Basics of Computing Environments for Scientists

ISG D-PHYS

Jan 26, 2024

Welcome to the "lecture notes" of the **Basics of Computing Environments for Scientists** modules at the Physics department of ETH Zurich. Please refer to the web site https://compenv.phys.ethz.ch for the latest version.

CONTENTS

CHAPTER

ONE

IT AT D-PHYS

This ~1 h introduction is targeted at new members of D-PHYS and IGP/D-BAUG and is meant to familiarize you with the following three topics:

- Introduce the IT Services Group (ISG) as the primary go-to place for all IT-related questions
- Give a general idea of "how IT works at D-PHYS", what there is and whom to ask for help
- Introduce **Basics of computing environments for scientists** lecture modules for advanced topics

1.1 IT Services Group (ISG)

D-PHYS, like several other departments of ETH Zurich, employs a group of IT professionals that support the researchers and departemental staff in all IT-related matters. We are currently 8 members (usually plus one apprentice), specializing in Linux, Windows and macOS on both the server and workstation side. We run all servers of the department and offer a long list of IT services.

We love solving complicated problems, so please don't hesitate to get in touch if you're got any question regarding computing. The easiest way to contact our helpdesk is via Matrix chat or our ticketing system. We provide a large documentation repository describing all aspects of our work.

1.2 How IT works at D-PHYS

1.2.1 The players

Aside from ISG, there are at least two more service providers in D-PHYS' IT landscape: local IT representatives in the research groups and Informatikdienste (ID).

Since the various research groups at D-PHYS and IGP have vastly diverging computing requirements and work them out differently, many of them have some sort of local IT representative (usually a PhD student) who serves as the first point of contact in IT matters and also acts as a liaison to us (ISG). While we certainly don't discourage each and every department member to contact us in case of questions, it's often a very good idea to consult your IT rep first just to keep things consistent in the group.

On the other side of the spectrum, there's the central Informatikdienste (ID) of ETH Zurich. Traditionally D-PHYS IT has been quite independent from ID, so for most members of the department, the list of ID services is rather short:

- Basic networking (wired and wifi, VPN). Higher protocol levels are handled by us.
- Email - while every member of D-PHYS has an `@ethz.ch` email address, most people forward it to their `@phys.ethz.ch` account and use that one. IGP/D-BAUG uses `@geod.baug.ethz.ch` addresses on Exchange (ID).
- IT Shop, the ETH-wide software/license portal.
- Printing, ISG serves as the liaison to ID's pia printing system.

Everything else (and there's a lot..) is handled by ISG.

1.2.2 Accounts

One of the side effects of having multiple IT service providers: multiple accounts. Both ETH and D-PHYS issue an account to access their services. The first one you'll encounter when starting your ETH career will ususally be the ETH account, so we try to use the same login (passwords can be different of course). Furthermore, the ETH account comes with two different passwords (one for network-related services ("Radius password") and one for the rest ("LDAP password")). For full details please refer to our documentation.

1.2.3 Main IT services provided by ISG

While it is virtually impossible to list every service we offer, we do try to cover the highlights. Here's an excerpt:

Managed Workstations

We offer fully managed workstations for all three major operating systems: Windows, Linux and macOS. We take care of both software installation and updates, so you can concentrate on your work. This makes them a good choice for many office computers (lab not so much). Self-managed computers are always possible (and often needed, for example as measurement PCs), but we can't support them on the same level as managed workstations due to their sheer number (~ 1000 in D-PHYS).

Storage and backup

One of our most important services is storage and backup. For group shares and personal home directories, we run an enterprise-grade SAN (currently 3.6 PB) plus the equivalent amount for backups. Your home directory and all group shares you have access to are automatically mounted on all Managed Workstations and can be accessed via SMB from non-managed machines. A backup of all data is taken each night and kept for at least 30 days. You even have direct access to all your backup data. We also provide a backup solution for non-managed laptops and lab PCs.

Please note that all your research data should go into a group share, not your personal home directory. Your colleagues will be thankful.

Email

We run a mail server with full spam and virus filtering and all relevant protocols and a powerful webmail interface. In addition, we have a groupware solution for collaboration that synchronizes with most calendars and mobile devices.
`@phys.ethz.ch` and `@ethz.ch` email accounts can be used independently or be forwarded in either direction.
`@phys.ethz.ch` accounts come in various flavors and can be migrated to collaborator status after leaving ETH.

Network

To ensure a stable and secure network environment at D-PHYS, all computers connecting to the wired network have to be initially registered with their MAC address.
Required information:

- MAC address (obviously…)
- two D-PHYS accounts that serve as primary contact persons
- type of machine (desktop, laptop, oscilloscope…)
- operating system (Linux, Windows, macOS…)

ETH wifi is managed by ID and requires an ETH Radius login.

Helpdesk and support

We operate a helpdesk that rotates within ISG on a weekly basis. You can reach us by

- Phone: 3 26 68 (at least in non-corona times)
- Chat: element.phys.ethz.ch
- Email: isg@phys.ethz.ch, feeds into our ticketing system
- Physical presence: HPT H 6-9
- Business hours: usually 07:00-17:00.

Our ticket system allows us to dispatch incoming requests and keep track of follow-ups.
Tons of documentation and many tutorials can be found in our readme.
Our website features a blog that contains service announcements. In order to stay informed, subscribing to our newsletter (very low traffic) is highly recommended.

Printing

ETH offers the campus-wide PIA printing system. We provide documentation.

Web services

We host and maintain a wide variety of web services for the department, e.g.

- D-PHYS shop
- Vademecum - helpful for new D-PHYS members
- Lecture experiments
- Personal website for all users: https://people.phys.ethz.ch/~LOGIN/
- CMS like Wordpress or Dokuwiki, e.g. for conferences
- Database hosting, for example InfluxDB

Computing resources

We operate a number of public Linux login hosts and a Windows Terminal Server for everyone to use. While it doesn't make sense to run HPC clusters on the level of D-PHYS, we collaborate with our colleagues of ID SIS for cluster usage. Many research groups have access to those resources. We also maintain a number of number crunchers for interactive workflows in different groups.

Data protection and privacy

As mentioned earlier, all your research data should be stored on our group shares - for at least three reasons:

- it's your most valuable research data!
- ETH regulations require a copy to be stored on ETH servers
- we take care of regular backups

You might be tempted to use external cloud services because they're "so convenient", but please always keep in mind that these companies then have full access to your data! That's why we try to offer local equivalents whenever possible: Matrix (with LaTeX support!) instead of Slack, Polybox instead of Dropbox, our mail server instead of Gmail…

Also take a moment to establish a good password management strategy.

Make sure to explore the other *exciting* topics on our IT security website.

Consulting and special projects

Bleeding edge research often requires innovative and creative IT solutions. Please come talk to us whenever you're facing a non-standard problem. We love those! Some examples of recent projects for/together with research groups:

- InfluxDB + Grafana for lab environment monitoring
- Raspi-based measurement setup on top of Prime Tower
- Remote management for D-PHYS' liquid helium system

1.3 Some pieces of advice from us. Pretty please with sugar on top.

- consult our readme, most of your questions have already been answered.
- when in doubt, come visit and discuss with us!
- when creating a ticket, please provide the necessary information (contact details, name/MAC address of your machine, operating system + version, detailed description of the problem…).
- do not create millions of small files on the file server. This will waste a lot of space and kill performance. Create tgz or zip archives instead.
- do no run computational jobs on network storage! Every machine has a local *scratch* folder to hold temporary data while the simulations are running. End results go to the group share.

1.4 Where to go from here?

This introduction should have given you a first glance of the IT service landscape at D-PHYS. If you'd like to learn about more advanced topics, we currently offer the following modules in our "Basics of computing environments for scientists" lecture series:

- Linux Basics *I* + *II*
- Python Ecosystem *I*, *II* + *III*
- *System Aspects*
- *IT and Information Security*

Our colleagues of ID Scientific IT Services also offer a range of courses and workshops (lecture notes here) that might be of interest.

CHAPTER

TWO

IT AND INFORMATION SECURITY

This is a ~1 h crash course that will hopefully prepare you to react to most common threats “on the internet” and make sure your data is safe. Since there is already a wonderful website with all the information covered in this module (and more!), we will not duplicate any contents here.

CHAPTER

THREE

LINUX BASICS (I)

The first steps in a Linux ecosystem may be quite challenging. Albeit there are a lot of material available most of it assumes already some familiarity with concepts and terms or some background knowledge. This lecture provides a gentle introductions and covers basic but important topics. It is by no means complete, but it should help to ask the right questions and understanding the answers.

3.1 Index

3.1.1 Introduction to Linux

Linux is much more than *computer operating* system. It is an example from a whole family of *UNIXoide* computer systems and ubiquitous in world of open source software. This section give a very brief overview to understand the role of operating systems, kernels, Linux distributions and shells.

Index

Operating Systems

It is hard to precisely define what exactly an operating system is. Here it suffices to to define an *operating system* as a software which manages the (hardware) resources of a computer system and is capable of running applications and programs for users.

Note: Strictly speaking this definition just includes the *operating system kernel*, which is a software encapsulating the hardware and resources of a computer to provide an interface for applications. But usually software vital to operate a computer is also counted as part of an operating system.

Popular examples of operating systems are *Android*, *Windows*, *macOS* and *Linux*. However there are many many more operating systems. An historical eminent example is the *UNIX* operating system. Dating back to 1969 is a common ancestor of many others. Various editions, versions and implementations gave rise to the computing landscape of today. An important release is the *UNIX System V* (five) release from 1979 which motivated first open source re-implementations and the *GNU project*. This project started to implement all system software under a free license. Later on in 1991 Linus Torvalds created the open source *Linux kernel*.

In this sense the Linux is an open source clone of UNIX, or an open source implementation of the UNIX standard. Sometimes Linux it is referred to as GNU/Linux.

The Linux kernel

The Linux kernel is now maintained by the *Linux Kernel Organization* and is the largest software development project in the world. It is mainly written in the C programming language and over thousand lines of code change every day. The development process is organized via mailing list and heavy use of the git version control system which was developed by Linus Torvalds for exactly that purpose. A hierarchy of kernel maintainers organize the code and decide which changes to the codebase are applied and published. The current kernel and everything about its development can be found on kernel.org.

The Linux kernel has version numbering scheme. For example 5.14.6 is the 6th stable release in the *5.14 kernel*. Every few months a new version, i.e. 5.15 is released resulting in subsequent stable releases 5.15.1, 5.15.2 and so on. Some kernels are chosen to be on *long term support* (LTS) kernels which are guaranteed to receive important updates for usually two years. It is completely legitimate to run an LTS kernel, for example 4.9.282. All in all many different versions of the Linux kernel are considered current and a wide variety is deployed in running systems.

Note: It is even more complicated as many distribution do not run the *plain vanilla kernels* from kernel.org, but apply a set of custom extensions and patches. The distributions have to maintain these patches and reapply them for every new stable release. A distribution release number might look like 5.13.13-200.fc34.x86_64 or 5.4.0-80-generic. The running kernel release version can be displayed by uname -r.

Multi-user systems

Following UNIX ideals, Linux is designed to be a multi-user system. Everything is set up in way that multiple users can use the system, even at the same time. Obviously user accounting and permissions are an important topic. Nevertheless Linux has a very simple way of organizing its users.

Users

First of all *users* are identified by unique numerical ID, also called uid and they have also (unique) *username*, say john. On every system there exist a user called root with uid=0. This user account has special privileges and is used by system administrators.

There is some more data associated to each user, for example the *login shell*, the *home directory* or a *primary group*, see below.

Groups

Moreover user can be member of *groups*. A group is also identified by a unique numerical ID (gid) and has also a name. Traditionally, for every user there exist a dedicated group (with the same name), with that particular user as the only member. So for example there is a group called john with only john as user. This is often used as the user's primary group.

Groups are just lists of users (or uids) and every user can be member of multiple groups. However groups cannot be nested.

Note: All (local) users can be viewed by inspecting the file /etc/passwd and all (local) groups are defined in /etc/groups. Information about a particular group or user is best retrieved by running gentenv passwd username or id username and getenv group groupname.

Open source software

The idea of free and open source software is inherently connected with Linux and there exist a lot of open source software written to be run on Linux systems. As the name suggests the source code of such software is available. However to actually use such a software it has to be compiled and installed. Often a software is not self contained but depends on many other softwares such as libraries. So to actually use a particular software all its *dependencies* (usually in particular versions) have to be installed as well. So while in principle available, it might be very cumbersome to actually install such a software successfully. To some extend all these problems are solved by *Linux distributions*.

Linux distributions

A *Linux distribution* is a set of software which makes a complete usable system. It is a well chosen and maintained choice of the many alternatives in the open source landscape. They all start to build upon a Linux kernel and the community or company behind a distribution maintains many applications which are chosen, installed and configured in a compatible way. Usually the distributions have a *packaging system* to install further software. These packages are already customized to the distributions needs and are available for download from so called *repositories* (very much like an *app store*). This makes it easy for users to use and extend their systems. More over the distribution maintain updates and security fixes.

Famous examples are: *Ubuntu*, *Fedora*, *Debian*, *Suse* or *Arch*.

Command Line Interface (CLI)

While there are graphical environments to work with a Linux system in a similar way to work with a Windows system. It is very common to work “on the shell”, that is just interact with system via a textual interface by typing commands. While working on remote systems, this is often the only way to interact with the system at all. The name stems from idea of providing the outer layer of the core operating system or kernel.

The interface where commands are typed in is called a *terminal* or *terminal emulator*. These software processing these commands is called a *shell* or *command interpreter*. This is actually an application just waiting for input to process. Famous examples of shell programs are bash (*Bourne again shell*),ksh (Korn shell) or zsh (*z-shell*). The commands typed in are typically names of other programs which are then executed or part of the shells own programming language, like loops, conditional statements or the use of variables.

Example

The pwd command to show the current working directory, actually executes the program /usr/bin/pwd which prints the current working directory. Similar typing firefox will execute the popular web browser typically installed as /usr/bin/firefox.

Terminal emulators

There is a whole list of terminal emulators. Their basic functionality is similar and the choice highly opinionated. Good terminals to start with are Terminator or GNOME Terminal. The latter is the pre-installed default terminal in the popular GNOME desktop and can simply be selected from the application menu.

Shell

The terminal spawns the users default shell, (e.g. bash or to precise /bin/bash) and a command prompt is shown and waits for further commands. The command prompt itself already provides some information. In the example below john@laptop means, this is a session of user john on the computer named laptop. This may seem superfluous, but makes sense while working with (multiple) remote machines. The tilde (~) is a shortcut and shows the current working directory is user john's home directory. The final $ indicates that this is a normal shell, opposed to a # indicating a privilege shell of the root user.

Commands can be typed in and are run by hitting return. Executing the pwd command produces a result like below:

```
[john@laptop ~]$ pwd
/home/john
```

To exit a shell the Terminal window can be closed or (on remote systems) the exit or logout command can be run. Alternatively sending *end of transmission* character which is typed as CTRL+d exits the shell as well.

Note: Which shell is started is defined in the Linux user database on the system. getent passwd username shows colon-separated fields with basic information: Besides the username, the numerical ids for the user (uid) and his primary group (gid), also the path of the user's home directory (here: /home/johndoe) and the user's login shell (here /bin/bash).

```
getent passwd johndoe
john:*:12366:12366:John Doe:/home/john:/bin/bash
```

3.1.2 File systems

A *file system* defines how data is actually stored and retrieved on storage media. It usually consists of multiple layers. Towards the operating system the *logical file system* provides functions for file operations like open, read and close. Behind the scenes the *file system implementation* actually implements these functions and defines how data is actually organized on storage media.

Usually these file system implementations are shortly referred as *file systems*. There are many file systems available for Linux. Prominent examples are ext4, btrfs, FAT, NTFS (actually used in Windows) or xfs. There are less intuitive file systems like *pseudo file systems* which are not backed by any storage media but create the data upon request or *network file systems* which access their data over a network.

The Linux *virtual file system (vfs)* abstracts all these implementations, such that they appear all (more or less) equal to the operating system and the user.

Index

File names

Starting from a Windows Systems there are some important differences regarding file names in Linux file systems. In Linux files paths are written with / as separators (opposed to Windows using back slashes \) and Linux files systems are usually case sensitive, i.e. file.dat and File.dat are different files.

On most Linux file systems the name of a file can be up to 255 characters long and may contain any characters except slashes / and NULL characters (\0).

Portable filename character set

It is advisable to use only filenames with characters from the *portable filename character set* [-._a-zA-Z0-9]. That is letters a-z and A-Z, numbers 0-9 and the characters -, . and _. Other characters as whitespaces, parenthesis or * have a special meaning and it is difficult to handle such filenames correctly. For example rm My Notes will delete the files My and Notes, instead of deleting the file “My Notes”.

Absolute and relative paths

There is a distinction between relative paths and absolute paths. An *absolute path* describes an absolute position in the file system, i.e. with respect to the file system root / it always starts with a slash:

```
/home/alice/Documents/notes
```

A *relative path* describes a file with respect to the *current working directory*. It is always written without a leading slash:

```
Documents/notes
```

Assuming the current working directory is /home/alice then the two paths from above refer to the same file.

More absolute paths

Sometimes paths are written with a leading dot such as in ./foo/bar. This is also an absolute path, as . is actually a name for the current working directory. So for example if the current working directory is /home/john, then ./foo/bar is equivalent to /home/john/foo/bar.

Another subtlety are expressions like ~/foo/bar. In this case ~ is a feature of the bash shell. Before the execution of a command bash substitutes ~ with the current users home directory. So ~/foo/bar will be replaced by the absolute path /home/john/foo/bar.

Note: There is a important distinction. While . is actually a file in the file system, ~ is not. It is just a handy short cut provided by bash.

Hidden files

By convention, files staring with a dot character (.) are *hidden files*. They are not included in normal file listings and are not shown graphical file explorers. A typical example would .config.

Each directory contains two files . and ... They are actually directories or merely alternative names: . is a name for the current directory
and .. is a name for the parent directory. For example foo,foo/., foo/bar/.. are all names for the same directory foo. Also see the *hard links*. In particular . and .. are hidden files, as their name begin with .!

Hard links

A *hard link* is an other word for file name. For most Linux file systems it completely fine to have files with multiple names. The hard link counter keeps track of how many file names refer to a particular file. It it shown by stat file or ls -l file. Usually files have a hard link count of 1, i.e. the file has exactly one name. However directories usually a higher count for example dirA, dirA/. and dirA/dirB/.. are all different names for the same file which makes the hard link count at least 3.

Note: A hard link is very different from a *soft link* or *symbolic link*. The latter is an actual file of special type (link) which has another filename as its content. While accessing a *symbolic link* its content is read and then used as target for the file access. The procedure is called *following* or *dereferencing* the link.

Directory Entries and Index Nodes

A file name does not directly correspond to the data of the file, instead it refers to an object called *dentry* (directory entry) in the containing directory. Each a dentry points to an *inode* (index node) which then refers to the actual data. Multiple dentries may refer to the same inode. By this mechanism a file may be known by several names. All these dentries are equally ranking. There is no such thing as primary dentry. If a file is deleted its dentry is removed. The hard link counter in the corresponding inode is decreased. If the counter is still non-zero, there are other dentries referring to it and the inode stys intact. If the counter is zero, the inode and eventually the data is released.

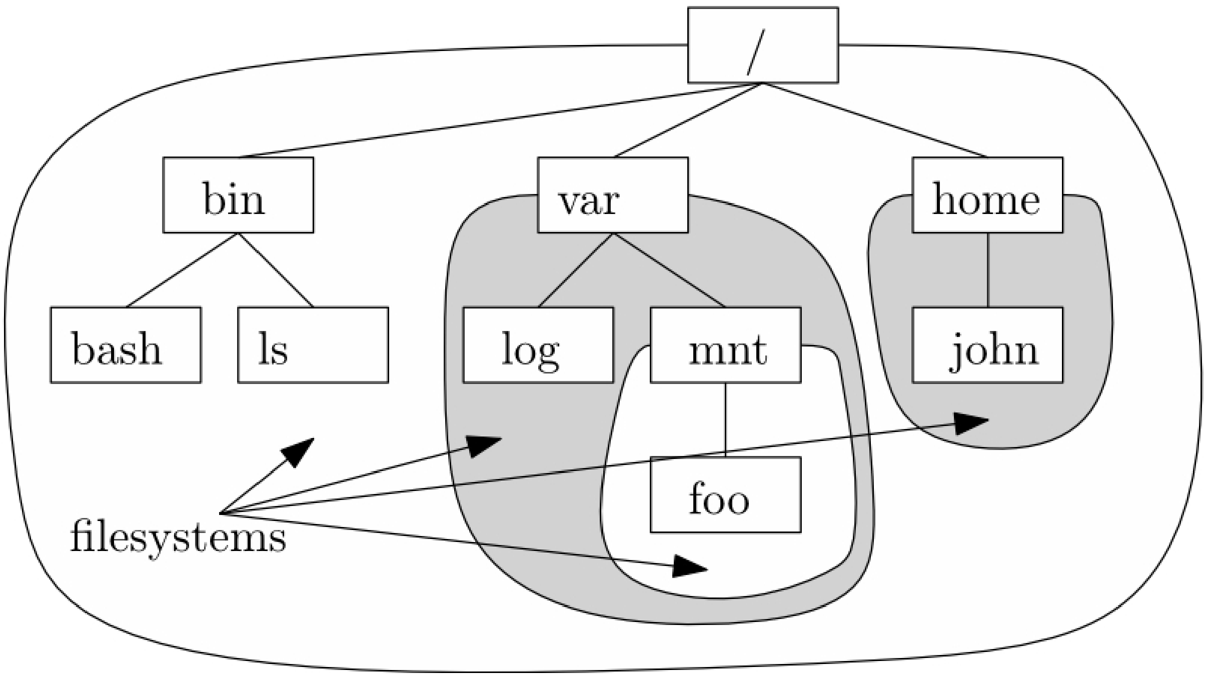
```
dentries         inodes                 data

+-------+       +----------------+
| fileA | ---> | metadata:       |      +-----------+
+-------+       |  hard links:2  | ---> | file data |
+-------+       |  permissions   |      +-----------+
| fileB | ---> |  ctime, ...     |
+-------+       +----------------+
```

Permissions, owner and other metadata is stored in the index nodes. That is why changing permissions or metadata is seen by using any file name.

Single directory tree

In Linux the totality of files is organized in one big directory tree starting with / which is called the *file system root*. However usually the directory tree contains multiple different file systems as subtrees. These file systems are said to be *mounted* inside the root file system. The root of these sub file systems is called the *mount point*.



In the example are 4 different file systems, the mount points are:

- /
- /var
- /var/mnt
- /home

Mounted file systems

It is important to be aware of the file systems. For example, some path can be part of a mounted network file system. This makes frequent accesses very inefficient. It may even be a volatile filesystem which may be lost on reboots. The fndmnt or df command can be used to find the containing file system of a file.

On a typical Linux systems there are many mounted file systems. A complete list is always stored in the file /proc/mnt or can be showed by running mount in a terminal.

Example

On a manged D-PHYS Linux machine the user home directories are not local on the individual machines but on a central server. This way users can access there home directories from any machine, but also this make the access much slower as all the data has to be transferred over a network. For example fndmnt Documents/notes.txt shows this is a network file system (nfsv4) actually located in /home/J/john on server phd-home.

```
findmnt --target Documents/notes.txt
TARGET     SOURCE                   FSTYPE OPTIONS
/home/john phd-home:/home1/J/john nfs4   rw,relatime,sync,vers=4.2,[...]
```

3.1.3 File Permissions

Linux file systems allow a simple permission scheme for each file. There are only three coarse *types of file access* defined and each can be granted or denied for three types of (user) *classes*. Moreover there are three special permissions to be granted, but these are only relevant for file execution. In total each file has 3×3 matrix (9 bits) of basic permissions.

However there are ways to extend this very basic permission scheme using *access control lists*.

Index

Basic file permissions

Access types

The three basic *file access types* are *read*, *write* and *execute* and for each file each permission can either be granted or not.

File access

For *file access* theses permission have the following interpretation:

Permission	Symbol	Meaning
read	r	ability to read the contents of a file
write	w	ability to modify the contents of file
execute	x	ability to run an file/program

While *read* and *write* permissions should be obvious, the permission to *execute* a file is less intuitive. It is not enough for a file to be an executable program (opposed to an image file for example), Linux does not allow execution without the correct permission. This has mainly historical reasons, but is still useful from time to time.

Note: If the file is not an ELF file, but for example a script, also read permissions are necessary to run the file. A common pitfall is to download or create a script and running it without granting execute permission first:

```
[john@laptop ~]$ ./script.sh
/bin/bash: ./script.sh: Permission denied`
```

Granting the correct permissions solve that problem: `chmod u+rx script.sh`

Directory access

For *directories* the permissions are interpreted slightly differently:

Permission	Symbol	Meaning
read	r	ability to list the names of the files inside the directory
write	w	ability to modify directory contents, i.e. create, delete or rename files inside
execute	x	ability to "search" inside the directory, i.e. access file contents, metadata

Note: Note there is no permission to *delete* a file. To delete the content of file (i.e. truncate it to 0 bytes) *write* permission to that file is necessary. The delete (or *unlink*) the file entirely, write permissions on the containing directory are needed. This makes sense after reading the section about *dentries* and *inodes*.

Access classes

There are three *access classes* which define for whom the permissions r, w, x apply. These are *user*, *group* and *other*. Each file is associated to exactly one *user* and one *group*. It also said the file is owned by the user and the group. The read, write, execution permissions apply for the associated user, the associated group and for the rest (*other*):

Access class	Symbol	Meaning
user	u	the single user that is owner of the file
group	g	all users that belong to the group associated with the file
other	o	all users that are neither owner nor member of the file's group

The r, w, x permissions for u, g and o entities respectively make a total 9 permissions to be set or denied.

Note: A common pitfall is to misinterpret "o" as abbreviation for owner instead of other. Then granting permission to the "o"-class may be disastrous.

This permission scheme is very restrictive. For example, two users with the same groups and none of them is the file owner share the same access rights (those of the class *other*). Neither is it possible to separate the permissions of the users within the associated group of a file. They all share the permissions of the group class. To work around these limitations *access control list* are necessary.

Special permissions

There are three special permission which rather modify a file's execution behavior than granting a permission. They are called *set-uid*, *set-gid*, and *sticky mode*. Again the meaning is different for files and directories:

Special permissions on files

Each program has also an associated user and an associated group. Normally these are inherited from the calling user. The *set-uid* and *set-gid* change this behavior.

Permission	Symbol	Meaning
set-uid	s	run the file as the file's owner
set-gid	s	run the file as the file's associated group
sticky	t	no effect

A typical use case is a program owned by root with enabled *set-uid*, for example /usr/bin/passwd or /usr/bin/su. Any user can run such programs, but they will be executed with root privileges. This allows users certain predefined tasks like changing passwords or impersonating other users which would otherwise require system administrator privileges.

Special permissions on directories

Permission	Symbol	Meaning
set-uid	s	no effect
set-gid	s	newly created sub-directories inherit the associated group from their parent directory
sticky	t	directory w-permission only applies to an user's own files

The *sticky mode* is common for shared /tmp or /scratch directories. Many users have write permissions but cannot delete or modify one anthers files.

Representation of file permissions

There are two notation schemes for file permission: *symbolical* or *numerical*. Different tools use either one, so it is necessary to understand both.

Symbolical representation

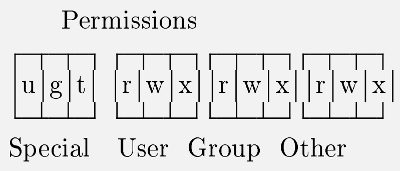
The symbolic representation lists the given access permissions in the order read, write, execute for each user class in the order user, group, other. A granted permission is indicated by its symbol a denied permission by a -. The special permissions are mixed within the symbolical representation by replacing the x-position in each access class by s or S (for set-uid and set-gid) and t or T (for sticky mode) .

This will be clear after a few examples:

permissions	numerical	user	group	other	special
rwxrw----	0750	read, write, execute	read, write,	none	none
r--r--r--	0444	read	read	read	none
rwxrwxrwx	0777	read, write, execute	read, write, execute	read, write, execute	none
r-x---r-x	0505	read, execute	none	read, execute	none
rwsr-s---	3750	read, write, execute	read, execute	none	set-uid, set-gid
r-Sr-sr-T	7455	read	read, execute	read	set-uid, sticky

Numerical representation

To obtain the numerical representation, the permissions are written as a bit field of length 12, divided in sections of three bits:



A group of three bits may represent $2^3 = 8$ different values. From the binary representation follows that r, w, x can be mapped to the numeric values $2^2 = 4$, $2^1 = 2$ and $2^0 = 1$. All possibilities are shown in the table below. The same holds true for the special permissions.

permissions	binary	octal
---	000	0
--x	001	1
-w-	010	2
-wx	011	3
r--	100	4
r-x	101	5
rw-	110	6
rwx	111	7

To obtain the numerical representation the octal values of the special, user, group and other section are concatenated.

Note: If no special permissions are used, the leading 0 in the numerical representation can be omitted. So for example 0755 and 755 both represent rwxrw-rw- with no special permissions. The other sections cannot be omitted.

3.1.4 Secure Shell (SSH)

SSH, short for *secure shell*, is network protocol using strong cryptography to protect traffic in untrusted networks. Its original use is to provide access to a shell on a remote system in secure manner. But many more applications are possible, like transmitting files or more generally providing a *secure channel* for any otherwise unprotected network traffic.

With respect to SSH there are two instances the *SSH client* and the *SSH server*. The server program constantly running on a remote machine, possibly a D-PHYS server, waiting for incoming connections. The client is program executed in an ad-hoc fashion to initiate and use an SSH secure channel.

Index

Encryption algorithms makes it easy to protect communication as long as the sending and receiving party share a secret. Clever Mathematics solve the problem of exchanging such secrets over untrusted channels. More worrisome is the problem of verifying the identities of the communicating parties.

First time connections

If the SSH client connects the very first time to a server, it cannot be sure about the identity of the remote server. It presents a *cryptographic fingerprint* of the server and asks whether to establish a connection. You should verify the fingerprint with an trusted list. After accepting this server, the client will recognize the server and don't ask again for a verification.

```
john@laptop:~$ ssh johndoe@login.phys.ethz.ch
The authenticity of host 'login.phys.ethz.ch (129.132.89.195)' can't be established.
ECDSA key fingerprint is SHA256:upncE1in1QVEyXEeafC/WOPpK8QtZ/skpxU7GwTlpUk.
Are you sure you want to continue connecting (yes/no/[fingerprint])?
```

SSH login with password

With any further configuration SSH falls back to authentication via passwords. So the your client will ask your for the password of the account on the remote side (johndoe on login.phys.ethz.ch).

```
john@laptop:~$ ssh johndoe@login.phys.ethz.ch
johndoe@login.phys.ethz.ch's password: *********
johndoe@phd-login1:~$
```

Public Key Cryptography

Public key cryptography or asymmetric cryptography use keys which have a public and a private part. The *private key* part, or *private key* for short has to kept secret. The *public key* can and has to distributed to other parties. The real protocols are more complicated but it here it suffices to think that public keys are used to encrypt, and private keys are used to decrypt messages. So if Alice wants to send an encrypted message to Bob, she uses Bob's public key to encrypt it. Afterwards Bob can read the message using his private key. To send an answer, he has to use Alice's public Key.

Using asymmetric cryptography is has many advantages over passwords. First of all no secrets have to be transmitted over untrusted channels and also brute force attacks on encrypted traffic are much harder.

Generating SSH keys

Depending on the underlying math and protocols there are different versions of keys. For SSH the most common choices are *RSA* and *ed25519* keys. RSA keys use the famous RSA algorithm based on factoring primes. Nowadays the key length (length of the modulus) should be at least 4096 bits. ed25519 keys are based on calculating discrete logarithms.

Use `ssh-keygen -t rsa -b4096` to generate a 4096 bit RSA key or `ssh-keygen -t ed25519` to create an ed25519 key. The key pair will be saved in the directory in a directory called `.ssh`. I consists of two files:

- `~/.ssh/id_rsa` is the private key. It has to be kept secret and is ideally protected with a strong password.
- `~/.ssh/id_rsa.pub` is public key. This file may be distributed to other parties.

In case of ed25519 the files will be named `id_ed25519` and `id_ed25519.pub`. The name of these files can be chosen on creation or they can be renamed later. Each key pair has also a *fingerprint* and a *random art image* to make it easier to identify and recognize the keys.

```
$ ssh-keygen -t rsa -b4096
Generating public/private rsa key pair.
Enter file in which to save the key (/home/john/.ssh/id_rsa):
Enter passphrase (empty for no passphrase): ************
Enter same passphrase again: ************
Your identification has been saved in id_rsa
Your public key has been saved in id_rsa.pub
The key fingerprint is:
SHA256:KsXhkRCJFq17V5hwgiQx+Wt/gbv4OgxmbovNim1fTEM john@johnsPC
The key's randomart image is:
+---[RSA 4096]----+
|++o=oo           |
|.o+ =...         |
| o . +Eo         |
|  o  +oo.        |
|   o .*.S        |
|.o+ o+oo         |
|++ o.o+.         |
|o*+.oo.          |
|=+B=+o           |
+----[SHA256]-----+
```

Authorized keys

To use SSH keys to establish a connection the server must be aware of which keys are trustworthy. This is done by registering the public keys as *authorized*.

On the remote machine as the remote user create or modify the file `~/.ssh/authorized_key` and add the public key as line in this file. The SSH server process reads this file and accepts all logins using these authorized keys. For obvious reasons the strict permissions to the `authorized_key` file are necessary, otherwise the SSH daemon will refuse to accept the keys.

```
# create the .ssh directory if necessary
mkdir -p ~/.ssh

# Add the line "ssh-ed5519 ... john@laptop" to ~/.ssh/authorized_keys it will
# create the file if it does not exist. Make sure to replace
# "ssh-ed5519 ... john@laptop" with the content of `~/.ssh/id_ed25519.pub` on the source machine

echo "ssh-ed25519 AAAAC3NzaC1lZDI1NTE5AAAAIG5Csv7paLFNcTTIry5jMX/
```

(continues on next page)

(continued from previous page)

```
↪4JK20mD3sUEUNm2I6pt1w john@laptop" >> ~/.ssh/authorized_keys

# This sets the permissions on the .ssh directory and the authorized_keys properly
chmod 700 ~/.ssh
chmod 600 ~/.ssh/authorized_keys
```

Graphical programs

```
john@laptop:~$ ssh -X johndoe@login.phys.ethz.ch
johndoe@phd-login1:~$ mathematica &
```

3.1.5 Regular expressions

Regular expressions originate from the theory of formal languages in mathematics or computer science. They are closely related to *finite state machines* and are a way to define a set of words which match certain criteria. The most common application of a regular expression, or *regex* for short, is parsing of textual data and recognizing or finding textual patterns.

The regular expression itself is given in one of several competing *regular expression languages* or *regular expression flavours*. The crucial point is to understand the syntax and grammar of the regular expression language. Unfortunately not all of these languages are compatible. A de facto standard is defined by *POSIX extended regular expressions (ERE)*. Further, many programming languages define their own regex flavour. A very wide spread example is the *perl (compatible) regular expression language (PCRE)*. The following, however, sticks to POSIX.

Index

POSIX regular expressions

It is easiest to think about regular expressions that they specify certain textual patterns. For example the expression: `t[io]m_[0-9]`

This is pattern standing for all words starting with `t`, followed either by an `i` or an `o`, then afterwards an `m`, an underscore `_` and a single digit number:

- tim_0, tom_1, tom_0, tim_9, …

Often regular expressions are applied to a *search space* which is most often a line of text but in principle could be any string. The question is, whether the specified pattern is contained in that string, or if the string *matches the regular expression*. In this sense `atom_23` and `Hi I' am tim_3!` both match, while for example `int time_0;` does not.

The POSIX regular expression language is built of blocks actually matching stuff and quantifiers changing the matching behavior of such blocks.

Characters

Most characters just match themselves and nothing else. But of course there are some exceptions, those are called *meta characters*.

Expression	Meaning
a	matches a, this hold true for almost all characters
^	matches the beginning of the search space
$	matches the end of the search space
.	matches any character
\\.	matches an actual ., see *escaping*
( ) [ ] { }	meta-characters used for *groups*, *classes* and *quantifiers*, see below
* + ?	meta-characters used for *quantifiers*, see below

Note: The *POSIX basic regular expression* flavour uses all parenthesis as normal characters matching themselves. The meta-character parenthesis have to escaped with a backslash, i.e. \{ instead of }. Some tools ere not very strict about this distinction an try to guess if a regex is basic or extended.

Character classes

Expression	Character set
[:digit:]	a single digit 0 to 9
[:lower:]	characters a to z
[:upper:]	characters A to Z
[:alpha:]	characters from [:lower:] and [:upper:]
[:alnum:]	characters from [:alpha:] and [:digit:]
[:space:]	white spaces
[:punct:]	punctuation characters

There are more classes defined and actually their exact definition depends on the systems *locale*.

Quantifiers

Quantifiers allow repetition of previous blocks, either single characters, character classes or groups (see below).

Expres-sion	Equiv-alent	Meaning
{m,n}		The preceding expression is repeated at least n times and at most m times. n can be left out, (i.e. {m,}) which means $n = \infty$
{n}	{n,n}	The preceding expression matches exactly n times
*	{0,}	The preceding expression matches arbitrarily often, maybe not at all
+	{1,}	The preceding expression matches arbitrarily often, but at least one time
?	{0,1}	The preceding expression matches at most one time

Greedy and reluctant matches

Of the variable quantifiers above {m,n}, + and * are *greedy quantifiers*. They try to consume as many characters as possible. For example the regular expression ".*" matches the whole sentence "Run", she said, "I know regular expressions" including the , she said,. To avoid this behavior the regular expression ".*?" has to used. This results in two matches ("Run" and "I know regular expressions"). This make the quantifiers *lazy* or *reluctant*, trying to match as few characters as possible.

```
|<---------------------------------------->| greedy match
"Run", she said, "I know regular expressions"
|<->|            |<------------------------>| reluctant matches
```

Note: Most regex engines do not consider overlapping matches. They linearly search through the given data and ignore previous matches. That is why ", she said, " is not considered as a match in the previous example.

Alternatives

Custom character classes can be defined by using square brackets. Inside square brackets ^, $ and . loose there special meaning and match themselves. However a leading ^ is used for negation. A pipe | is used for alternatives between sub expressions.

Expression	Meaning
[*class*]	characters from predefined *class*
[aeiou]	characters a, e, i, o and u
[a-f]	characters a to f
[-a-f]	characters a to f and -
[^<>]	any character except < or >
expr_A | expr_B	match expr_A or expr_B

Groups

Subexpressions can be grouped and used in back references or to apply quantifiers. A group is marked by parenthesis about the subexpression. (Abc){3} is a regular expression matching AbcAbcAbc. This is different from [Abc]{3} matches any string of length 3 just containing the characters A, b and c, e.g. Acc.

Moreover the groups in a regular expression are *captured*. This means, they are stored in the order of their appearance and can be referenced in another location of the expression using their number. For example:

```
(foo|bar) == \1
```

matches foo == foo and bar == bar but not foo == bar or bar == foo.

Note: Implementing back references is makes the implementation and of matching algorithms and much more complex and also increases the runtime. Therefore a second grouping syntax exists which avoids capturing. This may should be used if back references are not required (?:expr). So in in the example above (?:Abc){3} preferable over (Abc){3}.

Shell Globbing

While using the shell it often desirable to specify file names using regular expressions, for example to bulk move or rename files. Unfortunately bash does not exactly support regular expressions but a mechanism which is called *globbing* or *path name expansion*.

Path name expansion

In the very last step before a shell command is executed, bash performs *path name extension*. This is quite similar to matching regular expressions against path names. However the syntax is slightly different from POSIX regular expressions. The search space consists of all file names with respect to the working directory.

Expandable expressions

The general form of an extendable expression is given as

expr or quantifier(expr-list)

where expr and quantifier are in as the tables below and expr-list is a list of expressions separated by |.

Expandable expressions are replaced by all matching file named separated by a whitespace. For example du *.tex might be expanded to du chapter1.tex chapter2.tex tex main.tex:

```
[john@laptop thesis]$ ls
chapter1.aux chapter1.tex chapter2.aux chapter2.tex main.tex main.pdf

[john@laptop thesis]$ du *.tex
3123        chapter1.tex
2441        chapter2.tex
800           main.tex
```

Expressions in quotes are never expanded, so du "*.tex" is executed as du "*.tex", probably resulting in an error, as the file *.tex is unlikely to exist.

```
[john@laptop thesis]$ du *.tex
du: cannot access '*.tex': No such file or directory
```

Note: *Path name expansion* is a quite dangerous feature as it may alter the command in unexpected ways. For example, file names starting with a - can be a misinterpreted as options. Also if the expression has no valid expansion it used unchanged and especially meta-characters can cause trouble. Even if not used explicitly, weird file names, e.g. *.tex, could cause globbing in unexpected places. Globbing can be disabled entirely (in the current session) with set -f noglob

Simple expressions

Expression	Meaning
abc	match an actual string
[[:digit:]]	match a character in a POSIX character class
[a-f]	match a character in a character class
\?	match a meta-character
?	match a single character, except /
*	match any string of any length not containing /

Note: Globbing will never expand to hidden files by using the wildcards * or ?. This is another safety measure, commands like rm * would otherwise remove the current directory . and the parent directory ..!

Expressions with quantifiers

Quantifier	Meaning
?	zero or one matches
*	zero or more matches
+	one or more matches
@	exactly one match
!	do not match expression from the list

Examples

Assuming a directory structure like below.

```
[john@laptop ~]$ tree
.
├── bar
│   └── fileC
├── fileA
├── fileA.md
├── fileA.txt
├── fileB
├── fileB.md
├── fileB.txt
└── foo
    └── fileD
```

Expression	Meaning	Expansion
*	all files names	bar fileA fileA.md fileA.txt fileB fileB.md fileB.txt foo
*.txt	all file names ending in .txt	fileA.txt fileB.txt
!(fileA|*.??)	not fileA and no files with a 2 letter extension	bar fileA.txt fileB fileB.txt foo
file[[:upper:]]	all file names file followed by a single capital letter	fileA fileB
*/file[A-Z]	the same inside the every sub directory	bar/fileC foo/fileD
@(fileA|fileB.txt)	fileA or fileB.txt	fileA fileB.txt

3.1.6 Basic shell commands

Index

Basic file operation using the command line

Changing the working directory

The shell and actually any Linux process has a *(current) working directory*. File operations without absolute paths will always be interpreted with respect to this working directory. It is important to keep track where "the shell is located". As a hint the last level of the current path is shown in the command prompt. The pwd command (print working directory), well, prints the working directory and the cd command (change directory) takes a path as argument and changes the working directory accordingly.

```
[john@laptop ~]$ pwd
/home/john

# change the working using an absolute path
[john@laptop ~]$ cd /home/john/foo
[john@laptop foo]$ pwd
/home/john/foo

# change the working directory using a relative path
[john@laptop foo]$ cd bar
[john@laptop bar]$ pwd
/home/john/foo/bar

# change the working directory the users home directory
[john@laptop bar]$ cd
[john@laptop ~]$ pwd
/home/john
```

Note: A common pattern is go up the directory tree by one level with cd .. and using .. as name for the parent directory.

Inspecting directory contents

The ls command lists files in a directory and quick overview about the directory hierarchy itself is shown by tree. The output of both commands can be controlled with various parameters.

```
[john@laptop ~]$ ls foo
bar  baz  filaA  fileB

[john@laptop ~]$ tree foo
foo
├── bar
│   └── fileC
├── baz
│   └── fileD
├── fileA
└── fileB
```

Creating and deleting directories

There are two commands to "make" and "remove" directories mkdir and rmdir. They take a path as argument and are straight forward to use.

```
[john@laptop ~]$ mkdir new_dir
[john@laptop ~]$ rmdir new_dir
```

Note: rmdir will refuse to remove non-empty directories. This is a safety measure, all the contained files and directories have to be removed upfront. Files are removed with rm which also removes directories, so rmdir is seldom used.

Moving and renaming files

The mv command renames or moves files. The order of arguments is, like always, mv source target. As long as no file system boundaries are crossed a move or renaming action just manipulates the related dentry objects. Inodes and file data are usually not touched. In particular no data is copied and the operation is very fast.

If the target is an existing directory mv will keep source's name and move the file into the target directory. In that case it is also possible to give several source names to move multiple files at once. This can be conveniently combined with bash's *path name expansion*.

```
[john@laptop foo]$ mv fileA fileB bar
[john@laptop foo]$ tree
.
├── bar
│   ├── fileA
│   ├── fileB
│   └── fileC
└── baz
    └── fileD
```

Note: If the target is an existing file mv will overwrite that file! To be save there -i enables the interactive mode, which will ask for confirmation before overwriting files

```
[john@laptop foo]$ mv -i baz/filaA bar/fileC
mv: overwrite 'bar/fileC'? n
```

manipulate files

mv SOURCE TARGET cp SOURCE TARGET cp -r SOURCE TARGET rm FILE scp SRC host:TARGET scp host:SRC TARGET mkdir TRAGET rsync tree –du -h

ls - List files

The ls command is used to list files and directories and their meta data like access permissions or size. Most common options are -l, -a and -h. The command ls -lah shows a long listing of all files (including hidden ones) in the current working directory and display human-readable file sizes, e.g. 5.3M instead of 5266573.

A long file listing (-l) shows a lot of metadata information. The following could be an example of an ls -lh invocation:

```
lrwxrwxrwx 1 root root   9    Mar 13  2020  /usr/bin/python3 -> python3.8
-rwxr-xr-x 1 root root   5.3M Jun  2 12:49  /usr/bin/python3.8
-rwsr-xr-x 1 root root   67K  Jul 15 00:08  /usr/bin/passwd
drwxrwxrwt 1 root root   130  Sep  3 07:01  /scratch
-rw-rw----+ 1 john isg 652  Sep 10 15:13  my_notes
```

The following table breaks down the output into the individual fields:

Example	file type	owner permissions	group permissions	other permissions	ACL	hard links	owne	owning group	file size	modification time stamp	file name	info
1	l	rwx	rwx	rwx		1	root	root	9	Mar 13 2020	/usr/bin/	-> python3.8
2	-	rwx	r-x	r-x		1	root	root	5.3M	Jun 2 12:49	/usr/bin/	.8
3	-	rws	r-x	r-x		1	root	root	67K	Jul 15 00:08	/usr/bin/	
4	d	rwx	rwx	rwt		1	root	root	130	Sep 3 07:01	/scratch	
5	-	rw-	rw-	—	+	1	john	isg	66K	Sep 10 15:13	my_note	

File type

The file type field indicates the nature of the file. Most common are *regular files* indicated by - or a directory indicated by d. But there are other file types as well. Note there is no difference between an executable program, a video file or a text document, for the file system all are just *regular files*. See Examples 2, 3 and 5. Another file type is *symbolic link* indicated by l. This correspondents to *junctions* in Windows file systems. This is an actual file, and its content is just the file name of its target. So in example 1 /usr/bin/python3 is a file with the actual content python3.8 (9 bytes). The link target is shown the last column of the output. The permissions on links are *always* rwxrwxrwx, the final access rights depend on the link target.

Most programs *follow* or *dereference* symbolic links by default. So for example by executing the link from example 1 it dereferenced ans its target /usr/bin/python3.8 is executed. This way programs are often provided by different names.

Warning: Some programs actually check how exactly they were invoked and behave very different. An example are the symbolic links mkfs.ext2, mkfs.ext3 and mkfs.ext4 all referring to the same program mke2fs. It creates and ext2, ext3 or ext4 file systems depending by which link it was executed. You may use this “feature” in your own scripts by inspecting the environment variable $0.

There are more file types as shown in the following table.

indicator	file type
-	regular file
l	symbolic link
d	directory
b	block device
c	character device
n	network file
p	FIFO
s	socket

permissions

The next three blocks of 3 letters show the POSIX permissions of the file for *owner*, *group* and *others* respectively. There could be a + behind the POSIX permissions, see example 5, indicating that this file has extra permissions via an *access control list* (ACL).

There are some subtleties for executable files (and directories). The x in the owner or group triplet may be replaced by a s (example 3). In this case the file is executable and additionally the *set uid* or *set gid* permission is set. The process owner or group after execution will be set to the file owner or group.

Similarly the execution permission of the "others" triple for directories may be replaced by a t like in example 5. This marks an activated *sticky bit*: Only the owner of a file may delete or rename that file. That makes sense for shared directories like /scratch or /tmp

To be complete, there is a *sticky bit* for files indicated by T but this was historically important and has no meaning in modern Linux systems.

To review or modify ACLs on files use setfacl and getfacl respectively.

modification time

On most file Linux file systems are three time stamps which are tracked with each file. The are called *atime*, *mtime*, and *ctime*. It may be that file systems suppress updates on time stamps (notably *atime* via noatime mount option) to improve performance.

time stamp	name	ls option	meaning
atime	access time	-u	Time of the last read-only access of the files data
mtime	modification time	(default)	Time of the last write access to the files data
ctime	change time	-c	Time of the last change to the files metadata

other useful options

There are many options to ls. The following table shows the most common ones:

option	meaning
-a	Show also hidden files, i.e. files name beginning with a .
-s	Sort the output by file size
-t	Sort the output by modification time
-tu	Sort the output by access time
-tc	Sort the output by change time
-r	Reverse the ordering
-R	Recursively list the contents of all directories as well

top - Interactive view of system resources and processes

The top program is one of few interactive system inspection commands. It shows real time data about running processes, memory consumption and a basic summary about the system. It really useful to inspect what is currently going on a system ans provides a starter investigate resource consumption. It a interactive program, so after invocation it will show a lot of data which is updated every 3 seconds and meanwhile it waits for further user input.

There are a lot of short cuts to control a running top, most often a single key stroke is enough to trigger an action. Most importantly hitting q quits the top program.

A simple invocation of top shows a header with basic system information and below a table of running processes.

Below is a typical header as displayed by top. Breaking it down line by line already reveals a lot of information.

- top - 12:34:56 up 34 days, 2:51, 1 user, load average: 60.96, 60.96, 60.80
  - the current system time is 12:34:56
  - the systems is running (*uptime*) for 34 days and 2h 50m.
  - 1 user is currently logged in
  - the load average is
    - 60.96 over the last minute
    - 60.96 over the last 5 minutes
    - 60.80 over the last 15 minutes
- Tasks: 1409 total, 61 running, 1347 sleeping, 1 stopped, 0 zombie
  - There are 1409 processes of which
  - 61 are running, i.e. are executed right now
  - 1347 are inactive, usually waiting for data or disk I/O, etc.
  - 1 is stopped, usually by a user
  - There are no zombie processes, which is good
- %Cpu(s): 0.0 us, 20.6 sy, 26.2 ni, 52.9 id, 0.0 wa, 0.0 hi, 0.2 si, 0.0 st
  - This a break down of how many time (in percent) the CPUs are occupied which different tasks
    - us: userland processes (with default nice value)
    - sy: system processes
    - ni: userland processes (with non-default nice value)
    - id: idle, i.e. non of the above, $100 - (\text{us} + \text{sy} + \text{ni})$
  - The next values are for system administrators indicating time spent on handling interrupts (hi, si) and waiting for I/O (wa, st)

```
top - 12:34:56 up 34 days,  2:50,  1 user,  load average: 60.96, 60.96, 60.80
Tasks: 1409 total,  61 running, 1347 sleeping,   1 stopped,   0 zombie
%Cpu(s):  0.0 us, 20.6 sy, 26.2 ni, 52.9 id,  0.0 wa,  0.0 hi,  0.2 si,  0.0 st
MiB Mem : 2052001.+total, 1514273.+free, 233328.6 used, 304399.2 buff/cache
MiB Swap: 210215.0 total, 210206.5 free,      8.5 used. 1808182.+avail Mem
```

There is a more sophisticated version of top called htop the functionality is similar but the user interface is a bit simpler. Also nvtop is similar but for GPUs.

man - View documentation

There are many command line programs installed. It is impossible to remember their exact functionality and options. Almost all of them are well documented. The documentation is called *man page*, is installed on most systems for fast reference. The `man` tool is used to display the man-pages.

The basic invocation is `man pagename` where `pagename` is the topic of the desired reference page. Typically this is a command name, e.g. `man ls` or man `msn chmod`. In a sense little more has to be said than referring to manual page of the manual itself: `man man`

Each man page is divided in several paragraphs, the most important ones (in the order they appear in the manual page) are:

- `NAME`: A short summary about the tool or topic
- `SYNOPSIS` On overview how to invoke the command.
- `DESCRIPTION` A detailed listing of command options and their effects
- `EXAMPLES` Basic examples for common command invocations
- `SEE ALSO` References to further documentation and related topics

Searching man pages

Sometimes it is not clear which man page should be consulted or what is the exact name of the man page. There are some commands which help to search across all man pages:

Command	Result
`whatis name`	Search for *name* in all man page titles
`apropos name`	Search for *name* in all man page NAME sections
`man -K name`	Search for *name* in all man pages. This is very slow and usually produces a lot of results.

On D-PHYS managed computers there is also the (non-standard) tool `tldr name`, which provides a handy short version of the man pages.

man page sections

```
1   Executable programs or shell commands
2   System calls (functions provided by the kernel)
3   Library calls (functions within program libraries)
4   Special files (usually found in /dev)
5   File formats and conventions, e.g. /etc/passwd
6   Games
7   Miscellaneous (including  macro  packages  and  conventions),  e.g.
    man(7), groff(7)
8   System administration commands (usually only for root)
9   Kernel routines [Non standard]
```

CHAPTER

FOUR

LINUX BASICS (II)

4.1 Index

4.1.1 Input/Output redirection

The Bourne Again Shell supports multiple the input and output of programs. This I/O manipulation is the core functionality to combine simple individual tools.

Index

File Descriptors

A *file descriptor* (fd) represents an entry in the operation system's global *open file table*. Whenever a process requests to open a file, the kernel creates an entry in the open file table and returns a file descriptor to that process. Every process has its own table of file descriptors.

Multiple file descriptors can reference to same entry in the kernel file table. It does not matter which file descriptor is used. All meta-data is stored in the file table, such that, for example, seeking in that file changes the file position for all file descriptors.

Representation of File Descriptors

File descriptors are numbered, starting from 0. All processes have their own descriptors. They are exposed in /proc/$pid/fd and /proc/$pid/fdinfo. A file in proc/$pid/fd is a symbolic links and the target is the name of the corresponding file, device or internal object associated to that file descriptor.

Every process can access their file descriptors via /proc/self/fd.

Standard File Descriptors

Each process has at least 3 file descriptors. They have the numbers 0, 1 and 2 and are called the *standard file descriptors*:

fd	Name	Usage
0	stdin	Standard input
1	stdout	Standard output
2	stderr	Standard error, output of errors or additional information

In case of terminal window, they are usually attached to a (pseudo) terminal device in /dev/pts/. The descriptor stdin is used for reading from the character device (e.g. keyboard input), while stdout and stderr usually point to the same file entry which is opened for writing to the character device (e.g terminal output).

Note: Example: A terminal with bash (pid 17757) has four file descriptors. The three standard file descriptors are all pointing to the pseudo terminal device /dev/pts/1. FD 255 is bash specific. In fact file descriptors 1 and 2, (respectively 0 and 255) reference to same entry in the open file table. More information is shown by lsof.

```
$ ls -log /proc/17757/fd
lrwx------ 1 64 Oct 11 17:38 0 -> /dev/pts/1
lrwx------ 1 64 Oct 11 17:38 1 -> /dev/pts/1
lrwx------ 1 64 Oct 11 17:38 2 -> /dev/pts/1
lrwx------ 1 64 Oct 11 17:38 255 -> /dev/pts/1
```

I/O Redirection

The Bourne Again Shell supports multiple operators to manipulate file descriptors. They can be opened, closed, redirected to other files or duplicated. Most often redirection and duplication of the standard file descriptors is used. This I/O manipulation is the core functionality to combine simple individual tools.

Redirection Operators

The following table lists the most common redirection operators in their general form:

Operator	Result
<n file	Open file for reading on fd n
n> file	Open file for writing (truncate) on fd n. Redirect fd n if it already exist
n>> file	Open file for writing (append) on fd n. Redirect fd n if it already exist
n>&m	Make fd n a copy of fd m (which should be open for writing)
n<&m	Make fd n a copy of fd m (which should be open for reading)

Often they have abbreviated forms when dealing with the standard file descriptors:

Operator	Result
> file	Redirect stdout to a file (truncate)
2> file	Redirect stderr to a file (truncate)
>> file	Redirect stdout to a file (append)
2>> file	Redirect stderr to a file (append)
< file	Redirect stdin to a file
2>&1	Make stderr a copy of stdout
1>&2	Make stdout a copy of stderr
cmd_1 | cmd_2	Redirect cmd_1's stdout to cmd_2's stdin
cmd_1 |& cmd_2	Short for cmd_1 2>&1 | cmd_2

4.1.2 Shell Variables

The Bourne Again Shell and any other shells support the use of variables. The basic usage of variables is very simple. They do not need to be declared upfront and can just defined by an assignment. To reference a variable (i.e. use its value) a $ symbol has to be prepended. The name of a variable has to start with a letter and may contain numbers and underscores. Often uppercase names are used.

```
A=123
echo $A
123

A=abc
echo $A
abc

MY_VAR1="some string with spaces"
MY_VAR2=
```

Index

Variables

Names

A variable name can contain letters, digits and underscores, it just cannot start with a digit. It is a wide spread convention to use upper case letters for variable names, but this is not mandatory. It is important, to be aware of already defined *environment variables* to avoid name clashes.

Note: If it becomes necessary, names could be made explicit with curly braces while referencing. For example, in the expression 123${SEP}45 the variable referenced is SEP, and not SEP4 or SEP45. This is also important while using arrays.

When not declared otherwise, all variables are treated as *strings*. That means bash does handle A=123 as a string and not an integer. To explicitly define an integer variable use declare -i A=123 instead. This is seldom necessary as variables are casted on demand.

Types

Most importantly, every process has its own set of variables. They are referred to as *local variables* or just variables. A subset of these variables are *environment variables* (sometimes also called *global variables*). The difference is how they are inherited to child processes:

- *global/environment* variables/: Inherited to child processes
- *local* variables: Not inherited to child processes

It important to understand, that every process has its own copy of local and environment variables. The environment is inherited from the parent process on start up. But after that, any changes to an environment variable by any other process (even the parent process) do not effect the current environment.

Some programs, most notably shells, import variables from certain files (e.g. /etc/profile, ...) and add them to their environment. This interferes with variable inheritance.

Exception to Inheritance

An important exception to rule of inheritance are sub-shells. bash might start another bash process by splitting itself up into to possesses (*forking*). The new split-up process is called a *sub-shell*. It a perfect copy of the parent process (except its pid) including the whole process state and variables. This means a sub-shell does inherit local variables from its parent.

Actually, bash executes any other program as follows: First it invokes the *fork system call* to split itself up (duplicating all the variables). And then the child uses an *execv system call* to replace itself with the program to execute (and loses all local variables).

Defining Variables

By default all variables are local and the can just be defined by an assignment. It is important to leave no whitespace around the =. This would be a syntactic error. To define a global variable it has to be exported, either direct in an assignment or later by invoking export VARNAME. The export can be undone with export -n:

```
MY_LOCAL_VAR=123             # defining a local variable
export MY_GLOBAL_VAR=123     # defining a global variable
export MY_LOCAL_VAR          # this now also a global variable
export -n MY_LOCAL_VAR       # now it is local again
```

If not otherwise stated, all variables are treated as strings. E.g. MY_LOCAL_VAR is not interpreted as an integer. It is rarely needed but variables and their attributes can be declared explicitly:

```
declare FOO=bar              # equivalent to FOO=bar
declare -i N=100             # N is now defined as integer
declare -x MY_GLOBAL_VAR     # equivalent to export MY_GLOBAL_VAR
```

Sometime it is useful do delete a variable. An assignment VAR="" or even VAR= is not enough. This just assigns an empty string to VAR, but VAR is still defined. The command unset VAR undefines VAR.

Note: bash provides a special way to inherit or inject variables to its child processes' environments without setting them for the parent process. If variables are set on the same line they are inherited as environment to the child but do not alter the environment of the parent (i.e. the current shell) LC_ALL=C MY_VAR=120 ./my_prog In the previous example the (existing) environment variable LC_ALL is not altered and (the non-existent variable) MY_VAR does not become an environment variable in the current shell, but both will be seen as environment from my_prog.

Inspecting Variables

The attributes (including the scope) and the value of a variable can be checked with declare -p, this command displays an equivalent definition of the variable using the declare built-in command.

```
declare -p MY_LOCAL_VAR
declare -- MY_LOCAL_VAR="123"   # -- can be ignored

declare -p MY_GLOBAL_VAR
declare -x MY_GLOBAL_VAR="123" # -x indicates a global variable
```

There are several ways to inspect variables, some are listed in the following table.

Command	Effect
env	Print all environment variables
printenv	Print all environment variables
set	Print all shell variables (including functions)
declare -x	Print all environment variables declarations
export	Print all environment variables declarations

Using Variables

Variables are referenced by their name preceding a $ symbol. There is no distinction in local and global variables. bash will replace $VAR by its value (or empty string if VAR is undefined). Variables can be used inside strings and other expressions. They can even be used as commands themselves.

```
NAME=World
echo "Hello$NAME!"    # prints/echoes "HelloWorld!"

OPTIONS="-l"
COMMAND="ls"
echo Now running $COMMAND $OPTIONS  # echoes "Now running ls -l"
$COMMAND $OPTIONS                   # runs ls -l
```

Special variables

Bash maintains a set of special variables. They are read-only and very useful in scripting.

Variable	Meaning
$?	Exit code of the last command
$$	The pid of the current shell
$0	The name of the current shell or shell script
$n	The n-th positional argument for the current shell script. For $n \geq 10$ the form ${n} makes the expression unambiguous
$#	The number of arguments for the current shell script

The exit code is useful to check for errors in a shell script. For example in the snippet below a directory foo is created. But this may fail, for example because the directory already exists, in this case $? will be equal to 1 and the script can react accordingly. If mkdir foo succeeds, $? will be equal to 0.

```
mkdir foo
if [ $? -neq 0 ] ; then   # if "$? is not equal to 0" then
   # error handling
```

To illustrate the other special variables assume the following shell script my_script.sh:

```
#!/bin/bash

echo "Hello, this is $0"    # print out a greeting
if [ $# -neq 2 ] ; then     # if $# is not equal to 2
    echo "I need exactly 2 arguments"
    exit 1                  # exit with exit code 1
fi
echo "My arguments are $1 and $2"
exit 0                      # exit with exit code 0
```

The example invocations show the values of the special variables. In particular $0 includes

```
./my_script.sh foo bar
Hello, this is ./my_script.sh
My arguments are foo and bar

some_path/my_script.sh foo bar baz
Hello, this is some_path/my_script.sh
I need exactly 2 arguments
```

Important environment variables

There are a lot of environment variables set. Some are configuration, others are just informational. All currently set environment variables are listed by printenv. The following table shows a small excerpt of commonly used variables.

Variable	Usage
PATH	A : separated list of path to look for executables
EDITOR	The preferred (command line) text editor
HOME	Path of the current users home directory
USER	The user name of the current user
LANG	Preferred language for user interfaces
LC_ALL	A catch all setting for locale settings
LC_*	More fine grained locale settings, e.g. LC_COLLATE, …
IFS	Internal field separator
GIT_*	Many programs define their own variables, e.g. for git: GIT_AUTHOR, …

HOME and USER are useful to make shell scripts user agnostic. EDITOR is consulted by programs which start editors for the user, e.g. git or visudo. LANG can be changed ad-hoc to read manual pages in an other language. For example LANG=de_DE.UTF8 man bash shows the manual for bash in German (if available the system). IFS defines how certain tools (e.g. read) split data into individual fields. It often set in an ad-hoc per command fashion or stored and reset after a block of commands.

Many tools, for example git, read their configuration from files and (with higher precedence) from the environment. This is useful to override settings without touching the configuration files.

Note: A common pitfall is to override PATH instead of just adding a custom directory.

```
PATH=/opt/bin           # this sets PATH to /opt/bin,
                        # i.e. removes /usr/bin etc.
PATH="$PATH:/opt/bin"   # this extends PATH by /opt/bin
```

Often it is more desirable to keep the default PATH and create symbolic links in a location which is already in the PATH. For example in ~/.local/bin/:

```
ln -s /opt/bin/my_prog ~/.local/bin/my_prog
```

Permanent changes to the environment

A change or definition of an environment variable is only seen by the current shell and its child processes. It will be lost once the shell is closed. The mechanism to make permanent changes is straight forward. There are several files which will be *sourced* on login or when creating a new shell. Variables defined in these files will be available for every new shell. There are several files which will be read in different situations.

User specific environment

File	Usage
~/.bash_profile	Sourced at login, e.g. SSH logins
~/.bash_login	Alternative for .bash_profile
~/.profile	Common profile, sourced by bash in the absence of bash_profile and bash_login, but also read by other shells
~/.bashrc	Sourced on start of non-login shells, e.g. start of new terminal
~/.bash_logout	Sourced on logout

Note: It is very common to modify just ~/.bashrc and reference to it again in ~/.bash_profile. So a typical ~/.bash_profile looks like

```
if [ -f ~/.bashrc ]; then   # check if ~/.bashrc exists
  source ~/.bashrc          # if so, source its content
fi
```

This way everything from ~/.bashrc also applies to login shells, e.g. SSH logins.

Systemwide environment

The files mention in the previous section are located in a user's home directory and are only consulted by programs run by this user. There are analogous files which apply to every user. They are read before the user profiles, i.e. the latter have a higher precedence as they may override variables. Most notably there are

- /etc/bashrc
- /etc/environment
- /etc/profile.d/*.sh

Actually /etc/profile is executed on logins and reads all the *.sh files from /etc/profile.d/. These files should not be modified as they come with package installations and subsequent updates may delete any changes. Instead new files should be added.

Many distributions also come up with other or more files to restore an environment.

4.1.3 Scripting

Instead of giving all commands on after another to the interpreter, they can put into a file an executed as a whole. This way many tasks could be easy and conveniently automated. Even complete complete programs realized if necessary. This is much more lightweight than using any other compiled language.

Bash provides a range of typical control structures as conditional branching and loops. Even functions and arrays are supported to some extent.

Index

Script Interpreter

If an executable file is run, the kernel checks the first bytes of the file to choose the right handler to continue. If the file begins with #! it is treated as a script file. This marker is known as *shebang* or *hashbang*. It is followed by the full path of the interpreter to be used for the rest of the file. The kernel runs interpreter and passes over the script.

This first line does no harm to the logic of script itself as lines beginning with # are treated as comments.

The interpreter is usually a real scripting language interpreter as /bin/bash or /bin/ptython but could be any program. It has to be specified by full path. The which command helps to quickly find the correct path, e.g. which bash gives /usr/bin/bash.

A simple shell script (say hello.sh) may look like this:

```
#!/usr/bin/bash
echo "Hello World"
```

It can be executed like any other program.

```
$ chmod u+x hello.sh
$ ./hello.sh
Hello world
```

The same could be realized as a python script, e.g. hello.py:

```
#!/usr/bin/python
print("Hello world")
```

Conditional Statements

The basic structure of conditionals is similar to most other programming languages:

```
if test1; then    # if "test1 has exit code 0" then
   # block 1
elif test2; then  # else if "test2 has exit code 0" then
   # block 2
else              # else
   # block 3
fi
```

Here test1 and test2 can be any shell command. The exit code is evaluated, and only exit code 0 is treated as “true” and the subsequent block is executed. Any other exit code is handled as “false”.

More complex logical conditions, e.g. negations or conjunctions, have to be built into the test commands.

There can be any number of elif (*else if*) blocks, or none at all. Also the last catch-all else-block is optional. The fi statement (if spelled backwards) closes the if block.

Note: Depending on the style, the statement can be split into multiple lines, this makes the semicolons superfluous:

```
if test1
then
  # block1
elif test2
then
  # block2
fi
```

The most compact form would be everything on one line. In this case additional semicolons are needed:

```
if test1 ; then ... ; elif test2 ; then ... ; fi
```

Often there are spaces around the semicolons to increase readability, but their are not strictly necessary.

Tests

To make full use of conditionals a wide range of tests is helpful. Traditionally there have been two programs test (/usr/bin/test) and [(/usr/bin/[) which take arguments to build test conditions and logical expressions.

[is a weird file name to make the expressions more readable. It also expects a closing bracket as its last argument. To test, if $A is equal to 0, the following statements can be used:

```
test $A -eq 0     # Ok, 3 arguments for test
[ $A -ge 0 ]      # Ok, 4 arguments for [
[$A -ge 0]        # Syntax error, missing spaces
```

This explains the unusual syntax of the comparison operators and the necessary spaces around the brackets. Bash also provides a built-in command [[which is very similar to [and test. It has more features and is easier to handle.

```
[[ $A -eq 0 ]]    # exit code 0 iff $A is equal to 0
```

Test operations

Numerical comparison

The following table list comparison operators for integer values. It important to note that [[interprets empty strings a numerical value 0. For example [[$A -eq 0]] with A undefined has exit code 0, while [$A -eq 0] yields an syntax error. Variables do not have to be declared as integers (decalre -i) for these test to work, they just have to valid string representations for integers, e.g. A=10, B=-12, C=013, … Otherwise [[exits with an error and an exit code other than 0 or 1.

Operator	Math	Meaning
-gt	$>$	Greater than
-ge	$\geq$	Greater or equal
-lt	$<$	Less than
-le	$\leq$	Less or equal
-eq	$=$	Equal to
-ne	$\neq$	Not equal to

Note: Comparison of floating point numbers is seldom used and much more complicated. And extra tools like bc (basic calculator) have to be used. For example, testing $A $\geq$ $B with floating point numbers is achieved by:

```
(( $(echo "$A >= $B" | bc) ))
```

String comparison

The following table list the string comparison operators for [[. Instead of =, also == can be used to be more in line with most other programming languages.

Operator	Meaning
>	Lexicographical less than
<	Lexicographical greater than
=	Equal to, the right hand side can contain bash wild cards
=~	Equal to, the right hand side can be a regular expression
!=	Not equal to

All comparisons are case-sensitive. And the lexicographical order may depend on the environment (LC_COLLATE).

The equality test = (or ==) accepts shell wildcards on the right hand size, e.g. * or ?. In particular they have to be escaped (on the rhs) for literal matches, e.g.

```
[[ $A = foo\? ]]
```

is true only for A=foo?, while [[$A = foo?]] evaluates to true for A=foo? or A=food.

The =~ is similar but accepts regular expressions on the right hand size.

```
[[ $A =~ ^foo.$ ]]
```

is true for all values of A starting with foo followed by exactly one other character.

File tests

Often it useful to check if a file exits or its attributes, [[implements some file tests. All operators in the following table are unary operators, i.e. just take one argument.

Operator	Meaning
-e	File or directory exists
-s	File exists and is non-empty
-x	File is executable
-f	File is a regular file

For example [[-x my_script.sh]] has exit code 0 if and only if my_script.sh exists and is executable. A common bash idiom is to check if a file to executable before actually executing it: [[-x foo.sh]] && ./foo.sh.

Boolean operators

To build more complex expressions, simpler ones can be combined by boolean operations. All these are dedicated shell built-ins and treat exit code 0 as "true" and everything else as "false". However [[does implement them as well, so they can be used inside and outside of [[-expressions. It is better to use them inside since this makes grouped conditions easier to write and read. As usual "*not*" has a higher precedence than "*and*", which has higher precedence then "*or*":

Operator	Math	Meaning
||	$\vee$	(non-exclusive) Or
&&	$\wedge$	And
!	$\neg$	Negation

Example: Both of the following expression evaluate to true, if A is not 1 and between 10 and 20, i.e. $\neg(A = 1) \wedge (A \geq 10 \vee A \leq 20)$:

```
[[ ! $A -eq 1 && ( $A -ge 10 || $A -le 20 ) ]]
! [[ $A -ne 1 ]] && ( [[ $A -ge 10 ]] || [[ $A -le 20 ]] )
```

Loops

There are several looping constructs in bash. The most straight forward constructs are *loop-while* and *loop-until*. The *for-loop* is very powerful in combination with shell globbing.

Loop-while and Loop-until

Very much like conditionals the *while* and *until* statements evaluate the exit code of test command. The difference is, until loops as long as this exit code is non-zero and while loops as long as it is equal to zero.

```
until test-cmd ; do
  ... # repeat this as long as test-cmd has a non-zero exit code
done

while test-cmd ; do
  ... # repeat this as long as test-cmd has exit code zero
done
```

These loops are (compound) commands themselves. In particular they have exit codes and rules for redirection apply. A very common pattern is to loop over lines in a file:

```
while IFS="" read -r line ; do
    # to something with "$line"
done < data.txt
```

The bash built-in read -r reads words from stdin until a newline character occurs and stores them in given variables (here just line). The word boundaries are defined by the environment variable IFS (internal field separator). Setting this to on empty string (just for that read command) defines the whole line as one "word". Finally via input redirection stdin is replaced with the file data.txt.

Similar, if data.csv contains lines with key/value pairs (e.g. key1;value1), these fields could be read in directly:

```
while IFS=";" read -r key value ; do
    # to something with "$key" and "$value"
done < data.txt
```

CHAPTER

FIVE

PYTHON ECOSYSTEM (I)

The Python Ecosystem modules are meant to present the basics of the *environment* around Python. The goal is *not* to teach you the Python programming language, or its scientific programming libraries. Rather we want to discuss the «ecosystem» of the Python programs, where they "live" and the surrounding they interact with.

Nevertheless, as our presentation moves on, the amount of actual Python code will keep increasing. Even though our modules are targeted at beginners, we believe that even intermediate Python programmers can still learn a few tricks and improve their coding style.

In general we aim to provide and strengthen the understanding of various underlying concepts and best practices. We hope that, by providing you with a firm foundation, you can grow beyond the naive "search the web and copy-paste" and gain a deeper understanding.

This first module focuses on **getting ready to run Python code**.

5.1 Index

5.1.1 Interpreter Management

Which Python?

There is no single "Python". Obviously there's the programming language called Python, meaning the syntax and instructions the code is written in. Then there's the Python executable that interprets and runs the code. In most cases this refers to the CPython reference implementation. Together with the Python language it comes in different versions, like the obsolete 2.7.18 or the current 3.11.x release. To add yet another layer of complexity, there are various alternative Python implementations that can execute Python code.

Even when only considering CPython, the same computer can have several Python versions installed, and possibly multiple interpreter executables in different locations for exactly the same version.

It's crucial to understand which Python you are using and to know how to manage different interpreters and versions.

Index

Python implementations

The Python programming language has been implemented in different executables and runtime environments.

CPython

CPython is the reference implementation written in C and Python. It is the most wide-spread and de-facto standard Python executable. Almost all of the time “Python interpreter” refers to the CPython implementation. And most of the time it’s more than enough to work with CPython.

Alternative implementations

It is important to keep in mind that alternative Python implementations exist. They may provide a speedup or other benefit depending on your specific usecase or environment. On the downside they do not always support all features of CPython and may not support all packages.

PyPy

The PyPy implementation is written in a restricted subset of Python called RPython (Restricted Python). It often runs faster than CPython because it’s a just-in-time compiler instead of an interpreter. On the downside there are also differences compared to CPython, in particular when using C extensions.

IronPython, Jython

IronPython is a Python implementation written in .NET, whereas Jython is implemented in Java. These variants are useful if you need to combine Python with existing .NET or Java code.

Brython, Pyodide

Brython (Browser Python) is a rather exotic implementation that allows to run Python code directly in the browser by translating it into Javascript. Pyodide brings Python and various scientific libraries to the browser using WebAssembly.

MicroPython, CircuitPython

MicroPython and CircuitPython are Python implementations optimized to run on microcontrollers.

Note: From now on we always mean the CPython reference implementation when refering to the Python interpreter. As this is the de-facto standard for most users, it simplifies the discussion by not having to think about possible deviations when using other implementations.

Python interpreters

Most computers have multiple Python interpreters installed that need to be distinguished and managed.

System Python

Depending on the operating system and version, your computer may have zero, one or even multiple Python interpreters pre-installed. This is referred to as system Python, as it ships as part of the operating system.

Warning: If you use the system Python, never install packages globally (with a naive pip install <package>). Otherwise you may end up breaking your operating system because of incompatibilities between the package versions you installed and what the system requires to work properly.

Linux

Let's consider the example of our D-PHYS Linux server login.phys.ethz.ch running Ubuntu. Use the which -a command to list all available commands found in the $PATH.

```
johndoe@phd-login1:~$ which -a python3
/usr/bin/python3
/bin/python3
```

Given that /bin is a symlink to /usr/bin, both executables refer to exactly the same file.

Next we determine the actual Python version

```
johndoe@phd-login1:~$ python3 --version
Python 3.10.6
```

Lastly we check what lies behind the generic python command.

```
johndoe@phd-login1:~$ which -a python
/usr/bin/python
/bin/python
johndoe@phd-login1:~$ ls -l /usr/bin/python
lrwxrwxrwx 1 root root 7 Apr 15  2020 /usr/bin/python -> python3*
```

So in this particular case python is identical to python3 as it is a symlink. But this may not be the case on other systems.

You may notice that there is still a python2 installed.

```
johndoe@phd-login1:~$ python2 --version
Python 2.7.18
```

However always use Python 3+ exclusively, as the old version 2 is end-of-life and no longer developed since January 2020 (PEP 373).

macOS

The latest macOS ships only Python 3 as part of the operating system (or more precisely, its Xcode command line tools).

```
john@macbook:~$ /usr/bin/python3 --version
Python 3.9.6
```

But before macOS 12.3 /usr/bin/python was a symlink to the obsolete python2.

Windows

There is no Python installation included with Windows.

User Python

If the operating system comes without Python or ships an outdated version, the user has to manually install a Python interpreter. In contrast to the system Python this is commonly referred to as user Python.

Linux

Typically the system Python is sufficient for most users. Especially on our D-PHYS Linux workstations, where the user has no admin rights and cannot mess up the system-wide package installation. Otherwise we recommend to use *pyenv*.

macOS

The recommended way is to not touch the system Python at all and install a user Python with homebrew.

```
john@macbook:~$ brew install python
```

It even allows to install multiple (major) versions with `brew install python@3.x`, while keeping them easily up-to-date with `brew upgrade`. Experienced users may even use `pip install <package>` to install packages globally inside the homebrew Python.

Windows

Python can be installed directly from the store or using the official packages. Alternatively you may use the Anaconda distribution.

Selecting the proper interpreter

Note: From now on, whenever we use the generic `python` command, we mean the Python 3 interpreter and you may replace it with `python3` if needed.

Linux / macOS

On Unix-like systems Python programs are typically started from the command line. The actual shell command is the Python interpreter followed by the name of the Python script.

```
$ python myscript.py
```

As mentioned above (and in the Linux modules), `python` refers to the first command found in the list of directories in your shell's $PATH environment variable. You can use `which -a python` to see the full list. You can either adapt the $PATH or explicitly call a different Python interpreter by using its full path.

```
$ /usr/local/bin/python myscript.py
```

Alternatively a shebang can be used as first line in the Python script, to tell the system which interpreter to use when executing this file. Even though this could be the full path to a specific interpreter, it's best practice to let the shell's environment decide which interpreter to pick. This allows other users to select their favorite interpreter by adapting the $PATH variable. Also note how we explicitely use python3 to avoid picking up an obsolete Python 2.x on some systems.

```
#!/usr/bin/env python3
print("Hello World")
```

With a correct shebang, you can make the script executable with chmod and run it directly.

```
$ chmod a+x myscript.py
$ ./myscript.py
```

Despite the shebang, you are still free to run it with any other interpreter.

```
$ /path/to/some/python myscript.py
```

Windows

The selection of the interpreter in the Windows command line is conceptually similar to Unix. You also run a Python script using the Python executable.

```
$ python myscript.py
```

If multiple python commands are installed, a path environment variable defines the order of precedence of the various folders. The actual syntax depends on the shell used.

Command Prompt (cmd):

```
$ echo %PATH%
$ where python
```

PowerShell:

```
$ $Env:Path
$ Get-Command python -all
```

pyenv

The pyenv tool allows you to install and manage multiple versions of the Python interpreter. It also supports alternative Python implementations out-of-the-box, allowing you to test your code in different environments.

Under the hood, pyenv is a collection of shell scripts. In particular python-build provides the pyenv install functionality to download the source code, compile and install a given version of the Python executable.

Installation

Linux

The root user needs to install the packages required to build Python from source.

```
# apt install --no-install-recommends make build-essential libssl-dev zlib1g-dev libbz2-dev libreadline-
↪dev libsqlite3-dev wget curl llvm libncurses5-dev xz-utils tk-dev libxml2-dev libxmlsec1-dev libffi-
↪dev liblzma-dev
```

No admin rights are required to install and use pyenv itself

```
$ git clone https://github.com/pyenv/pyenv.git ~/.pyenv
$ cd ~/.pyenv && src/configure && make -C src && cd ..
$ ~/.pyenv/bin/pyenv --help
```

Note that you will have to adapt your $PATH if you want to use the pyenv command directly in your shell, without prepending the full path ~/.pyenv/bin/pyenv.

macOS

We recommend to use a package manager like homebrew.

```
$ brew install pyenv
```

Windows

Consider using the pyenv-win port on Windows.

Usage

Start by listing the ~500 Python interpreters that can be installed with pyenv

```
$ pyenv install --list
```

Then install any version you need

```
$ pyenv install <version>
```

The interpreter will be installed into ~/.pyenv/versions/<version>/. You can use the full path to the binaries to run a specific python interpreter,

```
$ ~/.pyenv/versions/<version>/bin/python your_python_script.py
```

or to create a new virtual environment and install packages within.

```
cd some_python_project/
~/.pyenv/versions/<version>/bin/python -m venv .venv
source .venv/bin/activate
pip3 install -r requirements.txt
```

Feel free to adapt $PYENV_ROOT to change the install location. Refer to the pyenv documentation for other ways to manage the Python version in the shell or per-project.

5.1.2 Package Management

Python ships with the so-called *Standard Library* containing over 200 modules. In addition, there are over 330k packages published on the Python Package Index (PyPI) that can be installed with the Python package manager pip. Its vital for any Python user to learn how to install and manage these extra packages.

See also:

Instead of relying on the native PyPI packages and virtual environments, you can also make use of the freemium Anaconda Python distribution and its conda package and environment manager. Especially on Windows it can provide an easy way to install scientific Python packages. In general we believe that the built-in pip and venv should fit most needs. And the concepts around dependency management and virtual environments are anyway similar.

Index

Package Location

Before we explain how to install additional packages, let's first investigate where Python looks for packages in order to use them.

When Python imports a module, it basically searches through the directories listed in sys.path. For debugging purposes it can be helpful to inspect the list of folders on your platform.

```
$ python -c 'import sys; [print(p) for p in sys.path]'
```

It typically consists of

- the empty string or the current directory, depending how Python was invoked
- optional directories from the PYTHONPATH environment variable
- the directories with the standard library
  - /path-to-python/lib/python311.zip for embeddable installations
  - /path-to-python/lib/python3.11 with Python modules
  - /path-to-python/lib/python3.11/lib-dynload with C extensions
- the site-packages directory with third-party packages
- optional directories defined by .pth files

Of particular significance is the site-packages directory. This is where any additional packages are located, that have been installed by pip. Our Linux machines will also include dist-packages folders that are used by the Debian packages.

It's important to understand that the site-packages location is determined dynamically by the site module. The lookup logic is crucial for virtual environments to work.

For sake of concreteness, let's consider the example import pandas. Python will loop through the sys.path directories and look for pandas.py or pandas/__init__.py. The first package found with the matching name gets imported. As direct consequence, only a single version of any package can be used. This is why package management and virtual environments are indispensable.

Tip: Most users should avoid "hacking" the PYTHONPATH or sys.path directory lists. We recommend to use proper *virtual environments* instead.

See also:

- How the Python import system works
- Python import system reference
- How Python sets sys.path

pip

The package installer for Python, pip, is used to install packages from PyPI and other sources.

Usage

Running pip

The pip tool should be included in your Python installation. Otherwise check their installation instructions. You can check its version and installation path.

```
$ pip --version
pip X.Y.Z from /path/to/site-packages/pip (python 3.11)
```

Instead of using the pip command directly you can also call the Python interpreter with the pip module.

```
$ python -m pip --version
```

This syntax allows you to easily select the pip for any given Python installation.

Upgrading pip

On some systems you may want to upgrade pip to the latest version. It's usually safest to do this inside the user scheme, as it does not require admin rights or affect the system-wide installation.

```
$ python -m pip install --user --upgrade pip
...
Sucessfully installed pip-X.Y.Z
$ python -m pip --version
pip X.Y.Z from /home/johndoe/.local/lib/python3.11/site-packages/pip (python 3.11)
```

Note how the pip installation path resides inside the user's home in ~/.local/lib/.

Important: Depending on how Python was installed on your system, pip may warn or even fail when used outside of a virtual environment, regardless of the --user flag. The underlying reason is that PEP 668 allows the system Python to be flagged as "externally-managed" and redirecting users to virtual environments instead of interfering with the system installation. Please consider migrating to proper venvs and disregard the --break-system-packages flag to make pip install packages nevertheless. The goal of PEP 704 is to require virtual environments by default in Python 3.13+.

Installing packages

The most common task is to install Python packages, either in their most recent version,

```
$ pip install <package>
```

or with a specific version number.

```
$ pip install <package>==2.0
```

Another common usecase is to install a Python package directly from a locally checked out git repository. Combined with the -e flag, the package is installed in "editable mode", where all modifications in the repo become directly available in the installed package. This simplifies active development in a project, while also keeping it installed as dependency for related projects.

```
$ git clone git@github.com:path/to/py-project.git
$ pip install -e py-project/
```

Tip: Some packages have C or Fortran bindings that may need to be compiled from source. This can require additional compilers and header files to be installed on your computer.

Most packages are also distributed as pre-built wheels. Installing a package using such binary files skips the requirement to compile from source, thereby making the installation much faster and simpler. Please make sure you have the wheel package installed.

```
$ pip install wheel
```

In case you explicitly want to compile from source (for instance to have platform-dependent optimizations), use the --no-binary flag.

```
$ pip install --no-binary :all: <package>
```

Changing package installation path

If you have admin rights, pip typically installs all packages to a system-wide location. With the additional --user option the packages are installed into the user's home (eg ~/.local/lib/ on Unix). This allows users without admin rights to install Pyhton packges.

In rare cases you may want to install a specific package and "bundle" it with a project's code by installing it into a vendor/ sub-directory.

```
$ pip install SomePackage -t vendor
```

The package can then be imported by prepending vendor/ to the sys.path.

```
import sys
sys.path.insert(0, "vendor")
import SomePackage
```

See also:

In general we recommend to use project-specific *virtual environments* whenever possible.

Working with requirements.txt

It's common practice to include a requirements.txt file in a Python project, that specifies the list of required packages and, if needed, the compatible versions thereof. The motivation is to ensure full portability and reproducibility of the code, as every user can install exactly the same packages.

```
some_package
other_package==1.0
```

The following pip command installs all packages listed in the requirements.txt file.

```
$ pip install -r requirements.txt
```

Conversely you can use pip freeze to generate a requirements.txt file from the currently installed packages.

```
$ pip freeze > requirements.txt
```

Upgrading packages

Any given package can be upgraded to the latest version using the --upgrade (or -U) switch.

```
$ pip install --upgrade <package>
```

See also:

Please read about other *tools around pip and venv* to manage project dependencies or upgrade all installed packages.

Other helpful commands

```
pip list              # show list of installed packages and their versions
pip list --outdated   # show list of packages with available upgrades
pip show <package>    # show info about installed package
pip show -f <package> # show info and include list of files
pip uninstall <pkg>   # uninstall given package
pip cache info        # show info and size of pip cache folder
pip cache purge       # delete pip cache folder
pip --no-cache-dir <cmd>    # run pip command without using any cache
```

5.1.3 Virtual Environments

Virtual environments allow to completely isolate one set of installed Python packages from another. This is a crucial ingredient when working on projects with mutually incompatible package version requirements. They have various advantages that make them indispensable:

- install specific package versions for each project
- isolate one project from another and the globally installed packages
- no admin rights required to install packages
- possibility to test an application against several different environments
- play with latest version of a package without affecting other projects

On the downside:

- the size of a virtual environment folder can quickly exceed hundred megabytes
- the packages of each virtual environment need to be upgraded individually
- the virtual environment may have to be recreated when switching platforms or after system upgrades

We recommend the use of virtual environments for most projects, as they provide full control over the package installation.

Index

venv

Python includes (since version 3.3 released in 2012) the venv module to provide lightweight support for virtual environments, without the need for any third-party tools.

Usage

When inside the folder of your Python project, the following command creates a new virtual environment

```
$ python -m venv .venv
```

where .venv is the conventional name for the virtual environment and the prepended dot makes the associated folder hidden on Unix systems.

Additional options allow to control some aspects during venv creation. See python -m venv --help for details. In particular you can grant your venv access to the system-wide packages. This can be useful on HPC clusters, where the number-crunching libraries have been optimized for the platform and should not simply be re-installed with default options inside the virtual environment.

```
$ python -m venv --system-site-packages .venv
```

Tip: When using a version control system like git, always make sure to ignore the .venv folder by adding it to your project's .gitignore file. The virtual environment can quickly grow to over 100MB and is platform-dependent. It must therefore be excluded from the git history.

Once created, the virtual environment needs to be activated.

Unix

```
$ source .venv/bin/activate
```

Windows

```
C:\> .\venv\Scripts\Activate.ps1
```

Afterwards the pip and python commands should refer to the binaries inside the virtual environment.

```
$ which pip
/path/to/your/project/.venv/bin/pip
```

In particular pip will install packages locally into the virtual environment.

When done, you can either close your shell or deactivate the virtual environment

```
$ deactivate
```

Behind the scenes

Let's take a closer look at what is happening behind the scenes when creating and activating a virtual environment.

Contents of .venv

We use the tree command to list the (first three levels of) contents of the .venv/ directory.

```
$ tree -L 3 .venv/
.venv/
├── bin/                  # directory with binaries and executable scripts
│   ├── Activate.ps1        # scripts to activate venv for different shells
│   ├── activate
│   ├── activate.csh
│   ├── activate.fish
│   ├── pip*              # local pip wrapper scripts
│   ├── pip3*
│   ├── pip3.11*
│   ├── python -> python3.11*
│   ├── python3 -> python3.11*
│   └── python3.11 -> /path/to/your/python3.11*  # symlink to your Python interpreter
├── include/              # directory to contain C header files
├── lib/
│   └── python3.11/
│       └── site-packages/  # directory to contain local Python packages
└── pyvenv.cfg            # config file
```

Activation of .venv

Feel free to inspect the code of .venv/bin/activate to see what exactly is happening when activating a virtual environment. We highlight the main points:

1. the VIRTUAL_ENV shell environment variable is defined with the absolute path to the .venv directory. This variable can be used to check if one is running inside a virtual environment and display its name in the shell prompt.
2. the .venv/bin folder is prepended to your PATH so that all executables inside take precedence. This ensures that running a pip command actually invokes the pip from inside the virtual environment, without having to give the full path .venv/bin/pip. Similarly for the python interpreter itself and any executables installed by the Python packages inside the virtual environment.

Note that, given the activation is merely changing your shell's PATH, you don't really *need* to first activate a virtual environment. Alternatively you may call the interpreter or scripts directly using their full .venv/bin/ path.

Conversely, when calling deactivate all changes are reverted by resetting the shell environment variables to their previous state.

virtualenv

Instead of the built-in venv module, virtual environments can also be managed with the PyPI package virtualenv. It offers some additional features and is for instance used by pipenv and poetry.

Usage

The syntax to manage virtual environments is similar to venv.

```
$ virtualenv .venv           # create new virtual environment .venv
$ source .venv/bin/activate  # activate the virtual environment .venv
$ deactivate                 # deactivate the current virtual environment
```

When juggling multiple Python versions, you can specify the python executable for the virtual environment.

```
$ virtualenv -p python3.11 .venv    # create .venv for the given python interpreter
```

5.1.4 Tools around pip and venv

The pip and venv discussed so far are the underpinning of Python package management. Often that's all you need. But sometimes you may want another layer of tools to further facilitate some aspects. On one hand, when working on a given project, the development and dependency management can be simplified with *pipenv* or *poetry*. On the other hand, when working with packages shared across projects, *pip-review* and *pipx* may come in handy.

Related projects

As a matter of fact, besides pipenv and poetry, exist a variety of Python package installers and dependency managers. On the one hand it's nice to be offered a choice, on the other hand the community could also profit from some convergence and mutual agreement on a common tool. The following list is sorted by GitHub stars:

- pip-tools: provides pip-compile to generate a requirements.txt with versioned dependencies, which can then be parsed by pip-sync to install those packages.
- pdm: the "Python dependency manager" installs packages and creates a pdm.lock file.
- hatch: a Python project manager that helps to build and publish your code.

Index

pipenv

The pipenv project aims to combine pip and virtualenv into one single toolchain. It is currently the recommended packaging tool of the Python Packaging Authority, while slowly being surpassed in popularity by *poetry*.

It introduces two new files to your project: Pipfile and Pipfile.lock. The first is similar to requirements.txt and tracks all packages that you explicitly installed. The latter is an extended version of the pip freeze output and precisely tracks the versions of all installed dependencies. This file is the key to recreating exactly the same environment on a different machine for full reproducibility. While you can still fine-tune the Pipfile by hand, both files can be fully managed by pipenv commands.

Usage

If you are working on a project with an existing Pipfile.lock, one single command is sufficient to create a virtual environment and install all listed dependencies.

```
$ pipenv sync
```

Tip: By default pipenv creates all virtual environments in the common ~/.local/share/virtualenvs/ directory. We recommend to set the environment variable PIPENV_VENV_IN_PROJECT=1 to instead use the .venv directory inside the project's root.

On a new project, simply start by installing the required packages with pipenv, thereby initializing the virtual environment and Pipfile[.lock] files.

```
$ pipenv install <package>
```

Tip: Always add Pipfile and Pipfile.lock to your version-control system. Not only does it help your collaborators to use the same package versions, but it also tracks all package updates in the git history.

Note: Some packages may fail to install on macOS, because the system is reporting a version number >10, but they expect 10.x. Even though this should be patched in the upstream versions, you may still encounter the bug in rare cases. A simple fix is to configure the system to report a compatible version number before installing the problematic packages.

```
$ export SYSTEM_VERSION_COMPAT=1
```

To run the code of your project you can, as usual, activate the virtual environment, or use pipenv helper commands.

```
$ pipenv shell           # spawn shell with activated virtual environment
$ pipenv run <command>   # run given command inside the virtual environment
```

Development dependencies

In addition to the standard package dependencies, one can define a list of dependencies useful when developing on the project. This could include formatters and linters that are important when writing code, but are not required in production.

```
$ pip install --dev <package>       # install package as development dependency
```

Similarly, use pipenv sync --dev to include dev packages when re-creating the virtual environment from an existing Pipfile.lock.

Upgrading dependencies

With the pipenv update subcommand it's easy to list outdated packages and install all available updates.

```
$ pipenv update [--dev] --outdated  # list outdated packages with available updates
$ pipenv update [--dev]             # install all available updates
```

Other commands

```
$ pipenv graph          # show graph of package dependencies
$ pipenv requirements   # generate a requirements.txt file with the package dependencies
$ pipenv clean          # uninstall all packages not specified in Pipfile.lock
$ pipenv --rm           # delete virtual environment
```

poetry

The poetry project is yet another tool around Python packaging and dependency management. It is gaining in popularity and probably the best option if you also plan to publish your code to PyPI.

It uses a pyproject.toml file (as motivated by PEP 518 to store the project's metadata and dependencies. An additional poetry.lock file keeps track of the precise package versions that are installed.

Tip: Always add pyproject.toml and poetry.lock to your version-control system. Not only does it help your collaborators to use the same package versions, but it also tracks all package updates in the git history.

Usage

The best way to get started with poetry in an existing project, is to let it interactively create the pyproject.toml file by prompting you to input the relevant data.

```
$ poetry init
```

Then install all specified dependencies inside a virtual environment and create the corresponding poetry.lock file.

```
$ poetry install
```

In contrast to pipenv it will install the development dependencies by default, unless the --no-dev option is passed. In addition you can pass --no-root to skip the installation of the project's package itself.

Tip: We recommend to instruct poetry to create the virtual environments inside the project's .venv directory, instead of the shared virtualenvs.path.

```
poetry config virtualenvs.in-project true
```

You may later add other package requirements to the list.

```
$ poetry add <package>
```

To run the code of your project you can, as usual, activate the virtual environment, or use poetry helper commands.

```
$ poetry shell           # spawn shell with activated virtual environment
$ poetry run <command>   # run given command inside the virtual environment
```

Upgrading dependencies

```
$ poetry update --dry-run   # list outdated packages
$ poetry update             # install available package updates
```

Other commands

```
$ poetry show [--tree]                # list all packages [in a tree structure]
$ poetry install --remove-untracked   # uninstall all packages not specified in poetry.lock
$ poetry config --list                # show current configuration settings
```

pip-review

The pip-review tool is a small wrapper around pip that provides an easy way to update all installed packages.

Motivation

When installing packages directly with pip, you can use `pip list --outdated` to view the packages with available updates. However there's no built-in command to upgrade all outdated versions. Instead of lengthy bash commands, we suggest to make use of a third-party tool.

Usage

When called without additional arguments, pip-review delegates to `pip list --outdated` and returns the list of packages with available updates.

```
$ pip-review
some-package-name==2.0 is available (you have 1.0)
```

When called with the -a option it will install all updates.

```
$ pip-review -a
```

In the background, the list of all outdated packages will be passed to a single call of `pip install --upgrade`, thereby attempting to upgrade all packages simultaneously. Occasionally it happens that the dependency resolver of pip bails out with an error message because the requirements of some updates are mutually exclusive. You can then use the interactive switch to manually select the list of packages to upgrade.

```
$ pip-review -i
some-package-name==2.0 is available (you have 1.0)
Upgrade now? [Y]es, [N]o, [A]ll, [Q]uit y
```

Related projects

- pipupgrade: similar to pip-review, but less minimal.

pipx

The pipx tool allows to install and run Python command-line applications with pip in isolated virtual environments. As such it fulfils a rather precise goal and should not be confused with the general-purpose installations of pip and development tools like pipenv or poetry.

Motivation

Besides the Python packages that install some libraries to be used inside your Python code, there are others that install command-line tools. Noteworthy examples are black and flake8, but also projects like ansible (to automate the configuration of a computer) or cookiecutter (to generate a project's file structure from a template) . These tools are often used outside of a given Python programming project and its dependency management system.

The naive approach to make them available globally across projects is to install them with pip. But if multiple packages are installed, you may run into issues of dependencies with mutually incompatible versions. Similarly to our previous discussion of virtual environments, a possible solution is to install each application into a dedicated venv. This is exactly what pipx does.

Usage

The following command installs a given package with pipx.

```
$ pipx install <package>
```

Behind the scenes pipx creates a folder structure inside ~/.local/pipx/ to contain the virtual environments for each installed application. The executables are linked into ~/.local/bin/. To make the tools available in the shell, add this directory manually to your $PATH or issue the pipx ensurepath helper command.

The already installed packages and associated command-line tools can be listed.

```
$ pipx list
venvs are in ~/.local/pipx/venvs
apps are exposed on your $PATH at ~/.local/bin
   package black 23.3.0, Python 3.11.2
    - black
    - black-primer
    - blackd
```

Other commands include

```
$ pipx uninstall <package>      # remove an installed package
$ pipx upgrade <package>        # upgrade an installed package
$ pipx upgrade-all              # upgrade all installed packages
$ pipx reinstall-all            # reinstall all packages (eg to fix issues after python upgrades)
$ pipx inject <package> <pkg>   # install additional pkg into existing venv of installed package
```

5.1.5 Best Practices

Python ecosystem

- *Manage your Python interpreter*
  - know that there is no single Python
  - know which Python you are using
- *Manage your Python packages*
  - know the names (and required versions) of your packages
  - know how to install them
  - share the package requirements with collaborators
  - ensure portability and reproducibility of your code
- *Use virtual environments*
  - isolate and freeze your dependencies
  - know precisely which package versions your code is using
  - keep updating the packages whenever possible

General tips

- Communicate with collaborators and peers
  - learn about their code and tools
  - when in doubt, use the same tools and libraries as your colleagues
- Write documentation for others and your future self
- *Write automation scripts*
- Version control all your code
  - including (converted) Jupyter notebooks
  - excluding sensitive date (like credentials or api tokens)
  - excluding all data in general (exceeding a couple of MB)
  - project-specific .gitignore (to complement the global core.excludesFile)

```
__pycache__
.venv
.ipynb_checkpoints
```

5.1.6 Noteworthy Packages

Non-exhaustive list of popular packages useful to scientists.

Scientific

- NumPy: multi-dimensional arrays and various numerical computing tools.
- SciPy: routines for numerical integration, interpolation, linear algebra etc.
- SymPy: symbolic mathematics.
- pandas: powerful data analysis with a 3500+ pages manual.
- matplotlib, seaborn, bokeh: plotting and data visualization.
- scikit-learn, TensorFlow, Keras, PyTorch: machine learning and deep learning.
- numba, dask: higher performance computing.

Not directly scientific

- sphinx: documentation generator (incl LaTeX export).
  - MyST: Markedly Structured Text is a Markdown flavor with extended features.
  - jupyterbook: create books and other documents from Jupyter notebooks.
- requests, httpx: make HTTP and API calls.
- click: command-line interface creation kit.
- jinja: templating engine to generate files from data, code and a template.
- cookiecutter: create projects and file structures from templates.
- flask, django: web frameworks to create APIs or web pages.
- SQLAlchemy: database toolkit and object relational mapper (ORM).
- pillow: “Python Imaging Library” for image processing.
- lxml, scrapy, beautifulsoup: web scraper and parser.
- pytest: testing framework.

CHAPTER

SIX

PYTHON ECOSYSTEM (II)

This module presents **tools for writing Python code**.

6.1 Index

6.1.1 Editors, IDE and REPL

Pretty much any editor can be used to write Python code. But not all offer the same features when it comes to actively developing and debugging code. For some situations, a minimal text editor is just fine. For quick debugging and testing an interactive Python shell, or read-eval-print-loop (REPL), can be irreplaceable. Other circumstances call for a tight integration of text and images, or even a full-fledged Integrated Development Environment (IDE).

Tip: As a general piece of advice, we recommend anyone to invest some time getting familiar with their editor of choice. Getting to know the features and keyboard shortcuts can boost your productivity. Ask your colleagues which editor they prefer and why. Take the opportunity to look over their shoulder to learn new tricks.

Index

ipython

Motivation

The classical development cycle is sometimes caricatured as Edit-Run-Debug loop. Some debugging tasks are greatly simplified if one can quickly run code snippets and immediately see the results. This is the starting point for “interactive programming” that enables exploratory coding. The Python interpreter offers a built-in Read-Eval-Print-Loop (REPL) shell.

```
$ python
Python 3.11.2
>>> 1+1
2
>>> exit()
```

However the default shell has limited functionality. Therefore, back in 2001, a PhD student in particle physics started developing Interactive Python (IPython). The same project later evolved into ipython notebook and finally *Jupyter*.

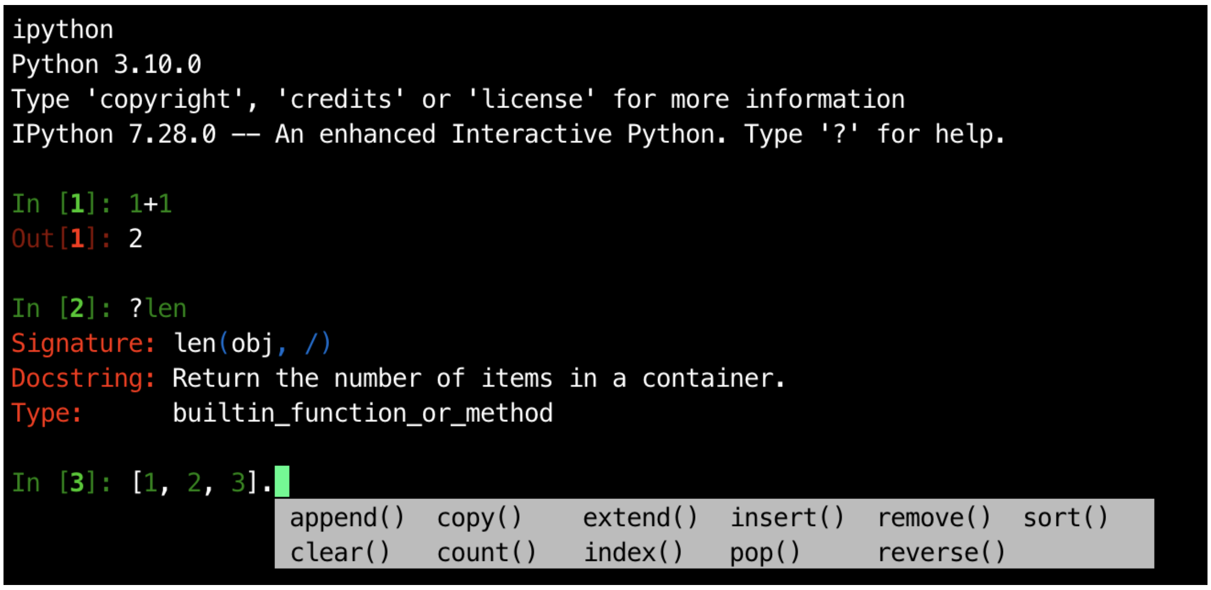
Usage

Interactive shell

The IPython shell offers many features

- object introspection
- multi-line editing and tab completion
- numbered input/output prompts with command history
- rich output of data structures with syntax highlighting
- magic commands and access to shell commands

Invoke the ipython terminal command to get started.



```
ipython
Python 3.10.0
Type 'copyright', 'credits' or 'license' for more information
IPython 7.28.0 -- An enhanced Interactive Python. Type '?' for help.

In [1]: 1+1
Out[1]: 2

In [2]: ?len
Signature: len(obj, /)
Docstring: Return the number of items in a container.
Type:      builtin_function_or_method

In [3]: [1, 2, 3].
                       append()  copy()   extend()  insert()  remove()  sort()
                       clear()   count()  index()   pop()     reverse()
```

Magic commands

IPython has so-called magic commands to enhance the interactive shell with additional features. Line magics are prefixed with a single %, cell magics with %% and commands starting with ! are shell magics. These commands also work inside Jupyter notebooks.

```
%quickref       # show quick reference of magic commands
%magic          # general information about magic commands

%aimport pkg    # import package and automatically reload on code changes

%pinfo obj      # print info about object. Same as obj? or ?obj
%pinfo2 obj     # print more info about object. Same as obj?? or ??obj
%pdoc obj       # print docstring of object
%psource obj    # print source code of object definition
%pfile obj      # print file containing object definition

%run file.py    # execute code from file
%prun file.py   # execute code from file with profiler
%mprun file.py  # execute code from file with memory profiler
%lprun file.py  # execute code from file with line profiler
```

(continues on next page)

(continued from previous page)

```
%%timeit        # time code execution of current cell
%%bash          # execute cell contents as bash code (or %%html, %%javascript, etc)

%cd ~/          # change directory
!ls             # execute shell command
files = !ls     # execute shell command and capture output in Python variable
```

Attach interactive debugger

Another useful feature is to start an embedded IPython shell directly from your code. This provides access to all the project's variables for live inspection and debugging, similar to setting a breakpoint in the IDE.

```
import IPython

a = 1
IPython.embed()
```

jupyter

Project jupyter is an *IPython* spin-off that provides a web frontend to do the coding inside your browser. It is not specific to Python but also supports kernels in other programming languages. The most notable feature is that code, markdown, equations and inline images can all be combined in the same document. The native file extension of Jupyter notebooks is `.ipynb` and all data is represented as JSON. Similar to IPython, Jupyter uses cells for code evaluation. This interactive approach simplifies coding, as it's easy to incrementally add computational steps and immediately see the results. The Jupyter notebook format is especially popular in data science where it enables exploratory analysis. On the downside users need to pay attention to the execution order of the cells in order not to lose reproducibility.

Installation

JupyterLab desktop app

JupyterLab can be installed as cross-platform, standalone desktop application. It is self-contained and bundles its own Python environment with selected packages. Please refer to their documentation on how to change this environment or install additional packages.

Package installation

JupyterLab can also be installed as a normal PyPi package with `pip install jupyterlab`. If you prefer the classical Jupyter Notebook interface, install it with `pip install notebook`.

Online playground

You can try Jupyter out in your browser, without installing anything.

JupyterLite

The JupyterLite project allows to run Jupyter directly in a web browser, without even having to install Python. Of course this is only meant as playground and not for computationally intensive code. With the JupyterLite Shinx extension you can easily embed interactive code environments directly inside your documentation.

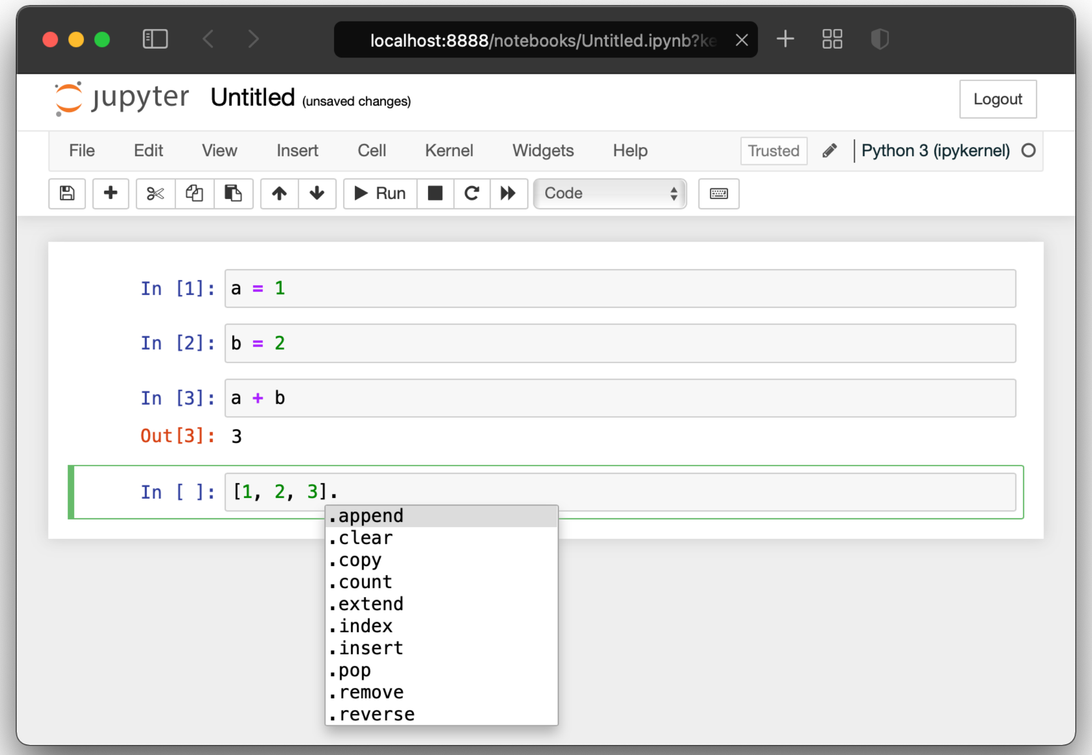
Usage

Use the jupyterlab or jupyter notebook commands to launch the server in your terminal.

```
$ jupyter notebook
[I 12:53:38.084 NotebookApp] Serving notebooks from local directory: /Users/johndoe/Desktop/jupyter
[I 12:53:38.084 NotebookApp] Jupyter Notebook 6.4.5 is running at:
[I 12:53:38.084 NotebookApp] http://localhost:8888/?token=454d32b4236c6e70836582f0f269ddf4ffed4251688e161a
[I 12:53:38.084 NotebookApp]  or http://127.0.0.1:8888/?token=454d32b4236c6e70836582f0f269ddf4ffed4251688e161a
[I 12:53:38.084 NotebookApp] Use Control-C to stop this server and shut down all kernels (twice to skip confirmation).
[C 12:53:38.094 NotebookApp]

    To access the notebook, open this file in a browser:
        file:///Users/johndoe /Library/Jupyter/runtime/nbserver-23775-open.html
    Or copy and paste one of these URLs:
        http://localhost:8888/?token=454d32b4236c6e70836582f0f269ddf4ffed4251688e161a
     or http://127.0.0.1:8888/?token=454d32b4236c6e70836582f0f269ddf4ffed4251688e161a
```

It should automatically open a web browser with the Jupyter dashboard (unless you added the --no-browser option). Next use the *New* button on the top right to create a notebook with the Python kernel.



```
In [1]: a = 1

In [2]: b = 2

In [3]: a + b
Out[3]: 3

In [ ]: [1, 2, 3].
           .append
           .clear
           .copy
           .count
           .extend
           .index
           .insert
           .pop
           .remove
           .reverse
```

Refer to the documentation on how to use Jupyter on the ETH HPC clusters Euler and Leonhard.

Danger: Do not configure your Jupyter server to be publicly reachable from the internet. Without SSL encryption the password or security token will be transmitted cleartext over the network. Any eavesdropper can connect to your Jupyter and have unrestricted access to your user account and computer. The Jupyter server should only listen on localhost. Use ssh to open a browser on the remote host or connect with an ssh tunnel. We recommend to use normal Python scripts for long-running computations.

Keyboard shortcuts

Productivity can be boosted by learning the modal keyboard shortcuts to navigate and edit cells.

Edit mode:

```
Shift-Enter     # run cell
Tab-Tab         # code auto-completion
Esc / Ctrl-m    # switch to control mode
```

Control mode:

```
h               # view help with all keyboard shortcuts
p               # open command palette
m               # convert cell to markdown
y               # convert cell to code
dd              # delete cell
a / b           # insert new cell above/below
j / k           # move cell down/up
c / v           # copy/paste cell
n / p           # select next/previous cell
Enter / Click   # switch to edit mode
```

The built-in help contains many more keyboard shortcuts, which are different depending on the operating system, and therefore not listed above.

Notebooks vs Scripts

As amazing as Jupyter can be for exploratory data analysis, its JSON file format does not play well with version control and the results are only reproducible if all cells were run in the correct order. Scientists should consider using Jupyter for exploration, but switching over to Python scripts for the final computations. The scripts can trivially be run on remote machines and their changes tracked in a version control system. The *VSCode* editor offers a nice compromise between the interactive programming features of Jupyter and classical scripts. But you can also rely on command-line tools to convert between different formats.

Jupyter includes the nbconvert tool to convert between .ipynb and various formats (Html, LaTeX, PDF, Markdown, Python script, etc).

```
$ jupyter nbconvert --to python my_notebook.ipynb
```

A powerful third-party tool is jupytext, as used by VSCode under the hood. Notebooks can even be paired and synced with their corresponding script text files.

```
$ jupytext --to py:percent my_notebook.ipynb
```

Another approach is to use nbdev and install its git hooks for a more git-friendly Jupyter experience.

VSCode

The Visual Studio Code editor has become a very popular IDE. Its main features include

- open-source and cross-platform
- sweet spot between lightweight editor and powerful IDE
- customizable via settings and extensions
- syntax highlighting and auto-completion of many programming languages
- code launcher, debugger (breakpoints, variable inspection) and tasks runner
- integration with terminal, version control, GitHub and remote development
- preview rendering of markdown
- command-line tool code

See also:

There are of course many alternatives to VSCode. From command-line editors like Vim with python-mode, over spyder, to the commercial PyCharm (for which ETH students can receive a free pro license.

Customization

Extensions

Thousands of extensions provide additional features, key mappings and themes.

Settings

The looks and behaviour of VSCode are heavily customizable. The settings can be made globally for all projects, but also defined and overridden for each workspace. The workspace-specific preferences are stored inside the project's .vscode/settings.json. This JSON file can be edited directly or via the GUI. If you include it in your version control, all collaborators can share the same configuration.

The following snippet shows a selection of noteworthy settings.

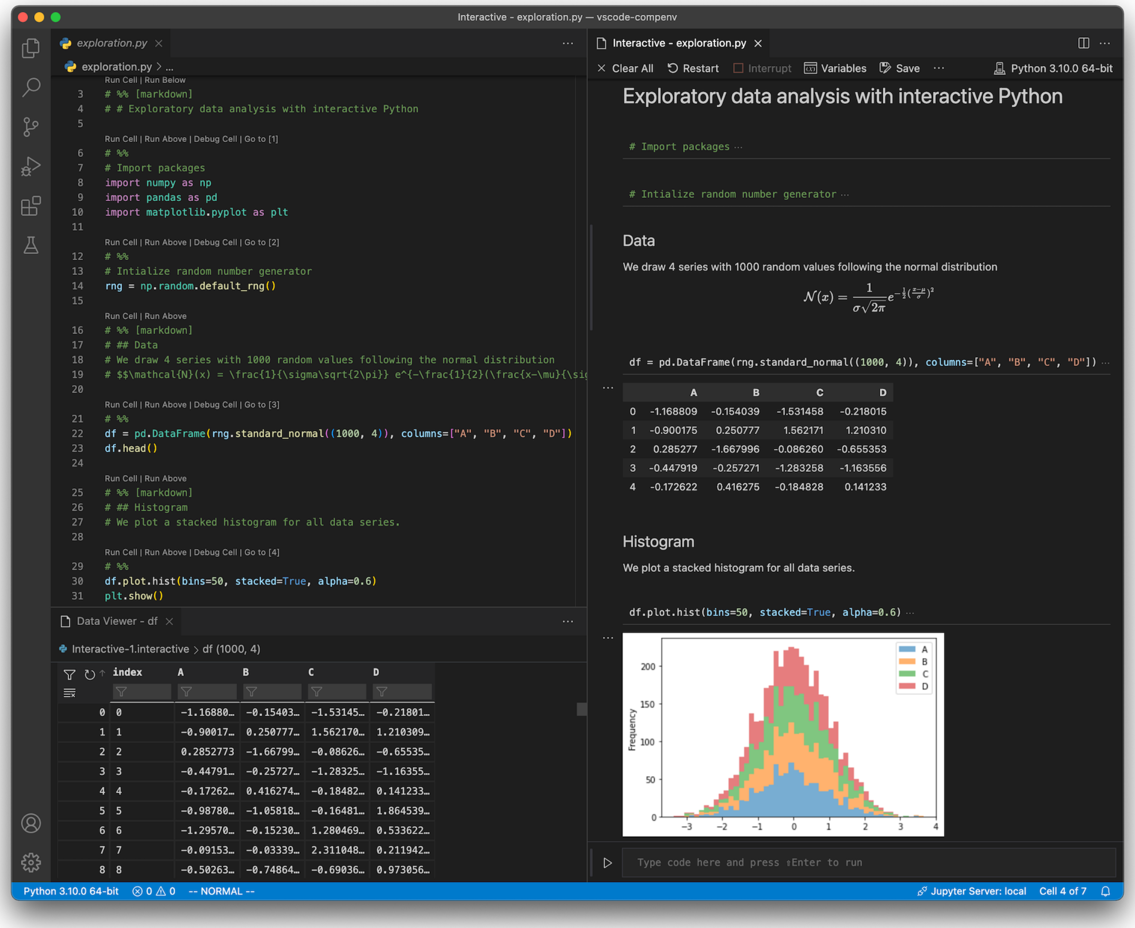
```
{
   "editor.bracketPairColorization.enabled": true,
   "files.insertFinalNewline": true,
   "files.trimTrailingWhitespace": true,
   "files.exclude": {
      "**/__pycache__": true,
      "**/.venv": true,
   },
   "jupyter.askForKernelRestart": false,
   "[python]": {
      "editor.defaultFormatter": "ms-python.python",
      "editor.formatOnSave": true,
   },
   "python.formatting.provider": "black",
   "python.formatting.blackArgs": ["--line-length=100"],
   "python.linting.flake8Enabled": true,
   "python.linting.flake8Args": ["--max-line-length=100"],
   "python.defaultInterpreterPath": ".venv/bin/python3",
   "telemetry.telemetryLevel": "off",
}
```

Tip: By default, VSCode collects telemetry data, including crash reports and usage data. For privacy reasons you should consider disabling this automatic data reporting by setting the `telemetry.telemetryLevel` user setting to off.

Python and Jupyter

The Python extension is indispensable for Python development and includes the closed-source Pylance language server as dependency. There is also the Jupyter extension. The official documentation and tutorials offer plenty of resources to explore Python and Jupyter in VSCode.

- Getting started with Python in VSCode
- Integrate linters and formatters
- Debugging and running Python code
- Data Science in VSCode and Jupyter notebooks
- And last, but not least, our favorite interactive evaluation of Python scripts



```
Run Cell | Run Below
# %% [markdown]
# # Exploratory data analysis with interactive Python

Run Cell | Run Above | Debug Cell | Go to [1]
# %%
# Import packages
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

Run Cell | Run Above | Debug Cell | Go to [2]
# %%
# Intialize random number generator
rng = np.random.default_rng()

Run Cell | Run Above
# %% [markdown]
# ## Data
# We draw 4 series with 1000 random values following the normal distribution
# $$\mathcal{N}(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}(\frac{x-\mu}{\sig

Run Cell | Run Above | Debug Cell | Go to [3]
# %%
df = pd.DataFrame(rng.standard_normal((1000, 4)), columns=["A", "B", "C", "D"])
df.head()

Run Cell | Run Above
# %% [markdown]
# ## Histogram
# We plot a stacked histogram for all data series.

Run Cell | Run Above | Debug Cell | Go to [4]
# %%
df.plot.hist(bins=50, stacked=True, alpha=0.6)
plt.show()
```

	index	A	B	C	D
0	0	-1.16880…	-0.15403…	-1.53145…	-0.21801…
1	1	-0.90017…	0.250777…	1.562170…	1.210309…
2	2	0.2852773	-1.66799…	-0.08626…	-0.65535…
3	3	-0.44791…	-0.25727…	-1.28325…	-1.16355…
4	4	-0.17262…	0.416274…	-0.18482…	0.141233…
5	5	-0.98780…	-1.05818…	-0.16481…	1.864539…
6	6	-1.29570…	-0.15230…	1.280469…	0.533622…
7	7	-0.09153…	-0.03339…	2.311048…	0.211942…
8	8	-0.50263…	-0.74864…	-0.69036…	0.973056…

Exploratory data analysis with interactive Python

```
# Import packages ···
```

```
# Intialize random number generator ···
```

Data

We draw 4 series with 1000 random values following the normal distribution

$$\mathcal{N}(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}(\frac{x-\mu}{\sigma})^2}$$

```
df = pd.DataFrame(rng.standard_normal((1000, 4)), columns=["A", "B", "C", "D"]) ···
```

	A	B	C	D
0	-1.168809	-0.154039	-1.531458	-0.218015
1	-0.900175	0.250777	1.562171	1.210310
2	0.285277	-1.667996	-0.086260	-0.655353
3	-0.447919	-0.257271	-1.283258	-1.163556
4	-0.172622	0.416275	-0.184828	0.141233

Histogram

We plot a stacked histogram for all data series.

```
df.plot.hist(bins=50, stacked=True, alpha=0.6) ···
```

```
Type code here and press ⇧Enter to run
```

6.1.2 Formatter and Linter

Having discussed where to write code, we now turn to what the code should look like. Code formatters like *black* automatically rewrite your code to adhere to certain style guidelines. Linters like *flake8*,*pydocstyle* or *ruff* perform static code analysis and check for potential shortcomings or stylistic errors.

Index

black

Black is the most widely used Python code formatter and backed by the Python Software Foundation.

Motivation

Not only can the same problem be solved by different code implementations, but the same code can also be *formatted* differently. Even though the following snippet

```
a=  [ 1,2
,3]
print(
a  )
```

is perfectly valid to the Python interpreter, adapting the white space and newlines

```
a = [1, 2, 3]
print(a)
```

makes it much more readable to the human mind. Granted that this example is a bit extreme, but the details of where to put commas, parentheses, whitespace and newlines are often a matter of *taste*, or even better, *convention*. The Python community introduced PEP 8 as a "Style Guide for Python Code". Adhering to a common coding style facilitates writing new code, as well as reading existing code.

Using a code formatter like black goes even a step further, as it will automatically format the code for you. No matter what the code looks like at first, the formatter will always make it look exactly the same and adhere to the Black code style.

> By using Black, you agree to cede control over minutiae of hand-formatting. In return, Black gives you speed, determinism, and freedom [. . .] You will save time and mental energy for more important matters.
>
> [. . .] Blackened code looks the same regardless of the project you're reading. Formatting becomes transparent after a while and you can focus on the content instead.
>
> [source]

Usage

Python source code can be formatted by invoking black on a single file

```
$ black some_file.py
```

or a whole folder, for instance the current directory

```
$ black .
```

On some projects you may need to increase the maximum line length (from its default of 88 characters)

```
$ black -l 100 .
```

Please refer to the official documentation for all usage and configuration options.

Instead of the manual invocation on the command line, the code formatter can also be integrated in various editors to automatically format the file on each save, or as a hook in your git repository. For instance in VSCode you can adapt your project's settings.json to include

```
"python.formatting.provider": "black",
"python.formatting.blackArgs": [
        "--line-length=100"
],
"[python]": {
        "editor.formatOnSave": true
},
```

Related projects

- blacken-docs runs black on code blocks in documentation files, like markdown.

flake8

Flake8 is one of the many projects backed by the Python Code Quality Authority. What makes flake8 particularly useful is that it glues together several tools:

- pycodestyle to check against PEP 8 style conventions
- pyflakes to check for syntax errors and unused variables or imports
- mccabe to check the cyclomatic complexity of the code

Especially when tightly integrated with the editor, linters can offer easy rewards for low effort. They can quickly identify syntax errors and help to avoid common bugs and pitfalls.

Usage

Use the flake8 command to check a given file

```
$ flake8 some_file.py
$ flake8 --max-line-length=100 some_file.py
```

When checking a whole folder you often need to exclude the virtual environment

```
$ flake8 --exclude .venv .
```

There are different ways to ignore certain violations. Either globally by adding the error code to the ignore list

```
$ flake8 --ignore=E402 some_file.py
```

or locally in the code using a noqa, short for "no quality assurance", comment

```
import os  # noqa: E402
```

If you want to use flake8 with VSCode, add the following lines to your settings.json:

```
"python.linting.flake8Enabled": true,
"python.linting.flake8Args": [
        "--max-line-length=100"
],
```

Related projects

- pylint is another Python linter.
- isort sorts and formats import statements.

pydocstyle

Yet another ingredient to improve the quality of your code is to have a good inline documentation of what each class or function is doing. In particular PEP 257 outlines the best practices around using docstrings. Pydocstyle is a style checker for docstrings. Not only does it help you to get the formatting right, but it also warns of missing docstrings and encourages you to document the key components of your code.

For instance, given the simple function

```
def square(x):
    """ computes the second power of x """
    return x**2
```

pydocstyle would complain

```
D210: No whitespaces allowed surrounding docstring text
D400: First line should end with a period (not 'x')
D401: First line should be in imperative mood (perhaps 'Compute', not 'computes')
D403: First word of the first line should be properly capitalized ('Computes', not 'computes')
```

and make us change the docstring to

```
def square(x):
    """Compute the second power of x."""
    return x**2
```

which is not only more consistent with other libraries but also correctly formatted for IDEs that show square.__doc__ as tooltip.

Usage

Use the pydocstyle command to check the docstring style of a given file:

```
$ pydocstyle some_file.py
```

At the beginning it may help to increase the verbosity of the output by also including a snippet of the source code and a detailed explanation for each error:

```
$ pydocstyle --source --explain some_file.py
```

The tool can be nicely integrated into VSCode with the following settings.json snippet:

```
"python.linting.pydocstyleEnabled": true,
```

ruff

Ruff is a Python linter implemented in the Rust programming language. Even though it is a relatively young project, it has become popular because it's very fast and incorporates over 600 linting rules. It can therefore be used as full replacement for several other tools, including *flake8* and *pydocstyle*.

Usage

Use the ruff check command to lint a given file or project directory.

```
$ ruff check some_file.py
$ ruff check .
```

Use the pyproject.toml settings to customize the linting rules.

```
[tool.ruff]
line-length = 100
select = ["E", "F", "D"]  # include pydocstyle "D" rules
ignore = ["D107", "D301"]

[tool.ruff.pydocstyle]
convention = "pep257"
```

6.1.3 String Formatting

At the end of the day, irrespective of what your Python code was doing in the middle, it typically finishes by outputting some result. Given the omnipresence of printing strings, we want to briefly discuss how to control the formatting and representation of strings, numbers and dates.

Index

Formatted String Literals

String interpolation

The process where a placeholder in a string is replaced by the contents of a variable is known as *string interpolation* or *variable interpolation*.

At long last it's time to let the obligatory "hello world" program enter the stage and show some of its multiple faces.

```
name = "world"
print("Hello " + name)  # prints "Hello world"
```

Instead of using the + operator for the string concatenation, one can also call print() with multiple arguments, which will be separated with a whitespace.

```
name = "world"
print("Hello", name)    # prints "Hello world"
```

Python still supports the % format specifier for string interpolation, although this printf-style string formatting is no longer recommended.

```
name = "world"
print("Hello %s" % (name))  # no longer recommended in Python 3+
```

Python 3 introduced the str.format() method for string formatting. The {} placeholder will be replaced with the contents of the format argument.

```
name = "world"
print("Hello {}".format(name))      # prints "Hello world"
print("Hello {n}".format(n=name))  # prints "Hello world" using a named replacement field
```

Python 3.6 announced the support of the so-called *f-strings* or *formatted string literals*, bringing the currently recommended syntax for string formatting and variable interpolation.

```
name = "world"
print(f"Hello {name}")  # prints "Hello world"
```

f-string expressions

Note: The following examples use pi, which is understood to have been imported with from math import pi.

The canonical usecase of f-strings is basic string interpolation,

```
>>> print(f"pi = {pi}")
pi = 3.141592653589793
```

or evaluation of expressions.

```
>>> print(f"pi/2 = {pi/2}")
pi/2 = 1.5707963267948966
```

Python 3.8+ even supports the following shorthand notation to print the name of a variable and its contents.

```
>>> print(f"{pi/2 = }")
pi/2 = 1.5707963267948966
```

Sometimes one has to work with long strings that do not fit on a single line of code. As Python ignores line breaks inside pairs of brackets, the following is a common way to split a long string across multiple lines.

```
multiline_string = (
    f"Lorem ipsum dolor sit amet, "
    f"consectetur adipiscing elit, "
    f"sed do eiusmod tempor incididunt ut labore et dolore magna aliqua."
)
```

Note how each line is a separate f-string and that all whitespace must be added explicitly to the string.

Format Specifications

Often it's not sufficient to simply print the contents of a variable. In addition one has to be able to control how the values are being presented, for instance how many digits of a decimal number should be visible. The format specification mini-language provides a powerful and yet flexible handle on string formatting.

Note: We use f-strings in all examples, but the format specifiers also work with the format() string method.

Formatting of strings

Strings can be represented with a minimal length, text alignment and padding.

```
>>> print(f"=== {'xxx':10} ===")
=== xxx        ===
>>> print(f"=== {'xxx':<10} ===")
=== xxx        ===
>>> print(f"=== {'xxx':^10} ===")
===    xxx     ===
>>> print(f"=== {'xxx':>10} ===")
===        xxx ===
>>> print(f"=== {'xxx':.^10} ===")
=== ...xxx.... ===
```

Formatting of numbers

Numbers can be represented with a fixed width, precision (number of decimal places) and padding.

```
>>> print(f"pi = {pi:.5f}")
pi = 3.14159
>>> print(f"pi = {pi:9.5f}")
pi =   3.14159
>>> print(f"pi = {pi:09.5f}")
pi = 003.14159
```

One can force the use of scientific notation with `e` or `E` separator and a given precision,

```
>>> print(f"{pi:e}")
3.141593e+00
>>> print(f"{pi:.3E}")
3.142E+00
```

or use the `g` general format specifier to dynamically switch between fixed precision and scientific notation, depending on the magnitude of the number.

```
>>> print(f"{1.23456:.4g}")
1.235
>>> print(f"{1234.56:.4g}")
1235
>>> print(f"{12345.6:.4g}")
1.235e+04
```

Large numbers can also be represented with a comma as thousands separator.

```
>>> print(f"{1234567890:,}")
1,234,567,890
```

By default, the sign of a number is only shown if it is negative. This behavior can also be customized, either with `+` to always output the sign, or with `  ` to output a leading space for positive numbers.

```
>>> print(f"{pi:+.5f}")
+3.14159
>>> print(f"{pi: .5f}")
 3.14159
```

A numeric value can be represented in different bases.

```
>>> n = 42; print(f"decimal {n:d} = hexadecimal {n:x} = binary {n:b}")
decimal 42 = hexadecimal 2A = binary 101010
```

Formatting of dates and times

Let `now` be the current date and time as returned by `datetime` of the standard library.

```
import datetime

now = datetime.datetime.now()
```

Various format specifiers can be used to customize its representation.

```
>>> print(f"{now: %d/%m/%Y %H:%M:%S}")
27/10/2021 10:26:33
>>> print(f"{now: %F}")
2021-10-27
>>> print(f"{now: %A}")
Wednesday
```

See also:

The Python documentation contains the full list of `strftime()` format codes.

See also:

We focus on black-and-white formatting of strings. If you are looking to style your output with colors or represent it in tabular form, have a look at the rich library.

6.1.4 Regular Expressions

Regular expressions, shortened as regexp, can be thought of as special character sequences to describe pattern matching. They are similar to globbing and wildcard matching (eg `ls *.txt`), but not identical in syntax and offer more advanced features.

Regexp can be extremely useful in a variety of situations, from data analysis (extraction, cleaning, parsing), over powerful find and replace operations in text and code, to command-line tools like `sed` and `grep`. Despite the fact that it's not necessarily an everyday topic for all scientists, we believe that it's valuable to know the basic concepts and be able to spot if a problem at hand can be solved with regular expressions.

Beware that there are various regexp engines that can have slightly different syntaxes and behaviors. So depending on the programming language or library at hand, the specific syntax and supported features can vary. The Linux module has already presented the *POSIX regexp*. Another well-known implementation is the Perl Compatible Regular Expressions (PCRE) library written in C. The Python standard library contains the `re` module, that is not fully PCRE-compatible. The third-party regex package brings additional features and improved Unicode support.

Index

Regex Patterns

Regular expressions are constructed from a sequence of atoms, that can consist of literal characters or metacharacters with special meaning. Let's start by looking into the list of commonly used characters and regexp patterns.

See also:

- Python regular expression howto
- Python `re` regular expression syntax

Literal characters

Most characters simply match themselves.

```
a       # matches 'cat' but not 'dog' or 'CAT'
```

Non-printable or whitespace characters can be matched with their backslash notation.

```
\t      # matches tab
\n      # matches newline
\r      # matches carriage return
```

Metacharacters

The metacharacters . * + ? ^ $ { } [ ] \ | ( ) have a special meaning.

```
.       # matches any character except newline
a|b     # matches 'a' or 'b'
()      # used for precedence or capturing groups
```

They need to be escaped to match the literal character.

```
\.      # matches any dot character
\\      # matches any backslash character
```

Character classes

A character class specifies the set of characters that should be matched.

```
[chars]   # any char from given set or range
[^chars]  # any char not in given set or range
```

Only] ^ - \ are treated as special in character classes, the rest are literal (eg []x] matches] or x). The - is treated as literal if it's the first or last character.

Shorthand character classes

Abbreviations exist for frequently used character classes.

```
\d      # any digit = [0-9]
\D      # any non-digit = [^0-9]
\w      # any alphanumeric plus underscore = [a-zA-Z0-9_]
\W      # any non-alphanumeric = [^a-zA-Z0-9_]
\s      # any whitespace char = [ \f\t\n\r\v] (space, form feed, tab, newline, carriage return, vertical
↪tab)
\S      # any non-whitespace char
```

Anchors and other zero-width assertions

Zero-width assertions do not match any specific character, but rather a position before, after or between characters.

```
^      # beginning of a line
$      # end of a line
\A     # start of string
\Z     # end of string
\b     # any boundary between a word char \w and a non-word char \W (eg ',.!?)
\B     # any non-word-boundary
```

Quantifiers

Quantifiers allow to specify how often a given portion of the regexp can (or must) be repeated.

```
*      # zero or more occurrences of preceding element
?      # zero or one occurrence of preceding element
+      # one or more occurrences of preceding element
{n}    # exactly n occurrences of preceding element
{n,}   # n or more occurrences of preceding element
{n,m}  # n or up to m occurrences of preceding element
```

Greedy vs lazy matching

Greedy quantifiers start by matching everything at first, and back off a character at a time only when it's obvious that the match will not succeed. This is called *backtracking* and can have a negative impact on performance.

Lazy quantifiers will prefer the shortest possible match and will increase the number of characters only if the current number fails to match.

```
.*     # longest anything
.+     # longest something
.*?    # shortest anything
.+?    # shortest something
```

For example a.*?b matches a b and ab, whereas a.+?b matches only a b but not ab.

Tip: It's best to use .* sparingly and prefer character classes that are more selective (for improved performance and less false positives).

Consider the example of matching the string inside double quotes. The naive ".*" does not work. Indeed, given the input a "b" c "d" e, the greedy matching will return as much as possible, namely "b" c "d". The lazy ".?" would return "b" and "d". Best practice is to make use of the negated character class "[^"]*".

Regex in Python

String matching without regex

For simple substring matching you don't need any regular expressions or additional modules.

```
data = "1.0 m"

if "m" in data:
    print("String contains an 'm'")
```

The built-in string method find even allows to return the first position where the substring was found.

```
>>> "John Doe".find("o")  # returns index of first match
1
>>> "John Doe".find("o", 2)  # idem, but starting at index 2
6
>>> "John Doe".find("a")  # returns -1 if no match was found
-1
```

String substitutions with regexp

Regexp can be used to replace portions of a string that match a given pattern. For example to replace any digit by #.

```
import re

re.sub(r"\d", "#", "abc123")  # --> abc###
```

Format validation with regex

A typical usecase of regex is to check that a given string is in the valid format.

```
import re

measurements = ["1.0 m", "1.000 m", "1m", "1 m", "1  m"]

for measurement in measurements:
    assert re.match(r"[\d.]*\s*m", measurement)
```

Best practice would be to define the regexp once outside the loop with re.compile() and add explanations using the re.X verbose flag to ignore whitespace and comments.

```
import re

measurements = ["1.0 m", "1.000 m", "1m", "1 m", "1  m"]

re_valid_format = re.compile(
    r"""     # raw string to treat \ as literal
    ^        # beginning of line
    [\d]+    # one or more digits
    (\.\d*)? # optional dot with decimals
    \s*      # any number of whitespace
    m        # the unit must be m
    $        # end of line
```

(continues on next page)

(continued from previous page)

```
""",
    re.X,
)

for measurement in measurements:
    assert re_valid_format.match(measurement)
```

Data parsing with regexp

Another important application of regexp is to parse a given string and extract the data from it. This can be done elegantly with *named capturing groups* (?P<name>regex).

```
import re

measurements = ["1.0 m", "1.000  m", "100cm", "1000 mm"]

re_parse_value_unit = re.compile(
    r"""
    (?P<value>[\d.]*)  # numerical value
    \s*                # whitespace
    (?P<unit>[\w]+)    # unit
""",
    re.X,
)

for m in measurements:
    data = re_parse_value_unit.match(m).groupdict()
    print(data["value"], data["unit"])
```

CHAPTER SEVEN

PYTHON ECOSYSTEM (III)

At the interface between Python code and external data files. We explain how to read or write files, discuss data types and file formats. We show how to handle configuration parameters and mention tools to automate the data analysis.

7.1 Index

7.1.1 Reading and writing files

Of course scientific computing relies on reading input parameters and exporting the computed results. But various things must be kept in mind with file input/output. So let's start by looking at general aspects when reading and writing files.

First off is performance. Disk I/O is orders of magnitude slower than working in memory. One distinguishes between the latency of accessing a given file, and the further throughput of the actual data. It's commonly recommended in HPC to bundle some data, then write it in bulk to disk, instead of appending small amounts of data at each computational step. In addition, always avoid writing millions of small files, for which the file system access if often much slower, than for a small number of larger files.

Next is portability. When collaborating across operating systems, additional care must be taken to make the code cross-platform compatible. Many aspects may depend on the platform: text encoding, line endings, path separators. Even simple cases, as accessing data from a particular directory, may need some thinking. In most cases you should prefer relative paths (with respect to the code location). However when accessing a network share, use the well-defined absolute path instead. Whenever possible, use code that abstracts away the differences of each platform. And otherwise try to keep as many parameters as possible customizable, so that others may adapt them to their needs.

Last but not least comes error handling. Writing data to a file may fail. For instance if the connection to the network share is not stable or the parent folder has unexpected permissions. Especially in long-running jobs try to react to such errors, maybe by re-trying again after some time, instead of losing your results.

Having mentioned some of the common obstacles, we now focus back on Python. We start by discussing text encoding, explaining how to interact with the file system (via `os`, `shutil` and `pathlib`) and finally how to handle files.

Index

Encoding

Besides numbers and punctuation, data files often contain text with characters, ranging from the well-known a-z, over accented characters ä, é, ù, to mathematical symbols ∞ and modern emojis . Most of the time the text is understood as human-readable string. But when this string must be saved as binary format in some file, the decision must be made how to map the characters to a binary representation. This operation is known as *encoding*. The inverse, converting a sequence of raw bytes (that may in principle represent any binary data) into a human-readable text, is called *decoding*.

Common text encodings

Many different encodings exist, depending on the characters used by a language or even the computer's operating system. And by just looking at the binary data in a text file, it's non-trivial to know which encoding was used. Moreover, most encodings are not compatible with each other. Hence the common problem of decoding errors or mis-mapped characters when opening the files with a wrong encoding.

ASCII

ASCII (American Standard Code for Information Interchange) dates back to the 1960s and maps each of its 128 characters (including non-printable and control characters) to an integer represented by 7 bits.

Latin1

Latin1, or rather ISO/IEC 8859-1 is an 8-bits extension of ASCII to include typical accented characters used in latin languages. The Windows operating system used the Windows-1252 counterpart, with some deviations.

Unicode and UTF

Already in the 1980s people realized, that a much larger set of characters must be supported to accommodate the world's languages and needs. The efforts lead to the birth of the Unicode Standard, which currently defines ~150k characters and can support over a million in total. Strictly speaking, Unicode is itself not an encoding. But the Unicode Standard defines for instance the utf-8 and utf-16 encodings. As a matter of fact, the 8-bit utf-8 is compatible with ASCII, but utf-16 or utf-32 are not. Unicode has become the de-facto standard and when in doubt you should be using utf-8 for everything.

Tip: On Unix, tools like iconv or uconv can help to convert files between different character encodings. The Python package chardet tries to automatically detect the correct encoding of a given file (by making an educated guess about its language).

Encoding in Python

The default encoding in Python 3 is utf-8. This applies to the source code files themselves, as well as to strings, which are unicode characters by default. The `str.encode()` and `bytes.decode()` methods convert between strings and bytes.

See also:

The Python documentation contains a section Encodings and Unicode with many details.

Different encodings of the Umlaut "ä" yield different bytestrings.

```
>>> "ä".encode("utf8")
b"\xc3\xa4"
>>> "ä".encode()          # shorthand for above
b"\xc3\xa4"
>>> "ä".encode("latin1")  # different encoding
b"\xe4"
```

Conversely, the decoding of the bytestring requires knowledge of the correct encoding.

```
>>> b"\xc3\xa4".decode()
"ä"
>>> b"\xe4".decode("latin1")
"ä"
```

Otherwise an incorrect string gets returned, known as mojibake,

```
>>> "ä".encode("utf8").decode("latin1")
"Ã¤"
>>> "Ã¤".encode("latin1").decode("utf8")  # reverting wrong encoding
"ä"
```

or the decoding fails completely

```
>>> "ä".encode("latin1").decode("utf8")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
UnicodeDecodeError: 'utf-8' codec can't decode byte 0xe4 in position 0: unexpected end of data
```

In the worst case, you can decide to handle the error by replacing the failing characters with “?” in ASCII, or the official Unicode replacement character “” in utf-8.

```
>>> "ä".encode("latin1").decode("utf8", errors="replace")
""
```

os & shutil

os for operating system interfaces

The os module of the Python Standard Library provides a wide interface to functionality that may depend on the operating system.

```
import os

os.environ["HOME"]          # access the $HOME environment variable
os.getenv("DEBUG", False)   # return the $DEBUG env var if set, or False as default
os.getpid()                 # get the current process ID
os.urandom(64)              # get 64 bytes of random data
os.chdir("data")            # change the current working directory
```

There’s a collection of methods to interact with file system paths. Even though they are still commonly used in many Python projects, new code should prefer the *pathlib* equivalents. Analogously the *subprocess* module should be preferred over the os.system() method to execute commands in a subshell.

shutil for high-level file operations

The shutil module of the Python Standard Library offers helper functions for common file operations, like copying or deleting.

Below some usage examples, where source, dest and path are understood to be strings or path-like objects referring to files or directories.

```
import shutil

shutil.move(source, dest)       # mv source destination
shutil.copy(source, dest)       # cp source destination
shutil.copy2(source, dest)      # cp -p source destination
shutil.copytree(source, dest)   # cp -rp source destination
shutil.rmtree(path)             # rm -r path
shutil.which("python3")         # which python3
shutil.chown(path, user="jdoe", group="jdoe") # chown jdoe:jdoe path
shutil.diskusage(path)          # return total, used and free disk space
```

pathlib

In order to access files, one must first define the path on the file system to that file, as not all files reside in the same folder as the Python script. When released in 2014, Python 3.4 introduced the pathlib module as high-level and object-oriented approach. As such it has a considerable overlap with the previously existing os.path. However, pathlib is nowadays the recommended library to handle file system paths.

Path

Note: The following code snippets presume that Path has been imported.

```
from pathlib import Path
```

The most basic usage is to define a path object from a string representation.

```
PROJECT_DIR = Path("path/to/project")
```

Strictly speaking, the path separator depends on the operating system, namely / on Linux/macOS and \ on Windows. Nevertheless the above code will work on both platforms, but return pathlib.PosixPath and pathlib.WindowsPath objects respectively. There are also dedicated methods to join paths without having to worry about the separator.

```
PROJECT_DIR = Path("path") / Path("to") / Path("project")
PROJECT_DIR = Path("path", "to", "project")
PROJECT_DIR = Path("path").joinpath("to", "project")
```

Tip: Use raw string literals for Windows paths.

Given that the Windows path separator \ is also used as an escape character, it's best to use raw strings r"C:\new_folder" to always treat the backslash sign as literal and avoid any confusion with non-printable characters like newline \n.

Path-like objects

Before Python 3.6, file system paths have been represented as strings. But with the addition of the object-oriented pathlib representation, most of the Python methods were adapted to accept path-like objects as arguments, which can either be the path represented as str or an object implementing the os.PathLike protocol. This means that for example the open() function now works with file.txt as well as Path("file.txt"), as we will shortly see when discussing *file handling*.

Path methods

We briefly mention a selection of methods that path objects support.

```
some_path = Path.cwd()      # path of current working directory

some_path.exists()          # return True if path exists
some_path.is_dir()          # return True if path is a directory
some_path.resolve()         # resolve as absolute path
some_path.parent()          # parent directory
some_path.parents()         # immutable sequence of all ancestors
some_path.name()            # file or directory name
some_path.suffix()          # file extension (or empty string)
some_path.stem()            # file name without suffix
```

Some yield generator objects that can be looped over.

```
# Loop over the contents of a given directory
for contents_path in some_path.iterdir()
    print(contents_path)

# Loop over files with given extension inside subfolders
for img_path in some_path.glob("**/*.jpg"):
    print(img_path)
```

Another common usecase is to reference the base directory of the project /path/to/project/, by starting from the location of the executed script /path/to/project/src/cli/cli.py.

```
BASE_DIR = Path(__file__).parents[2]
```

Also note that, since Python 3.9+, __file__ directly returns the absolute path of the script, we no longer need to add resolve().

Replacing os with pathlib

The following table summarizes the correspondence between pathlib and os methods.

pathlib	os
Path.cwd()	os.getcwd()
Path.readlink()	os.readlink(path)
Path.unlink()	os.remove(path)
Path.resolve()	os.realpath(path)
Path.stat()	os.stat(path)
Path.home()	os.path.expanduser("~")
Path.chmod(mode)	os.chmod(path, mode)
Path.mkdir()	os.mkdir(path)
path1.join(path2)	os.path.join(path1, path2)
path1 / path2	os.path.join(path1, path2)

File handling

Having discussed all pre-requisites, from encoding to interacting with the file system and juggling paths, we can finally move on to reading and writing files.

open

The following snippet shows how to use open() to read the contents of a file. It's best practice to use a with context to automatically close the file handle when done.

```
with open("file.txt") as f:
    content = f.read()
```

Instead of slurping the whole file contents at once, you can also iterate over the individual lines, for instance to directly convert them to floats.

```
contents = []
with open("file.txt") as f:
    for line in f:
        contents.append(float(line))
```

Mode

As a second argument, one can provide the mode in which the file should be opened.

Mode	Description
"r"	read-only mode (default)
"w"	write mode, overwriting existing files
"a"	append mode, write by adding to the end of file
"x"	exclusive mode, write mode that fails if the file already exists
"+"	update mode, open to read and write
"rb"	binary read mode to read bytes objects instead of str
"wb"	binary write mode

The following writes a string to a file.

```
with open("file.txt", "w") as f:
    f.write("lorem ipsum\n")
```

Encoding

The optional encoding= and errors= arguments define the encoding to use and how to handle encoding/decoding errors (as already mentioned when discussing *encoding* in general).

```
with open("file.txt", encoding="latin1", errors="replace") as f:
    lines = f.readlines()
```

open with pathlib

When making use of pathlib, there are multiple equivalent notations to read files.

```
from pathlib import Path

some_path = Path("file.txt")

with open(some_path) as f:
    content = f.read()

with some_path.open() as f:
    content = f.read()

content = some_path.read_text()
```

Similarly there are Path.read_bytes(), Path.write_text() and Path.write_bytes() shorthands.

7.1.2 Data types and file formats

Working with data requires the definition of appropriate "containers" to store that data. On the one hand, during program runtime, one must choose the data structure of the variables. This is a standard topic of most lectures on programming and we will only briefly skim over the common data types in Python. On the other hand, at the end of program execution, the results must be exported into some file format for later access. We will discuss some common text and binary formats as well as how to read or write them with Python.

Note: Don't forget about metadata. The bare results of a simulation typically fail to capture all of its features. It's of utmost importance to also include the metadata. This includes of course the input parameters (and units thereof), details of the related experimental setup, but also the version (or commit sha) of the programs used to run the code and perform the data analysis.

Index

Data types

Basic data types

Let's briefly look at the most common data types in Python. All data is represented by objects. The type of an object, type(some_object), determines which operations are supported. For instance all sequence objects (eg lists) have a length len(some_list) defined. Another important distinction is mutability. Depending on whether the value of an object can be changed after its creation, the object is called *mutable* or *immutable*.

See also:

The Python documentation contains a section on the data model providing a good overview of the data types and their hierarchy.

NoneType

The built-in name None is used to signify the absence of a value. As an example it's returned by any function without explicit return statement. A common mistake is to try to loop over a variable that may be empty,

```
data_rows = [[1, 2], None]

for row in data_rows:
    for element in row:
        print(element)
```

which fails with the TypeError: 'NoneType' object is not iterable error message.

Numbers

The numeric types include int for integers, its subclass bool for Booleans, as well as float and complex for real and complex numbers respectively.

```
ten_thousand = 10_000     # large int with optional thousands separator
z = 1.0 + 2.0j            # complex numbers
assert type(7/2) == float # floating point division of two integers
assert int(7/2) == 3      # convert float to integer
```

Sequences

Sequences represent finite ordered sets indexed by non-negative numbers. They support len() and index slicing a[i:j]. Examples for immutable sequences are str (strings), tuple and range. Conversely list (and array) are mutable sequences.

```
a_string = "Lorem Ipsum"
a_tuple = (1,)            # tuple with single element (defined by comma)
a_list = [1, 2, 3]
```

Set types

Set types represent unordered, finite sets of unique, immutable objects. Common uses are fast membership testing, removing duplicates or set operations like intersection, union or difference. The type set is mutable, frozenset immutable.

```
empty_set = set()        # as {} is reserved for empty dict
a_set = set("AABC")      # set of unique characters in a string
b_set = set(["AABC"])    # set with whole string as single element
```

Mapping types

Mapping types are mutable objects that map hashable keys to arbitrary objects as values. Most of Python's immutable built-in objects are hashable. Mutable objects however are not hashable. The dict (dictionary) is the only mapping type.

```
a_dict = { "a": 1, "b": 2}
assert a_dict["a"] == 1     # retrieve value of key "a"
a_dict["c"]                 # throws "KeyError" exception
a_dict.get("c", "3")        # returns value or the default "3"
```

(continues on next page)

(continued from previous page)

```
a_dict.keys()            # returns keys in a dict_keys object
a_dict.values()          # returns values in a dict_values object

for key, value in a_dict.items():   # loop over key value tuples
    print(f"{key=} -> {value=}")

nested_dict = {}
nested_dict["first"] = {}  # nested keys need to be initialized first
nested_dict["first"]["second"] = "value"
```

Data structure

In practice, the actual data structure is often some nested mixture of the above basic data types.

```
john = {
    'firstname': 'John',
    'surname': 'Doe',
    'age': 42,
    'emails': [{'email': 'johndoe@example.com', 'type': 'private'}],
    'married': False,
    'children': None,
}
```

Dataclass

When opting for object-oriented programming in scientific computing, writing out the class definition with the initialization of all attributes involves a lot of boilerplate code. A noteworthy addition since Python 3.7+ are dataclasses, which provide a short-hand notation. The @dataclass decorator implicitly adds an __init__() method (among others) for the attributes.

```
from dataclasses import dataclass


@dataclass
class Person:
    name: str
    weight: float
    children: int = 0


john = Person(name="John", weight=74.0)
john.children = 1
```

Scientific data types

The scientific libraries bring a variety of additional data types for numerical computing and data analysis. The details are beyond our scope and presented in lectures on scientific computing.

- NumPy data types
- NumPy ndarray
- Pandas data structures Series and DataFrame

See also:

There are beautiful and instructive visual guides for both NumPy and Pandas to illustrate the data structures and how the operations manipulate them.

Text formats

A first choice, when printing results to the console is no longer sufficient, is to export them to some variant of text files. The advantage of text formats is that they are human-readable, non-proprietary, cross-platform and future-proof. We will discuss how Python can interact with some common text formats.

CSV

The canonical choice is to dump all numbers with a chosen separator between, resulting in either Comma-Separated Values (CSV) or Tab-Separated Values (TSV). This format is best for flat, tabular data structures. It's best practice to include a header as first line with the names of the columns.

```
A,B,C
1,2,3
```

Standard Library

The csv module providers reader and writer objects for CSV files.

Tip: The below example uses an io.StringIO object as proxy for a real CSV file. In a nutshell this is a string-like object that behaves like a file, but in-memory and not on the filesystem. This allows our snippets to contain the data directly inside the code, thereby making the examples self-contained and hopefully easier to grasp. Feel free to adapt the `with csv_file as f:` contexts to `with open("file.csv") as f:` in order to work with actual files.

```python
import csv
import io

csv_file = io.StringIO(
    "A,B,C\n"
    "1,2,3\n"
    "a,\"b,b\nb\",c\n"
)

with csv_file as f:
    csv_reader = csv.reader(f)
    # Save the first row with the headers
    csv_header = next(csv_reader)
    # Save all others rows as nested list
    data = [row for row in csv_reader]
```

(continues on next page)

(continued from previous page)

```
assert data == [["1", "2", "3"], ["a", "b,b\nb", "c"]]

# Export again as csv (with the same contents as csv_file)
with open("file.csv", "w") as f:
    csv.writer(f).writerow(csv_header)
    csv.writer(f).writerows(data)
```

Note how the second element of the second row is quoted, in order to support nested newlines and commas.

There are also dedicated DictReader and DictWriter methods to work with rows as dictionaries instead of lists.

```
import csv

data = [
    {"A": 1, "B": 2, "C": 3},
    {"A": 4, "B": 5, "C": 6},
]

with open("file_dict.csv", "w") as f:
    # Use the dict keys of the first row as field names
    csv_writer = csv.DictWriter(f, fieldnames=data[0].keys())
    # Write header with field names and the data
    csv_writer.writeheader()
    csv_writer.writerows(data)
```

NumPy

When working with scientific libraries, it's recommended to use their helper methods to import/export text data, instead of the csv standard library module. The numpy library offers np.loadtxt() as a faster reader for simple text files. In addition there's np.genfromtxt() with more sophisticated handling of missing values. Conversely np.savetxt() can export data to a text file.

```
import numpy as np
import io

csv_file = io.StringIO(
    "value_x,value_y\n"
    "2.0,3.0\n"
    "2.2,3.3\n"
)

data = np.loadtxt(
    csv_file,      # csv file to import as ndarray
    delimiter=",", # use comma as separator
    skiprows=1,    # skip the one line with the header
)
```

Pandas

The data analysis library pandas offers read_csv() and to_csv() to import/export CSV files. These methods can offer a better performance and more configuration options than the numpy equivalents. They can also readily handle and parse datetime values.

```
import pandas as pd
import io

csv_file = io.StringIO(
    "date , value_x, value_y\n"
    "1-2-2023, 2.0, 3.0\n"
    "2-2-2023, 2.2, 3.3\n"
)

df = pd.read_csv(
    csv_file,                         # csv file to import as dataframe
    header=0,                          # treat the first csv line as header
    names=["datecol", "col1", "col2"],  # override header and assign custom names
    skipinitialspace=True,              # ignore space after comma in csv
    engine="c",                         # use faster C engine for parsing
    parse_dates=["datecol"],              # parse given column as date
    dayfirst=True,                      # dates start with day (eg DD-MM-YYYY)
    compression=None,                     # on-the-fly decompression (zip, tar, ...)
)
```

JSON

The JavaScript Object Notation (JSON) allows to encode more complicated and nested data structures. Its notation is very close to the basic data types in Python. Moreover it has become the standard data exchange format of most HTTP and REST APIs, for instance submitting measurement data to the InfluxDB time series database.

```
{
    "firstname": "John",
    "surname": "Doe",
    "age": 42,
    "married": false,
    "children": null,
    "emails": [
        {
            "type": "private",
            "email": "johndoe@example.com"
        }
    ]
}
```

The following Python snippet uses the builtin json module to import the above JSON, modify it, and export the changes.

```
import json
from pprint import pprint

with open("johndoe.json") as f:
    johndoe = json.load(f)

pprint(johndoe)
```

(continues on next page)

(continued from previous page)

```
#  {'age': 42,
#   'children': None,
#   'emails': [{'email': 'johndoe@example.com', 'type': 'private'}],
#   'firstname': 'John',
#   'married': False,
#   'surname': 'Doe'}

johndoe["married"] = True
johndoe["spouse"] = "Jane Doe"

with open("johndoe.json", "w") as f:
    json.dump(johndoe, f, sort_keys=False, indent=4)
```

Watch out that not all data types can directly be serialized as JSON. In particular, when expanding the above example to naively include a date, the json.dump fails with TypeError: Object of type date is not JSON serializable. A simple fix could be to add the default=str argument, so that str() is applied first to non-serializable objects, yielding the "YYYY-MM-DD" string representation of the date.

```
import json
from datetime import datetime

johndoe = {}
johndoe["weddingdate"] = datetime.now().date()

with open("johndoe.json", "w") as f:
    json.dump(johndoe, f, sort_keys=False, indent=4, default=str)
```

XML

The Extensible Markup Language (XML) is a close relative to the well-known HTML of the world wide web.

```
<?xml version="1.0" encoding="utf-8"?>
<data>
 <row>
   <shape>square</shape>
   <sides>4</sides>
 </row>
 <row>
   <shape>triangle</shape>
   <sides>3</sides>
 </row>
 <row>
   <shape>circle</shape>
   <sides/>
 </row>
</data>
```

Different XML libraries exist in the Python world:

- The Standard Library ships the xml library containing the xml.etree.ElementTree parser.
- Noteworthy mentions of PyPI are the very lightweight xmltodict and the full-fledged lxml
- Pandas includes the read_xml() and to_xml() methods to import/export dataframes to/from XML.

Others

- Python Standard Library includes the ConfigParser module to read/write Initialization (INI) config files.
- Since Python 3.11 tomllib is included to parse Tom's Obvious Minimal Language (TOML) files.
- The third-party ruamel.yaml provides a powerful interface to Yet Another Markup Language (YAML) files.

Binary formats

Especially for large data sets, text formats do not scale and one should consider binary formats instead. They are designed to be machine-readable and can offer a better input/output performance, as they are optimized to be closer to the machine representation of the data. Another common advantage is built-in support for sparse data and compression. Many different formats exist, some proprietary, some restricted to specific research domains. We will discuss some open-source formats that are commonly found in scientific computing.

Pickle

The pickle module allows to serialize Python objects into binary files, and restore them later on by de-serialization. Make sure to only unpickle data that you trust, as it basically allows for arbitrary code execution. Even though it's important to know pickle exists, there are certainly better formats for scientific computing.

```python
import pickle

some_object = {"A": 1, "B": [2, 3]}

with open("object.dump", "wb") as f:
    pickle.dump(some_object, f)

with open("object.dump", "rb") as f:
    imported_object = pickle.load(f)

assert imported_object == some_object
```

Sqlite3

The sqlite3 module provides an interface to SQLite databases. These are single-file databases with remarkable performance. They are typically used as relational databases but also support JSON data. The query language is a dialect of Structured Query Language (SQL).

See also:

If you need to access other SQL databases, have a look at pandas.read_sql() or SQLAlchemy for a full-fledged object-relational mapper (ORM).

```python
import sqlite3

# Initialize connection to the SQLite DB
con = sqlite3.connect("measurements.db")

# Let queries return Row objects accessible by name
con.row_factory = sqlite3.Row

# Initialize cursor to execute SQL statements
cur = con.cursor()
```

(continues on next page)

(continued from previous page)

```
# Create table in empty database
cur.execute("""
    CREATE TABLE IF NOT EXISTS
    measurements(device, temperature)
""")

# Prepare insertion of measurement data
measurements = [("dev1", 77.2), ("dev2", 294.5)]
cur.executemany(
    "INSERT INTO measurements VALUES(?, ?)",
    measurements,
)

# Commit transaction to write data to DB
con.commit()

# Retrieve all measurements from the DB
for row in cur.execute("SELECT * FROM measurements"):
    print(f"{row['device']} -> {row['temperature']}")

# Close connection to DB
con.close()
```

NumPy

NumPy has its own binary serialization format. Single arrays are persisted in .npy files using np.save(), while multiple arrays are combined into .npz files with np.savez(). Conversely there's np.load() to import both types of files back in.

Excel

In case you got handed an Excel .xlsx spreadsheet with lab data, consider using Python for the further data analysis, instead of the proprietary Office software.

Pandas provides read_excel() and to_excel() as powerful interface to various spreadsheet file formats. Under the hood it uses the openpyxl engine to read and write Excel files natively from Python.

HDF

The Hierarchical Data Format (HDF) is designed to store and organize larger amounts of scientific data and aims to provide fast I/O processing. The current version is HDF5 and has libraries for basically all programming languages. In addition, the HDFView Java software offers a GUI to visually navigate and modify the data.

Yet again, pandas offers convenient helper methods read_hdf() and to_hdf(). It makes use of the lower-level PyTables library. Alternatively there's also the h5py Python package.

7.1.3 Reading config parameters

Besides the input data sets for computational analysis, most coding projects rely on some form of configuration parameters. These range from booleans to toggle certain flags (like debug logging), over paths and file names to access data, to miscellaneous variables to tune the algorithm.

It's best practice not to directly insert "magic" values deep inside your code, but rather declare them as constant variables early on, to that others can easily spot what assumptions have been made and how to adjust the parameters.

Especially the tunable parameters that will often be modified, should be moved outside of the computational code. This not only facilitates parallel execution of simulation runs with varying parameters, but also allows to easily bundle the files with input parameters with the generated output files for better reproducibility.

In addition to reading the config parameters from files in your preferred format (as explained in the previous section), you may want to directly parse some as command-line arguments for fast access. Alternatively, we will explain how to encapsulate your configuration in a Python module for highest flexibility.

Warning: Always treat sensitive data (credentials, API tokens, etc) with extra care. Try to read these variables from a separate file, which you exclude from your version control.

Index

Command-line arguments

Python scripts can parse command-line arguments provided by the user.

argparse

Python contains the argparse module for parsing command-line arguments.

```python
#!/usr/bin/env python
import argparse


def say_hello(name):
    """Small CLI tool to say hello."""

    print(f"Hello {name}!")


if __name__ == "__main__":
    # Initialize the parser using the function docstring as description.
    parser = argparse.ArgumentParser(description=say_hello.__doc__)
    # Define the supported command-line arguments.
    parser.add_argument("--name", "-n", help="Your name", default="world")
    # Parse the provided command-line arguments into args.
    args = parser.parse_args()
    # Call the function with the name argument.
    say_hello(args.name)
```

Presuming you saved the above code as hello_cli.py, you can then provide the command-line options at runtime.

```
$ ./hello_cli.py
Hello world!
$ ./hello_cli.py --name Jane
Hello Jane!
```

(continues on next page)

(continued from previous page)

```
$ ./hello_cli.py -n Jane
Hello Jane!
$ ./hello_cli.py --help
usage: hello_cli.py [-h] [--name NAME]

Small CLI tool to say hello.

options:
  -h, --help            show this help message and exit
  --name NAME, -n NAME  Your name
```

Note that the --help option was added automatically and will show the function docstring and usage options.

click

The third-party package click provides a "Command Line Interface Creation Kit" to build advanced command-line applications. It's syntax makes heavy use of function @decorators, which boil down to a fancy notation (also known as *syntactic sugar*) for function-of-a-function. This syntax allows for a terse, yet highly readable code.

```python
#!/usr/bin/env python
import click


@click.command()
@click.option("--name", "-n", help="Your name", default="world")
def say_hello(name):
    """Small CLI tool to say hello."""

    print(f"Hello {name}!")


if __name__ == "__main__":
    say_hello()
```

Config module

In larger projects, where the Python code spans over several files, it's common practice to extract all tunable parameters and constants into a dedicated config module. Not only does this approach bundle all declarations together, but the direct support of Python code inside the module offers the highest flexibility. For instance one can implement default values that can be readily overridden. In advanced setups, there's even the possibility for more complicated secret management using GPG or querying some parameters directly from an API.

Sample project structure

```
├── README.markdown
├── cli.py*
└── pyproject/
    ├── __init__.py
    ├── analysis.py
    └── config/
        ├── __init__.py
        ├── overrides.py
        └── secrets.py.example
```

Where the executable cli.py provides command-line access to running the code

```python
#!/usr/bin/env python
"""Pyproject command-line interface."""
from pyproject.analysis import run_data_analysis


if __name__ == "__main__":
    run_data_analysis()
```

The pyproject/__init__py can be empty and serves to define the package namespace. The analysis.py loads config parameters and defines how to perform the data analysis.

```python
"""Data analysis algorithms."""
from pyproject.config import UNIVERSAL_RESULT, SCALE_FACTOR


def run_data_analysis():
    """Wrapper to run full data analysis."""
    print(f"Scale factor: {SCALE_FACTOR}")
    print(f"Scaled result: {UNIVERSAL_RESULT * SCALE_FACTOR}")
```

Last, but not least, the config/__init__.py module encapsulates the declaration of all parameters.

```python
"""Config parameters for the py-project."""

UNIVERSAL_RESULT = 42

"""
Some default values below can be overridden by
adding definitions to the optional `overrides.py` file.
"""
try:
    from pyproject.config.overrides import SCALE_FACTOR
except ImportError:
    SCALE_FACTOR = 1.0

"""
Loading sensitive data from `secrets.py`.
"""
try:
    from pyproject.config.secrets import SECRETS
    API_CREDENTIALS = SECRETS("API_TOKEN")
except ImportError:
    print("Warning: running without definition of config/secrets.py")
```

All sensitive data is contained inside a dedicated secrets.py file, which is excluded from version control. However, a secrets.py.example is included, to serve as template and show what the file should contain.

```python
"""
Secret parameters with sensitive data.

Copy this secrets.py.example file to secrets.py
and adapt the credentials below.
"""

SECRETS = {
    "API_TOKEN": "1234567890"
}
```

7.1.4 Automation

When performing numerical simulations or applying a data analysis pipeline, it's crucial to keep track of all the steps that where necessary to obtain the results. The idealistic goal is *scientific reproducibility*, which in practice boils down to instructing peers (and your future self) how to redo your computations. Any time-consuming and error-prone manual steps should be avoided and replaced with an automated workflow. This automation will help to remove tedious and repetitive tasks from your everyday research. And it's anyway required if you want to redo similar computations with varying input parameters, for instance as batch jobs on a HPC cluster.

Even though you may start by documenting all required steps in some text document, you should quickly move on and automate as much as possible. This can be achieved with Python or Shell scripts. An alternative is to make use of available task runners. Another significant ingredient is to run external commands from inside Python code. Lastly, we will show examples of executing code in parallel using concurrency.

Index

Task runner

Having stressed the importance of automation, we now proceed by looking at task runners. Their goal is to run a set of pre-defined commands in the correct order. The archetype usecase is compiling software, where dependencies dictate in which order the libraries must be built, before linking them all into the final executable. A similar type of dependencies can also happen in data science pipelines, were the input data must first be cleaned, transformed and merged, before the actual analysis can take place. Given the dependency tree, the task runner can watch for changes of the input files, and only execute the necessary steps to re-generate the output files. Not having to run the whole pipeline each time can save time, especially when some long-running computations are only re-run when absolutely needed.

Make

The GNU make software (and derivatives) is probably the most used build automation tool in Unix software development.

When running make, it searches the current directory for a file named Makefile, which contains the rules to build the individual targets, and how they depend on each other. Schematically

```
<target>: <prerequisites>
<TAB><commands to build target>
```

where it's important to use TAB for indentation and not spaces.

Consider the following example Makefile combining several targets.

```
# Declaration of variables
INPUT_FILES := data1 data2
INTERMEDIATE_FILE := merged_data

# Targets that do not correspond to file names
.PHONY: all clean

# First target is the default goal when calling `make`
all: $(INPUT_FILES) $(INTERMEDIATE_FILE)

clean:
	rm -f $(INPUT_FILES) $(INTERMEDIATE_FILE)

data1:
	echo "1" > data1
```

(continues on next page)

(continued from previous page)

```

data2:
	echo "2" > data2

# The intermediate file depends on the input files
$(INTERMEDIATE_FILE): $(INPUT_FILES)
	cat data1 data2 > $(INTERMEDIATE_FILE)
```

It's sufficient to call make [all] to generate the input and intermediate files, in the correct order.

```
$ make
echo "1" > data1
echo "2" > data2
cat data1 data2 > merged_data
```

When calling make again, nothing will be done, as all targets already exist. However, when modifying one of the input files, the intermediate file will be regenerated, as make notices that it's out of date.

```
$ make
make: Nothing to be done for `all'.
$ echo "3" > data2
$ make
cat data1 data2 > merged_data
```

Finally, make clean will delete all generated files.

```
$ make clean
rm -f data1 data2 merged_data
```

See also:

There are many online ressources and tutorials available to help you dive deeper into Makefiles, for instance the official make manual or the makefile tutorial.

Alternatives

There are many alternatives to Make, depending on personal taste, the programming language or surrounding community. We only mention a few.

snakemake

The snakemake workflow management system is written in Python and targeted at data science pipelines.

doit

Another task runner implemented in Python is doit.

task

Task is implemented in Go and parsed a Taskfile.yml file.

Subprocess

Instead of using dedicated task runners, one can also be in need of launching external commands directly from inside a Python script. The subprocess module is part of Python's Standard Library and helps spawning new processes and managing their input/output.

Simple example

Let's consider the following shell command as an example,

```
$ seq 20 | grep 3
3
13
```

where seq 20 prints lines with the integers 1 to 20, | redirects this output as input to the next command, and finally grep 3 keeps only lines containing the digit 3.

We can use subprocess.run() to execute this command in Python and get its output.

```
import subprocess

some_cmd = "seq 20 | grep 3"
some_cmd_result = subprocess.run(some_cmd, shell=True, capture_output=True)
```

The shell=True argument executes the command in a shell (which on POSIX defaults to /bin/sh), whereas the capture_output=True is a short-hand to keep the output of stdout and stderr (instead of simply ignoring it). The latter is crucial if you want to parse the output of the command (and be able to handle any error messages sent to stderr). This data can be accessed as attributes of the returned object.

```
print(some_cmd_result)  # CompletedProcess(args='seq 20 | grep 3', returncode=0, stdout=b'3\n13\
↪n', stderr=b'')
print(some_cmd_result.stdout.decode("utf-8"))  # '3\n13\n'
print(some_cmd_result.stdout.decode("utf-8").splitlines())  # ['3', '13']
```

Given that .stdout is a bytestring, we must first use .decode("utf-8") to convert it into a normal Python string. An additional .splitlines() then converts the string with newlines into a Python list.

Example wrapper function

If you need to call subprocess.run() multiple times, the bytestring decoding and error handling may become repetitive. The following snippet is meant to simplify this by wrapping the required steps in a Python function.

```
import subprocess


def run_command(command, shell=True, timeout=3):
    """Run generic commands via subprocess wrapper and return their stdout as string."""
    try:
        cmd = subprocess.run(
            command, shell=shell, stdout=subprocess.PIPE, stderr=subprocess.PIPE, timeout=timeout
        )
```

(continues on next page)

(continued from previous page)

```
        if cmd.returncode != 0:
            raise Exception("command returned non-zero exit code: " + str(cmd.returncode))
        if cmd.stderr:
            stderr = cmd.stderr.decode("utf-8")
            raise Exception("command returned output to stderr: " + stderr)
        else:
            command_output = cmd.stdout.decode("utf-8")
    except subprocess.TimeoutExpired:
        raise Exception("command timeout")

    return command_output
```

Here an example usage of this wrapper function.

```
some_cmd_results_list = run_command("seq 20 | grep 3").splitlines()
```

Concurrency

The ability to execute multiple tasks in parallel is often crucial in scientific computing, as it may help to significantly improve the overall runtime of various algorithms. The scientific libraries like numpy do a lot of the heavy-lifting under the hood and let the user directly run their computations on multiple cores, without having to worry about the implementation details. Nevertheless it can be instructive to know how to write simple multi-threading or multi-processing code directly in Python. Besides the asyncio approach, the Standard Library also contains concurrent.futures, which is probably easier to get started with.

HTTP requests

Let's consider the scenario where you need to download multiple internet pages and parse their contents. This could be any HTTP REST API where you need to collect some data, or, as in our case, simply the first few lines of a Wikipedia article.

```
import requests


def wiki_extract_for_title(wiki_title):
    """Query the Wikipedia API and return the first 5 sentences of the text with given title."""

    resp = requests.get(
        "https://en.wikipedia.org/w/api.php"
        "?action=query&prop=extracts&exsentences=5&explaintext=1&format=json&titles="
        + wiki_title
    )
    # Convert the JSON into a Python dict
    resp_dict = resp.json()
    # Get the extract text from the Python dict
    page_id = list(resp_dict["query"]["pages"].keys())[0]
    extract = resp_dict["query"]["pages"][page_id]["extract"]

    return extract


wiki_titles = ["Albert_Einstein", "James_Clerk_Maxwell"]
wiki_extracts = []
```

(continues on next page)

(continued from previous page)

```python
for wiki_title in wiki_titles:
    wiki_extracts.append(wiki_extract_for_title(wiki_title))

print(wiki_extracts)
```

Note: We use the requests package instead of the built-in urllib, as it is a very popular HTTP library with a more concise syntax. Another alternative would be httpx.

Now assume you need to make hundreds or thousands of such HTTP requests. With a linear execution as above, the processor spends most of its time idling and waiting for the response of the web server. This is a perfect use case where concurrency provides a considerable speedup.

concurrent.futures

The concurrent.futures module provides a high-level interface for concurrency via multiple threads or processes. In a nutshell, you define a pool of workers (either ThreadPoolExecutor for threads, or ProcessPoolExecutor for processes) and submit your tasks to its queue. The returned object is an instance of concurrent.futures.Future, which is conceptually similar to a promise or callback. It does not immediately contain the result of your queued task, but promises to have it in the future.

```python
import requests
import concurrent.futures


def wiki_extract_for_title(wiki_title):
    """Query the Wikipedia API and return the first 5 sentences of the text with given title."""

    resp = requests.get(
        "https://en.wikipedia.org/w/api.php"
        "?action=query&prop=extracts&exsentences=5&explaintext=1&format=json&titles="
        + wiki_title
    )
    # Convert the JSON into a Python dict
    resp_dict = resp.json()
    # Get the extract text from the Python dict
    page_id = list(resp_dict["query"]["pages"].keys())[0]
    extract = resp_dict["query"]["pages"][page_id]["extract"]

    return extract


wiki_titles = ["Albert_Einstein", "James_Clerk_Maxwell"]
wiki_extracts = []

# Use a pool of 10 threads for concurrent execution
with concurrent.futures.ThreadPoolExecutor(max_workers=10) as executor:
    queued_futures = []
    # Submit tasks to the worker queue
    for wiki_title in wiki_titles:
        future = executor.submit(wiki_extract_for_title, wiki_title)
        queued_futures.append(future)
    # Wait for all tasks to complete and gather the results
    for future in concurrent.futures.as_completed(queued_futures):
        wiki_extracts.append(future.result())
```

(continues on next page)

(continued from previous page)

```
print(wiki_extracts)
```

CHAPTER

EIGHT

SYSTEM ASPECTS

This module takes a deep dive into the hardware-related aspects of Scientific Computing. As there are many different hardware configurations out there, and also many different types of scientific computing problems, you'll need to understand how the hardware affects your computational performance and vice versa. Adapting your algorithm to the hardware (or even the other way around) will not only optimize run time, but sometimes make the difference between "not computable" to "yeah can be done". We will not be talking about purely algorithmic code optimizations, but rather architectural considerations dictated by the underlying hardware.
Before going *in medias res*, it's worth noting that IT is notorious for its abbreviations. Since not all of them might be familiar to you, we've compiled a *glossary*.

8.1 Anatomy of a scientific computing problem

A scientific computing problem ususally consists of three parts:

8.1.1 The science part

Rather counterintuitively, the science aspects of a problem are often not the hardest. That's not to say that physics is less complicated than IT, but once you've understood and formulated your problem, you probably have an idea of the equations it's governed by. The science is 'clear', but your problems are only about to start..

8.1.2 The computing part

For any non-trivial problem, you'll probably spend most of your time on converting these equations into working code that produces reasonable output. It's also the stage at which you'll encounter the various limitations that are typical for scientific computing: memory shortage, non-converging algorithms, run times measured in decades… Quite often it may be an iterative process in which computational limits will force you to go back to the science part in order to simplify things or remove higher order corrections.

8.1.3 Data analysis

When your code runs and produces output, you'll advance to the analysis stage in order to generate nice plots or apply data mining or statistical methods to harvest your results. Again iterative visits back to the computing or even the science part might be required, for example because the amount of data you've produced is simply too large.

8.2 Computational limits

True to the title of this module, we'll now focus on potential computational limits that might impact your code implementation, but also possibly the data analysis stage.

When you start tackling your scientific computing problem and it looks easy to implement and you expect results in no time, you're either very lucky or you simply missed something and your problems will become apparent later on. Most of the times however, your computing problem will be a real challenge (this is ETH, after all!), in which case your headaches will be caused by one of two things (or both, if you're unlucky):

8.2.1 Implementation or algorithmic challenges

This is pure software engineering. You're having trouble converting your physical equations into code, or the algorithm you come up with doesn't converge. The point here is: no amount of hardware you could throw at it will make the problem go away. You have to solve it in software. The way to successfully address this depends on the very specifics of the underlying physical equations, on the way they're implemented, on the programming language and libraries used, and so on. There is no real general advise here and we'll not even pretend to give any. If you're stuck at this stage, one possible solution might be to talk to our colleagues of ID Scientific IT Services who deal with these issues on a daily basis. They also offer a range of courses and workshops (lecture notes here) that cover some of these topics.

8.2.2 Computational limits induced by hardware

This is the true topic of our module: you've implemented this beautiful algorithm to solve your physics equations, you've optimized it for run-time and memory usage, used the fastest libraries out there, but still you just cannot make it produce the desired output in any reasonable time (or at all), because you run into hardware limits: be it CPU, memory, storage or the amount of interprocess communication required by parallel threads on a cluster - you're being stalled by system aspects.

In order to find possible ways to address this, we need to learn about the different types of hardware limits and their interactions - with each other and with your code.

8.3 The hardware itself

One of the first and most obvious questions when you start working on your scientific computing problem might be: what hardware will you be running it on? For large projects, you might get the chance to buy dedicated hardware for your code. Since there is a wide variety of configurations out there, you should tailor the system to your needs - this is not limited to the amount of CPU cores or memory, but affects the very architecture of the machine. More about this in a minute. In most cases however you'll have to use hardware that is already available - this will require you to understand the architecture of the machine(s) your're dealing with so you can identify computational limits and work around them.

Before taking a look at different hardware architectures, we have to determine another important aspect:

8.4 The "size" of a scientific computing problem

The key question here is:

Does a reasonable implementation of your scientific computing problem fit on one machine in terms of CPU and memory?

"One machine" might be the laptop on your desk, or one cluster node at CSCS, but the answer will fundamentally influence the way you'll have to address the implementation of your scientific computing problem.

8.4.1 Yes, it fits

This is the best-case scenario as it gets rid of most of the headaches that would plague you if the answer was ‘no’. When writing your code, you just have to deal with ‘local’ hardware and not worry much about hardware architectures. You’d still be well advised to optimize your code in order to get results as fast as possible, but this might be as simple as starting multiple threads or processes on the same machine. You might also benefit from GPGPU acceleration if available.
The run-time of your code in this case will be determined by the CPU/GPU performance or the amount of memory available on your machine.

8.4.2 No, it doesn’t fit

This is where things get interesting. You might be able to cheat and return to ‘yes, it fits’ land by making the “one machine” beefier, but often that’s just not possible and you have to deal with a more complicated problem. The next question to answer now is:

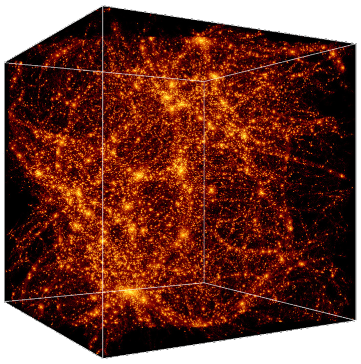
Can you partition the problem into N independent smaller tasks?

Independent in this context means that each task can work on some part of the problem without having to interact with the other tasks.

Yes, that’s possible

In this case you’ll end up with N independent computing tasks that all look like the ‘Yes, it fits on one machine’ case from above. This allows you to run them sequentially on one machine or in parallel on N machines, or anything in between. You’ll suffer from some scheduling overhead from having to spread the tasks across your available hardware and the necessary accounting, but otherwise you can again make full use of the available CPU performance and memory. In terms of hardware, you have several options: you can use one or several ‘regular’ nodes, NUMA nodes, a cluster, or ‘the cloud’. More about those in a minute.
A typical example of this configuration is parameter sweeping in Monte Carlo simulations: you have a huge parameter space to cover with one fixed algorithm, and each run is independent from all others with no communication between them. Just spread those tasks across all hardware available to you, but still make sure you optimize the code as discussed later.

No, communication is required



If your tasks are not independent but need to exchange information, you've reached the most complicated case of distributed scientific computing. You can expect heavy inter-process communication (IPC) costs that will grow with the number of tasks squared.
The showcase example here would be large N-body simulations in which the number of particles require the problem to be distributed across many machine due to memory limitations, but still all particles interact with each other, so constant updates must be shared between nodes.

Now that you've understood the underlying nature of your scientific computing problem, let's finally talk about the different hardware architectures and how they influence your implementation.

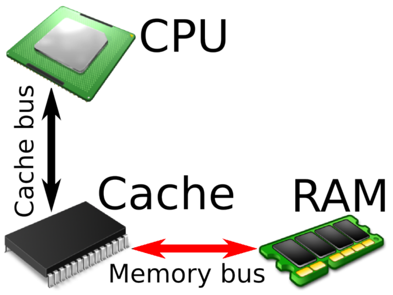
8.5 Hardware architectures and limiting interconnects

Assuming the 'worst case' of an interdependent distributed scientific computing problem that needs inter-task data exchange, we will now try to determine the communication performance bottlenecks on several levels, starting at a single CPU and working our way up to "the cloud".
We will measure the size of a system in **M** *cores*, **N** *CPUs* and **O** *nodes*. The relevant performance metrics are **bandwith** in *bytes / s* and **latency** in *s*.

8.5.1 The CPU

Modern CPUs are true technology marvels. They contain billions of transistors on the area of a thumbnail and perform hundreds of billions of operations per second. They offer between **M** = 2 and 32(ish) cores for multithreading and their power consumption ranges from sub-1 W mobile processors to 200+ W server CPUs. In order to always be able to provide all cores with sufficient instructions and data and never have to wait for comparatively slow main memory (RAM), CPUs usually come with three levels of cache memory. L1, the fastest, but smallest (usually 8-64 kB) one exists for each CPU core separately, while L2 and L3 (a few MB and tens of MB, respectively) are shared across cores. Highly sophisticated algorithms in the CPU microcode (branch prediction, prefetching..) try to always keep relevant data in these caches and cache hit ratios in best case scenarios can be close to 100%. Data in the cache can be accessed at **hundreds of gigabytes per second** and with single-digit **nanosecond latency** - the speed of the cache bus.
In order to harvest the full performance of the CPU, your code has to be multithreaded to keep all cores busy.

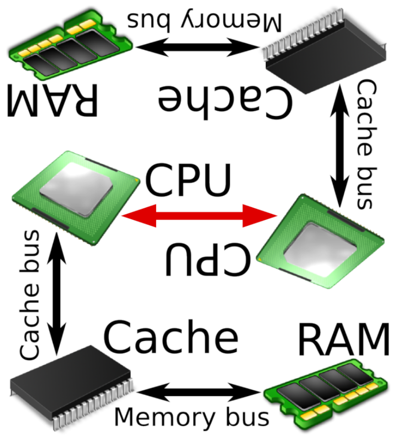
8.5.2 Single-CPU node



While caches are super fast, they are way too small for a typical combination of operating system + program + data, so a ‘real’ computer always has gigabytes of main memory (RAM). In a single-CPU node (**M = 2..32** and **N = 1**), all RAM is connected to the one CPU via the caches. Every memory access that results in a cache miss will be limited by the speed of the memory bus, typically **> 30 GB/s** and **tens of nanoseconds**. So it’s immeditely obvious that for optimal performance, you should keep as much of your code and data as possible in the cache.

Single-CPU nodes are architecturally simple, but they don’t scale: the CPU can only have so many cores (currently at most 32), and it also can only address so much memory (realistically 1 TB). As much as this sounds, in HPC this is often not nearly enough.

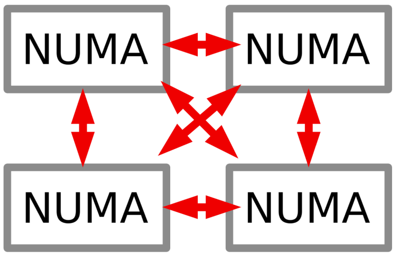
8.5.3 Non-uniform memory architecture



As soon as there’s more than one CPU socket in our machine (there are up to 16-way machines, so **M =2..32** and **N = 2..16**), things get a bit more complicated: each CPU has its own memory (and caches, for that matter), but as the user of the machine, you want to be able to deal with one flat memory model that combines all RAM connected to all CPUs. This can only work in a non-uniform memory architecture (NUMA) setup, since any memory not connected to a specific CPU will only be reachable via a CPU-CPU interconnect like QuickPath/Ultra Path Interconnect (Intel) or HyperTransport/Infinity Fabric (AMD). While these interconnects are fast, they’re once again about one order of magnitude slower (as was the case from cache bus to memory bus) at around **20 GB/s** bandwidth and **hundreds of nanoseconds** latency. On a NUMA machine, you will see all memory and all cores, but performance will be non-uniform.

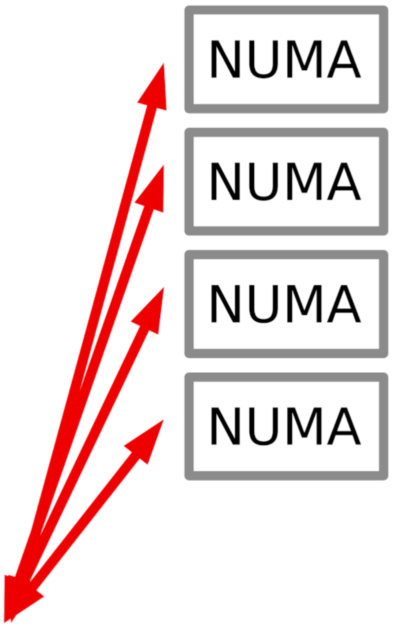
This is as far as we get in “one computer”. If you need even more resources (especially RAM), you have to combine multiple nodes to form a cluster.

8.5.4 Cluster



A cluster introduces yet another level of interconnect: instead of CPUs like in NUMA it connects nodes via network. Another important difference is that now you'll have to deal with multiple operating systems: one on each node (at least they're typically all identical). Since you're trying to keep **M =2..32** cores times **N = 2..16** CPUs times **O = 2..104** nodes busy, it's not sufficient to just multithread, you'll also have to multiprocess (**O** separate OS..) and you'll have to employ some form of inter-process communication (IPC) across nodes unless your tasks are completely independent. Simpler and smaller clusters might use regular ethernet (1G or 10G) networking, but typically HPC clusters are built around a fast Myrinet or Infiniband network. Once again, there's a performance penalty for scaling out onto multiple nodes: we're now at **single-digit GB/s** bandwidth and **microsecond** latency. Two software components usually help you to get the most performance out of a cluster: a **job scheduling system** like TORQUE, slurm or HTCondor will take care of running submitted jobs on all available nodes, while **IPC libraries** like MPI and TIPC will help you to exchange efficient inter-process messages with other nodes in the cluster.

8.5.5 Cloud



While serious HPC almost always happens on a dedicated cluster, there are certain scientific computing workflows that use "the cloud". For example, gene sequencing pipelines in molecular biology might be deployed on a Kubernetes cluster, which is a typical cloud setup.
"The cloud" typically differs from a real HPC cluster in 3 ways:

- usually you'll get a bunch of virtual machines or containers that you can employ to run your jobs instead of 'real' hardware in a cluster
- the network is more off-the-shelf: 10G ethernet, **1 GB/s** and **> 10** μ**s**
- IPC happens on a higher level: REST or SOAP interfaces instead of MPI

Now that you've understood the different system architectures, let's talk about some

8.6 Implications and specialities

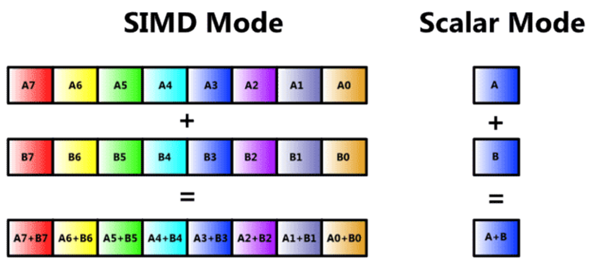
8.6.1 Keeping everything busy

As we've just seen, any modern computer will have a lot of cores. In fact, for the last ~ 15 years, per-core performance hasn't really grown that much, and most speed gains in modern CPUs have come from adding ever more cores. This creates a problem for software engineering in general and HPC in particular: in many cases it's not at all easy to keep those dozens (or in a cluster: thousands) of cores busy all the time. Of course you can run multiple instances of your code or spawn multiple threads within one process, but if these individual threads are not completely indepedent but need to exchange data, you'll suffer from IPC penalties that scale with the square of the number of threads involved. It therefore pays off in the long run to put some thought into the layout of your code, the data structures used and how to exchange them early on.

8.6.2 Parallelism vs IPC costs

One aspect to keep an eye on is your level of parallelism: the smaller your individual thread is, the easier it'll be to keep all cores on all nodes occupied as you'll have more threads to work with. IPC costs however will be higher since more communication partners are involved. You might have to experiment to find the best granularity for your system.

8.6.3 SIMD



SIMD stands for *single instruction, multiple data* and means that you're performing an operation not just on one variable, but a whole set of them. Typical examples are vector or matrix operations. If your code executes a lot of those, you might benefit greatly from

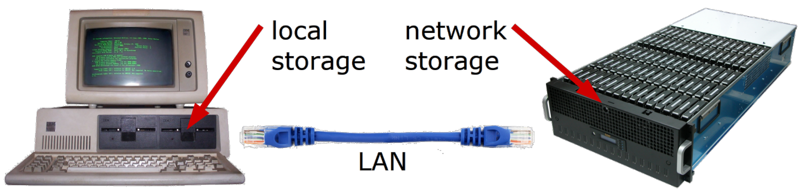
GPGPU

General-purpose graphics processing units denote graphics cards being used for general computation. Graphics cards owe their impressive 3D performance to their vast number (thousands) of execution units that can run a limited set of instructions extremely fast. If the calculations in your code can be mapped onto this model, you can profit from impressive speed gains (100x fold or more). GPGPU is not applicable to all scientific computing problems and has downsides too (e.g. limited onboard memory, limited instruction set), but it's definitely something to keep in mind and play around with. There's NVIDIA's powerful CUDA framework and the more general OpenCL to get you started.

AVX

In recent years, also CPUs (both Intel and AMD) have gained SIMD capabilities: AVX (Advanced Vector Extensions). You'll have to consult your compiler documentation for the corresponding compile flags.

8.7 Storage



We've talked a lot about CPUs, memory, threads and so on, but now it's time to address another part in every scientific computing system: storage. After all, what good are all your beautiful results if you have nowhere to store them? Typically you'll find two types of storage:

8.7.1 Local storage

Local storage is directly built into your PC or node in the form of hard drives or SSDs. It's usually only meant to be used on this very machine, even though it might be shared in some cases. It's typically referred to as **scratch** and is supposed to be used as a temporary place for your data while the calculation is ongoing - often, there's no backup for scratch. In most cases, it's not very large (GB to a few TB), but pretty fast (esp. for SSDs).

8.7.2 Network storage

Additionally you'll usually find a much larger, permanent storage space mounted over the network. This is accessible from all nodes, there's backup, but it comes at a cost: since it's a network file system (NFS or a cluster file system), it's usually slower than local scratch. As soon as your data has reached any semi-permanent state, it should probably go onto network storage.

8.7.3 Storage as a bottleneck

While it is true that it might be possible to simply generate too much data (especially in simulations, often by mistake..), in most cases storage **capacity** will not be a major limit to your scientific computing endeavours - storage is well understood and it's not too difficult to have enough of it. The same is not true for **IOPS** however. It's trivially possible for one process to saturate the I/O capabilities of a given machine, no matter how fast. This is especially true for network storage, but even local HDD storage can only handle so many IOPS. In the worst case, your code will bring the whole I/O subsystem of the machine to a halt, effectively rendering it unusable for you and everybody else. The I/O cascade of hell goes something like this:

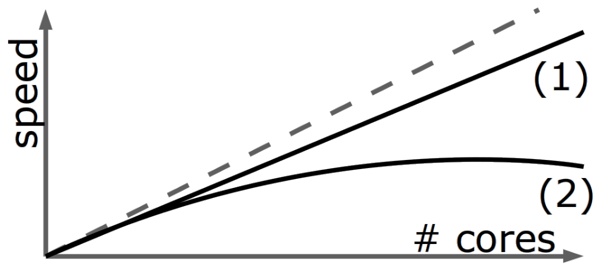
everybody happy < sequential read OR write < sequential read+write < random read OR write < random read+write < random read+write from many threads < machine kaput

Sequential means that you're reading or writing chunks of reasonable size (hundreds of kB to a few MB) at a time while *random* I/O wildly jumps around in a file and reads/writes very small amounts of data. Random I/O creates a high number of IOPS which will kill any storage subsystem, especially if it's HDD based.

So in order to get the best performance out of your code and not render the hardware unusable for everyone else, you should make sure to

- always run your calculations on local scratch, then copy the results to network storage when finished
- open as few files as possible and keep them open as long as possible
- read and or write contiguous chunks of data instead of jumping around in those files

8.8 Limits revisited



With all this in mind, it's time to come back to our original question "*Can you partition the problem into N independent smaller tasks?*" in order to identify the probable bottleneck for your problem and get an idea of how well it will scale.

8.8.1 Yes, independent tasks

As we've seen before when we talked about bus speeds, in this case the follow-up question is: "*Does each task (code + data) fit into the L1-L3 caches?*". If this is true, your performance will likely be limited by the raw processing power ("**CPU bound**"). A CPU with faster cores (often: more GHz) would make things faster. If false, your bottleneck might turn out to be the memory bus ("**memory bound**"). If you can't optimize your problem to fit (better) into the cache(s), faster memory could help (but the speedup is quite limited here).

In both these situations, your overall performance across many cores on many nodes will probably scale linearly with the number of cores involved (line 1 in the graph above). Due to all sorts of 'losses' (scheduling, network file systems..) your gradient might not reach 1, but adding more cores will give you more performance.

8.8.2 No, I have communication between my tasks

As discussed previously, if your tasks need to communicate, you'll encounter IPC performance penalties, and your bottleneck will be the communication overhead and the interconnect (at least for larger numbers of cores). The overall performance of your system will most likely scale like line 2 in the graph: for a certain number of cores your performance gain will saturate, and might turn negative if going even further.

8.9 The software side

In the previous chapters we've learned how the underlying hardware architecture will influence your code's performance. Now let's turn to the software side of things. First:

8.9.1 Programming languages

There's an almost innumerable variety of programming languages out there, and most of them can be (and probably have been) used for scientific computing. Some have been around for more than 60 years (Fortran) while others are pretty new (Julia). They all have their strengths, weaknesses and specialties, and the choice of programming language (and the accompanying libraries) *will* influence the progress of your project in at least two ways: development time and final performance. In some cases you simply won't have a say in the choice of the language to be used, but if you do, it's important to know what to look out for. Here are some guidelines:

What may you use?

- not everything might be available on the target machine
- there might be libraries or existing code you want or have to use
- you might need to collaborate with others

What should you use?

- which language would be the fastest in terms of
  - development time
  - execution speed
- have (parts of) your problems already been solved in some code or library in one language or other?

What can you use?

- Which language(s) are you familiar with?
- How fast can you learn (and master) another one?

Compiled vs interpreted languages

Above all other dichotomies (statically vs dynamically typed, procedural vs functional, object oriented vs structured) the one between a compiled and an interpreted language will probably have the largest impact on your scientific computing project.

Interpreter languages

Interpreter languages (also called scripting languages, examples include Python, PHP, Perl..) originally used the following workflow: each time you 'run' such a program, the operating system starts the appropriate interpreter which executes your script line by line. No executable file will be created on disk. This allows for 'rapid prototyping' programming since you can immediately test each small change in the program. Downside: execution isn't very fast, since the interpreter is basically going through the problem step by step. In a nutshell: "fast coding, slow code".

Compiler languages

Classic compiled languages like C/C++, Rust, Fortran, Go etc require a dedicated compile run each time you've changed your code and want to test it. The compiler compiles and links the code and generates an executable file on disk that you can start. Compared to interpreter languages, this requires an additional step before you see the result of your code changes. On the upside, the compiler has a lot more possibilities to optimize execution, so compiled programs usually run faster: "slow coding, fast code".

In recent years interpreter languages have changed a lot and nowadays usually don't get interpreted line by line, but are compiled on the fly into an intermediate bytecode that also offers most of the performance benefits of a compiled language. Furthermore, some compiler languages (most obvious example: Java) also compile into bytecode, so the gap between compiled and interpreted languages is getting narrower. The latter still have the advantage of easier handling and faster turnaround though.

8.9.2 Optimizations

Aside from picking the "right" language for the job, there are a lot of other factors that can dramatically increase the performance of your code. Here are some ideas:

- pick the right compiler (gcc, llvm, Intel, PGI..) and the compile flags best matching your problem. This can easily yield a factor of 2 or even more
- use optimized libraries (BLAS, LAPACK, BOOST, NumPy..). For most interpreter languages you'll find bindings or wrappers for those highly optimized C libraries that speed up HPC tasks tremendously.
- use data types that reflect your problem (e.g. bit fields for Ising spin glasses, single vs double precision floats…)
- think about cache + memory alignment and locality
- use a profiler to identify code hot spots
- if you have access to the hardware and OS, both can be tuned for specific workloads
- individual threads can be pinned to specific cores
- pick sensible file formats (HDF, packed binary, XML, CSV…)
- think about what you need to write to disk, how often, how many files

8.10 Tips, hints and anecdotes

We'll end this module with a small collection of tips and real-life stories from ISG's stock of experience.

- use checkpointing to prevent data loss in case of a crash, power outage or network problem.
- good random numbers are surprisingly hard to generate - especially if you need lots of them.
- use job queuing systems if available. It'll make your and other people's lives easier.
- never ever swap, ever! Disks are so much slower than memory that you'll completely kill your performance. The situation is a bit better for SSDs, but you should still avoid it if you can.
- once again: make sure you're using all available cores.
- if you have to use network storage for long-running jobs (even though you really shouldn't), be aware of network interruptions.
- on ISG's behalf: please don't move / rename a 10T folder without telling us. It'll create headaches for our backup system.
- on ISG's behalf: do not create millions of small files on the file server. This will waste a lot of space and kill performance. Create tgz or zip archives instead.

8.11 Web content

There is a nice interactive Pluto notebook that lets you explore many of the topics of this module.

8.12 Glossary

Abbreviation	Explanation
CPU	Central Processing Unit $\rightarrow$ the physical processor in a computer
Core	individual execution unit within a CPU
Process	one ‘program’ running on a computer
Thread	sequence of instructions that may execute in parallel with others, within one process
Cache	small + fast memory in or close to CPU
Interconnect	data bus between two units
Throughput	amount of data transferred per time unit
Latency	packet delay time
GPGPU	General-purpose graphics processing unit, use graphics cards for computation
NUMA	non-uniform memory access, one way of building multi-CPU computers
IPC	inter-process communication
IOPS	I/O operations per second