is given.

5.6.1 Running the Python interpreter

The default Python interpreter is cPython. It is implemented in the C programming language and can be invoked on the command line by its versioned interpreter name `python3` or, as used throughout this book, its alias name `python`. As the Python programming language evolves, checking the version with `python -V` is advisable. For pragmatic reasons, the book assumes the version to be at least 3.10, although previous versions 3.x should also work fine for most of the examples.

The Python interpreter enters interactive mode by default similar to Bash. More specifically, it follows the Read-Evaluate-Print-Loop (REPL) paradigm. First, it reads user input with the prompt indicating a new command (`>>>`) or a continuation of a previous line (`...`). Then, it evaluates the input in the form of an expression or a statement, such as variable assignments, mathematical formulas, functions, control flow instructions. Afterwards, it prints out the return value of any statement or expression, unless it is `None`, before closing the loop and reading the next input. The interpreter can be quit with the command `exit()` or with the keyboard combination Ctrl+d.

When invoked in the form of `python <scriptname.py>`, the interpreter instead executes the specified script in batch mode. Nothing is then printed unless the `print()` function is used or an uncaught exception is raised, and no interaction happens unless the `input()` function is used. A one-liner script consisting of one statement or expression or a sequence of those separated by semicolon can also be passed directly on the command line with `python -c "<command>"`. Similarly, `python -m <module>` loads a module and executes it as script, i.e. typically the guarded main code.

There are several environment variables and options that influence the behaviour of the Python interpreter. Not all of them can be documented here, but among the more often used ones are running `python -u` for unbuffered output especially when piping the output to another tool invoked on the shell; and informing about additional module search paths beyond the default (informed as `sys.path`) with the environment variable `PYTHONPATH`.

Python scripts can be made executable by using `chmod +x` and setting

[8]Python modules: `https://docs.python.org/3/library/index.html`

`#!/usr/bin/env python` as their shebang line. Due to Python being an interpreted language, similar to Bash, all Python scripts appear in the OS process list as merely `python`. The script name appears at least as second parameter so that processes can be distinguished with `ps x`, but using process management tools such as `pidof` or `killall` is then challenging. An external Python module `setproctitle` can be installed from PyPI (`pip install setproctitle`) or from the distribution repository (e.g. on Debian-based systems with `sudo apt-get install python3-setproctitle`) as solution. At the start of a Python script, calling `setproctitle.setproctitle("title")` then allows for giving a unique name.

In general, it is important to consider the effects of Python code on performance and memory use especially when handling very large volumes of data. Performance may slightly degrade especially for multi-threaded and iterative operations, which can be remedied by choosing the right interpreter and data-processing modules. Compared to the standard cPython interpreter, PyPy is optimised for higher performance and better threading support and can be invoked as `pypy3` to give a faster and more scalable drop-in replacement. Further alternative interpreters exist, notably iPython, which is known for its integration into Jupyter notebooks, and MyPy as an emerging interpreter with support for static typing enforcement. Memory usage in the interpreter may also increase due to data being represented with metadata in memory. Using libraries such as Numpy and Pandas that manage raw data without the metadata helps to avoid this issue.

> Commands to repeat in alphabetic order: python, pypy3
>
> Module functions: setproctitle.setproctitle (external)

5.6.2 Modules 'os' and 'sys'

The `os` module, along with its submodule `os.path`, represents the main interface to low-level operating system functionality.[9] It wraps many of the functions contained in the main system libraries to make them accessible from Python in a portable manner. Achieving portability requires discipline in using the module's OS abstractions without compromises. For this matter, differences in operating systems are masked, for example, path separators (`/` versus `\` as `os.pathsep`). On Linux, the `os` module is primarily the interface to `libc`, which in turn provides convenience functions to access low-level functions in the kernel itself. Therefore, calling one of this functions switches the execution context from user space to kernel space temporarily. This is not different

[9] OS module documentation: `https://docs.python.org/3/library/os.html`

from `print`, which is, however, used so often that it was turned into a Python built-in function instead of being exported through this module.

Among the commonly used functions is `os.system` to spawn a new process. On Linux, the command is interpreted as a shell command, so that all shell facilities such as pipes and redirections are available. An example invocation is `os.system("ls -l > /tmp/listing")`. Proper care must be applied to avoid using untrusted input to that function and for properly quoting the command. Other useful commands are `os.cpu_count` to get information about the possible parallelism in multiprocessing and `os.kill` to send signals to processes.

To interact with environment variables, the functions `os.getenv` retrieves values of variables passed to the Python interpreter with default values for unset variables, for instance, `os.getenv("LANG", "C")`. Its counterpart is `os.putenv` to update the values or introduce new environment variables. It should be noted that this only affects processes spawned from the current one. To effectively update variables within the running Python session itself, the dictionary-like data structure holding the environment must be modified directly, following the form `os.environ["LANG"] = "C"`. To avoid a spoofed user identity, `os.getlogin` is preferred over reading out the value of `$USER`, albeit it does not follow Sudo semantics.

Among the often-used functions related to directory trees are `os.getcwd` to report the current working directory, `os.chdir` to change to a different directory, `os.makedirs` to create a new directory unless it already exists, and `os.listdir` to report the directory contents. Helpful functions from the submodule are `os.path.join` to construct subpaths for the navigation, the reverse with `os.path.basename` to split off the rightmost file or directory name and `os.path.dirname` for the remainder, and `os.path.isdir`/`os.path.isfile` to check the type of entry when iterating over a directory as reported by `os.listdir`.

The `os` module is complemented by `sys` for some OS interaction, mostly related to the process itself.[10] This encompasses the handover of parameters to the Python interpreter or script (`sys.argv`), the search paths for executables to be spawned (`sys.path`) and the default communication channels for input, output and errors, respectively (`sys.stdin`, `sys.stdout`, `sys.stderr`).

To request the termination of a Python process, even multiple functions are available: The built-in functions `exit` and `quit` take either a numerical exit code argument (0 meaning success, otherwise failure) or a text message (`None` meaning success, otherwise failure message). Failures are treated as exit code 1, represented by a raised `SystemExit` exception, with appropriate error message, if specified, written on the standard error channel. These methods can be overridden in interactive sessions and are unavailable in some

[10] Sys module documentation: `https://docs.python.org/3/library/sys.html`

interpreters like iPython when running scripts. Therefore, it is advisable to use `sys.exit`, which takes the same parameters. Still, the exception may be caught by higher-level code, which is often the desired behaviour but as a consequence does not guarantee termination. Alternative options for forced termination are `os._exit`, taking only a numerical exit code, and `os.abort`, immediately terminating the process with a failure code.

> Module functions: exit (built-in), quit (built-in); os.abort, os.chdir, os.cpu_count, os.getcwd, os.getenv, os.kill, os.listdir, os.makedirs, os.putenv, os.system; os.path.basename, os.path.isfile, os.path.isdir, os.path.join; sys.exit

5.6.3 Module 'shutil'

This module[11] refers to *shell utilities* and represents the equivalent of many file and directory management commands typically invoked on the OS shell. These include commands to copy or move files (`shutil.copy`, `shutil.copytree`, `shutil.move`), to change their ownership (`shutil.chown`) or to get the cumulative disk usage of a directory path (`shutil.disk_usage`). To copy a single file, for instance, the syntax would be `shutil.copy("origfile.txt", "filecopy.txt")`. Like all I/O operations, it should be enclosed in a try-except block to check for typical errors such as disk full or insufficient permissions. Recursion is used by `shutil.copytree` and `shutil.move` if the second parameter is a target directory so that entire directory hierarchies can be copied or moved. The module can also be used to pack and unpack an archive such as a ZIP file or compressed TAR files (`shutil.make_archive`, `shutil.unpack_archive`). This resembles the invocation of the compression utilities in the shell, whereas for more fine-grained support, dedicated Python modules for compression and archiving are available.[12]

> Module functions: shutil.chown, shutil.disk_usage, shutil.copy, shutil.copytree, shutil.make_archive/ shutil.unpack_archive, shutil.move

5.6.4 Module 'tempfile'

The module enables the safe creation of temporary files and directories with unique names in a shared directory, by default the standard directory for temporary files, which is usually at risk for race conditions when multiple applications would create a file with the same name. On Unix-like systems, the directory for temporary files is `/tmp`, whereas on Windows, they can be found in

[11] Shutil module documentation: `https://docs.python.org/3/library/shutil.html`

[12] Python modules for compression: `https://docs.python.org/3/library/archiving.html`

`C:\Windows\Temp` and `C:\Users\<user>\AppData\Local\Temp`. The canonical call for the safe opening of temporary files for write access is to the constructor of the class `NamedTemporaryFile` with appropriate parameters. For example, in a web service to handle image uploads, the following code would ensure that all write access is redirected to an unprivileged area first: `f = tempfile.NamedTemporaryFile(suffix=".jpg"); print(f.name)`, followed by `f.write(b"..."); f.close()`. As long as the file is not closed, the attribute `name` of the object can be used to share the file content with other processes.

Module classes: tempfile.NamedTemporaryFile

5.6.5 Module 'argparse'

To facilitate the interface between shell and application, this module provides a structured way to access command-line parameters that influence the actions of the application. It is based on the convention that an application can be invoked with a binary parameter or flag (e.g. `app --strict`), a qualified parameter with argument value (`app --strictness 5`), subcommands (`app check ...`) and an arbitrary number of arguments (`app 1.txt 2.txt 3.txt`). The module provides the `ArgumentParser` class that is first constructed with information about the permissible flags and arguments. The information encompasses even typing information to inform that an argument takes only numeric values or references to files. Then, the parser object is applied on the standard Python interface for command line arguments, i.e. `sys.argv` excluding the script name itself, and sets attributes on the returned arguments object that the application can evaluate. A minimal use case would thus be as shown in the listing below.

```
import sys
import argparse
p = argparse.ArgumentParser()
p.add_argument("--url", default=None, help="URL to load")
args = p.parse_args(sys.argv[1:])
print(args.url)
```

Documentation about the supported invocation options is automatically created from the parser object and made available via the long-style `--help` parameter or its short-style equivalent `-h`. Moreover, in case incorrect options are supplied, an error message with a short synopsis of correct options is generated automatically, written to the standard error channel and causing a non-zero exit status. The module is therefore the first step in input validation, although further checks on the validity of parameters and arguments should be performed within the application code.

Module classes: argparse.ArgumentParser

5.6.6 Module 'subprocess'

Python processes can spawn synchronously executing (i.e. blocking) subprocesses via `os.system("command ...")`. This function returns the exit code and therefore lets the calling application code decide on the convention of zero meaning success and non-zero meaning failure how to proceed. However, this function is unable to return any output from the executed command. The output is instead just written to the standard output channel of the terminal. Moreover, it does not easily permit the execution in the background, instead requiring the use of multithreading for this purpose or appending `&` to the command but then returning immediately without being able to track when the command finishes. Hence, the `subprocess` module is a more versatile alternative, giving more control about subprocesses handling to the application engineer, in return for a slightly more complicated invocation interface. The canonical invocation behaviourally equivalent to `os.system` is as follows: `p = subprocess.run("command ...", shell=True, stdout=subprocess.PIPE)`. The return value is an object encapsulating information about the spawned process. This can be followed by `p.wait()` to wait for the command to finish, `print(p.returncode)` to inform about the success (0 meaning success by convention), and `print(p.stdout.read().decode())` to access the command's standard output.

Module functions: subprocess.run

5.6.7 Module 'socket'

The socket module allows re-implementing the Netcat command in Python and adding application-specific logic on top of it. On the server side, it permits the creation of a socket by calling `s = socket.socket()` and binding it to a network interface specified by IP address and port number, as follows: `s.bind(("0.0.0.0", 9999))`. The special IP address, equivalent to the empty string `""`, instructs the OS to bind to all interfaces. In practical terms, the service is then accessible from incoming connections originating outside of the computer. Alternatively, the address `"127.0.0.1"` can be used to allow only local connections. Once the socket is created, it can be used to listen to incoming connections (`s.listen(1)`) and to represent the client-specific socket along with client address information after a successful connection (`c, caddr = s.accept()`). The client object then offers basic network interaction with the `recv`, `send` and, eventually, `close` methods. Sending and receiving data works similar to reading and writing binary files. On the client side, the

interface is similar but omits the creation of additional sockets. A socket offers the `connect` method that mirrors `bind`, and then allows for sending and receiving directly on that socket.

While these socket methods mirror the low-level system functions, a slightly more comfortable interface is available. On the server side, the function `socket.create_server` creates a socket ready to accept without requiring the binding and listening steps; on the client side, the corresponding function `socket.create_connection` sets up the connection with control over timeouts.

The historic OS behaviour is that, whenever a program terminates, any port that it bound to as a server remains occupied for a short period of time. Restarting then usually fails unless precautions were taken to tell the OS to disable this reservation. The higher-level interface makes these easy to integrate, as in: `socket.create_server(("", 2000), reuse_port=True)`.

To access HTTP services, the Python equivalent of the Curl/Wget functionality can be achieved with the default module `urllib.request`, as well as the third-party module `requests`. Both provide extensive support for HTTP GET and POST requests, but are not explained in detail here.

Module functions: socket.create_connection, socket.create_server, socket.socket

Repetition

1. What would the command `python -c 'import os; print(os.getpid())'` do?
2. How to programmatically create a directory in an idempotent way, i.e. adding it if it does not already exist, and do nothing otherwise?

5.7 Package management

Customising and maintaining an operating system requires understanding the lifecycle of software and data, their representation as layered and strongly related packages and the use of package managers to achieve a desired system state. While essential functionality and many basic tools might be already pre-installed, several complex data-processing and analytics packages for data scientists require not only a one-time additional installation, but even tracking new releases systematically and upgrading the workstation or server accordingly. One advantage of using provisioned packages through appropriate package managers is that all necessary dependencies are automatically installed. This reduces the cognitive load and allows focussing on what functionality

should be present, without having to understand how that functionality is implemented. A second advantage is that, instead of having to trust dozens or hundreds of different download locations, many package repositories take at least basic precautions to avoid low-quality and malicious packages.

Packages emerge within certain ecosystems that are developed by communities. Some communities work close-knit almost like teams, while others are rather loose sets of people. Ecosystems are also either thematically focused on operating systems, languages or technologies, or they are rather universal. Navigating these ecosystems is not trivial. In this book, the emphasis is on Python packages, OS-level packages, and (in the subsequent section) Docker images. Understanding those concerning lifecycle, tool support and obstacles also opens the door to understanding others.

5.7.1 Python package management with Pip

When it comes to extending the functionality of the installed Python interpreter, the language-specific ecosystem of packages provides a huge diversity and generally a high quality of popular packages. The `pip` tool is the package installer for Python and the main interface to this ecosystem. Before the installation, the scope needs to be defined. A package can be installed system-wide by invoking Pip as super-user, or per user by invoking Pip as that user, or even confined to a single project's virtual environment in conjunction with additional tools such as `virtualenv`[13] and the derived `venv` library. To avoid clashes with packages already provided by the operating system, the use of `virtualenv` or `venv` can even be mandatory.[14]

On most Linux installations, the default directory for system-wide Python package installation from third-pary sources is in the `/usr/local` hierarchy, specifically `/usr/local/lib/python3.`X`/dist-packages`. Correspondingly, for per-user packages it is `~/.local/lib/python3.`X`/site-packages`, with X referring to the minor version (e.g. 10 for Python 3.10). The OS distribution itself may have placed its curated packages into the `/usr` hierarchy, specifically `/usr/lib/python3/dist-packages`, which is also understood by `pip`. By default, `/usr/local` overrides `/usr` in the module loading mechanism. The creation of a virtual environment is possible with `python -m venv <directory>` or alternatively `virtualenv <directory>`. It is then supplied with its own copies of the Python interpreter and package installer, and using them confines all installations to within the specified directory, specifically into the relative subdirectory `lib/python3.`X`/site-packages`.

The command `pip list` shows all installed packages, and a subsequent `pip show <package>` shows details including about where they are installed.

[13] Virtualenv website: `https://virtualenv.pypa.io/en/latest/`
[14] PEP 668: `https://peps.python.org/pep-0668/`

New packages can in principle be found with `pip search <term>`, but over time this has caused too much load on the servers, and therefore lookups now need to be done interactively on the package repository websites.

Python packages may depend on other packages such that installing a single package triggers the implied installation of many other packages. These are either pure Python packages that are simply placed into the corresponding interpreter folder, or native packages that are compiled on the spot in order to work on the computer's hardware architecture. There is only one category of dependencies with such packages, so that excluding rather optional ones is not possible; however, it is possible to exclude all dependencies for tests with the flag `--no-deps`.

Packages may not only contain pure Python code but also natively compiled code in other programming languages such as C. In this case, the installation will automatically trigger the compilation and placement of the resulting shared libraries. As a prerequisite, a C compiler along with other build tools (assembler, linker) needs to be present.

The canonical invocation for user-wide package installation from the global Python Package Index (PyPI[15]) follows the format `pip install <package>`. Popular packages for data science include for instance `pandas` or `dsfaker`. For better control over the versions, a concrete version of a package may be specified (`pandas==2.0.0`), or a range of permissible versions may be given. In more complex scenarios requiring multiple packages, it is a convention to create a `requirements.txt` file with one versioned dependency per line. The installation of all packages can then be automated with one command: `pip install -r requirements.txt`.

Details on using pip to install packages are documented on the Python website.[16] The complementary view on how to produce such packages from own code are documented as well.[17]

Commands to repeat in alphabetic order: pip, virtualenv

5.7.2 Advanced Python package management with Pipx and Poetry

While `pip` handles the heavy duty tasks of package management, by itself it is often insufficient for the needs of complex yet productive data science environments. Two extensions are available to ease the installation of applications and the handling of dependencies.

[15] Python package index portal: `https://pypi.org/`
[16] Pip user documentation: `https://docs.python.org/3/installing/index.html`
[17] Pip package production: `https://docs.python.org/3/distributing/index.html`

With `pipx`[18], executable Python applications can be installed from various sources that run in non-privileged, isolated directories on a per-user basis. This essentially combines the functionality of virtual environments, package installations and taking care of custom `PATH` environment variable settings, so that executables from the packages can be invoked directly in the shell. Moreover, `pipx` supports continuous development scenarios by referencing code repositories. For instance, to install the Datadiary package that summarises the outcome of machine learning processes, the command can be invoked to proceed with the installation directly from a Git repository: `pipx install git+https://github.com/itsayellow/ datadiary`. Hence, `pipx` is applicable to all Python packages that provide executables on the command line.

From the user perspective, the installation is therefore as easy as it can get. Still, providing own packages takes some effort in describing the packages, formulating dependencies and performing the publishing process on package repositories. With Poetry[19], providing custom Python packages is made easier in contrast to the conventional approaches of distutils/setuptools (`setup.py`, `pyproject.toml`). It allows for fine-grained specification of dependencies, caching during dependency resolution and publishing to PyPI or other package distribution sites.

Additional projects and distribution channels for Python packages exist, including Conda and the C++ reimplementation Mamba. They can be consulted when needed.

Commands to repeat in alphabetic order: pipx, poetry

5.7.3 Package management for other programming languages

Beyond Python, many programming language communities have developed ecosystems to distribute software. In typical data science scenarios, it is common to mix applications and libraries from these ecosystems. Through files, databases, services and containers, different software implementations can interact with each other despite differences in the implementation language. The following thus summarises how to install packages written in popular languages, using the respective mechanisms for selection and deployment.

Projects using R can rely on the comprehensive R archive network (CRAN)[20] to find additional packages. They are organised into libraries, represented by local directories. The syntax for the installation of a locally available package (archive file `packagename.gz`) from the shell is `R CMD INSTALL -l`

[18] Pipx website: `https://pypa.github.io/pipx/`
[19] Poetry website: `https://python-poetry.org/docs/`
[20] CRAN: `https://cran.r-project.org/`

`/path/to/library <packagename.gz>`. There are also alternative techniques to download packages directly from within an R script or interactive session, and those can also be combined with the shell invocation to access CRAN with the command: `R -e "install.packages('<packagename>')"`. An exemplary invocation would be with the `LGDtoolkit` package. By default, R installs packages system-wide into the directory `/usr/local/lib/R/site-library` and therefore requires privileged invocation with `sudo`.

JavaScript packages based on the NodeJS runtime are distributed via the Node Package Manager (NPM[21]). For instance, installing the JavaScript equivalent of the Pandas module is conducted with the command `npm install pandas-js`. Apart from the code, such packages have a package file in JSON format with metadata and recursive information about their own versioned dependencies, in two sets: for running the code and for development. When such a file exists in a project directory, simply calling `npm install` automatically covers all listed dependencies, effectively preparing the project for use. In contrast to Python packages, NPM packages are installed in a per-project scope by default into a `node_modules` folder. The flag `-g` activates a global, system-wide installation, with the location `/usr/local/lib/node_modules`.

Java artefacts are packaged with Maven[22] and distributed via Maven Central[23]. The `mvn` tool build prepares the project and ensures the download of Maven dependencies. Alternative approaches exist, such as building Java projects with Gradle[24]. In general, these approaches are more software engineering-centric and are typically used within other Java development project structures, regulated via files such as `pom.xml` or `build.gradle`. This also applies to other compiled languages such as C and C++, where build commands like `make` or `cmake` are common, indicated by the presence of either a `Makefile` or a `CMakeList.txt`. If these build files do not yet exist, they may be produced by a locally existing executable auto-configuration script `configure`, which produces these files from templates based on the *Autotools* framework. If even that script does not exist, it might be possible to bootstrap it with another script, by convention called `autogen.sh`. C/C++ projects are more scattered over the Internet, although recently, with the C++ package repository[25], a package catalogue with dependency tracking support has become available.

Commands to repeat in alphabetic order: cmake, make, mvn, npm, R

[21] NPM website: `https://npm.io/`
[22] Maven website: `https://maven.apache.org/`
[23] Maven package search: `https://search.maven.org/`
[24] Gradle user guide: `https://docs.gradle.org/current/userguide/userguide.html`
[25] C++ repository: `https://cppget.org/`

5.7.4 System package management with APT

Not all applications and tools are implemented in Python or one of the other covered languages, and even some of those that are require a deeper integration into the operating system. Additionally, it is convenient to have a unified, language-independent way to install well-curated packages whose dependencies are balanced out.

Hence, an OS-specific software catalogue is needed. On Debian or Debian-based systems such as Ubuntu, the Advanced Package Tool (APT) performs this role. It does so in conjunction with a well-maintained, tightly integrated and policy-driven repository containing packages around the OS itself as well libraries, applications, datasets and documentation. Thousands of packages including their dependencies can be installed with a tool invocation of the form `sudo apt-get install <package>`. The command looks up package locations in a system-wide catalogue cache configured with a sources list file `/etc/apt/sources.list` containing references to installable packages (`deb` lines) and, if properly configured, corresponding source packages (`deb-src`). An example line for Debian is `deb http://ftp.ch.debian.org/debian/ bookworm main`, indicating the named version of the OS distribution and the main archive that contains exclusively free software.

Before the installation succeeds, the system-wide catalogue cache must first be created in the directory `/var/lib/apt/lists` via the command `sudo apt-get update`, which is also updated occasionally with the same command as the repository evolves. Rarely, for finalising a system and making it immutable, it can also be deleted again `rm -rf /var/lib/apt/lists`. The installation command itself also caches packages, which fill the disk over time, in the directory `/var/cache/apt/archives/`. They can occasionally be cleaned up with the more accessible command `sudo apt-get clean`. Packages are archive files created with `ar` and containing `tar` files. Apart from metadata and installation scripts, the main file `data.tar` contains directory hierarchies that are applied relative to the system's root directory `/`.

A typical example would be `curl` in order to ensure the system-wide availability of this command that is a Swiss army knife for interacting with web services. However, while `pip` works both in user mode and in privileged system-wide mode, `apt-get` installation and maintenance commands require system privileges, i.e. the use of `sudo`. There are some ways around this requirement. The first way is to retrieve only the installable binary package with `apt-get download <package>` or the corresponding source package with `apt-get source <package>`, if the proper source lines were added to the sources list. As a downside, this way skips the dependencies. The second way is to use a fake root environment, for instance, by working entirely within a user-level OS hierarchy through the command `fakeroot -s fakechroot.save fakechroot debootstrap --variant=fakechroot book-`

`worm /tmp/osroot`. This results in a the directory `/tmp/osroot` containing an entire operating system, which can then be activated with the follow-up command call `fakeroot -i fakechroot.save fakechroot chroot /tmp/osroot/ /bin/bash` to obtain a shell with fake root privileges. The third and most complex way, apart from the use of non-fake virtual environments with `chroot`, is virtualisation or containerisation to obtain virtual root access.

This requirement for privileged invocation does also not apply to simple package description search through `apt-cache search <term>`. A related command is `sudo apt-file update` that creates a system-wide database of mappings of file paths to packages containing those paths. While it requires root privileges, the subsequent search command `apt-file search <partialpath>` does not.

If the sources list gets modified to include a new version of the distribution, a more complex upgrade command in the form of `sudo apt-get dist-upgrade` should be run to accomodate larger changes in the packaging.

Despite the high curation quality, the package updates sometimes get stuck. Diverse calls such as `apt-get -f install` or to the underlying Debian package management tool (`dpkg`) may help in resolving the issues.

Over time, one learns the package-naming conventions, such as `python3-`X for Python packages often taken from PyPI or `r-cran-`X for all R packages taken from CRAN. The online package catalogues can also be searched, for instance, for Debian[26] or Ubuntu[27], which in contrast to local system searching is especially useful to find packages not currently offered for the running distribution version.

Commands to repeat in alphabetic order: apt-file, apt-get, debootstrap, dpkg, fakechroot, fakeroot

Repetition

1. In case the editor Vim is not yet installed on a system, how would it be installed?
2. With which command can a user count the number of externally installed Python packages?

5.8 Container management

Application isolation and containerisation has become widely popular since the mid 2010s due to a number of convenient technology stacks. Isolation

[26] Debian packages: `https://www.debian.org/distrib/packages`
[27] Ubuntu packages: `https://packages.ubuntu.com/`

itself has for many decades been confined to file systems with the change root (`chroot`) command, but has been expanded to other OS-managed resources with *CGroups*, which paved the way for convenient containers. Containers are therefore a fine-grained runtime isolation mechanism at the OS level with representation in `/sys/fs/cgroup/`. At the same time, tangible container images have complemented the isolation with greater portability and more options to ship code and data from creators to users.

The most widely used containerisation stack and distribution channel especially for mainstream use by data scientists is *Docker* and the Open Container Initiative (OCI).[28] OCI is an evolving set of specifications on container images, runtimes and distribution mechanisms, and Docker Hub the most popular public distribution site. In recent years, *Podman* has emerged as mostly-compatible container interaction tool with security benefits. It has simplified configuration requirements and integrates into the operating system process management without requiring elevated privileges, allowing for better self-healing capabilities not available in the plain Docker client.[29] This part of the book introduces primarily Podman, explains how to work with existing containers, and also informs about how to create custom containers for isolating and shipping own code and data. Occasionally, references are made to Docker and to OCI specs.[30]

5.8.1 Introduction to Podman

Podman consists of the intuitively `podman` command for individual container and container image management. Docker moreover provides support to operate composite applications through Docker Compose and Docker Swarm. Due to operational complexities with the Docker stack, there are alternative runtimes for the same container images, with some like Podman almost offering a drop-in interface, and others that operate slightly differently.

With Podman, the other package management approaches (programming language-specific and OS-level packages) are complemented by the ability to distribute complex applications in a pre-configured way encapsulated in container images. At the target machine, only some system bindings such as port numbers (for services) or volume directories (for applications requiring data persistence) need to be set up. Containerised applications can therefore run multiple services implemented in different languages without interfering with the host system such as a data scientist's workstation or virtual machine. A downside of this approach is the duplication of code in dependency libraries, which also requires discipline to keep container images updated to not be ex-

[28] Docker/OCI ecosystem: `https://opencontainers.org/`

[29] Podman documentation: `https://docs.podman.io/en/latest/`

[30] OCI runtime specification: `https://opencontainers.org/release-notices/v1-1-0-runtime-spec/`

posed to security issues. The `podman` command offers many subcommands, each with its own set of options and arguments. In the following, only the most-needed commands are explained.

5.8.2 Fetching and running containers

Container images use a layered file format to store containerised application code and data. The user workflow consists of fetching the layered images from a container registry, combining them into a local directory tree representing an entire filesystem hierarchy (minus OS kernel), and then executing commands within that filesystem through CGroups isolation. Docker Hub[31] is the default public registry on which thousands of container images are available. Collaborative programming platforms also offer integrated container image registries, and, moreover, it is possible to operate standalone instances for private use.

Finding the right image on any registry requires searching, trying out and trusting the provided images or the provider behind them, and gradually building up experience on what to look for. Image names on Docker Hub usually consist of one or two stem components (user name being the first one which could be omitted for official images) as well as a version number or practically unique version hashsum. With `podman search <registry>/<term>`, an approximated search for published images can be conducted, with the same caveats applying to Python packages and OS packages.

The canonical form of invoking Podman interactively is by specifying the name of the container image, optionally extended with the version information and the launch command, as in: `podman run -ti <container-image>[:<version>] [<command>] [<arguments>...]`. By default, the start command given at container image build time is used, but it can be overridden. In particular, debugging an image that does not start as intended is often possible by overriding the default start command with a shell invocation: `podman run -ti ... /bin/bash`. The run command implies a `podman pull <registry>/<container-image>` with all images stored as overlay file systems in the per-user path `~/.local/share/containers` in case there is no local image replica yet. If there is, pull can also update from a moving version tag.

If the image does not yet exist locally on the machine the tool is executed on, its layers are transparently downloaded and merged, leading to a slight delay in invocation. For testing purposes, a `hello-world` image is provided on Docker Hub that runs without further arguments and exits after delivering a message, and `alpine` is another convenient and small image to get a basic containerised shell. An example invocation of a useful container with configurable launch command would thus be as follows: `podman run -ti curlimages/curl:latest https://www.zhaw.ch/de/hochschule/`.

[31] Docker Hub: `https://hub.docker.com/`

Port numbers (`-p <hostport>:<containerport>`) and directories serving as volumes (`-v <hostdir>:<containerdir>`) connect the container's OS resources to the one from the host OS. For non-interactive invocation, the flags `-ti` are dropped, and the new flag `-d` leads to an invocation in the background, informing about the container identifier (id) as only output.

With `podman images` and `podman rmi [-f] <image-id>`, container images can be managed. Further commands such as `podman ps` or `podman kill <container-id>` are available to monitor containers and to manage the lifecycle of containers beyond starting up. Their semantics has been derived from the respective OS-level commands. In case a container has exited, `podman ps -a` shows an archived list of previous executions that still remain until `podman rm` is invoked on them. Well-maintained container images also provide endpoints for tracking the health status, showing up in the status column of the process list, in conjunction with specifying a container restart policy as self-healing instrument. With advanced container orchestrators, further self-management, autoscaling and other operations-supporting features become possible.

5.8.3 Building custom container images

A container image is built by incrementally modifying an existing image and storing the differences as another layer or set of layers. In the most trivial case, this would mean adding some files and customising the default startup command. Hence, selecting a base image to start with becomes a crucial task and a skill to master.

From a security viewpoint, often only a subset of 'official' images are considered trusted and are used exclusively as a basis for deriving custom container images. For example, instead of directly running the *Azure ML Inference* container, one would fetch an appropriate base image such as *Alpine Linux*, and then install the Machine Learning (ML) tools such as Tensorflow, PyTorch and Scikit-Learn manually into a derived image. Naming (tagging) the images for private use, uploading them to a private registry to share, and uploading them to Docker Hub for world-wide access are further options to consider after having modified an image that way.

Building such derived images works by specifying the base image as well as any modification commands in a `Containerfile`. This file can be written manually, or for some types of application even be generated automatically, at least to some degree. A Containerfile consists of a specification of the base image (`FROM` tag), files to be copied or commands to be run within the image (`COPY` and `RUN`, respectively), and the default launch command (`CMD`). The build command is then `podman build -t <tag> .`, with the last dot being a reference to the current directory which contains the Containerfile but also all files referenced from it.

Commands to repeat in alphabetic order: chroot, podman

Repetition

1. Can Podman be used to run a different version of the operating system?
2. How can PyPy be run interactively in a container?

5.9 Data management and version control

Keeping track of data, code and configuration files requires data-centric tooling for basic shuffling of data between locations. Before starting to accumulate data, certain criteria need to be assessed. This concerns primarily the protection of sensitive data. Such data contains privacy-related identification of persons or business secrets. It should never be stored on unencrypted physical media such as disks and should never be exposed to the Internet. In contrast, in order to not lose valuable data, safety replicas and backups need to be arranged. Automating this process may require lifting the exposure criterion for a short amount of time, still with the aim to minimise risks. This minimisation also includes occasional replacement of physical media and regular checks on the presence and recoverability of data. Once the criteria on data safety and security are fulfilled, the question becomes more technical concerning the best tools to use for content-agnostic data management on the file level.

Over the years, delta transfer and version control systems have emerged as technologies of choice for software engineers but also for data scientists with potentially large files. From early centralised approaches in version control such as CVS and Subversion (SVN), the collaborative nature of many projects has led to the proliferation of decentralised approaches. The widespread adoption and commercial success of Git has led to it being the de-facto standard for data management especially in development and operations, whereas unversioned data might also be effectively managed with RSync. This section explains both tools to ensure that file-based data can be handled in a safe manner.

5.9.1 Delta synchronisation with RSync

The previously introduced commands to copy files have basic support for incremental synchronisation and backup. Using `cp -auv <source> <target>`, or `--archive --update --verbose` in the long form, copies all files from a source directory including their metadata but only if they have changed compared to the last run, based on a comparison with the contents of the (previously existing) target directory, and informs about the progress of copying. Running the same command twice while keeping the source files unchanged shows that no

copying takes place anymore on the second run. Copying the files to a remote machine requires a syntax like `scp -r <source> <machine>:<target>` that unconditionally performs a copy operation without consideration of whether changes have occurred.

In summary, while `cp` and `scp`/`sftp` are suitable to copy few files and directories locally and remotely, respectively, the remote copy process always assumes the full file, leading to unnecessary occupation of bandwidth and processing time. When the amount of data is large and the amount of modified data is small, it is more practical to synchronise only the changes, represented as deltas between the old and the new version. With `rsync`, delta transfer happens transparently both on local systems and across the network. RSync is therefore a suitable single and efficient interface for all data shuffling, no matter whether system boundaries need to be crossed.

A typical invocation is `rsync -avz <source>[/] <destination>[/]`, with the flags representing archive mode (keeping all file attributes), verbosity, and compression of the deltas. The trailing slashes are significant concerning the reference to the directory itself or its contents. Source and destination either refer to local directories or to remote ones based on SSH in the form of `<user>@<machine>:<path>`. In case certain file and directory patterns should not be part of the replication, the option `--exclude <pattern>` (only available as long-style option) can be added to the command.

If low copy time is less of a concern compared to low bandwidth use, RSync supports a bandwidth limitation in the form of `--bwlimit <rate>`. The rate is typically specified with a data size per second unit suffix, for instance as `2.3m` to allow for 2.3 MB/s. Units can be chosen flexibly, including `g` for GB/s which is closer in line with evolving network speeds. For comparison, `scp -l <rate>` requires the value to be specified with high numbers in kB/s, whereas `cp` has no bandwidth limitation support, making RSync also a suitable choice overall for predictable rate data transfers independent of local or remote operations.

RSync can also run in daemon mode on a server to grant public read access to files organised as areas, offering a delta-capable alternative over the File Transfer Protocol (FTP) or HTTP. In that case, a listing of the exported areas can be obtained using `rsync <machine>::`, and the source location in the copy process is then addressed as `<machine>::<area>`. In contrast to version control systems, datasets obtained that way will be smaller albeit not collaboratively editable.

File transfers with RSync are documented on its website.[32] While RSync is not a backup solution in itself, several backup tools and strategies are built on top of it, such as RSnapshot[33].

[32] RSync website: `https://rsync.samba.org/`
[33] RSnapshot website: `https://github.com/rsnapshot/rsnapshot`

5.9.2 Version control with Git

Git is a tool to shuffle files between machines, but also a Version Control System (VCS) to track the evolution of those files, containing data and software with version history and a clear indication of what changed. Using Git properly aids in distributing and backing up data, ensuring provenance of data, code and models in larger projects and traceability of modifications.

The central data structure of Git is a *repository* containing binary files with the full history of all tracked files. Repositories can be cloned (replicated) and delta-updated multiple times from each other, leading to a decentralised model, which is, however, in practice often subject to agreeing on one centralised master repository, not for technical but for social or legal reasons. Repositories grow over time, but due to delta storage of the modifications, the growth especially with text files is modest. The advantage of keeping the entire history on all machines the repository is cloned to is that most operations such as search through the history work locally without requiring permanent network access. Especially for mobile workstations, this is a practical design choice. Each file tracked in a repository exists in versions that start with the addition of the file and run up to the lastest version. These versions might be located in different branches, so that each branch contains a subset of versions. The default branch is often called `master`. Version numbering in Git is not consecutive but based on file content hashsums, which are hexadecimal numbers often abbreviated as long as the short version is unique within a repository (e.g. `510906fa` as shortcut for `510906faecd6e8eae8c74b1cea9294347b3f1a97`). The latest version of each branch with all latest versions of the files in that branch is referred to as `HEAD`.

The second data structure of Git is the *index*, which memorises intended changes to the repository. An index is a local companion to a clone. The third data structure is the *working directory*, essentially the directory in which the files representing the currently checked out version of a branch in a repository are located and are being worked on. A working directory may therefore be either clean, or consist of a number of untracked files and/or files currently marked for changes (addition, removal, modification) in the index.

5.9.3 Basic usage of Git

The Git usage from a data scientist perspective follows a number of workflows centered around working on a cloned repository, adding files to branches within the index, commiting those files to the repository locally, and propagating the modifications to other repository clones while also ingesting modifications from them. Delta transfers work similar to RSync, with the advantage of versioning, but with the slight disadvantage of requiring double the space on checked-out working copies.

The standard command to invoke the Git client is just `git`, with a plethora of subcommands. There are also graphical frontends to Git available such as Github Desktop, and many collaborative environments automatically synchronise changes through Git. Often, Git is not yet properly preconfigured upon first use. The basic configuration requires two commands: `git config --global user.email <email>` to set a contact address, which may not necessarily be an e-mail address, and `git config --global user.name "<name>"` for a corresponding full name. The global setting ensures that the configuration becomes usable across all working copies. The configuration settings are then stored in the file `~/.gitconfig` which with some experience can also be edited directly to modify the identity, add custom commands and perform other settings.

When using Git on the shell, an apparent characteristic is that all files are tracked in branches of local repositories, effectively represented by hidden directories called `.git` at the top folder of a checked out clone, i.e. working directory, also called working copy or workspace. The repositories are initially empty when created with `git init` within an arbitrary directory that then becomes a working copy. Repositories can also be created with `git init --bare` in a dedicated directory meant purely for cloning without being usable as a working copy. When a repository already exists, it can be cloned with `git clone <url>`, with the URL either being a local file path, an HTTP(S) URL to a web-served Git repository or an SSH account on a server managing Git repository access based on SSH keys.

In a working copy, files can be prepared to be added to the default branch, typically called *master*, by marking them with `git add <file>`, which only adds them to the local index in an intermediary step. Directories are tracked implicitly as paths to those files, while empty directories are not tracked. Nevertheless, the `add` command also works recursively if a directory is specified. Only directories and files matching the names or patterns given in a `.gitignore` files are excluded. That file can be placed in the top-level directory of a repository or in subdirectories to override the configuration for those. Consequently, `git rm <file>` removes files or recursively removes directories again. To maintain an overview, `git status` shows information about the branch, the index and untracked files.

The current branch can also be queried with `git branch`, with the option `-a` to show all available branches. If desired, to try out experimental changes, a new branch can be created with `git branch <name>` and switched to with `git checkout <name>`. Whenever the working copy ends up in a broken state, for instance, due to versioning conflicts, it may also be disassociated from any branch, requiring more work to be aligned again. In those cases, `git merge` with the default merge strategy or a custom one helps to recombine independently conducted changes across branches or from different people.

A repository as well as the index both contain hashsum references to files. Whenever a file changes, the hashsum no longer matches, and the file is detected as modified. The modifications can then be staged to be added to the repository as well by invoking `git add` once again, keeping the index current. The actual modification of the repository based on the index then requires another command, `git commit`, which produces a new revision. In this command, a commit message can be entered either interactively or for short messages via `git commit -m "<message>"`. To combine the two steps (adding to index and finally adding to local repository), `git commit -a` can be used.

The history of all changes on the current branch can be shown with `git log`, and all changes themselves with `git log -p`. This command also shows whether all branches and remotes are synchronised. Remotes refer to the machines that are known to host other clones of the repository. When a working copy is initially cloned from an existing repository, that one gets recorded as known. Other clones can be registered manually with `git remote add <name> <url>` and queried with `git remote -v`.

Git then becomes decentralised by the ability to push revisions on a branch, in other words: to synchronise a branch, with remote copies of the same repository. Hence, a data server may run a Git daemon process or SSH-wrapped Git executable to receive modifications via the `git push <remote>` command, where the remote is a symbolic name usually pointing to an HTTP or SSH URL. Calling only `git push` attempts the default remote. If the modifications are purely additive, the push command is accepted and the repository branches are synchronised. Otherwise, for instance, after concurrent modifications by multiple users, the user intending to push is instructed to resolve the conflicts first. This usually requires merging the user's changes with those of other users by running a sequence of `git pull`, manual resolution via file editing, and finally again adding, committing and pushing.

A guided introduction to Git is given in the 'Pro Git' book. Further information including client download options are available on the website.[34]

5.9.4 Advanced usage of Git

Git offers the ability to execute repository-side hooks upon receiving modifications. This feature can be used to prevent the modification in the first place, by executing sanity checks on the files, as well as to trigger external actions such as sending the current files to an external service. Being activated by repository changes applies both to the local repository (after `commit`) and to any attached remotes (after `push`). This ability allows for setting up continuous delivery schemes. For instance, a data scientist may train a model, and pushing the files to Git ensures that the model is also properly deployed in

[34] Git clients: `https://git-scm.com/downloads/guis`

all inference services. On remotes, the `pre-receive` and `post-update` hooks are primarily of interest for this functionality to enforce policies, whereas in the working copy, the `commit-msg` hook can perform a first sanity check on what got modified, and the `pre-push` hook can further check whether a series of commits are worth pushing and likely allowed by the remote's policy. The hooks are implemented as shell scripts invoked by Git. All of those scripts must be placed as executable shell scripts into the repository's hook directory, which is in `.git/hooks` relative to the top level of any working directory and by default already contains templates for common hooks.

Git is optimised for text content. Smaller binary files can be stored without a problem. However, larger binary files such as compressed models or multimedia content not only become inefficient in terms of double space handling (for the checkout and the local repository) but also in terms of change detection and other Git management tasks. For this purpose, extensions are available such as Git LFS (Large File Storage), Git-Annex and DVC (Data Version Control). Git LFS stores only metadata about the files in the repository and can therefore inform about the absence of expected files. The files themselves are stored on a special LFS server. Git LFS must first be activated inside a working copy with `git lfs install`. Large files are then marked for tracking with the command `git lfs track <pattern>`, producing a `.gitattributes` file which must be added to the repository. Git-Annex follows a similar design, but does not require a dedicated server. Instead, it can synchronise large file content flexibly via RSync. The usage pattern is `git annex init` followed by `git annex add <largefile>` which replaces the file with a symbolic link to it, putting the actual file into the folder `.git/annex`, and marks the file within `git annex status`. The command `git annex sync` then synchronises repositories. Finally, `git annex copy --to <remote>` replicates the large files to a remote manually.

Commands to repeat in alphabetic order: git, git annex, git lfs, rsync

Repetition

1. Which merge strategies are available in Git to resolve conflicts after a pull?
2. You would like to dive deeper into extended file system attributes and have an idea to extend the custom `attr` tool for that, hosted at `https://git.savannah.nongnu.org/git/attr.git`. Which steps would be needed?

5.10 Data processing tools

Programming languages such as Python allow for very powerful text processing. Yet sometimes, as part of a shell-level automation workflow, it might be more convenient to remain on the shell for trivial processing steps, or it might take too much effort to implement a certain text processing algorithm.

Coreutils are therefore available as a collection of several small tools invoked from a shell to facilitate the processing of data stored in files. This encompasses the formatting of data, the creation of checksums and statistical information about file contents, performing string-based and numeric processing as well as search. A subset of the functionality is briefly summarised here, in conjunction with Sed, Grep and other tools also suitable for command-line data processing. While individual commands are documented with manual pages, a more systematic documentation is often provided in the so-called info format, especially for the individual utility programs that are part of Coreutils; hence, `info coreutils` brings up relevant pointers. A broader introduction to use coreutils and other command-line utilities for data science tasks is given in online books.[35]

5.10.1 Text search

The `file` command is able to inspect and inform about the content of any given file. If it is a text file, in contrast to a binary file, the text encoding is also determined heuristically. The default output is for human consumption, whereas a technical representation as MIME type is given with the parameter `-i`. A file may also be reported as a symbolic link; in that case, using the option `-L` to dereference the link is advisable. Working with text data files from various sources may also require a more sophisticated way to determine the encoding. Although some text editors are capable of showing a text file encoding, this is always subject to heuristics. The `chardet` tool includes a confidence value to make this decision more transparent.

Text files can be summarised with `wc` to count lines, words and bytes. One should be careful not to mistake the bytes with characters, as depending on the encoding the number might be different. Hence, to determine the number of characters for instance to check form limits of significance to humans, the option `-m` or long `--chars` for character counting should be used. Again, care must be applied to not use `-c`, which unintuitively is the short form of the default counting in `--bytes`.

The canonical tool to find occurrences of text in unstructured and semi-structured files is `grep`, along with similarly invoked tools such as `ack`. These tools can also search directory trees recursively and report the findings in detail

[35] CLI data processing tutorial website: `https://datascienceatthecommandline.com/2e/chapter-1-introduction.html`

or summarised as binary result whether or not certain text has been found. Searching a text token in a file works by calling `grep <token> <file>`. All complete lines containing the token are printed to standard output, and if the token was found at least once, the command exits with code 0, or else with 1. The commonly used option `-q` (`-quiet`) suppresses the printing. By default, search is case sensitive, whereas case-insensitive search can be instructed with `-i`, so that searching for either `word` or `WORD` matches either occurrence in a file.

To search through an entire directory recursively, the option `-r` is used. When the directory traversal encounters a binary file, it also attempts a search there and may report a match if by chance a short sequence of letters also appears therein. To prevent this behaviour and skip binary files, the option can be extended to `-rI`. With `ack`, recursion is the default when either no file or a directory is given as argument to search in.

The dot in a search term stands for arbitrary characters; hence, to search for a dot, it must be double-quoted once for the shell and once for Grep itself, by using either `grep "\."` or `grep \\.` as search command. A number of characters are interpreted as basic regular expression but in a Grep-specific interpretation. To achieve support for standard expressions, the switch `-E` for extended regular expressions (e.g. `[[:digit:]]`) or `-P` for Perl-compatible regular expressions (e.g. `\d`) should be used.

Text search is slow when conducted repeatedly. There are fulltext search engines such as Xapian that depend on prior indexing of content that is not supposed to change afterwards, such as archive files, so that subsequent searches are faster. In practice, non-indexed search is fast enough for most purposes.

The `expr` command evaluates expressions related to text strings. For instance, the command `expr index abcdef cz` determines the first occurrence of any character of the character set `cz` in the string `abcdef` as a 1-based position, i.e. 3, whereas 0 would signal that no character from the set was found in the string. Alternative invocations are `expr match` for matching regular expressions, `expr substr` to find out substring relationships, and `expr length` to calculate the length of a string.

> Commands to repeat in alphabetic order: ack, chardet, expr, file, grep

5.10.2 Text processing

The text-processing commands work on files or text passed on standard input as their input and write the results to standard output, with the option to redirect them into other files through shell redirection techniques or explicit output file specification (often `-o <outputfile>`).

While `cat` and `tac` were already introduced for outputting text content, the `nl` command works like `cat` but prepends line numbers.

The `sort` command sorts a text file line by line, either alphabetically or, with the `-g` option, numerically. Sorting can be further configured to be case-insensitive (`-f`), strictly numeric (`-n`) and in reverse order (`-r`). To speed up the sorting of large amounts of text, the built-in parallelisation (`--parallel=`N) to use N CPU cores is advisable.

The related command `shuf` performs a random permutation of all lines, without repetitions by default or with infinite repetitions by the parameter `-r`. To reduce the number of repetitions, this option is usually combined with `-n <lines>`.

The `uniq` command drops all duplicate consecutive lines and with the option `-c` also counts the occurrences of each line in a file. To enforce that identical lines are consecutive, a text must be sorted first; a pipeline of the form `sort <file> | uniq -c` is therefore commonly used.

The `cut` command is able to extract separated columns. For instance, `cut -d , -f 1 <file.csv>` outputs the first column of a CSV-formatted file. The logical opposite is `paste`, merging content from several files as columns into a single output line per line. Similarly, `join` merges lines based on a common join key, dropping all lines without occurrence of the key across all files.

Sometimes, using `cut` is difficult because there may be one or more spaces used as separator between words. The `awk` command is an alternative in such cases as it can also extract based on token positions independently from repeated separators. Running `awk '{print $2}'`, for example, always prints the second word on a line, and `awk 'NR>10 {print $3}'` always prints the third word while skipping over the first ten lines (with `NR` being the number of records skipped).

Rearranging text to fit into a specific width can be achieved with `fold -w 20 <file>`.

Search and replace functionality based on regular expressions can be achieved with Sed. A typical use case for automated text replacement would be to substitute all occurrences of A with B in a file with the following invocation: `sed -i -e 's/A/B/' <file>`.

> Commands to repeat in alphabetic order: awk, cut, fold, join, nl, paste, sed, shuf, sort, unique

5.10.3 Numeric processing

Basic mathematical operations on integers are built into shells, not as commands, but as arithmetic expressions that can be evaluated. For instance, the expression $((1+1)) evaluates to 2, as can be verified with the command: `echo $((1+1))`. If `a` has the value 5, `$(($a*2))` returns 10. Supported operators include addition, subtraction, multiplication, division, exponentiation,

remainder (`+ - * / ** %`) as well as bitwise and comparison operators. Incorrect results can be expected with the division (`/`) which resembles the integer division operator (`//`) in Python.

For arbitrary-precision calculations, the `bc` tool can be used, taking an input formula on standard input. The standard math library needs to be activated to achieve the full functionality including floating-point numbers support. For instance, `echo 7/2 | bc -l` outputs the mathematically correct result, like all floating-point operations only subject to minor discretisation errors, and `echo "s(1.0)" | bc -l` returns the sine value for $X = 1$.

Moreover, several coreutils assist in numeric calculations. With `factor`, a number can be divided into its prime factors. For example, `factor 999999993` yields the factors 3, 17 and 19607843, all of which are prime. The `seq` command produces a sequence of numbers over which an iteration can be performed, similar to the `for` loop in Python.

> Commands to repeat in alphabetic order: bc, factor, seq

5.10.4 Media formats

Beyond text and numbers, files may contain multimedia information, often in binary format, such as documents, images, audio and video. Working with that kind of data requires format-specific tools. Typical operations include retrieving format-specific metadata, scaling, other manipulation and cross-format conversion. Due to the variety of formats, only a brief glimpse at the possibilities to automate those tasks in the shell is given here.

PDF documents are inspected with `pdfinfo`. They can be scaled with the GhostScript (`gs`) tool, as follows: `gs -sDEVICE=pdfwrite -dCompatibilityLevel=1.4 -dPDFSETTINGS=/<level> -dNOPAUSE -dQUIET -dBATCH -sOutputFile=<out.pdf> <in.pdf>`. The target level is one of `screen`, `ebook`, `printer` or `prepress`, in ascending order of quality. With `pdftk`, pages can be extracted (`pdftk <in.pdf> cat 2 output <out.pdf>`) or stitched together (`pdftk <in1.pdf> <in2.pdf> cat output <out.pdf>`). PDFs can be converted to text with `pdftotext`.

Raster images are usually stored as lossless PNG or lossy JPEG files. The `file` command shows resolution and colourspace. With `convert`, formats may be changed as well as the resolution and other settings, for instance: `convert -scale 800 <in.png> <out.png>`. Videos may be scaled with `ffmpeg`, which, despite the name, covers a lot of video formats through plugins, including MPEG/MP4, AVI, MOV and OGV. A typical invocation for resizing and quality adjustment in terms of data points per second for the video stream is `ffmpeg -i <in.mp4> -s 800x600 -b:v 1000k <out.mp4>`. Frames can be extracted from videos as still images with a command following this pattern: `ffmpeg -i`

`<in.mp4> -r 1 -s 800x600 -ss 10 -t 2 out-%03d.jpg`. This creates two images at seconds 10 and 11 of the video, and give each of them a consecutively numbered filename.

Commands to repeat in alphabetic order: convert, ffmpeg, gs, pdftk, pdftotext

Repetition

1. How can you look up all phone numbers in a collection of text files?
2. How can a picture be reduced in size to a quarter, i.e. with half of its original width and height?

5.11 Structured data processing

Several command-line tools exist to perform basic checks, queries and modifications on structured data files in the common formats CSV, XML and JSON. Although the choice of tools is large, their names are usually related to the data formats they can work on, making it still possible to find the right tool given a dataset and an associated task. Moreover, some commands exist to perform basic data science and machine learning on specifically structured files with tabular formats.

5.11.1 Format-specific processing

For all structured data files containing text content, the same advice applies as previously given for text-file processing. In particular, using `file` or `chardet` to determine the character set and encoding is recommended before starting the processing.

For JSON, `json_pp` performs pretty-printing (and basic validation). It reads from standard input and is therefore invoked either in the form of `cat <file.json> | json_pp` or, more elegantly, `json_pp < <file.json>`. The output is indented and dictionary keys are sorted alphabetically, giving a uniform representation that does not matter for programmatic processing but is more suitable for humans and for maintaining a JSON file in version control systems.

To learn more about JSON files and especially the evolution between multiple versions, the Pip-installable `genson <file.json>` generates a basic schema from a data file, which is again in JSON format following the JSON Schema specification.[36] The schema is written to standard output. A schema contains

[36] JSON Schema: `http://json-schema.org/`

a description of the structure and rules about the permitted values. Auto-generated schemas may be too strict or too relaxed and might therefore need manual refinement afterwards. With the tool `jsonschema`, such a schema (possibly refined) can be checked against various instances of JSON files (`-i`) whether or not they conform. Hence, the following sequence of commands should work by first generating the schema from one file and then checking that same file against the schema: `genson <file.json> > schema.json; jsonschema -i <file.json> schema.json`.

If there are multiple versions of a JSON data file, `jsondiff` creates a machine-readable (JSON-formatted) difference representation (diff) between pairs of two data files. The inverse command is `jsonpatch` that takes such a diff and the first file and reproduces the second file. Therefore, the following sequence should again work: `jsondiff <file1.json> <file2.json> > diff.json; jsonpatch <file1.json> diff.json > <file2b>.json; diff -u <file2.json> <file2b.json>`. The two files `<file2.json>` and `<file2b.json>` should then represent the same content, and if they are properly sorted, the text diff at the end is empty.

The `jq` tool queries over the nested list and tree structures that may exist in a JSON file. For instance, `jq .<key>` returns the value of a key top-level dictionary from a JSON file. More complex queries are possible by pipelines that resemble shell pipes. For example, to select dictionaries with key Y that are in a list that is defined by a top-level key X, use the following command: `jq '.X | .[] | select(.name=="Y")'`. Typical string processing methods can be chained as additional operators, for instance, `.name | contains("substring")`.

Linting and pretty-printing for XML files is done with `xmllint` based on XPath expressions. The equivalent of the example above with the modification that the key could be anywhere in the document structure would then be `xmlstarlet sel -t -v "//key"`.

CSVKit contains many small tools to work with CSV files. Using `csvclean` helps in preprocessing and removing common formatting glitches. Lines with a column matching an expression can be extracted with an adjusted grep command, for instance, on entries with price equal to 3: `csvgrep -d -c price -m 3 <*.csv>`. The `csvtool` command is another way to work with CSV files.

Conversions between formats are also possible. With `csvjson <*.csv>`, a CSV file can be converted to JSON and with `csvsql` correspondingly to insertion statements for several relational databases. With `csv2`/`2xml` and `xml2`/`2csv`, bidirectional conversions between flat XML formats and CSV can be achieved. For instance, to convert a flat XML file to JSON, the following pipeline may be set up, assuming the source XML has an element $L2$ under its root representing the record which in turn contains $L3$ containing the field:

`xml2 < <file.xml> | 2csv L2 L3 | csvjson`. More complex conversions and transformations are the domain of data integration tools such as Meltano that will be presented in a later chapter.

Commands to repeat in alphabetic order: csv2/2csv, csvclean, csvgrep, csvjson, csvsql, csvtool, genson, jq, json_pp, jsondiff, jsonpatch, jsonschema, xml2/2xml, xmllint, xmlstarlet

5.11.2 Training and inference

Tabular data is often the input for data analytics and machine-learning tasks such as training, testing and validation. Many of these tasks are conducted programmatically due to the sheer number of options and algorithmic decisions. Nevertheless, a few tools exist to perform basic tasks on the command line. One such tool is Vowpal Wabbit, which assumes training data in a line-based tabular format `<label> | <feature:value> <feature> ...` and test data in the same format with purposefully removed labels (that should be predicted). With Vowpal Wabbit, the training data is first used to produce an optimised binary mathematical model with `vw -d train.txt -f model.vw`. Next, the missing test labels are predicted in order with `vw -d test.txt -i model.vw -p predictions.txt`. The predictions can then be compared with the labels that were removed to determine an accuracy score.

Binary classification of text based on trained word frequencies (Bayes model) is possible with several tools, for instance, `spamoracle` for e-mails. First, training data is specified to contain either good or bad e-mails, as follows: `spamoracle add -good <mbox>` (and `-bad`, respectively). Then, newly incoming messages can be tested with `spamoracle test <mbox>`, giving a record with score between 0 and 1.

Commands to repeat in alphabetic order: spamoracle, vw

Repetition

1. A RESTful API provides a compact JSON representation on a GET endpoint that should be formatted adequately for human inspection. Which command pipeline could be used?
2. A CSV file using semicolons as field diver has a column A with product names. How can all lines with A containing the character sequence *PRODUCT* be extracted?

Chapter 6

Middleware

Middleware, such as generic compute services, remote file systems, databases, message queues and message brokers as well as event processing frameworks, fulfil an important role of bringing scalable and stateful data processing capabilities to applications while keeping those applications lean. As opposed to rather static file management systems such as Git that only interact occasionally with servers, most middleware is operated continuously as standalone services, although some provide strong local processing support in the form of a tool as well or allow for configuring the service to listen only to local clients. From a programming perspective, for instance, from a Python application, it is therefore possible to connect to these services either locally or remotely to gain powerful functionality. This section covers middleware in a number of broad and partially overlapping categories: custom programmatic data and service provisioning, file system abstractions, database management systems, frameworks for data processing and integration, model serving and workflow execution.

6.1 Programmatic data serving

In addition to accessing web services from a client perspective, there is often the need to serve data and program logic from a service perspective. Serving means either to the human (from plain text to interactive websites based on structured and unstructured formats, i.e. HTML/XHTML) or to client applications (APIs or programmable web, i.e. structured data formats such as JSON, CSV, XML or YAML). Apart from the data formats and layout in the form of directories and subdirectories, the protocol needs to be defined. Data serving on the web is appropriate in most situations, giving the choice of both anonymous and authenticated access for read and/or write operations. Further protocols such

as FTP/SFTP, RSync or S3, or even decentralised protocols such as Bittorrent, each come with their own characteristics and can be chosen depending on the needs.

For static data serving on the web, simple tools like `netwox 125 -P 8080` or `thttpd` or even the Python built-in module `python -m http.server` can be used on the simplistic end of the spectrum and sophisticated web servers such as Apache HTTPd[1] on the other end. As explained in the section on system administration tools, long-running servers would be set up with service supervision and logging to ensure availability and to provide means to detect and correct any issues with the service provisioning. In static data serving, the endpoints for clients typically correspond to the physical file system layout, although advanced web servers offer support for virtual folders within virtual host definitions and interfaces to custom logic through Custom Gateway Interface (CGI) scripts.

Often, more complex logic elements such as differential data transfer and sophisticated endpoints are required. This leads to programmatic web-serving in which a user-defined application runs as service and waits for client requests. In Python, this functionality can be realised through appropriate modules. A few of these modules are briefly introduced in the next paragraphs. Which one to choose depends on the functional and non-functional needs as well as on the serving model.

6.1.1 Third-party module 'flask'

Several web frameworks make it easy to serve data, both of the static and of the dynamically generated kind. Data, including model files, can be made available over the network to clients on other machines, aiding the integration of pieces of software towards more complex software. One of the most widespread web framework is Flask[2]. In Flask, Python functions or methods are marked via decorators as accessible via the web protocol HTTP and interfaced on the server side with either an external web server or, for convenience in case scalability is not an issue, with the built-in Flask web server.

First, the flask object is created by calling `import flask` followed by `app = flask.Flask("Myapp")`. Functions serving as exported endpoints are decorated with the endpoint path and optionally with the list of supported HTTP method beyond the default of GET. Segments of the path may be typed to indicate that a field should be numeric or a wildcard path follows. For example, `@app.route("/hello")` or `@app.route("/<path:path>", methods=["GET", "POST"])`. The module import and use of decorators also works conditionally inside functions or methods in a class, using inner functions to make sure the

[1] Apache HTTP server: `https://httpd.apache.org/`
[2] Flask website: `https://www.fullstackpython.com/flask.html`

decorators only apply in case Flask is available. Finally, the application main loop is entered with `app.run(host="0.0.0.0", port=8080)` or a variation thereof, indicating the listening network interface determined by the IP address, and the respective port number. In this case, `0.0.0.0` indicates any interface, whereas `127.0.0.1` would listen only on localhost and thus prevent access to the service from outside the machine.

Within the methods, metadata and data can be accessed. For instance, the context-dependent attribute `flask.request.method` informs about the HTTP method used, and the method `flask.request.get_data()` gives access to any raw submitted payload in POST requests. Special methods also exist to process form data from HTML uploads.

For practical use, extensions such as Flask CORS and HTTPS activation need to be used to build applications that can be deployed and invoked across machines in secured environments. The CORS extension is imported with `import flask_cors` and instantiated atop a Flask application object with `flask_cors.CORS(app)`. Its behaviour can be further configured, for instance, through `app.config["CORS_HEADERS"] = "Content-Type"` to allow POST requests on JSON and other specific data types. HTTP is activated by passing an additional `ssl_context` parameter to `app.run()`. It refers to either a certificate/private key pair or an anonymous context. The first can be established through the Python SSL module with `context = ssl.SSLContext(ssl.PROTOCOL_TLSv1_2)` followed by the configuration step `context.load_cert_chain("cert.pem", "key.pem")` based on the two previously generated files, whereas the second is more trivial with `context = "adhoc"`. Both options nevertheless produce various security warnings in most web browsers unless a cross-signed certificate is used. More Flask extensions such as Flask Talisman[3] attempt to provide more security options.

Flask is primarily concerned with low-level HTTP request-response mechanics and less with the content of the messages. While HTTP responses can deliver HTML pages, their parameterisation calls for using Flask extensions that permit easy template rendering, such as Jinja[4]. Another extension is Flask-RESTful, easing the development of resource-oriented APIs.[5]

The command-line tool `flask` can help debugging a Flask application, when invoked with the following command: `FLASK_APP=x.py flask routes`.

Multiple alternatives exist with a basic functionality similar to Flask and similar integration into the web server gateway interface (WSGI) of Python, such as Bottle[6] or, specifically for continuous web sockets updates, Tornado[7].

[3]Talisman: `https://pypi.org/project/flask-talisman/`
[4]Jinja website: `https://jinja.palletsprojects.com/`
[5]Flask-RESTful website: `https://flask-restful.readthedocs.io/en/latest/`
[6]Bottle: `https://bottlepy.org/docs/dev/`
[7]Tornado: `https://www.tornadoweb.org/en/stable/`

6.1.2 Third-party module 'streamlit'

When not only raw data should be provisioned, but instead human-consumable visualisations, a data visualisation library can be helpful. Streamlit[8] is more oriented towards frontend prototyping and development. It is not often packaged yet and needs to be installed separately from PyPI through `pip install streamlit` in a virtual environment previously created with `python -m venv <venv-dir>`. The `streamlit` command documents its usage and gives examples, whereas the equally named `streamlit` Python module allows for integration into code. The idea behind Streamlit is that only the program logic is programmed and any visualisations emerge from that code, without data scientists having to worry about frontend/backend separation or content delivery via HTML.

For example, a Pandas dataframe created in a Python script can be visualised as a linechart with a single command `streamlit.line_chart(df)`, which renders an HTML page with embedded interactive chart. This script then needs to be executed with `streamlit run <script.py>` to let the visualisations take effect. The command launches a local web server on port 8501 and automatically opens a web browser window to it. Headless mode can be configured automatically by absence of the `DISPLAY` environment variable or by passing environment variables `STREAMLIT_SERVER_HEADLESS` or the option `run --server-headless true`.

More chart types are possible including `streamlit.bar_chart` and the generic `streamlit.pyplot` for anything manually plotted with Matplotlib. Tabular visualisation can be achieved with `streamlit.table` for static tables and `streamlit.dataframe` for dynamic tables. There are many integrations to visualise complex data structures such as maps, using the built-in `streamlit.map` function that requires geocoordinate columns in a dataframe (`lat`, `lon`) and defaults to using Mapbox or alternatively the more advanced `streamlit_folium` module, which directly interfaces with OpenStreetMap.

6.1.3 Third-party module 'bokeh'

Bokeh is another library aimed at human consumption of data. It integrates with Python data science libraries and generates visualisations as static files or as dynamically served web pages consisting of HTML markup and JavaScript code. Similar to Streamlit, it is not widely packaged and needs to be installed manually into a virtual environment through `pip install bokeh`. The submodule `plotting` then allows for creating the first visualisations. Similar to Matplotlib, a `bokeh.plotting.figure` object is created first (`p = figure(title="...")`), followed by plot commands such as `p.line(x, y, ...)` and finally `p.show()`. When invoking `bokeh <script.py>`, a temporary

[8] Streamlit website: `https://streamlit.io/`

HTML file is produced and the web browser is opened to display it. Bokeh also supports plotting circles and colourful scatter plots, colour maps, bar charts and stacked areas.

Bokeh can be combined with Flask by calling `flask.render_template` along with the utility function `bokeh.embed.components()` that returns both an HTML `div` element and a portion of JavaScript code (`script`, based on BokehJS) to embed into a rendered page template. This can be accomplished as follows: `flask.render_template("template.html", plot_script=script, plot_div=div, ...)`. Bokeh also ships with a standalone Tornado-based server running by default on port 5006. The command `bokeh serve --show <app.py>` deploys the application to the server and shows the generated web page in the browser on `http://localhost:5006/myapp`. Within the Python code, HTTP requests can then be accessed through the function `curdoc()`.

The reference documentation for Bokeh is available online.[9]

Repetition

1. Why does CORS support have to be added to web services that should be accessible from a dynamic web frontend?
2. Why can Streamlit-using scripts not simply be executed with the Python interpreter but instead require the `streamlit` command?

6.2 File system abstractions and network storage

Virtual file systems are the most basic form of middleware for hierarchical data storage. They provide a regular file system structure to applications while physically storing the files in arbitrary locations such as other file systems, network services or data encoding applications. Hence, the application does not need to take care of the specific interactions with any such service itself, including protocol and data representation details. Instead, the interface to the application is always the standard file system structure with directories, files and – with varying degrees of support – metadata. File systems in userspace (FUSE) are the basis for most virtual file systems on Linux. Some are well-maintained and can be used in production, while others are more experimental in nature. FUSE mounts run as processes so that the data availability depends on the availability of the service, which can be ensured with process supervision, e.g. SystemD units. Some FUSE implementations do not

[9] Bokeh website: `https://docs.bokeh.org/en/latest/`

handle all low-level operations on the file systems, but most applications write and read data from such mounts without problems.

6.2.1 Basic FUSE operations

The generic form for mounting FUSE file systems is via the `mount` command by specifying the type, like `mount -t fuse.<fusefs> [options] <source> <target>`, where the target represents a local directory and the source is a type-specific identifier such as a path, a URL or something else. File systems based on FUSE that are pre-configured by the system administrator in the file system table `/etc/fstab` can be mounted to the configured location within the local file system by privileged users with the command `mount <target>`. Usually, and especially for unprivileged users, the mount command name along with the type of file system (option `-t`) are also substituted by an executable with type-specific command name such as `<fusefs> <source> <target>`. To unmount again, the command `fusermount -u <target>` is used on Linux or `umount <target>` on Mac OS X. Such file systems can be layered in modular combinations, for instance, to achieve transparent compression, encryption and distribution of files. The application, which is unaware of these layered data modifications, writes into a file on a path of a mountpoint of file system A that encrypts it, and the underlying file system B delivers the data to an online service, for example.

Slightly confusingly, not all FUSE mounts show up in the `df` command output, but all show up when running `mount`. Unprivileged user mounts are by default only visible to those users, but the line `user_allow_other` in the otherwise almost empty configuration file `/etc/fuse.conf` along with the mount option `-o allow_other` changes this behaviour. This is also necessary for loop-mounted and bind-mounted file systems passed as volume to Docker containers due to the Docker daemon running under its own user identity.

Among the more commonly used abstractions are SSHFS, providing transparent file access over SSH connections, EncFS, adding encryption on the storage path, Fuse2FS, for loop-mounting disk images containing an ext2/3/4 file system, and BindFS, for bind-mounting one directory on top of another one, which may or may not be a FUSE mount location. Less commonly used but still useful are HTTPFS, allowing to interact with files on web servers as if they were local (unfortunately not supporting HTTPS), and Restic, for backing up data to various network services. An example for HTTPFS is to create an empty directory with `mkdir ~/cifs` and then mounting a website's HTML file to it with `httpfs2 http://cifs.servicelaboratory.ch/ ~/cifs`.

There are also more obscure abstractions such as π-FS or FUSE filesystems exporting the online mailbox as a virtual directory. Further information is

available online about the FUSE base technology[10] and, albeit incomplete, about selected file systems created with FUSE.[11]

6.2.2 Selected file systems and synchronisation

In case a user has access to a remote machine via SSH, then using SSHFS is a straightforward way to view a directory on that machine's file system. All files remain physically on that machine but can be listed and modified within the mountpoint. A production-grade invocation of SSHFS to resiliently mount the remote machine's `work` directory to the same name on the local machine is `sshfs -o reconnect,ServerAliveInterval=15,ServerAliveCo untMax=10 <server>:work ~/work` (on Mac OS X, where it supports slightly fewer parameters: `sshfs -o reconnect <server>:work ~/work`). This command mounts the remote folder of a machine serving SSH to the local equivalent and instructs SSHFS to overcome temporary connection drops, providing an always-on experience for remote file access. To automate this access without the need for manual authentication, apart from pre-confirming the host fingerprint with a first interactive SSH connection, a passwordless SSH public key needs to be deployed on the target machine, and an autostart file (e.g. via SystemD) needs to be provided. After mounting, the command `ls ~/work` works transparently on both the server (running the server's `ls` command on the local file system) and the client (running the client's `ls` command on the local mountpoint translating the access to the server's file system).

Disk images contain a file system within a file of fixed size and are a suitable mechanism to mount capacity-limited storage. They can be trivially produced by running `dd` followed by file system creation, for instance, with `mkfs.ext4`. Fuse2FS can then loop-mount these images: `fuse2fs <fs.img> <target>`. BindFS is able to mirror both loop-mounted and other directories to another location: `bindfs [-o allow_other] <source> <target>`. This makes the content accessible in two locations and can also be used to work around problems with low-level file system operations on the source mount.

EncFS and its alternative implementation, GoCryptFS, allow producing encrypted data from any application. When running `encfs <source> <target>`, a configuration mode and then a password need to be specified interactively. By default, any content is encrypted symmetrically with AES-256. The command `echo "plaintext" > <target>/text` leads to a 10-bytes unencrypted virtual file on the mounted target and a 26-bytes encrypted physical file in the source directory.

RClone builds on FUSE to provide a more integrated package for compression, encryption, chunking, as well as serving files both as local file system

[10] FUSE: `https://github.com/libfuse/libfuse/`
[11] List of file systems: `https://github.com/koding/awesome-fuse-fs`

and as locally operated file network service, while physically storing the files in arbitrary locations.[12]

Repetition

1. How can a filesystem contained in a single file be mounted?
2. SSHFS only grants access to certain files on a remote computer, not to the computer itself. Correct?

6.3 Database interaction and management

Databases (DBs) are useful collections of structured or unstructured data. The collection format can be very strict with schema information as in relational databases or rather loose as in schemaless/schema-flexible document databases. Database management systems (DBMS) assist in the collection process by providing data management commands and queries. Consequently, there are relational/tabular, tree-/graph-based and document-oriented DBMS as well as key-value stores, matching the respective database contents.

The term relational refers to normalised tabular data in which values in certain columns of a table form a reference to the same values in a column of another table. For instance, a table with addresses might contain country codes, and a second table maps these codes to country names. Tables are managed following the CRUD paradigm for creating (inserting), reading (selecting), updating and deleting rows. Read and update operations are also possible for a subset of columns. Read, update and delete operations are moreover possible on a subset of rows by filtering. Tables may be further grouped into schemas, databases and other higher-level structures. In non-relational databases, the main structures are collections or graphs.

Most DBMS offer a command-line client to manage tables, collections or graphs interactively and to insert or query data in the form of records or documents. For automated mass processing, programming interfaces specific to the chosen language are available, typically called connectors. Both clients and connectors require access to a running database service, although some also work on in-process embedded data structures.

6.3.1 Embedded relational databases with SQlite

The standard API from Python to relational databases is DB-API, although there is also support for such databases in Pandas due to the tabular data dominating in relational DBMS. Many DBMS require a complex server setup.

[12] RClone homepage: `https://rclone.org/`

For more modest use cases, an embedded DB entirely contained within one file is a good alternative. One such system is SQlite, an embedded relational DBMS with support for the standard query language SQL and the ability to interface from both Pandas and DB-API. All commands starting with a dot are non-SQL internal commands.

The canonical invocation on the shell level is `sqlite3`. Within its command prompt, a database file is created with `.open <filename>`. Then, tables can be defined with schema and populated, for example, with: `CREATE TABLE left (x int, y str); INSERT INTO left (x, y) VALUES (3, 4);`. The second value ought to be written as `'4'`, but the parser is flexible concerning the types. With `.mode table`, the default output formatting optimised for scripting is made more human-friendly, and a `SELECT * FROM left;` confirms the table contents.

From a Python script, that data can be imported into a Pandas dataframe by running a schema-dependent query. This is accomplised by importing the respective modules `import sqlite3; import pandas` and by running two successive commands `conn = sqlite3.connect(<filename>); response = pandas.read_sql("SELECT * FROM left", conn)`. The corresponding DB-API connection starts by setting up the connection first with `conn = sqlite3.Connection(<filename>)`, then creating a queriable cursor object on top with the query `cur = conn.execute("SELECT * FROM left")`, and eventually obtaining the query results through `cur.fetchall()`. This returns a list of tuples covering all columns, which should obviously only be used for smaller tables. Alternatively, one can iterate using `cur.fetchone()`, which returns one tuple at a time, or `None` at the end of the table.

SQlite also supports transient in-memory databases with a pseudo `:memory:` filename. The complete SQL syntax of SQlite is documented online[13], whereas the built-in Python module has its own documentation.[14]

6.3.2 Networked relational database systems

MySQL/MariaDB and PostgreSQL are relational DBMS that have seen active development and high adoption for many years. Both run either as standalone services or in various scalable cluster combinations. They support conventional relations but also binary-optimised structured data, as evidenced by the JSONB data type in PostgreSQL. MariaDB listens on port 3306, whereas it is 5432 (or merely a local socket in localhost connections) for PostgreSQL. The following describes an exemplary setup for PostgreSQL. First, the DBMS needs to be installed in the right version, for instance, with `sudo apt-get install postgresql-15`. Next, a user for local DB access needs to be created:

[13] SQlite syntax: `https://sqlite.org/lang.html`
[14] SQlite Python module: `https://docs.python.org/3/library/sqlite3.html`

`sudo -u postgres createuser $USER` and a corresponding database `sudo -u postgres createdb -owner=$USER <dbname>`. Finally, this database can be accessed interactively from the shell: `psql`. Automation is possible via the batch command execution `psql -c "<sql-statement>"`. In Python, DB-API is implemented by the widely packaged `psycopg2` module. A local database can be connected to with `conn = psycopg2.connect(database="<dbname>", user=os.getenv("USER"))`.

6.3.3 Beyond relational databases

The system families of key-value stores, document databases, graph databases and timeseries databases all offer means to manage data beyond tabular formats and relations. Many of the respective database management systems are however not easily installable or maintainable in operation.

Key-value stores offer low-latency access to dictionary structures held on disk or in memory. One of the earlier embedded database options for key-value storage was Berkeley DB. Later, networked alternatives such as Memcached[15] and Redis became popular. To improve network latency, they support setting and getting the values for multiple keys at the same time.

Document databases support collections of large structured and unstructured documents. In structured documents, such as JSON-serialised data, searches can be performed. Popular implementations beyond the mentioned document support in relational databases include MongoDB, CouchDB and BaseX (specifically for XML documents).

Graph databases represent graphs as labelled, weighted and directed relations between nodes. They implement graph algorithms such as shortest path search. Neo4J and OrientDB are typical examples of graph databases.

While conventional databases are suitable for mostly static data that needs to be retained over long periods of time, the nature of data is sometimes closely bound to its production time. Specific timeseries DBMS exist to handle such chronological data. Examples include TimescaleDB (an extension to PostgreSQL), as well as Prometheus and Victoria Metrics, both of which offer a native HTTP API for data management. One advantage of using a timeseries database is in being able to set up automatic compressions of events that have been longer back in the past, saving storage capacity.

TimescaleDB is installed into a running PostgreSQL extension by preparing the environment (`sudo apt-get install cmake libkrb5-dev postgresql-server-dev-15`) and executing the bootstrap script from the most recent version.[16] Next, the extension library needs to be loaded by setting `shared_preload_libraries = 'timescaledb'` in the global configuration file `/etc/postgre`

[15] Memcached: `https://www.memcached.org/`
[16] TimescaleDB downloads: `https://github.com/timescale/timescaledb/releases`

`sql/14/main/postgresql.conf` and activated within the client with `CREATE EXTENSION timescaledb;`. Afterwards, so-called hypertables on top of existing tables with a time column of type `TIMESTAMPTZ` are created with, for instance: `SELECT create_hypertable('<tablename>', 'time');`.

Repetition

1. Inserting a million records into a relational DBMS might be very slow. What could be done to speed it up?
2. Why does PostgreSQL not ask for a password when connecting to a local database?

6.4 Message brokers for real-time data processing

When data should be forwarded or processed immediately as soon as it arrives, message brokers are another form of middleware that is suitable for ephemeral and event-based scenarios. Events are received and need to be processed or distributed to other machines without necessarily storing them beyond the processing stage. This is typically called Event-Driven Architecture (EDA), Event Stream Processing (ESP) or, especially when certain state data such as counters and aggregate values are retained, Complex Event Processing (CEP). The encapsulated processing logic in this context is referred to as operators, which often contain standing queries applied to the incoming data-stream.

Message brokers are related to message queues and publish-subscribe systems. In all of these systems, there is a notion of events, event producers and event consumers. A variety of networked systems with overlapping functionality exist, for example, ZeroMQ, RabbitMQ, NATS, Kafka and Pulsar. Pulsar has the unique functionality that it executes small code functions to modify messages and influence the routing. ZeroMQ works daemon-less but also supports the setup of persistent devices to handle multiple dynamic producers and consumers. There are also file-based systems such as SEC, reacting on lines, for instance, continuously appended to log files, and Fever, listening to JSON events on a local socket.

In Python, ZeroMQ can be used with `import zmq`, setting up an event producer queue with `sock = zmq.Context().socket(zmq.REQ)` and connecting it to a consumer with `sock.connect("tcp://localhost:5555")`. The blocking consumer sets up its counterpart with `sock = zmq.Context().socket(zmq.REP)` and listens for producers with `sock.bind("tcp://*:5555")`. On TCP, acknowledgements for sent events need to be received; thus, a message exchange may look like: `socket.send("<msg>".encode()); sock.recv()`.

Pulsar is addressed via its default port number `pulsar://localhost:6650`. In Python, this works after `import pulsar` with `client = pulsar.Client(<url>)`, creating a first producer with a full topic URL `prod = client.create_producer("non-persistent://public/default/<topic>")` and then sending events with `prod.send("<msg>".encode())`. No acknowledgements need to be read. With `cons = client.subscribe(<topic-url>)` and `cons.receive(timeout_millis=50)`, events can be received in a non-blocking way. Finally, with the `pulsar-admin` command, functions can be deployed between input and output topic paths.

Due to the wide variety of systems, these are not further explored here but should be kept in mind when designing a data science architecture for streaming data. More information can be found in the documentation of ZeroMQ[17] and Pulsar[18].

Repetition

1. What is the main advantage of message brokers over polling approaches?
2. Why is the recv() method invocation necessary for ZMQ sockets?

6.5 Parallel and distributed computing

Parallel computing concepts were described before and command-oriented tools such as `parallel` were introduced earlier in this book as well. In this section, more capable data-oriented frameworks that combine automated parallelisation on one machine with distributed computation across several machines are introduced. These frameworks typically involve an initial overhead, but for larger quantities of data, their benefits become clear quickly. Among them are a local speedup by using multiple CPUs and even GPUs relative to the wall clock (at the expense of higher resource usage) as well as the enablement of processing of data volumes that no longer fit into the main memory of a single computer.

To make parallel computation available programmatically, several middleware systems and language-specific libraries exist. Among the more popular frameworks based on at least the map-reduce paradigm are Hadoop, Spark, Ray, Dask and Lithops. Further paradigms are supported by some of them with parallel and distributed processing capabilities, including composable pipelines, graph algorithms, windowed stream aggregation and data warehousing. The use of such a framework is not always required, especially when

[17] ZeroMQ: `https://learning-0mq-with-pyzmq.readthedocs.io/`
[18] Pulsar: `https://pulsar.apache.org/docs/`

software supports distributed operation internally. This is the case with some shells (`dsh` multiplexing) and compilers (`distcc`). In the following section, Spark is introduced from a Python (PySpark) perspective as exemplary framework for horizontal scaling of queries and user-defined functions in a compute cluster.

6.5.1 Data processing with Spark

Apache Spark[19] is a framework for parallel and distributed computing on sequences, tables and graphs as well as running relational database queries on tables. It offers interfaces for multiple programming languages, in particular Java, Scala and Python (PySpark). The execution model of Spark is driver-master-worker, with each worker covering one machine, each with potentially multiple CPU cores and GPUs. The application runs the driver that needs to be reachable from the workers, which limits deployment options. The master runs its main service on port 7077, to which the driver and workers connect, and a web interface on port 8080. Each worker runs a web interface on port 8081 to expose logs in addition to their main service port (e.g. 6666), and the driver additionally runs a web interface on port 4040. Furthermore, the driver opens a port to be reachable from the master and a second port for the block manager (e.g. 5555 and 4444, respectively). Many of those port numbers are only the starting points, as the driver automatically counts up in case a port number is already occupied. This complex handling of ports makes Spark not trivial to operate in a larger setting. Hence, in the following explanations, the focus is on the application interface for Python applications.

In a downloaded and extracted Spark folder, the interactive PySpark interpreter can be invoked with the command `bin/pyspark`. It can be used as a regular Python interpreter, but by embedding a Spark driver it offers access to the PySpark API through its pre-defined objects such as `sc` (Spark context), referring to a local in-memory worker, and `spark` (session object and SQL context). Alternatively, PySpark can be invoked as a wrapper around standalone applications (`spark-submit`) with the code importing the `pyspark` module to create the Spark context explicitly. This way, or alternatively by parameterising the PySpark interpreter, a Spark context to a remote cluster with potentially many worker nodes can be established with the line `sc = pyspark.SparkContext("spark://<sparkmaster>:7077", appName="...", conf=pyspark.SparkConf())`. This context then allocates the resources on the cluster until it is explicitly stopped with `sc.stop()`. Spark can also be embedded into other Python execution contexts such as scientific notebooks. For that purpose, instead of downloading the entire Spark release, one would typically run `pip install pyspark` in a virtual environment, along with op-

[19] Spark: `https://spark.apache.org/`

tional dependencies such as PyArrow and Pandas. A full download is however necessary to operate the cluster itself. If no cluster is available or needed, Spark can also parallelise computation within one computer (`local[*]`).

A number of options can be set on the `SparkConf` object as string key-value pairs. This includes `spark.driver.host` as driver hostname resolvable from the worker nodes along with the corresponding `spark.driver.port`, `spark.port.maxRetries` to raise the limit of concurrent connections over the default of 16 (e.g. 50 would be appropriate for a classroom setting), and `spark.cores.max` for the maximum number of requested cores. The assigned cores may be less if fewer are available, or temporarily zero (putting the application into waiting state) if none are available. Another option is `spark.jars.packages` to activate extension packages such as Graphframes for graph processing, Glow for genomics data, sparkMeasure to obtain metrics, and RAPIDS to get GPU acceleration.

Programming pipelines for data queries in Spark requires a good understanding of asynchronous programming concepts. Instructions are evaluated lazily, not necessarily at the line the instruction is written on, but rather when results have to be calculated. Usually this is at the end of pipelines or at explicit caching instructions in between. Moreover, despite being able to handle large amounts of data, Spark is not safe against out-of-memory situations. When performing an operation that joins data from all worker nodes or otherwise brings data to one place, the driver in PySpark may run out of memory.

Spark offers two main data structures, resilient distributed datasets (RDDs) for unstructured data and dataframes for structured, tabular data. A prerequisite is the setup of the context object as mentioned: `import pyspark; sc = ...`, for instance, with `spark-submit`: `sc = SparkContext(appName="<name>")`. An RDD can be produced programmatically with the command `rdd = sc.parallelize(<list>)` or (assuming shared storage access) by reading in a text file with `sc.textFile(<filename>)`. The distribution of the data partitions can be verified with the command `rdd.glom().collect()`. For structured data, the session/SQL context object must be set up first: `import pyspark.sql; spark = pyspark.sql.SparkSession.builder.getOrCreate()`. A dataframe can then be produced by adding a tabular schema to an RDD, by converting a Pandas dataframe, or by reading tabular data right away, such as in the command: `df = spark.read.format("csv").option("header", "true").load("file.csv")`. Relational SQL queries, graph processing and structured streaming are all implemented atop the dataframe API. Spark also provides a drop-in Pandas API for easier conversion of existing code towards distributed environments (`pyspark.pandas`). Further information about Spark programming is available from the respective guide.[20]

[20] Spark programming guide: `https://spark.apache.org/docs/latest/quick-start.html`

Repetition

1. What is the difference between using spark-submit and PySpark applications not making use of spark-submit?
2. The line `df = spark.read.format("csv").load("file.csv")` loads the indicated CSV file. Correct?

6.6 Model serving

Model serving refers to the provisioning and lifecycle management of trained machine-learning models, essentially statistical models, as a service. Such models contain read-optimised information about quantitative or categorical features found in the training data. The models allow inference and subsequent decision-making based on input data sent as a request to the service. The lifecycle management includes input data preparation, versioned training, testing and validation with the primary goal of achieving a high accuracy of predictions. Specific management tasks therefore encompass systematic model (re)training (split, test, validate, regression testing), version control (input data provenance, cleaning, lineage), delivery of model data and code (packaging, containerisation, registration, discovery, serving at scale) and operational concerns (authentication, logging, metering, monitoring). Access to such models for the purposes of inference and prediction should be possible from multiple applications, with high reliability and low latency. Accordingly, APIs for matching and prediction services should be generated automatically and integrate with workflow managers and message brokers. Among often-used model serving implementations are Iguazio, Kubeflow/KFserving, TensorFlow Serving/TFX, MLflow, Clipper and BentoML. The latter is briefly introduced here.

6.6.1 BentoML model serving

BentoML is a specialised middleware system to register machine-trained mathematical models and perform scalable inference on them. It comes with support for a number of ML libraries such as MLflow, TensorFlow, PyTorch, Keras, CatBoost, LightGBM, ONNX and Scikit-Learn. It is called a unified model serving framework due to this multi-format support. The models are transmitted as Python code via function call to BentoML which then physically stores them in a local directory in Pickle format. Hence, model persistence is achieved through files and directories, which can be optionally versioned by the use of Git. Moreover, BentoML allows for the automatic containerisation of models with Docker, with the containers starting a ready-to-use API for matching and inference. Moreover, it integrates with Airflow for model training pipelines or

workflows, specifically with Airflow's PythonOperator and PythonVirtualenvOperator as well as with Flink's stream processing capabilities for streaming model inference and with MLflow.

A common way to use BentoML is to create the predictor code in the form of a generic Python model class. The class should have a `__call__(inputlist)` method. Instantiated with input data, this becomes a predictor as Python model instance with data already loaded. This model can be persisted as BentoML saved model via `bentoml.pickable_model.save_model(<modelname>, <modelinstance()>)`, and executed with a runner. The runner may be implemented as follows:

```
import bentoml
# next line is only for testing
lmodel_content = bentoml.picklable_model.load_model("<
    modelname>:latest")
lmodel = bentoml.picklable_model.get("<modelname>:latest")
runner = lmodel.to_runner()
runner.init_local()
prediction = runner.run(<inputlist>)
print(prediction)
```

All saved models may be inspected in the shell with `bentoml models list`. For production use and access by multiple applications, custom service logic can then be defined in Python and delivered through an instance of the BentoML service. The implementation requires setting up a service object by instantiating `bentoml.Service(<service-id>, runners)` which then uses the `@<service-object>.api` decorator to specify input and output data formats. Again, a brief example is given:

```
import bentoml
import bentoml.io
runner = bentoml.picklable_model.get("<modelname>:latest").
    to_runner()
svc = bentoml.Service(<modelname>, runners=[runner])
@svc.api(input=bentoml.io.JSON(), output=bentoml.io.JSON())
def predict(inputlist):
    return runner.run(inputlist)
```

These services are launched through `bentoml serve <python-filename>:<service-object> --reload`. Tools like Curl can then be used to run predictions, for example: `curl -X POST -H "Content-Type: application/json" --data "[...]" http://127.0.0.1:3000/predict`.

BentoML services can also be exported as self-contained units with both algorithmic service logic and model data, through the command `bentoml build` based on instructions given in a service description file `bentofile.yaml`. A minimal service description would be `service: "<modelname>:svc"`. All

such built units can be inspected with `bentoml list`, and exported for model transfer with `bentoml export <modelname>:latest <modelname>.bento`.

These units can in turn then be converted into Docker container images via `bentoml containerize <modelname>:latest`, based on additional `docker` entries in the YAML file specifying the base image and Python interpreter version. Containers then serve the model and allow for using it (e.g. for predictions) on port 3000. Hence, they can be run individually with the command `docker run -it -rm -p 3000:3000 <modelname>:<tag> serve`.

A step-by-step documentation with further information is available on the BentoML website.[21]

Repetition

1. Why are there dedicated model serving implementations when instead a web server could be used for distributed access to a model?
2. Why does Bento need a POST request for the prediction? Predictions are read-only and thus should work with a GET request.

6.7 Data integration

Data integration refers to the ability to shuffle data between data stores, or between applications, while performing conversion and transformation. Format conversion would, for instance, take CSV input from a source and deliver it as JSON to a sink. Transformations happen within the same format and would filter, aggregate or augment the input data. There are many approaches to data integration referred to as ETL or ELT, a reference to the order of steps in Extract-Transform-Load integration processes. Recent implementations to set up generic integrations include Meltano, Arrow Flight and dbt-core.

6.7.1 Meltano data integration

Meltano aids in setting up pipelines for large-scale data transformations from sources to destinations. It is installed through pipx with the command `pipx install meltano` and can then be activated for a project directory via `meltano init <projectname>`. For the affected directory, Meltano creates a number of default files and empty directories and administers several runtime environments such as development, staging and production. The main configuration file is `meltano.yaml`, which has the default environment set to development (`dev`). The project directory is designed in a way that it can be curated in Git, so that an additional `git init` is recommended. In their initial state, all

[21] Bento documentation: `https://docs.bentoml.org/en/latest/concepts/model.html`

environments are empty and waiting to be configured with a sink-source combination or, in Meltano terms, either extractor-loader or tap-target combination. The configuration may additionally encompass transforms and transformers, files, utilities and other pipeline ingredients.

The commands `meltano discover extractors` and `meltano discover loaders` are used to define the data transformation pipeline. There are more than 500 extractors available, alls prefixed with `tap`, and more than 30 loaders, all prefixed with `target`. Many exist in multiple implementation variants, so that, in practice, extracting data from a system or loading it into another system may only work well with specific variants. For the purpose of testing and interoperability, there are some generic loaders such as JSON files (`target-jsonl`).

The extractor-loader combination is installed into the current Meltano environment, but not yet configured, with `meltano add <extractor/loader> [--variant=<variant>]`. This command adds the chosen extractor or loader to `meltano.yaml` but also places its implementation into the `plugins` directory. Subsequently, a mandatory configuration is conducted with `meltano config <extractor/loader> set --interactive`, which asks the user about few to many key-value settings such as endpoints, credentials and options. Moreover, `meltano select <extractor> --list --all` followed by `meltano select <extractor> <table> <column>` selects a source table (entity) and column (attribute) in case multiple are offered by the extractor.

Finally, `meltano run` runs the pipeline that in turn extracts data from the configured source and sends and loads transformed data into the configured destination. While the run command runs once, a regular run to catch updates to the data source can also be configured with the `meltano schedule` command and activated with `meltano invoke`, both taking additional parameters. Schedules in particular can be set up with commands of the form `meltano schedule add <schedname> --extractor <extractor> --loader <loader> --transform [run|skip|only] --interval "@hourly"`. Many commands are predefined and others can be added with some configuration, including Jupyter notebooks as steps within a pipeline. To maintain an overview about pipelines, a web-based user interface is also available through the blocking command `meltano ui` for subsequent access at `http://localhost:5000/`.

A detailed documentation is available online.[22] Hundreds of plugins, specifically for different data sources, are available from the hub.[23] A managed service for Meltano is not yet available but expected to arrive sometime in the near future. There are also more complex alternatives to Meltano such as Pachyderm that require a more sophisticated deployment and configuration.[24]

[22] Meltano documentation: `https://docs.meltano.com/getting-started`
[23] Meltano hub: `https://hub.meltano.com/`
[24] Pachyderm: `https://www.pachyderm.com/`

Repetition

1. There are targets, loaders, extractors and taps. Which of these terms are practically synonyms?
2. What is the advantage of maintaining the data integration configuration in a Git repository?

6.8 Workflows and distributed scheduling

Meltano and other data integration tools represent static workflows with a source, a sink and a set of transformation rules. In many scenarios, more complex workflows with tree and graph structures need to be set up and triggered through various events including matching times. Scheduled task invocation beyond the use of Cron is broadly supported in programming environments, for instance, through Celery or, with emphasis on parallelisms similar to Spark, through Dask. Such workflows may however also occur in continuous delivery scenarios triggered by a change in data or code, for instance, via Git hooks and corresponding workflows like Gitlab Pipelines. The complexity increases in workflow languages permitting subtasks, branching, looping, nesting and other internal control structures. Corresponding workflow systems need to support automation (scripting and APIs), robustness (checkpointing, restarts) and optimisation (caching, adaptive re-entrant execution). They also need to support regulated environments through logging as well as lifecycle and user management. Finally, they should be flexible concerning the task implementation: as shell commands, Python functions, API calls, ETL processes and others. In the following, Airflow is introduced as a representative system for both scheduled invocations and workflows, supporting many of the listed requirements.

6.8.1 Airflow task and workflow specification

Apache Airflow[25] is a workflow engine, a scheduler and a programming interface to express complex executable workflows. Workflows are expressed as DAGs in Python in three main parts: metadata, a set of tasks and an execution order specification for those tasks. A task is triggered by a timer event, by a previous task having finished, or by a data dependency having changed. For that purpose, Airflow makes available an `airflow` Python module, and a script file importing this module and making use of it represents a workflow. The package with module and executable can be installed into a virtual environment with `pip install airflow`. The service start command `airflow`

[25] Airflow website: `https://airflow.apache.org/`

`standalone` listens on two ports: 8793 for internal scheduling purposes and 8080 for the web interface. The following listing shows a customised and decomposed start sequence.

```
airflow db init
airflow users create \
--username admin \
--firstname X \
--lastname Y \
--role Admin \
--email Z # enter password, or use --password
airflow webserver --port 8080
airflow scheduler
```

With `airflow dags list --subdir .`, all DAGs present in the current working directory are summarised. If needed, calling `airflow dags test --subdir . <dag_id>` tests the execution of a specific workflow.

The `DAG` class from the `airflow` module needs to be instantiated with a unique identifier given as `dag_id`. A particular design decision of Airflow is that workflows are addressed later by this identifier, independently of the name of the containing Python file. Further parameters refer to the workflow ownership and its timing information, including start date and interval in Quartz cron syntax. Hence, a typical Airflow specification may start as follows:

```
from datetime import timedelta
import datetime
import airflow
import airflow.utils.dates

with airflow.DAG(
  dag_id="helloworld_bash",
  default_args={"owner": "airflow"},
  schedule_interval="0 0 * * *",
  start_date=airflow.utils.dates.days_ago(2),
  dagrun_timeout=datetime.timedelta(minutes=60),
) as dag:
  ...
```

Tasks are specified as instances of operator classes along with a unique identifier and operator-specific parameters. The identifier is a string given as `task_id` parameter. The instances can be named differently but for simple cases may follow a convention of `t1`, `t2` and so forth. The following operator classes are among the often used ones:

1. EmptyOperator. This operator does nothing (no-op) and has no further parameters. It can be used as placeholder in a workflow, similar to a `pass` instruction in Python.

2. PythonOperator. Executes a Python function that must be reachable, either within the same file or from an imported module. The function is referenced with the `python_callable` attribute, and is completed by the keywords argument `op_kwargs` to specify parameters that should be passed to the invoked function. Alternatively, the Python operator can directly be specified as decorator of an existing function, like this: `@task(task_id="...")`.

3. PythonVirtualenvOperator. A variant that supports custom module deployment into virtual environments.

4. BranchPythonOperator. This special operator makes it possible to choose one out of several branches by evaluating a condition.

5. BashOperator. This operator works similar to `os.system()` by synchronously executing one shell command. The command is passed as `bash_command` parameter.

6. PapermillOperator. Loads a Jupyter notebook from the absolute path in `input_nb` and, since such notebooks are modified during execution, saves the result notebook as `output_nb`. Optionally, a dictionary `parameters` may be passed to parameterise the notebook according to the Papermill conventions, which are documented in the Jupyter notebook section of the book.

Further operators exist for system and data integration, including SimpleHttpOperator, EmailOperator, MysqlOperator and PostgresOperator. With those operators, Airflow overlaps functionally with Meltano, although both can also be integrated with `meltano add orchestrator airflow` followed by `meltano invoke airflow scheduler` and `meltano invoke airflow dags list`. The invocation of operators follows tristate semantics: 0 signals success, 99 signals skipping, and all other numeric codes signal failure.

The last line of a simple Airflow file specifies the order of execution and therefore constructs the actual workflow. Two greater-than signs are used to create a sequence, e.g. `t1 >> t2`. Tasks that can be parallelised due to not depending on each other's results are grouped in square brackets, e.g. `[t3, t4]`.

Repetition

1. A BashOperator t1 creates a file, and a PythonOperator t2 reads that file. Would [t1, t2] be a valid workflow?
2. What does the schedule interval used in the example (* * 0 0 0) mean?

Chapter 7

Collaboration and Governance Platforms

In contrast to the last two sections that focused on locally run shell tools and locally operated middleware, this section dives deeper into provisioned and managed middleware online platforms to accomplish data science-related governance and collaboration tasks. These platforms typically offer a web interface and, while they can be operated locally for a single user, have the capability to serve multiple users in teams and thus facilitate collaboration supported by extensive authentication and authorisation schemes. Consequently, they are often consumed as pre-provisioned setups and managed services supporting the data science workflows on a team or institution basis with oversight, change management and compliance concerns.

Three exemplary collaboration platforms with suitable open source licencing are introduced: Jupyter, Gitlab and Open Data Discovery. Data scientists should be familiar with the main workflows within these platforms and are then able to also use similar platforms. Such alternatives are mentioned in each respective section. Where applicable, possibilities for integration of the previously mentioned tools and middleware are highlighted.

7.1 Scientific notebooks

A digital scientific notebook is a web-based environment subdivided into input cells that either contain static information or run dynamic code, such as Python, R or Julia. Output cells then contain text, plots and other content. In contrast to a script, the cells can be run selectively. This might speed up explorative data analysis, but it might also lead to inconsistencies when the dependency order between cells is not manually considered. Moreover, there

is no explicit termination unless the code execution context (kernel) is terminated. This means allocated resources may be blocked unless explicitly freed up programmatically or implicitly by kernel termination.

Various notebook implementations with single-language or polyglot kernels exist, such as Jupyter which is described next, Apache Zeppelin[1], Querybook[2], and Polynote[3].

7.1.1 Jupyter notebooks

Jupyter is a popular implementation of notebooks that can run in isolation, but also shared among team members. In Python Jupyter notebooks, the kernel can be chosen; by default, the iPython interpreter is used with support for special commands starting with `!` (system execution) or `#`. Big data frameworks such as PySpark can be integrated into notebooks through the corresponding Python modules to perform parallel computing tasks by offloading them to a larger cluster. There are also Jupyter extensions such as Toree and BeakerX for Spark, and Elyra to visually design workflows that can then be mapped to Airflow or Kubeflow implementations.

Jupyter can be operated as single instance, catering to a single user or multiple users in a fully shared environment without isolation. It can also run as JupyterLab, combining the notebook with web-based text editors, terminals, launchers and custom widgets, while still catering to single users. For multi-user environments with isolation between users, JupyterHub can launch containerised notebooks, including on container orchestrators such as Kubernetes.

7.1.2 Working with notebooks

Launching an interactive single-user or fully shared notebook environment locally is achieved with the subcommand `jupyter notebook` or the alias command `jupyter-notebook` (with dash). The command hangs to serve a web browser at the URL containing a secret token indicated at standard output, and can be terminated with the keyboard combination Ctrl+C. The token can be disabled by running `jupyter notebook password` and thus setting a password permanently. Headless mode can subsequently be used by adding `--no-browser`. The default port number is 8888 and can be changed with the parameter `--port`. Furthermore, notebooks only listen to local connections by default. To be network-enabled, the parameter `--ip 0.0.0.0` needs to be set. The notebook serves the entire current working directory available. One

[1] Zeppelin website: `https://zeppelin.apache.org/`
[2] Querybook website: `https://www.querybook.org/`
[3] Polynote website: `https://polynote.org/latest/`

further important parameter is therefore `--notebook-dir <dir>` to confine the access to a certain directory.

A number of configuration settings becomes available through files only. This includes the ability for users to switch off the running Jupyter instance with a Quit button. In the file `~/.jupyter/jupyter_notebook_config.json`, the setting `"quit_button": false` removes this ability. JavaScript-based customisations such as default contents in new notebook can furthermore be performed by placing appropriate files into the directory `/usr/share/jupyter/nbextensions/` and activating the extension with a JSON configuration file in `/etc/jupyter/nbconfig/notebook.d/`.

The batch execution of notebooks including parameterisation is the task of Papermill[4], which can be installed with `pip install papermill` and requires adding `~/.local/bin` to `$PATH`.

Notebook files are stored with the extension `.ipynb` (from the original name iPython notebook) and can also be exported as Python scripts to facilitate automation.

Jupyter notebooks can be tried out for free[5] although that is not recommended for important scripts or confidential data. A more convenient service is Binder to execute notebooks already stored on the Internet in a Git repository.[6]

7.2 Code and data lifecycle management

The management and easy use of versioned repositories, possibly backed by Git or other version control system, is the domain of online repository frontends. They provide functionality for creating hierarchies of projects and subprojects with multiple repositories, assigning access rights, web-based read access to existing files and even web-based write access for the creation of new files. Such collaborative platforms also allow for planning and discussing the content in terms of code and data, for forking variants and for submitting wishlist and bug reports. Moreover, they support actions based on changes to the code or data, such as the automated rebuilding of binary packages.

Apart from shell-oriented tools such as Gitolite, and the built-in Gitweb as well as cgit, there are many web-based Git management platforms. Gitlab is one of them and is presented next. Potentially leaner alternatives include Gogs[7], Gitea[8] and Trac[9].

[4] Papermill documentation: `https://papermill.readthedocs.io/`

[5] Jupyter demonstration: `https://jupyter.org/try-jupyter/retro/notebooks/?path=notebooks/Intro.ipynb`

[6] MyBinder: `https://mybinder.org/`

[7] Gogs website: `https://gogs.io/`

[8] Gitea website: `https://gitea.io/en-us/`

[9] Trac website: `https://trac.edgewall.org/`

7.2.1 Gitlab as repository management platform

Gitlab is a collaborative environment around a fully managed set of Git repositories. The most obvious functionality of the platform is the creation of personal projects and project groups and the definition of Git repositories within these projects. On the collaboration side, it allows sending invitations to users who are added with differentiated access rights into the projects depending on the chosen role. Full access rights are available to owners. Creating a new project results in ownership automatically. Fewer access rights are available for maintainers, developers, reporters and finally guests. For instance, only an owner can delete a project, and only owners and maintainers can rename a project or change its settings.

The platform supports patch management in the form of pull requests for those not sufficiently privileged or unsure about direct file modification in one of the Git branches. Moreover, it supports the definition of server-side hooks that run whenever a commit is pushed. This feature can be used to implement a continuous data processing pipeline, for instance, updated training whenever new data arrives.

Finally, Gitlab also contains registries for packages (e.g. Python packages similar to PyPI) and container images. The Docker container registry runs on port 5050. The command `docker login <gitlabserver>:5050` configures the local Docker client to make use of it. Images can then be pushed to `<gitlabserver>:5050/<user>/<project>/<image>`.

Due to many system integrations and complex configuration, the native installation of Gitlab requires elevated privileges and several required post-installation configuration steps. The installation process is described in the documentation.[10] A more portable approach is to run Gitlab as a container, in the following form:

```
export GITLAB_HOME=$HOME/gitlab
sudo docker run -ti \
  --hostname localhost \
  --publish 30000:80 \ # Container port 80 becomes 30000 on
      the host
  --name gitlab \
  --restart always \
  --volume $GITLAB_HOME/config:/etc/gitlab \
  --volume $GITLAB_HOME/logs:/var/log/gitlab \
  --volume $GITLAB_HOME/data:/var/opt/gitlab \
  --shm-size 256m \
  gitlab/gitlab-ee:latest
```

[10] Gitlab documentation: `https://about.gitlab.com/install/#ubuntu`

For security reasons, a password is auto-generated upon the first invocation. The additional command `sudo docker exec -it gitlab grep 'Password:' /etc/gitlab/initial_root_password` retrieves the password from the container and displays it on standard output.

Gitlab as a managed service[11] is similar to Github, with the added advantage of being open source and thus more fit for self-hosting. The Gitlab CLI can be used to interact with Gitlab instances. An exemplary invocation is to list all active projects with `python-gitlab --server-url https://<gitlabserver> project list`.

7.2.2 Gitlab as delivery platform

Based on any changes to data, code or other managed artefacts, Gitlab can run processes for continuous integration (CI) or continuous delivery (CD). Both terms largely overlap, but the former typically refers to a build–test–release process, whereas the latter refers to the deployment of artefacts to production systems, ready to be delivered to users.

To define what should happen upon a change, Gitlab uses the concept of Git hooks and connects them to pipelines and web hooks. Web hooks are external APIs that get called on certain events such as repository pushes or changes in the issue tracker.

Pipelines executing internally in Github are also represented by a webhook mechanism. The web hook endpoint then follows the form `https://<gitlabserver>/api/v4/projects/27/ref/REF_NAME/trigger/pipeline?token=TKN`. Despite the name, pipelines can be not only basic pipelines in the form of sequences but also DAG workflows representing full workflows or even multi-project pipelines to automate a larger release process. Each pipeline works in defined stages such as build, test or release as well as jobs representing steps of each pipeline. The pipeline definition is edited as a YAML file and can be compared to a makefile in terms of providing targets that trigger command sequences, with each command being a job. The following listing gives an example of a two-stage pipeline sketch:

```
stages:
  - prepare
  - test
image: alpine # global or per job; must contain job scripts
prep_a:
  stage: prepare
  script: # before_script + after_script also available
    - echo "This job prepares something."
    - wget ...
```

[11] Gitlab.com service: `https://about.gitlab.com/`

```
prep_b:
  stage: prepare
  script:
    - echo "This job prepares something else."
    - jupyter-execute ...
test_a:
  stage: test
  script:
    - echo "This job tests something. It will only run when
        all jobs in the"
    - echo "preparation stage are complete."
```

Gitlab runners execute the jobs, for instance, in the form of containerised tools to provide isolation. This way, Gitlab can also execute an Airflow image so that a more sophisticated workflow is initiated under the control of Airflow. Gitlab metadata can also be pulled by Meltano, by using the `tap-gitlab` extractor based on a configured access token in the project's settings menu next to the webhook configuration.

The global Gitlab registry contains re-usable pipeline components such as linters that can be used to rapidly construct useful pipelines for production scenarios.

7.3 Data catalogues and governance

There are two main flavours of these platforms. Some are more focused on publishing single datasets, often file-based, along with metadata. Implementations include CKAN[12] and Magda[13]. Others are more focused on ETL processes, data provenance and lineage. Those include many emerging platforms that are often still non-trivial to deploy, such as Open-Metadata[14], Datahub[15], Amundsen[16] and Apache Atlas[17]. The Open Data Discovery platform (ODD) is also in this camp and are briefly introduced next.

7.3.1 ODD deployment

ODD[18] can be up and running with moderate effort when using the provided container composition. First, the code repository needs to be cloned to get access to all relevant launch files (size ca. 50 MB). The referenced container

[12] CKAN website: `https://ckan.org/`
[13] Magda website: `https://magda.io/`
[14] Open-Metadata website: `https://open-metadata.org/`
[15] Datahub website: `https://datahubproject.io/`
[16] Amundsen website: `https://www.amundsen.io/`
[17] Atlas website: `https://atlas.apache.org/#/`
[18] ODD website: `https://opendatadiscovery.org/`

images are then automatically downloaded (size ca. 900 MB). The instructions are as follows:

```
git clone https://github.com/opendatadiscovery/odd-platform
cd odd-platform/
docker compose -f docker/demo.yaml up -d odd-platform-
    enricher
# or docker-compose in older Docker environments
```

The web interface to the platform is then running on port 8080, whereas the platform's PostgreSQL database can be accessed on port 5432. If the setup happens on a remote virtual machine, the ability to use SSH port forwarding becomes handy again: `ssh -L 10080:localhost:8080 ubuntu@<vmhost>`.

Data can be pulled from many sources through collectors including files, relational tables and message broker topics. MySQL, PostgreSQL, MongoDB, Airflow and Kafka are among the supported systems. In addition to input data, ODD supports transformers, quality checkers and other typical data integration pipeline elements.

Custom collectors can be added as well. In the management interface, one would click on 'add collector' and choose an arbitrary name for it. An access token is generated and can be copied from the interface for pasting into the collector configuration file. For demonstration purposes, one such file for a custom PostgreSQL connector is already provided in the code repository: `docker/config/collector_config.yaml`. With the token pasted into the empty token string within this file, the command `docker compose -f docker/demo.yaml up -d odd-collector` pulls additional Docker images (size ca. 2 GB) and adds the custom data source. In the web interface's catalog menu, filtering can be used to focus only on this source (named `postgresql-step2-test` by default) to show all datasets and other pipeline elements available from this source. For curation purposes, string tags and key-value metadata entries can be added to each element.

Repetition

1. How does the process hierarchy look like if the command os.system("ls") is invoked from a Jupyter notebook?
2. What is the command for pulling Docker images from a Gitlab container registry?

Chapter 8

Execution and Orchestration Platforms

Most of the software covered in this book is deployable in the form of system-wide packages per-user packages, or container images. When software becomes more complex, it might require a lower-level configuration of storage and networking resources, autoscaling rules, proxies and orchestration logic. The execution of containers or of virtual machines under this configuration is then controlled by dedicated platforms. In this chapter, two representative platforms for the execution of virtual machines and containers are briefly introduced: OpenStack and Kubernetes. Moreover, a mapping of those platforms and the previously discussed middleware and collaboration platforms into cloud services is given. The coverage is not meant to be an extensive guide into orchestration but rather to convey sufficient knowledge to be able to deploy data science tools should they require such a setup.

8.1 Virtual machines management

Virtual machines are highly isolated execution contexts on top of abstracted hardware, including both kernel and userland applications in a guest context on top of the host operating system. A hypervisor configures the available hardware resources and ensures that all guest system calls are appropriately translated into host calls. Hardware-supported virtualisation is often available for this task. On Linux, the Kernel Virtual Machine (KVM) has become the dominant hypervisor. However, this still requires managing the virtual machine images that should be executed, along with their configuration known as instance types. For example, a machine learning application might require 4 virtual CPUs (vCPUs), 1 GPU, 8 GB of memory, 100 GB of disk

space, and a public IP address. Tools like `virsh`, `virt-install` or plain `kvm`/`qemu-system-x86_64` are powerful but not easy to operate.

Instead of interacting with the hypervisor directly, setting up all configuration is eased by web-based infrastructure management software. In the next section, OpenStack is introduced. Possible alternatives include Apache CloudStack[1], OpenNebula[2] and Proxmox[3]. Eucalyptus[4] has been one of the first platforms in this space but might be no longer actively maintained. Similarly, Danube Cloud[5] might be a candidate platform to look at with the same caution.

8.1.1 Using OpenStack web interface and API

OpenStack[6] is a set of named components to manage infrastructure, primarily virtual machines and containers. Not all components need to be active, and therefore OpenStack instances differ in their functionality. A typical component combination might be Glance to manage virtual machine images, Nova for interaction with the hypervisor running virtual machine instances, Swift or Cinder for object or block storage, respectively, Keystone for identity management and Heat for orchestration.

All services integrate into Horizon, the web-based management interface. Users log into this interface, see their assigned projects with quotas, and set up virtual machines and other resources like virtual block devices within the project. Accounts, instance types, quotas, projects and other privileged configuration settings are handled by a system administrator. Hence, to get started with an instance that is not self-operated, a user would first have to request access from the administrator, by specifying the desired resources, and would subsequently get the account and the ability to log into Horizon.

Inside the interface, users would register SSH keys, network policies and, optionally, secondary persistent block storage devices. Then, they would launch VMs, each of which comes with a primary block storage device of reasonable but perhaps limited capacity. A typical instance type might have 4 vCPUs, 1 GPU, 8 GB of memory and 40 GB disk space. Then for the aforementioned machine learning application, a secondary block device is necessary. Within the Horizon dashboard, the block device would receive a label, which can then be used to automount it inside the VM by putting a line like `LABEL=seconddisk /home/ubuntu/mountpoint ext4 defaults 0 0` into the file `/etc/fstab`.

[1] CloudStack website: `https://cloudstack.apache.org/`
[2] OpenNebula website: `https://opennebula.io/`
[3] Proxmox website: `https://www.proxmox.com/en/proxmox-ve`
[4] Eucalyptus website: `https://www.eucalyptus.cloud/`
[5] Danube Cloud website: `https://danube.cloud/`
[6] OpenStack website: `https://www.openstack.org/`

By default, VMs are only accessible on an internal network accessible by all VMs, with IP addresses of the form `10.0.x.y`. OpenStack has the concept of floating IP addresses which depending on the quota settings can be assigned to running VMs to make network services on them accessible from outside. A VM then possesses three network interfaces that can be distinguished to enforce access policies: localhost, the internal IP, and the floating IP.

Access to VMs can be granted to users who do not have an OpenStack account, by requesting the desired resource configuration from them, as well as the SSH public key, and setting the VM up for them. The owner of a virtual machine, possibly a data scientist, then merely has the tasks of maintaining the VM itself, including the regular installation of security updates, and ensuring that no unnecessary services are exposed to the world. Regular maintenance also involves checking for resource shortage using `df`, `free` and similar tools introduced previously in this book.

Access to OpenStack is also possible programmatically. The lifecycle management of virtual machines serves as an example. Due to such VMs being managed by Nova, the first step is importing the Python submodule `novaclient.client` and setting up a client object. Next, the list of virtual machines is retrieved. Each machine has a current state and, in case a task such as powering on or off a VM is still running, also a task state. The following code exemplifies starting all switched off VMs inside a project, unless they are already in the process of being started. The authentication URL follows the pattern `https://<openstack-domain>:5000`.

```
import novaclient.client

nova_client = novaclient.client.Client(version="2",
    username=os.getenv("OS_USERNAME"), password=os.getenv("
    OS_PASSWORD"), project_name=os.getenv("OS_PROJECT"),
    auth_url=os.getenv("OS_AUTH_URL"), user_domain_name="
    default", project_domain_name="default")
servers = nova_client.servers.list()
for server in servers:
  status = server._info["status"]
  taskstate = server._info["OS-EXT-STS:task_state"]
  if status == "SHUTOFF" and taskstate is None:
    print("Starting machine", server.name)
    server.start()
```

A nested dictionary of network interfaces is also available through the object attribute `server.addresses`. Each inner dictionary can be checked for the presence of a fixed IP address assignment, for instance, with: `ip["OS-EXT-IPS:type"] == "fixed"`. The field `ip["addr"]` then contains the IP address.

8.2 Container management

A lot of software is shipping with the option to run in containerised form, or even requires container-native deployments. Container orchestration platforms ensure that they can run in production with the right provisioning and scaling characteristics. Over the past years, Kubernetes has emerged as one of the dominant orchestrators, with many features but also correspondingly high complexity. It is briefly introduced in the next sections.

8.2.1 Kubernetes ecosystem

Kubernetes runs as a standalone system or more typically as a cluster involving multiple virtual or physical machines. Within a running instance, different namespaces can be set up to increase isolation. By default, the namespace `kube-system` is reserved for Kubernetes-internal resources.

Kubernetes orchestrates resources of different types, each with corresponding declarative configuration. YAML or JSON files describe resources such as containers instantiated from images (with the types `Deployment`, `StatefulSet`, `Job`/`CronJob` and others), exposed network interfaces (`Service`) and storage areas (`PersistentVolumeClaim`). Many of these resources have a physical representation, such as CPU and memory limit assignments in deployments. Custom resources such as bindings to other platforms can be developed and deployed. Operators are further containerised software components that automate the lifecycle.

Multiple implementations of Kubernetes exist, especially to account for smaller (single-node) operation when horizontal scaling is not required. Among the well-maintained implementations are Minikube[7], K3s[8], MicroK8s[9] and Kind[10]. They differ especially in terms of how easily extensions can be installed and how strictly authentication needs to be performed. There are also multiple platforms building on top of Kubernetes, especially with additional features for application engineers, such as OpenShift and CloudFoundry.

The command-line utility `kubectl` is used to interact with a Kubernetes cluster. It works based on contexts, so that for each cluster a specific context can be set up and working across clusters becomes possible. With this tool, applications can be deployed and configured and the cluster status can be monitored. With OpenShift, the `oc` tool can be largely used as a substitute. Similarly, with Minikube, K3s and MicroK8s, bundled wrapper commands exist (e.g. `k3s kubectl`) so that a separate installation is not necessary.

[7] Minikube website: `https://minikube.sigs.k8s.io/docs/`
[8] K3s website: `https://k3s.io/`
[9] MicroK8s website: `https://microk8s.io/`
[10] Kind website: `https://kind.sigs.k8s.io/`

Helm[11] is a package manager for Kubernetes applications. On the shell, the tool `helm` allows for updating information about packages and installing them to a cluster namespace. These packages, also called Helm charts, are in essence templated resource descriptions along with metadata and dependencies, allowing for a higher-level handling of more complex applications. The Artifact Hub[12] is a popular repository for such applications. There are also alternative tools to adjust applications to specific needs, for instance Kustomize.

8.2.2 Kubernetes installation

In the following, MicroK8s is used as exemplary implementation to obtain a self-hosted Kubernetes instance. On an Ubuntu system, the installation of Kubernetes servers is performed with a dedicated meta-package: `sudo apt-get install kubernetes` followed by `kubernetes install` and choosing option 1, MicroK8s. The metrics server can be enabled as extension with `microk8s enable metrics-server`, and then works out of the box without requiring authentication. With `alias kubectl="microk8s kubectl"`, client-side access to the Kubernetes instance becomes possible.

In case a Kubernetes instance is already provisioned elsewhere, an alternative path needs to be taken. First, the instructions to download it need to be followed[13], and, second, an access context needs to be configured. The following listing shows an exemplary use:

```
# packaged: if available, use that and skip the three
    manual lines below
sudo apt-get install kubernetes-client
# manual: get current version first, and use it instead of
    1.27.2 below
curl -LS https://dl.k8s.io/release/stable.txt
curl -LO https://dl.k8s.io/release/v1.27.2/bin/linux/amd64/
    kubectl
sudo install kubectl /usr/local/bin/kubectl

kubectl config set-cluster <clustername> --server=<host>
kubectl config get-clusters
```

As a first confirmation of a successful deployment, `kubectl get all` should show all resources in the `default` namespace. In particular, there should be a Kubernetes service (`service/kubernetes`) with a cluster IP address. Other namespaces can be chosen, for instance by appending `--namespace kube-system`. To get a list of all namespaces, `kubectl get namespaces` is

[11] Helm website: `https://helm.sh/`
[12] Hub website: `https://artifacthub.io/`
[13] Kubectl guide: `https://kubernetes.io/de/docs/tasks/tools/install-kubectl/`

helpful. With `kubectl top nodes`, statistics about the resource consumption can be shown, and with `kubectl api-resources`, all resource types known to the instance are shown as well. Access to internal information can be achieved by combining the tool with post-processing, as in: `kubectl get service --namespace kube-system -o json | jq ".items[0].spec.clusterIP"`.

When these commands work as expected, the Kubernetes instance can be used to deploy applications.

8.2.3 Working with Kubernetes

Assuming a working Kubernetes instance and properly installed tools (`kubectl` and `helm`), an application can be installed in two ways. If it is packaged for Helm, the steps follow the sequence of commands listed below, assuming a self-chosen release name:

```
kubectl create namespace <namespace>
helm repo add <name> <repo-url> # for custom applications
    not in Artifact Hub
helm repo update
helm search repo <searchterm> # or 'hub' for Artifact Hub;
    lists charts
helm install <release-name> <chart> # simple test
    installation; more complete:
helm upgrade --namespace <namespace> --install <release-
    name> <chart>
```

Otherwise, if only YAML files are provided, a typical sequence would look as follows:

```
kubectl create namespace <namespace>
kubectl apply -f <app>.yaml --namespace=<namespace>
kubectl get deployments # verify that deployments show up
```

Static scaling of containers can be set up directly as configuration instructions on the container-related resources such as `Deployment`. For vertical and horizontal autoscaling, more sophisticated resources exist, including `VerticalPodAutoscaler` (VPA). Whether or not a certain autoscaling mechanism is offered in a Kubernetes instance can be found out by querying the available APIs with `kubectl get --raw /apis`. It contains the namespaced list of API groups. Those of relevance to autoscaling are `autoscaling/hpa` for horizontal scaling, `autoscaling.k8s.io/vpa` for vertical scaling and finally `keda.sh/keda` for event-driven scaling. Most scaling mechanisms (static, VPA and HPA) are reactive based on CPU and memory consumption, whereas KEDA can also react on network traffic and other events. In contrast to the

others, VPA is able to calculate recommendations on predicted resource requirements. Usually, the autoscalers are used in combination with the metrics server, `metrics.k8s.io/metrics`.

```
# configure HPA autoscaling
kubectl autoscale deployment <deployment> --cpu-percent=50
    --min=1 --max=3
# verify HPA autoscaling status
kubectl get hpa -o json | jq ".items[0].status"
```

8.3 Cloud services

Cloud computing refers to the on-demand provisioning of programmable services representing infrastructure, platforms, applications and data. Application orchestration in commercially offered clouds combines execution in virtual machines, containers and other environments with managed middleware services. Long-running virtual machines and containers are typically offered as Infrastructure-as-a-Service (IaaS), containers also as Container-as-a-Service (CaaS) hosting. Short-running containers and function implementations are also offered as runtime for Function-as-a-Service (FaaS). Likewise, databases may be offered as DBaaS, and other middleware as Backend-as-a-Service (BaaS).

Many cloud providers run Kubernetes for CaaS, along with KNative for scale-to-zero containers as backing for FaaS. Moreover, some providers also run registries for VM and container images. Hence, working with a cloud is not difficult when the fundamental technologies and platforms are understood, primarily by mapping the cloud service product names to the names of the platform and infrastructure technologies. Sometimes, cloud providers are explicit about what technology is used under the hood, and sometimes this can be found out by trying. Nevertheless, some cloud services also run proprietary software not available outside of the provider, and some run heavily modified or customised software. In the interest of interoperability and avoidance of vendor lock-in, the subscription to such services should be assessed carefully.

Repetition

1. Is a virtual machine managed in OpenStack aware of its public IP address?
2. Does Kubernetes support proactive autoscaling?

Chapter 9

Global Infrastructure

Beyond the individual hosted tools and online platforms, some online platforms offer much more combined functionality and thus qualify as complete self-service infrastructure for data scientists. While it may be technically possible to operate them locally, their complexity no longer permits doing so in an affordable manner. This holds for platforms located in a single region and even more so for truly globally distributed infrastructure such as content delivery networks (CDNs) and edge clouds, serving their users in geographic proximity. To give a second reason, the reliance of the platforms on social interaction workflows often stimulates a single global instance instead of isolated personal or institution-internal instances. Similarly, having a single global entrypoint facilitates discovery and bootstrapping, and a single global data broker for non-confidential data facilitates integration. Hence, the selection of appropriate platforms, account creation, subscription management and reliance on internal administrators or external operators needs to be factored into the equation when designing automation for data science.

In general, there are two paths towards global platforms: federation and dominance. To compare, consider the case of online social networks: In many parts of the world, Twitter had become the dominant commercial platform for broadcasting short messages. Mastodon, XMPP and other implementations allow for federation and therefore decentralised control of the hosting but, lacking the budget power, would not cause the same necessary network effect. In data science, few platforms support federation, and therefore global infrastructure is primarily reliant on dominant players, both full-stack cloud/CDN providers and providers of other platforms. Examplary platforms with lock-in risks include Github, social networks and hyperscalers. While most of those platforms are commercially operated based on mandatory subscriptions, there are always long-lasting global infrastructure providers with free tiers or even entirely free offerings that are especially suitable for learning the technology.

In the following, two worthwhile global infrastructure platforms based on open source implementations are described: Renku Lab and Zenodo. They are complemented with a national infrastructure service on the data input side: OpenTransportData. Together, they can be used to form a whole data pipeline without self-operated infrastructure. Complementary services for discovery (Etcd) and streaming (Dweet) are also described to facilitate distributed applications with various data processing and analytics components. One commonality is that these services and platforms are available for free and invite studying their functionality without having to put a credit card on file. They are also fairly easy to get started with and automate without unnecessary obstacles such as two-factor authentication, which may have their justification in production but are not helpful in learning situations. As with all online services, it is advised to carefully consider the use of pseudonym identities and temporary e-mail addresses while trying out the services, while maintaining discipline knowing that they are offered free of charge although real cost occurs for their operation.

9.1 Data pipeline infrastructure

A typical data pipeline involves one or more data sources, reproducible insights generation, and a long-term archive for the results data. In the following, the already available global infrastructure to set up such a pipeline is described. It starts with the acquisition of public data from OpenTransportData, followed by its collection and processing in Renku, and the public archiving in Zenodo. In turn, Zenodo might again serve as one of the input data sources for further pipelines...

9.1.1 OpenTransportData

There are many sources for public data. Well-curated ones with suitable licencing and reasonable long-term availability are open government data (OGD) available from many countries and their subordinate administrative levels. A global view on OGD in selected countries is available from the Open Data Barometer[1] or the older and now archived Global Open Data Index[2]. Individual collections are available from countries like Switzerland[3] or the USA[4] or organisations like the European Union[5] or the United Nations[6].

[1] OGD barometer: `https://opendatabarometer.org/`
[2] Index: `http://index.okfn.org/`
[3] CH data: `https://opendata.swiss/de`
[4] US data: `https://data.gov/`
[5] EU data: `https://data.europa.eu/en`
[6] UN data: `https://data.un.org/`

Complementary to such top-down approaches, community curation of data has led to DBpedia[7] and Wikidata[8]. A third source category are real-time updates mostly from measured data. One can fetch public real-time data on earthquakes[9], consume dweets[10], and inspect live camera feeds[11].

OpenTransportData serves as exemplary input data platform for a global pipeline. It offers access to static and real-time data related to public transportation such as networks, routes, schedules and delays.[12] While it is country-specific, many of the data formats and associated algorithms can also be found in other places, as evidenced by the global Mobility Database referencing almost 2000 static schedules.[13] The OpenTransportData site is among the higher-quality contenders as it is based on a tightly integrated national transportation system and also offers various APIs such as a journey planner. Registration is necessary to obtain an API key for the journey planner and the real-time feed, whereas static data retrieval and some of the other APIs do not require registration as long as temporal request limits are adhered to. The first example translates a coordinates pair to a list of nearby stations, returned as a JSON structure. It should be noted that x refers to the latitude and y to the longitude, in degrees:

```
curl 'http://transport.opendata.ch/v1/locations?x=47.006001
    &y=9.106130'
```

The second example demonstrates retrieval of the live data, including historic live updates dating back to up to a week, in JSON format amounting to hundreds of thousands of lines:

```
# register key first, or use test key for occasional use
testkey=57c5dbbbf1fe4d000100001842c323fa9ff44fbba0b9b925f0c
    052d1
baseurl=https://api.opentransportdata.swiss/gtfsrt2020
curl -H "Content-type: text/xml" -H "Authorization: $testkey
   " $baseurl?format=JSON
```

9.1.2 Renku Lab

Renku Lab[14] is the reference deployment of the open source Renku infrastructure. Renku fosters the collaboration between data scientists and supports

[7] DBpedia: https://www.dbpedia.org/resources/
[8] Wikidata: https://www.wikidata.org/wiki/Wikidata:Main_Page
[9] Earthquakes: https://earthquake.usgs.gov/earthquakes/feed/v1.0/
[10] Dweets: http://dweet.io/see
[11] Insecam: http://www.insecam.org/en/byrating/
[12] OpenTransportData: https://opentransportdata.swiss/en/
[13] Mobility Database: https://database.mobilitydata.org/
[14] Renku website: https://renkulab.io/

reproducible data-driven workflows based on version controlled data, containers and a unique knowledge graph. With Renku, the implications of changes on the input side (i.e. a modified dataset) on the output (i.e. workflow results) become traceable.

In Renku, data and scripts are stored in an integrated Gitlab instance. The Git repository needs to adhere to a specific structure, which is accomplished by running the appropriate Renku commands that create the directories and perform the file modifications as needed. Scripts are typically defined as Jupyter notebooks containing cells of Python code mixed with documentation and reference results.

Experiments are defined as sequential workflows that run the scripts and build a knowledge graph that shows the dependencies and how changes in the code or input data lead to different results in consecutive executions. This way, accidental degradations can be detected and mitigated quickly. The following example shows how to use the `renku` wrapper command to track a script invocation. This commands can be invoked in Renku's integrated Jupyter Lab terminal and will produce a knowledge graph on how modifications to a CSV file were made:

```
# make sure there is a dataset; if not, create it with an
    arbitrary CSV file and register it under a specific name
     <dsname>
renku dataset add --create <dsname> <filename.csv>
renku dataset ls
# inspect the dataset
ls data/<dsname>
# track script invocation
renku run --name <wfname> --no-output rm data/<dsname>/<
    filename.csv>
# check status and save session for later use in case there
     are changes
renku status
renku save
# re-run the recorded invocation for all operations
    affecting a certain file
renku rerun data/<dsname>/<filename.csv>
```

Further commands stress the workflow nature of recorded invocations.

```
# show the workflow
renku workflow show <wfname>
# re-execute a recorded invocation
renku workflow execute <wfname>
# compose a workflow as sequence of multiple existing
    workflows
```

```
renku workflow compose --link-all <wfname> <wf1name> <
    wf2name>
```

Renku also offers an API in addition to the graphical web interface, and a command-line client (Renku Client) to interface with this API from the command line. Alternatively, the API can be used from custom Python applications. Moreover, Renku integrates with public data repositories such as Zenodo and Dataverse in order to import existing datasets.

How to use Renku step by step is well explained in the first-steps tutorial.[15]

9.1.3 Zenodo

Working with data in research settings requires platforms to exchange files and larger datasets. While Git is suitable for data engineering processes, it has comparatively few ways to add publishing information to data or declare relationships between datasets. Moreover, despite technical support for large files, it might not be the best tool to store such files for long-term public access in the first place.

Zenodo[16] is a platform for sharing research data which also archives larger datasets and thus guarantees their availability for further exploration and analysis. Most of the interaction with Zenodo is web-based, such as uploading datasets, describing it with metadata, creating a community and tagging the dataset with the community. Datasets can be imported from Git (and thus also from Renku) and receive a unique publication record in the form of a digital object identifier (DOI). An HTTP API is available to upload datasets but also to search for existing data. An access token must be obtained before being able to use the API for write access such as data deposits. Reading works without such a key, as evidenced by the following example that retrieves the first ten records from a search result and also gives the URLs for navigating to further records: `curl https://zenodo.org/api/records/?q=serverless | json_pp > records.json`. The complete search syntax is described online.[17]

9.2 Distributed applications infrastructure

Building a globally distributed application requires the ability to reach out from one component to another. This is accomplished through a naming and discovery service. Etcd offers such a service. Once the addresses and other metadata of application components are known, communication in the form

[15] One of the Renku tutorials: `https://renku.readthedocs.io/en/latest/tutorials/01_firststeps.html`

[16] Zenodo website: `https://zenodo.org/`

[17] Zenodo search: `https://developers.zenodo.org/#records`

of message delivery between them can start. Dweet offers the appropriate interface for the sending and receiving components.

9.2.1 Etcd Discovery

Distributed systems, and even fully decentralised systems, need a way to bootstrap and identify endpoints. There are many ways to accomplish that in a fault-tolerant way beyond local-only approaches such as environment variables: with a list of meta-servers to maintain a decentralised approach, with registration of records into the DNS of a domain name or informative files placed into the root directory of a web server running on these domain names or finally with a global discovery service. Etcd is a distributed reliable key-value store which for its own purposes, but also for other applications, makes such a discovery service available.[18] In Etcd, key-value pairs can be nested, essentially forming a hierarchy of such entries.

The first step is to request a universally unique identifier (UUID), which is mathematically speaking not entirely unique but sufficient for all practical purposes. The secured HTTP request `curl https://discovery.etcd.io/new?size=5` would create such an identifier allocated for an application with 5 components. Those may be replicas of a single service, or arbitrary components. The response is a single URL that encodes the identifier as a path on the base URL and serves as endpoint for further operations. A typical undashed UUID may thus look like `24f7beedc1e99d5f02ca3cab05124b4e`.

In a subsequent step, naming information about the components are registered. For instance, a component `node1` may want to inform other components of its presence and endpoint. The HTTP write request `curl -X PUT https://discovery.etcd.io/<UUID>/node1 -d value="myip=127.0.0.1"` would register that information. The response in JSON format contains the key as `/<UUID>/node1` and the value as specified in the request.

Hence, a subsequent read request on the identifier makes that information available to other processes or application components:

```
$ curl https://discovery.etcd.io/<UUID>
{"action":"get","node":{"key":"/<UUID>","dir":true,"nodes"
    :[{"key":"/<UUID>/node1","value":"myip=127.0.0.1","
    modifiedIndex":...,"createdIndex":...}],"modifiedIndex"
    :...,"createdIndex":...}}
```

9.2.2 Dweet

Sending and receiving non-confidential messages on arbitrary streaming channels, for instance, in IoT scenarios or other forms of machine-generated data,

[18] Etcd discovery service: `http://discovery.etcd.io/`

is the task of several IoT platforms. They range from standalone or niche platforms to IoT integration capabilities of clouds. A globally usable platform with low entry barrier for occasional needs is Dweet[19].

The interface to publish simple key-value information is, perhaps slightly deviating from protocol conventions, an HTTP GET request of the form `https://dweet.io/dweet/for/<channelname>?k=v[&k2=v2...]`. An HTTP POST mechanism is also available which conforms better to standards and accepts more complex JSON-formatted structured data. Both interfaces respond with a JSON message about the success status and a transaction number. For instance, a script may regularly publish simple information about the available free memory on a machine. It uses a static channel name related to the local identity of the machine, but by combining with the Etcd discovery service introduced beforehand, globally unique names could also be achieved, as follows:

```
freemem=‘LANG=C free -m | grep Mem | awk ’{print $7}’‘
msg="lots%20of%20free%20memory"

while true
do
        curl "https://dweet.io/dweet/for/my::notebook?
            fremem=$freemem&msg=$msg"
        sleep 5
done
```

A consumer script would then retrieve this information. The HTTP connection is kept open to facilitate streaming. Thus, a continuous retrieval could be achieved with the following invocation: `curl https://dweet.io/listen/for/dweets/from/my::notebook`.

Repetition

1. How many stations can be found at the coordinate 47.00° north and 9.00° east?
2. How many datasets with R scripts does Zenodo provide on the topic of rattlesnakes?

[19] Dweet website: `http://dweet.io/`

Solutions

This part of the book contains answers and solutions to the repetition questions and tasks.

2. Concepts: Programming, Data Representation and DataOps

1. Tasks may be in sequential order, connecting the input of one with the output of another; or executing in parallel. Or they may be unrelated, for instance, in a sequence that has other tasks in between. 2. Map invocations are isolated from each other and can be parallelised without side effects or mutual dependencies. The map-reduce computing paradigm relies on this characterisation. 3. It might be a cost-effective and rapid alternative, but it comes with risks such as vendor lock-in and inability to customise the analytics logic.

3. Concepts: Operating Systems

1. The OS attempts to use swap files or swap partitions if these have been set up. If they are not set up or also full, the OS terminates a process. It may be the one that requested additional memory pages. 2. Containerisation typically provides lightweight isolation, with emphasis on fast start-up times. Stronger isolation can be achieved with virtualisation. 3. This website is not registered in the DNS. Either it is registered locally in the hosts files and the server is running, in which case the website is shown. Otherwise, a browser error message is provoked.

4. Concepts: Infrastructure

1. No. This IP address is bound to the local network interface and not to any of the physical network cards (NICs) that would accept traffic from other

computers. 2. OpenAPI is the most widely used language to describe web services, although RAML and other alternatives also exist. 3. Due to the latency sensitivity, cloud computing would be more suitable. Despite clouds not supporting hard realtime constraints, they typically have provisioning models with short response times, contrasting the rather batch-oriented HPC models.

5. Applications and Tools

Shells

1. Logins use the current username by default, but that can be changed with: `ssh Y@X`. 2. The transfer would happen with the following command: `scp Z user@server:`. 3. This command: `ssh user@server screen editor`. 4. At least seven wrapper tools have been introduced at that point: `bash -c`, `ssh`, `sudo`, `screen`, `parallel`, `stdbuf`, `timeout`.

Useful shell tools

1. The TERM signal asks the process to terminate itself, but this request could be overridden so that the process keeps running. The KILL signal is unconditional and always leads to immediate termination. 2. With a construct in the following form: `ps wauxf | grep PROC | grep -v grep`. The last part filters all occurrences of grep in the input. 3. The output of `last` contains one line per login session plus two extra lines at the end. Thus, `echo $((‘last | wc -l‘-2))`.

Shell programming

1. There is no (well-known) command `xyz`. In that case, nothing happens. But it could be a custom-installed program with this name, or a local alias definition, or a shell command. Therefore, when unsure, the nature of the command should be found out first with `type` and `which`. 2. This command would ask to terminate the current shell session. It is however intercepted by the shell for safety reasons. Only a `kill -KILL $$` can override the interception.

Python modules for OS interaction

1. It would print the PID of the just invoked Python interpreter process to standard output, and then terminate. 2. With `os.makedirs(<dir>, exist_ok=True)`.

Package management

1. By running this installation command: `apt-get install vim`. In case the package is already installed, this command may nevertheless lead to an upgrade of the package. 2. Assuming these installations happened via Pip, an appropriate command might be: `pip list | wc -l`.

Container management

1. Not of the OS kernel, but – assuming basic compatibility between kernel and userland – certainly another version of all other components of the OS distribution, or even a different distribution. This could be accomplished and verified best with a versioned run command: `docker run --rm -ti ubuntu:20.10 cat /etc/lsb-release`. 2. This command first requires finding an appropriate image via `docker search pypy`. It should be noted that this search is not resilient and might need several attempts. The output shows that there are lots of choices, but there is also one image plainly called `pypy` hinting at its official nature, and furthermore conveying credibility by having lots of stars assigned to it from users. Therefore, `docker run --rm -ti pypy:latest` might be a good choice.

Data management and version control

1. According to the manual page `man git-merge`, the following six strategies exist as of version 2.39: ORT (Ostensibly Recursive's Twin), Recursive, Resolve, Octopus, Ours, Subtree. 2. First, the repository would have to be cloned: `git clone https://git.savannah.nongnu.org/git/attr.git`. Then, the tool would have to be built. It is evidently an Autotools-based project, hence: `cd attr; ./autogen.sh` followed by `./configure` and `make`. Next, modifications can be applied for instance in `tools/attr.c`, and tested by a rebuild. If everything works, then `git add tools/attr.c` and `git commit -m "<message>"` extends the changes into the local repository clone, with `sudo make install` also making the tool available on the system.

Data processing tools

1. First, a unified phone number format needs to be established, for instance with international prefix. Turned into a regular expression, it can then be grepped out of arbitrary text as follows for notations both with and without spaces in between numbers: `grep -rIP "\+\d{2}\s?\d{2}\s?\d{3}\s?\d{2}\s?\d{2}" .` with the final dot representing the top-most search path. 2. By combining a query of the original size and a conversion, for instance, for PNG

files: `convert -scale $((‘file <in.png> | awk ’{print $5}’‘/2)) <in.png> <out.png>`.

Structured data processing

1. The following pipeline gives a suitable JSON formatting for human consumption based on a provided endpoint URL: `curl -s ’http://.../<file>.json’ | json_pp`. 2. This can be accomplished by a conversion to JSON, as follows: `csvjson <file.csv> | jq ’.[] | select(.A | contains("PRODUCT"))’`, or more directly: `csvgrep -m PRODUCT -c A -d ";" <file.csv>`.

6. Middleware

Programmatic data serving

1. Web browsers enforce CORS. Web services not properly reporting CORS headers do not receive any HTTP requests from the browser, effectively rendering the dynamic functionality useless. 2. The minimal Python code is augmented to listen for HTTP requests and to deliver complete HTML content as responses. This augmentation needs code generation which the streamlit command contains.

File system abstractions and network storage

1. Either using loopmounting with the privileged command `mount -o loop <file> <mountpoint>` or, in case the file system is supported by Fuse2FS, then by calling `fuse2fs <file> <mountpoint>`. 2. SSHFS itself yes; but to automate the mounting, a passphrase-less key is typically used. This means the remote computer can also be reached with an interactive login with the same key, and hence full access to that computer (under the indicated user privileges) becomes possible. In other words, if `sshfs <user>@<server>:<dir> <mountpoint>` works, then `ssh <user>@<server>` also works.

Database interaction and management

1. First, the reason for the slowness needs to be identified. It is caused by many network requests and many small database transactions. A bulk/batch insert operation, if supported by the DBMS, massively speeds up the inserts. 2. Local user authentication refers to the operating system's authoritative information on user identities. Hence, a password authentication is not necessary. Over a network, no such authority exists, and a password must always be supplied.

Message brokers for real-time data processing

1. Polling refers to active checks for new messages with high frequency. It corresponds to a pull mechanism that consumes unnecessary system resources, whereas by using a broker, a process can sleep until it is woken up by the arrival of a new message, corresponding to a push notification. 2. The ZMQ protocol mirrors the underlying TCP protocol. In that, the receipt of each sent packet is acknowledged, and the acknowledgement must be read before another packet can be sent.

Parallel and distributed computing

1. By using spark-submit, the application only needs an empty SparkContext. All resource allocation parameters are set by this wrapper command. In contrast, applications executing without spark-submit must parameterise the SparkContext object regarding local or remote (Spark master) resources and other settings. 2. Not quite. As Spark follows a lazy evaluation approach, it merely records the series of instructions to be run whenever strictly needed. In that case, outputting the dataframe (df.show()) would require the results and run all instructions.

Model serving

1. For the access part, indeed a web server could be used, but it would need logic to upload models, revert to older versions and so forth. The model-serving implementations contain all that. For high scalability, a caching proxy web server could be used in front of a model server. 2. In principle, this is correct. However, the HTTP standard only foresees limited key-value parameters for GET requests, whereas entire request bodies can be supplied with POST requests. Only that way can user-defined data structures required for the inference/prediction process be guaranteed to be expressable in a request.

Data integration

1. Both extractors and taps refer to data sources or read access. Both loaders and targets refer to data sinks or write access. 2. Change management. All changes to the configuration can be verified and traced, and upon misconfiguration, a previous configuration can be easily reinstated.

Workflows and distributed scheduling

1. No. Due to the dependency between both operators, they can only be executed sequentially: `t1 >> t2`. 2. It is a cron expression, meaning that a

job should be executed at midnight (both hour and minute being 0), on all days (day of month, month, and day of week all being irrelevant).

7. Collaboration and Governance Platforms

1. The top-most process would be Jupyter itself. Its child process is the iPython interpreter. And the `ls` process is in turn a child of that one. Each child process is controlled by its parent. 2. There are two commands: first, `podman login <gitlabserver>:5050` as auxiliary step and, second, `podman pull <gitlabserver>:5050/<u>/<project>/<image>`.

8. Execution and Orchestration Platforms

1. No. Floating IPs are assigned by OpenStack to a network proxy that runs outside the VM. Within the VM, only the internal IP address is visible. Software such as web servers needs to be consciously configured to react to public IP addresses or public (fully-qualified) hostnames. 2. No. Only reactive (after-the-fact) autoscaling is supported by the container orchestrator. Applications or helper containers can use their intrinsic knowledge, for instance, upcoming events, to emit proactive autoscaling decisions.

9. Global Infrastructure

1. The stations can be counted with the instruction: `curl 'http://transport.opendata.ch/v1/locations?x=47.00&y=9.00' | jq | grep name | wc -l`. Accordingly there are ten stations nearby. 2. This is a slightly tricky question due to incomplete Zenodo search syntax documentation. With some attempts, the answer can be constructed as follows: `curl "https://zenodo.org/api/records/?q=rattlesnake&file_type=r" | jq ".hits.hits | length"`. Accordingly, there are six such datasets published at the time of writing.

Data Science

Operating Systems and Infrastructure in Data Science

Modern data scientists work with a number of tools and operating system facilities in addition to online platforms. Mastering these in combination to manage their data and to deploy software, models and data as ready-to-use online services as well as to perform data science and analysis tasks is in the focus of Operating Systems and Infrastructure in Data Science.
Readers will come to understand the fundamental concepts of operating systems and to explore plenty of tools in hands-on tasks and thus gradually develop the skills necessary to compose them for programming in the large, an essential capability in their later career.
The book guides students through semester studies, acts as reference knowledge base and aids in acquiring the necessary knowledge, skills and competences especially in self-study settings.
A unique feature of the printed book is the associated access to Edushell, a live environment to practice operating systems and infrastructure tasks.

Print Version:
ISBN 978-3-7281-4167-5

eBook:
ISBN 978-3-7281-4168-2 / DOI 10.3218/4168-2

v/dlf