

Università di Pisa—Dipartimento di Fisica "E. Fermi"

Computing Methods for Experimental Physics and Data Analysis

Lecture Notes for the Basic Module

Luca Baldini

<https://codeberg.org/lbaldini/cmepda-basic>

Table of contents

Preface	6
License	7
You are free to:	7
Under the following terms:	7
Notices:	8
1 Introduction	9
1.1 Reproducibility	9
1.2 More ranting	10
1.3 Developing software collaboratively	11
1.4 And so what?	12
2 Python basics	13
2.1 Coding conventions	14
2.2 Python basic types	15
2.2.1 Numeric types	16
2.2.2 Strings and string formatting	17
2.2.3 Containers	18
2.2.4 More fun with types	19
2.3 Control flow	20
2.3.1 Two types of comparisons	22
2.4 Functions	23
2.4.1 Default argument values	24
2.4.2 Variadic functions	25
2.4.3 Keyword arguments	28
2.4.4 Type annotations	29
2.5 The Python import system	30
2.5.1 Import basics	30
2.5.2 Import aliases	32
2.5.3 Importing your own modules	33
2.6 The Python standard library	34
2.6.1 Beyond the standard lib	34

3	Version control	36
3.1	Version control basics	36
3.1.1	Centralized and distributed systems	37
3.1.2	Versioning single files or the entire repository?	39
3.2	Hosting a Git repository	40
3.2.1	Self-hosting or not?	40
3.2.2	GitHub	41
3.2.3	Codeberg	41
3.2.4	Digression: exchanging ssh keys	42
3.3	Daily workflow	43
3.3.1	More nomenclature	43
3.3.2	Git overview	43
3.3.3	Top-level good habits	45
3.3.4	Beginning your day	45
3.3.5	Getting started on the next, great feature	46
3.3.6	Doing the actual work	47
3.3.7	Merging changes	48
3.3.8	Resolving conflicts	49
3.3.9	Tagging the repository	50
3.3.10	What shall I track in the first place?	50
3.4	Ancillary tools	51
3.4.1	Issue tracking	51
3.4.2	Actions	52
3.4.3	Forking	52
4	Python packaging	54
5	Algorithms (sort of)	55
5.1	The Euclidean algorithm	55
5.1.1	How many iterations?	56
5.2	Asymptotic behavior and complexity	58
5.2.1	Big-O notation	59
5.2.2	Big-O in practice	61
5.3	Sequential vs. binary search	61
5.4	Sorting	65
5.5	Data structures	67
5.5.1	Lists	67
5.5.2	Dictionaries (aka hash tables)	69
6	The Python scientific stack	72
6.1	NumPy	72
6.1.1	Array and native Python types	73
6.1.2	Indexing and slicing	76

6.1.3	Broadcasting	78
6.1.4	Vectorization	80
6.2	SciPy	86
6.2.1	Interpolating splines	86
6.3	pandas	89
7	Object oriented programming	90
8	Advanced Python	91
8.1	Error handling	91
8.2	Iterators and iterable	91
8.3	Generators	91
8.4	Decorators	91
8.5	Miscellanea	91
8.5.1	Dataclasses	91
9	Testing	92
9.1	Unit testing	92
9.1.1	Unit testing basics	93
9.1.2	Unit testing in practice	94
9.2	Static code analysis	95
10	Documentation	98
10.1	Sphinx	99
10.1.1	General documentation	99
10.1.2	Documenting the interfaces	100
10.1.3	Compiling the documentation	101
10.2	Making the documentation available.	101
10.3	A concrete example	101
11	Optimization	102
	Appendices	103
A	Floating-point arithmetic	103
A.1	Base 2 numeral system	103
A.2	Floating-point representation	104
A.2.1	A simple example	105
A.3	Back to the problem	106
A.4	Properties of floating-point arithmetic	107
A.4.1	Range and resolution	107
A.4.2	More on arithmetics	109
A.4.3	Floating-point comparisons	110

A.5	What did we learn?	111
B	Assignments	112
B.1	Version control—assignment 1	112
B.2	Basic Python—assignment 1	112
B.3	Basic Python—assignment 2	113
B.4	Basic Python—assignment 3	113
B.5	Packaging—assignment 1	114
B.6	Advanced Python—assignment 1	114
C	Contributing	116
C.1	Note for agents	116
D	Acknowledgements	117
E	Use of AI for these notes	118
F	Release history	119
0.9.2	(2026-10-09)	119
0.9.1	(2026-10-05)	119
0.9.0	(2026-10-04)	119
0.8.1	(2026-10-02)	119
0.8.0	(2026-09-30)	119
0.7.0	(2026-09-30)	119
0.6.0	(2026-09-30)	120
0.5.0	(2026-09-23)	120
0.4.0	(2026-09-21)	120

Preface

Version 0.9.2 ([history](#))

These lecture notes are meant to cover the basic module of the *Computing Methods for Experimental Physics and Data Analysis* course.

I am in the process of migrating here, and expanding, the material traditionally hosted in the form of slides at the [github repository](#) of the course.

I plan on completing the migration by the end of the 2026/27 academic year. Up to that point these notes should be considered as a rough draft.

Found a typo, or have a good suggestion about making these notes better? Have a look at our guidelines for [contributing](#)!

License

All the material in this work is distributed under the *Creative Commons Attribution-ShareAlike 4.0 International (CC BY-SA 4.0)* license.



© 2026 Luca Baldini

You are free to:

Share — copy and redistribute the material in any medium or format for any purpose, even commercially.

Adapt — remix, transform, and build upon the material for any purpose, even commercially.

The licensor cannot revoke these freedoms as long as you follow the license terms.

Under the following terms:

Attribution — You must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.

ShareAlike — If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original.

No additional restrictions — You may not apply legal terms or technological measures that legally restrict others from doing anything the license permits.

Notices:

You do not have to comply with the license for elements of the material in the public domain or where your use is permitted by an applicable exception or limitation.

No warranties are given. The license may not give you all of the permissions necessary for your intended use. For example, other rights such as publicity, privacy, or moral rights may limit how you use the material.

1 Introduction

I will start with anecdote—an email thread that has actually happened in real file, in which I am only replacing one of the names for preserving the privacy of the persons involved.

Luca: good morning, Dave! Would you mind sharing the exact model parameters for your nice paper about [Cyg-X1](#)? And, if at all possible, the entire [XSPEC](#) session for the spectral analysis, so that I can re-run it? I am trying to simulate the source for a data analysis challenge I am preparing for the [IXPE](#) collaboration and I would like to start from the state of the art.

David: sure thing! The parameters are in the paper!

Luca: well, I have *read* the paper. Not all the relevant parameters are listed in table 2, and I am unable to repeat the analysis at this time.

David: oh, well... unfortunately *that* particular analysis was done on my old laptop, and I had to give it back to my previous employer, when I changed institution a couple of years back. I have no more access to that material.

David: ah! And, by the way, a couple of the fit parameters were running all the way up to their boundaries, so I would not trust the fit too much...

1.1 Reproducibility

Reproducible: (adjective) *able to be shown, done, or made again*. Source: the [Cambridge Dictionary](#)

Pretty simple, eh? Most of us would agree that research should be correct *and* reproducible. You don't just want to get things right the first time. You also want to be able to get them right *again* the second time around—be that tomorrow, a year or a decade from now. You also want your fellow colleagues scientists to be able to reproduce what you published, should they be interested in doing so.

Reproducible research is data analysis that arrives to the same answer when starting from the same raw data.

The fact is, often times physicists are better at correctness than reproducibility. There are entire communities (which I will refrain from naming) that are lagging behind on the best

practices that are ubiquitous among professional software developers. This, in turns, tend to result in poor-quality software.

While there are many reasons why that is actually happening—none of us is a software engineer by training, and it is totally legitimate for us to focus on science rather than software—but we argue that software plays a paramount role in modern experimental physics, and is therefore a crucial ingredient in reproducibility of research. It should be developed under controlled conditions, just like the experiments are performed.

This is what these notes are about. It will be boring all along the way, but if you do pay attention and are willing to entertain the notion that this might be actually useful in the long term... well, you might be surprised!

1.2 More ranting

Most certainly, all of you have written code in their life—no matter in which language or how complex that was. Now, think about that for a second, and try answering a few simple questions.

- Would somebody else able to use it without your help?
- Would it run as is on somebody else's computer?
- Would you be able to adapt it for somebody else's computer?
- Would *you* be able to run it on your computer now?

If the answer to any of those questions is no, that doesn't necessarily mean that something went horribly wrong, but you shall agree that things might have probably gone better than they did.

(And note we are just ignoring the *efficiency* of your code, at this time. Could your software run significantly faster, and would you care if it did? We shall come back on this one, and learn that *premature optimization is the root of all evil*, but just to be clear: we are not even asking if the code you produces was optimal in any sense—just if you think you could get it to do the same thing twice. Not a very high bar.)

And now take a quick look at the last paper that you have read. (You do read papers, do you?) Do you think that, given enough time, it would be possible to re-run the same analysis and obtain *exactly* the same results? Note this implies a number of things, none of which is immediately obvious...

- Have the raw data been preserved in a readable format?
- Is the analysis software still available?
- Is there still somebody around understanding what the software does?
- Are there machines around that can run the analysis software?

Spoiler alert: you would be *shocked* in many cases.

Due to a number of coincidences I cannot even explain, I happen to be a high-energy astrophysicist, and I know first-hand that NASA makes a deliberate effort for all the data from the high-energy mission to continue to be readable across the decades. This is largely achieved, at least in principle, via the [FITS standard](#) and the [HEASARC](#) archive of analysis software and calibration products.

Is this a good thing or a bad thing? Well, the intent is admirable—although being anchored to a standard that was cast in stone in 1993, and builds upon software and formats that were already decades old at that time does not exactly strike as something that would look anywhere close to high-performance computing in 2026. (Is it only me or the [FITS standard](#) page still has that distinct nineties look and feel?)

But even letting that go for a second, this brings us back to the anecdote at the beginning of the chapter: if we do not proactively seek reproducibility in our day-to-day workflow, there is very little that standards can do!

1.3 Developing software collaboratively

Most modern high-energy physics experiments are run by large teams—the big LHC collaborations count thousands of members. This immediately brings up a number of difficult questions. For instance:

- How do you handle development in such an environment?
- How do you distribute the source code?
- How do you know which version is any single person running?
- How do hundreds of people develop the same code concurrently?
- How do you make sure that your last awesome change isn't breaking everybody else's code?

Now, depending on what you decide to do for a living, you might or not end up in a big collaboration. You might work in a small team, instead. The thing is: the scale of the challenges will be different, but their *nature* will be pretty much the same. At the end of the day you will have to decide how you exchange code with your collaborators, how you collect feedback, how you distribute updates—in a sentence how you *keep track of the development*.

(If you bring this to its extreme consequences, you will immediately realize that in fact all of the above still holds even when you develop software *by yourself*.)

1.4 And so what?

All right. Lots of hard questions. How do we fix all of this?

Well, the first thing is a terribly boring thing most likely you have never heard of: *version control*. Much of these notes is directly connected, one way or the other, to version control. Hopefully when you get to the last page you will have *everything* under version control—from your master thesis down to your shopping list.

But before we dive into the matter we have to make sure we have something to version control, which is why in the next [chapter](#) we shall take a brief detour wondering into the magic of Python.

2 Python basics

Let's make it immediately clear what this chapter is and what isn't.

This is *not* a fully-fledged, self consistent Python course—there are so many good ones on the web that coming up with a new one would be just like solving a problem that did not exist in the first place. If anything, this chapter is a semi-random collection of things that you might stumble across as you work your way through becoming proficient in developing scientific software.

For each and single one of those thing, first of all you have to *acknowledge* its existence. By the end of the semester you will not qualify as a software engineer by any reasonable metrics. Hopefully, though, as more and more of the things we touch upon will come biting at you, your reaction will be: “hey, wait a moment—I’ve heard this one before”. And that’s where the real study will begin, with patience and practice.

Over the course of this notes we shall see tons of snippets of (mostly Python) code. Each one comes with a small copy button on the top right that you are encourage to use. Copy and past them in your favorite text editor, change them, play with them and develop the familiarity that you need in order to be able to use the different techniques that they illustrate whenever you need them.

And, since we are at it, here is the first one.

```
import this
```

The Zen of Python, by Tim Peters

```
Beautiful is better than ugly.  
Explicit is better than implicit.  
Simple is better than complex.  
Complex is better than complicated.  
Flat is better than nested.  
Sparse is better than dense.  
Readability counts.  
Special cases aren't special enough to break the rules.  
Although practicality beats purity.  
Errors should never pass silently.
```

Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than **right** now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!

Take a second to go through all the sentence in this easter egg originally devised by early Python adopter and developer [Tim Peters](#). It is very possible that any of these make much sense to you, now; by the end of the course you should be able to get something out of each one, and laugh out loud to at least a few of them.

2.1 Coding conventions

Oh. My. God. This has been going on for several pages, already, and we have not seen anything useful, yet. Shouldn't we be concerned with variables, functions and all of that?

You were warned: this is not a Python course for beginners in the usual sense. Of course we will get to the good stuff in due time, but remember our goal, here, is to develop the necessary habits to write decent code—code that you won't necessarily have to throw away tomorrow—rather than just code that *does something*. And, since we are at it, I have to tell you: you will have to survive in a hostile environment, where your supervisor will come to your office every other day asking for something important (the most important thing on the face of the Earth, in fact) to be completed by the end of the day. When that happens make sure you have clear in your heart that every shortcut you take in order to speed up the process will come hunt you down sooner or later. In the long term, it will be a net negative. (Whether you can convince your supervisor of that is a totally different story.)

Although never is often better than right now. Rings a bell?

Now back to the coding conventions. Coding conventions are *guidelines about how to write code* and, it goes without saying, they are different for different languages and, in many cases, different across different teams. Look at this small snippet of code.

```
width = 3.  
Height =4.  
aspect_ratio = width / Height  
RectangleArea=width*Height
```

```
print(f"Aspect ratio: {aspect_ratio}")
print(f"Area: {RectangleArea}")
```

```
Aspect ratio: 0.75
Area: 12.0
```

If by the end of the semester looking at this is not physically hurtful, something has gone wrong. Look carefully:

- `width` is lowercase, and `Height` is uppercase;
- `aspect_ratio` is *snake case*, while `RectangleArea` is *camel case*;
- the spacing around the `=` operator is inconsistent.

And yet the things runs correctly. This is the bad things about coding conventions: *you are encouraged to stick to them, but your code will happily run if you don't*. Then why should one care, you are asking? Well, this stems from one the insights popularized by Python creator and former BDFL (benevolent dictator for life) [Guido van Rossum](#)—namely: code is read much more often than it is written. In other words: it makes sense to spend time crafting the code while you are writing, because that will benefit all the persons that will be reading the code in the years to come.

Readability counts. (And there it is, the Zen of Python again.)

The one-line summary, here, is: think about it but don't be obsessed by it The bible of coding conventions for Python code is the Python enhancement proposal [PEP 8](#). Print a copy and keep it under your pillow.

By the way: there are automatic tools out there, called *linters*, to help you but we shall get back to that in due time.

2.2 Python basic types

If you have ever programmed in a compiled language such as C or rust, Python is different in many respects. For one thing, you don't have to declare variables, as the type is inferred at the time of assignment based on the value being assigned. Python offers a number of native type, among which the most common are

- integers;
- floating-point numbers;
- lists and tuples;
- dictionaries.

```

i = 3
x = 3.0
s = "Hi there!"
l = [1, 2, "a string"]
t = (1, 2, "a string")
d = {"key1": 1, "key2": 2}

for item in (i, x, s, l, t, d):
    print(item, type(item))

```

```

3 <class 'int'>
3.0 <class 'float'>
Hi there! <class 'str'>
[1, 2, 'a string'] <class 'list'>
(1, 2, 'a string') <class 'tuple'>
{'key1': 1, 'key2': 2} <class 'dict'>

```

Also, unlike C, variables can change type in the lifetime of a program. Despite that, every variable has a unique and well defined type at each point in the execution.

```

a = 11
print(a, type(a))
a = "Hello!"
print(a, type(a))

```

```

11 <class 'int'>
Hello! <class 'str'>

```

2.2.1 Numeric types

The most interesting fact about numeric types in Python is perhaps the fact that, unlike most languages, integers come with *infinite precision*. In practice this means that their footprint in memory is not pre-determined in size (e.g., 8, 16, 32 or 64 bits) but can grow automatically as the RAM memory permits, depending on the actual value that the variable is storing—no need to worry about overflows.

Small piece of advice: do not get used to that, as this is a luxury that you can hardly afford in the vast majority of other languages. Actually, when we shall start operating with numpy arrays, we shall see that we are thrown right back into the fixed-size arithmetic hell. We shall discuss the pros and cons when we get there.

The floating-point arithmetic is such an important subject for us, that we shall defer its discussion to a separate [chapter](#).

2.2.2 Strings and string formatting

Technically speaking, strings are *immutable sequences of unicode code points*. (If any of this is unclear to you, do your research and get up to speed, as strings are fairly common in many different settings.)

One of the things that, sooner or later you will have to master is how you actually *format* strings. From this standpoint, Python does not have a particularly straightforward story to tell, as there are at least four different ways to achieve the same things.

```
name = "Luca"
age = 42

# The ugly way.
print("My name is " + name + " I am " + str(age) + " year(s) old.")

# The old way (% operator)
print("My name is %s I am %d year(s) old." % (name, age))

# The new way (.format)
print("My name is {} I am {} year(s) old.".format(name, age))

# The newer way---new in Python 3.6. This is awesome!
print(f"My name is {name} I am {age} year(s) old.")
```

```
My name is Luca I am 42 year(s) old.
My name is Luca I am 42 year(s) old.
My name is Luca I am 42 year(s) old.
My name is Luca I am 42 year(s) old.
```

There should be one— and preferably only one —obvious way to do it. This is very un-Pythonic. Also, this suddenly reminds me that I have been teaching this class for seven years and I am now 49; damned.

What do I have to commit to memory in all this? Simple: use [f-strings](#) whenever possible, which is in Python 3.6 and later, which should be pretty much in any Python installation these days. And, of course, there is a lot of legacy Python code out there, so chances that you might stumble across any of the old variance, sooner or later, are fairly high.

2.2.3 Containers

Containers, as the name suggests, is a data structure meant to hold other objects. As usual Python is fairly flexible when it comes to containers—they typically can include any kind of objects (with a few limitations we shall discuss in due time) and they don't even have to be homogeneous, as we have seen already.

(This, again, is at odds with numpy arrays. As we shall see, in loose terms, we pay this extra flexibility in speed.)

When it comes to the container zoology, the most important top-level difference is between *sequences* and *mappings*. Lists and tuples are sequences: their elements are identified by position, starting from 0. Dictionaries are mappings: their values are identified by keys. They all preserve the insertion order, and you access the elements with the `[]` syntax.

```
l = [1, 2, 3]
t = (1, 2, 3)
d = {"a": 1, "b": 2, "c": 3}

print(l[1])
print(t[1])
print(d["a"])
```

```
2
2
1
```

(Stop for a second and think hard what the relevant use cases might be for each one of them. What would you use for a series of measurements from a data acquisition system? And for a phone book?)

If you are wondering what the difference is between a list and a tuple... easy: the former is *mutable*, while the latter is *immutable*.

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

l[0] = 12
print(l)

t = (1, 2, 3)
print(t)
```

```
# Ops... this causes a traceback!  
t[0] = 12
```

```
[1, 2, 3]  
[12, 2, 3]  
(1, 2, 3)
```

TypeError: 'tuple' object does not support item assignment

```
-----  
TypeError                                Traceback (most recent call last)  
Cell In[7], line 11  
      8 print(t)  
     10 # Ops... this causes a traceback!  
--> 11 t[0] = 12  
TypeError: 'tuple' object does not support item assignment
```

2.2.4 More fun with types

There's a couple more Python basic type we should mention. The first one is the boolean type that pops out when we compare objects

```
print(3 == 3, type(3 == 3))  
print(3 == 4, type(3 == 4))
```

```
True <class 'bool'>  
False <class 'bool'>
```

The other is the None singleton. It does absolutely nothing, it does not compare equal to anything other than itself, and does not support any specific operation.

```
print(None, type(None))  
print(None == "None")  
print(None == False)  
print(None == 0)
```

```
None <class 'NoneType'>  
False  
False  
False
```

Dumb as it seems, it is ubiquitous in Python code, mainly used a sentinel value for function arguments that have no other obvious default, or as a return value for functions that do not actually return anything.

2.3 Control flow

Although we have to go super-fast on this one, there's a few interesting bits of Python control flow that are worth mentioning.

At the tree-level, Python offers the same basic control mechanisms that most of the other languages do offer: the `for` loop, the `while` loop and the `if` statement.

```
i = 2

# Conditional expressions
if i == 2:
    print("Apple")
elif i == 3:
    print("Peach")
else:
    print("Cheese")

# For loops
for i in [1, 2, 3]:
    print(i)

i = 4
# While loops
while i != 0:
    print(i)
    i -= 1
```

```
Apple
1
2
3
4
3
2
1
```

Quite often, you will find `for` loop iterating over the `range()`.

```
for i in range(3):  
    print(i)
```

```
0  
1  
2
```

This is where you should start paying attention, because the semantic of the `for` loop in Python is not necessarily what you would expect if you come from a different language, e.g.,

```
for i in range(10):  
    print(i)  
    if i == 5:  
        i = 8
```

```
0  
1  
2  
3  
4  
5  
6  
7  
8  
9
```

And, since we are now proficient with the Python container, we can mix and match them with our control flow facilities in a few interesting ways:

```
list1 = ["a", "b", "c"]  
list2 = [10, 11, 12]  
  
# Horrible (and very un-Pythonic, too)!  
for i in range(len(list1)):  
    print(i, list1[i])  
  
# Nice-looking.  
for i, item in enumerate(list1):  
    print(i, item)
```

```
# Zipping iterables
for item1, item2 in zip(list1, list2):
    print(item1, item2)

# List comprehension---this works with dictionaries too.
print([x**2 for x in list2])
```

```
0 a
1 b
2 c
0 a
1 b
2 c
a 10
b 11
c 12
[100, 121, 144]
```

Boy, that was quick.

2.3.1 Two types of comparisons

We have seen lots of comparisons so far, in this chapter; we have used the `==` operator every single time. Now, if you read enough Python you will see instances of

```
a = None

if a is None:
    print("Busted!")
```

Busted!

Wondering why that is not spelled as

```
a = None

if a == None:
    print("Busted!")
```

Busted!

Well, it turns out `==` and `is` test two different things: the first will tell you if two objects *compare* equal, while the second will tell you if two objects are really the *same* object. What does it mean to be the same object, you ask? Well, it means to point to the same address in memory, as you can easily find out yourself using the `id()` built-in function.

```
a = [1, 2, 3]
b = [1, 2, 3]

print(id(a), id(b))
print(a == b)
print(a is b)
```

```
140476678679168 140476678684096
```

```
True
```

```
False
```

This also explains why `is` is more idiomatic to test whether an object is `None`: the latter is a singleton in Python, that is, there exists one and only one `None` object within the scope of a Python program. If something compares equal to `None`, then it is *the only* `None` that exists.

2.4 Functions

DRY (Don't Repeat Yourself) is better than WET (Write Every Time), so computer science go. When you find yourself doing the same thing over and over again (that is: more than once) in a program, it is *always* best to encapsulate that thing into a function for re-usability.

In their simplest form, Python functions are defined with the `def` keyword.

```
def square(x):
    return x * x

print(square(2.))
```

4.0

Let's take this chance to build a little bit of common nomenclature. In this case `square` is the *name* of the function; `x` is the first (and only) *argument*; function can return one or more values via the `return` statement. (If no such statement is explicitly present, the function returns `None`.)

Generally speaking, the use of functions comes with a few major advantages:

- you end up with less lines of code;
- if there is a bug in your function, you have one single place to fix, not multiple;
- it is easier to convey the intent of what we are doing.

The last point was probably not obvious in the first, simple example, but it becomes clear as soon as the body of the functions stops being trivial:

```
import math

def cartesian_to_polar(x, y):
    r = math.sqrt(x**2. + y**2.)
    phi = math.atan2(y, x)
    return r, phi

print(cartesian_to_polar(1., 1.))
```

```
(1.4142135623730951, 0.7853981633974483)
```

Function definition in Python is explained in all the gory details [here](#), but a cursory look at some of the more important facts is worth a few minutes.

2.4.1 Default argument values

Each argument can be provided a *default value*, the contract being: if we do not provide a value for a given argument, the default is used.

```
def square(x=1.):
    return x * x

print(square(2.))
print(square())
```

```
4.0
1.0
```

All the arguments we have seen so far are called *positional* arguments, as their meaning is determined by the position in the function signature. There is a catch to keep in mind: all the argument without default values must come before any argument with a default value—this works just fine

```
def power(x, exponent=2):  
    return x**exponent  
  
print(power(2., 2.))  
print(power(2.))  
print(power(2., 3.))
```

```
4.0  
4.0  
8.0
```

while this one doesn't.

```
def power(x=2., exponent):  
    return x**exponent
```

```
SyntaxError: parameter without a default follows parameter with a default (606433495.py, line  
Cell In[21], line 1  
    def power(x=2., exponent):  
        ^  
SyntaxError: parameter without a default follows parameter with a default
```

2.4.2 Variadic functions

It doesn't take too much familiarity with Python to realize that some functions are *special* in that accept an arbitrary number of arguments. Take the builtin `print()` function, for instance:

```
print("a")  
print("a", "b")  
print("a", "b", "c")
```

```
a  
a b  
a b c
```

(We could go on forever, but you got the point.)

What is it that makes the magic in this case, and how can I achieve that with a user-defined function? Well, first of all these functions are called *variadic* functions, and implementing this behavior in Python is significantly easier than in other languages—all you really need is the magic of the star

```
def add(*values):
    total = 0.
    for value in values:
        total += value
    return total

print(add(1.))
print(add(1., 2.))
print(add(1., 2., 3.))
```

```
1.0
3.0
6.0
```

If you paid attention you might have noticed that what we have done here is a poor man equivalent of the `sum()` builtin, and this is not something that would ever want to do in real life, but the point is: you prepend a `*` to your argument name and all of a sudden you can provide how many arguments as you want. Nice, isn't it?

You can get a clearer picture of what happens in the body of your variadic function by a printing experiment

```
def func(*values):
    print(values)

func(1.)
func(1., 2.)
func(1., 2., 3.)
```

```
(1.0,)
(1.0, 2.0)
(1.0, 2.0, 3.0)
```

This is easier than anticipated: the `*` in the function definition collects all positional arguments into a tuple that you can use in the body of the function itself.

Interestingly enough, the magic of the `*` also works in the other direction—that is you can *unpack* an iterable in place within a function call using essentially the same syntax. This is something that most likely you have seen when fitting with `scipy`. Take for instance

```
[omissis]

# Define the fitting model
def model(x, m, q):
    return m * x + q

# Run the fit.
popt, pcov = curve_fit(model, x, y, sigma=dy)

# Unpack the best-fit parameters.
m_hat, q_hat = pop

# Plot the data points.
plt.errorbar(x, y, dy)

# Plot the model.
plt.plot(x, model(x, m_hat, q_hat))
```

This all makes sense because our fitting model takes exactly three arguments (the x data points, along with the slope and the intercept of the line) and therefore we have to unpack the two-element numpy array with the best-fit slope and intercept.

But now that we know the trick we might as well do business in a more expressive fashion:

```
[omissis]

# Run the fit.
popt, pcov = curve_fit(model, x, y, sigma=dy)

# Plot the data points.
plt.errorbar(x, y, dy)

# Plot the model.
plt.plot(x, model(x, *popt))
```

which works out of the box, no matter how many parameters has our fitting model. Cool, isn't it?

2.4.3 Keyword arguments

There is one last important thing when it comes to Python functions: *keyword arguments*. Keyword arguments are passed to a function by explicitly naming the corresponding parameter, which is something that can be generally done

```
def square(x=1.):
    return x * x

print(square(1.))
print(square())
print(square(x=1.))
```

```
1.0
1.0
1.0
```

Now: in this particular case the things may look silly, as there is only one argument—naming it explicitly does not add much context. But when a function has 12 arguments, 8 of which have default values, and we only want to set one among these, well... all of a sudden calling it by name makes a lot of sense—think about [curve_fit\(\)](#)!

As a function becomes more and more complex, this can be so useful that Python provides a mechanism to make an arbitrary subset of the arguments *keyword only*—you just add a *phony* * argument as a separator:

```
def power(x, *, exponent=2):
    return x**exponent

print(power(2., exponent=2.))
print(power(2., 2.))
```

```
4.0
```

```
TypeError: power() takes 1 positional argument but 2 were given
```

```
-----
TypeError                                Traceback (most recent call last)
Cell In[26], line 5
      2     return x**exponent
      4     print(power(2., exponent=2.))
----> 5     print(power(2., 2.))
TypeError: power() takes 1 positional argument but 2 were given
```

(Of course this is a silly example, but if you do look carefully you will realize that `curve_fit()` is actually doing this for real.)

Finally, Python also provides a `**` syntax that collects additional keyword arguments into a dictionary, pretty much in the same way that `*` collects positional arguments in a tuple

```
def func(**kwargs):
    print(kwargs)

func(color="green", size=12)
```

```
{'color': 'green', 'size': 12}
```

While this can be useful at time (e.g., when you have to dispatch arguments from one function call to another), you should use this feature with caution, as all of a sudden you can pass literally *anything* to your function, and there is no way for the interpreter to tell, e.g., whether you mistyped one of the names.

I know, this was a lot. And we did skip over a number of details, too. This might be a good time to take a thorough look at the actual [documentation](#).

2.4.4 Type annotations

One last thing about functions. If you happen to read over some modern, idiomatic Python, you might stumble across something along the lines of

```
def square(x: float = 1.) -> float:
    return x * x

print(square(1.))
```

1.0

Now, wait a moment! What the hell is this extra `float` stuff—were not we clear that in Python you don't need to declare the variable type?

Well, yes; and this is still true. These extra bits are called [type annotations](#).

The funny thing is that the interpreter does absolutely nothing with the annotations. (Well, sort of. There are some odd corner of modern Python, such as [dataclasses](#) where type annotation have a structural function, but we shall pretend this is not the case, for the time being.) What are they for, then? Type annotations serve two main purposes:

- they help you *reason* about the code;
- there are tools, such as [mypy](#) that can do clever things via a combination of static analysis and type annotations.

We shall come back on the matter later, but for the time being be aware that annotations exists—nowadays they are fairly popular.

2.5 The Python import system

We have already stumbled across the `import` statement, and it's now the time to take a closer look at the thing.

When you fire up the Python interpreter, there is a limited number of things you have access to: the language syntax, a handful of keywords and a number of builtin [types](#) and [functions](#), which we have partially explored. (And, by the way, it is somewhat amusing that we got this far with these limited facilities.) Anything else must be *imported* in order to be usable.

The fact that you need to opt-in for most of the stuff should not be surprising, as it is a sensible way to limit the footprint in memory of the interpreter session—presumably there is no point in loading the Python module handling dates and times if you are working on combinatorial mathematics, would you think? Loading only a small subset of the things that are most commonly used and letting the user actively decide on anything else they might need seem only reasonable.

The mechanism through which additional Python modules are loaded in memory on top of the builtins is called the *import system*. And, although the mechanism is exactly the same, it is somewhat customary to divide the Python module ecosystem in two distinct levels:

1. the Python [standard library](#);
2. an enormous collection of third-party modules.

The main difference is: you get the standard library as part of the Python installation, with no question asked; and you have to install any third-party module by yourself, typically via [pip](#) or some other package installer. Just to be clear: the [math](#) module is part of the standard library, while [numpy](#) and [scipy](#) are third-party.

And, before we forget: the `import` system will let you import your own files too.

2.5.1 Import basics

The syntax of the Python `import` system is (unsurprisingly) rich. Let's go through some of the possibilities and try and establish the best practices.

Let's say you want to use the `sqrt()` function in the `math` module. Python will let you use a wildcard to minimize the hassle

```
from math import *  
  
print(sqrt(4.))
```

2.0

This is literally like saying: “get me anything that is defined within a given module”, and, to this day, you still see it being done. Avoid it, unless it's in a small script that you plan on throwing away. Doing this will pollute your scope with names that you don't even know are there and increase the likelihood of clashes.

By the way: you really want to what you have done?

```
import math  
  
dir(math)  
  
['__doc__', '__file__', '__loader__', '__name__', '__package__', '__spec__',  
'acos', 'acosh', 'asin', 'asinh', 'atan', 'atan2', 'atanh', 'cbrt', 'ceil',  
'comb', 'copysign', 'cos', 'cosh', 'degrees', 'dist', 'e', 'erf', 'erfc',  
'exp', 'exp2', 'expm1', 'fabs', 'factorial', 'floor', 'fma', 'fmod', 'frexp',  
'fsum', 'gamma', 'gcd', 'hypot', 'inf', 'isclose', 'isfinite', 'isinf',  
'isnan', 'isqrt', 'lcm', 'ldexp', 'lgamma', 'log', 'log10', 'log1p', 'log2',  
'modf', 'nan', 'nextafter', 'perm', 'pi', 'pow', 'prod', 'radians', 'remainder',  
'sin', 'sinh', 'sqrt', 'sumprod', 'tan', 'tanh', 'tau', 'trunc', 'ulp']
```

Lots of names, eh? Do this for a dozen of modules and it will be hard for you to find a name for your own functions that is not in use.

A slightly better alternative is

```
from math import sqrt  
  
print(sqrt(4.))
```

2.0

Now you are only importing one name from the module. Sure enough, if you happen to need another function you will need to add an import, but it's definitely better than before. You still have the potential issue that, if you have lots of code and you are using the `sqrt()` function at line 823, with the corresponding import at the top of the file, you might need a lot of back and forth to figure out where each function is coming from.

Look at this instead:

```
import math

print(math.sqrt(4.))
```

2.0

This gives you the best of both worlds—you only import one name (`math`) and it is immediately obvious which module the `sqrt()` function belongs to. I will give you that you also have to type 5 extra characters every time you use `sqrt`, but remember: code is written one time and read many times. Whenever you can, go for it; it's a no brainer.

2.5.2 Import aliases

If you start using numpy and matplotlib regularly, you will see that most of the time they are imported in the form of an *alias* with the `as` keyword

```
import numpy as np
from matplotlib import pyplot as plt
```

This is literally all over the place. Every single example on the web is spelled like that.

You might wonder why? Well, this saves you a few keystrokes any time you actually use a name exported by the module. Going from `numpy` to `np` gives you exactly 3 less keystrokes. Is that a good idea? Well, yes and no—it depends on the context. For *recognizable* aliases that everybody is using there is no particular harm, but if you start using aliases a lot, chances are you will sooner or later run into something like

```
from math import *
import logging as log

# ... 1000 lines of code in the middle
x = log(2.)
```

```
TypeError: 'module' object is not callable
```

```
-----  
TypeError                                     Traceback (most recent call last)  
Cell In[32], line 5  
      2 import logging as log  
      4 # ... 1000 lines of code in the middle  
----> 5 x = log(2.)  
TypeError: 'module' object is not callable
```

Good luck debugging that in a large codebase.

2.5.3 Importing your own modules

There will be a time in your career when you will start importing your own modules. When that happens you might start wondering how the hell you can signal Python where your module lives, and where the interpreter should look for it.

If your module lives in the *present working directory* things work out of the box, and you should be all right.

If you have properly installed your module, e.g., via pip, then again you are fine, because the files are where they should be. By the way, you can always tell where a Python module is physically located looking at the `__file__` attribute

```
import math  
  
print(math.__file__)
```

```
/home/users/lbaldini/.pyenv/versions/3.13.1/lib/python3.13/lib-dynload/math.cpython-  
313-x86_64-linux-gnu.so
```

Ideally this should be the case, and if you structure your repository [correctly](#) you should be all set.

If everything else fails, all you need to know is that you can add more places for Python to look for modules by properly setting the `PYTHONPATH` environmental variable. In 2026 this should be hardly necessary, but keep this in mind.

2.6 The Python standard library

Python has a name for coming *with batteries included*. When you install Python you get the full [standard library](#). And, just to be clear, the standard library is huge. Just scroll the documentation web page, and you will find hundreds of modules. Many of the things that you might think of are there.

Here a short list of modules that you might find interesting, sooner or later:

- [datetime](#): basic date and time types
- [collections](#): container datatypes
- [math](#): mathematical functions
- [random](#): pseudo-random numbers
- [pathlib](#): object-oriented filesystem paths
- [shutil](#): high-level file operations
- [os](#): operating system interfaces
- [time](#): time access and conversions
- [argparse](#): command-line options
- [dataclasses](#): data classes

Lots of interesting stuff—take the habit of browsing around in your spare time.

2.6.1 Beyond the standard lib

Awesome how the standard library is, there are use cases where you will grow to prefer third-party modules.

For mathematical functions and random number generation, for instance, the `math` and `random` modules in the standard library are ok in simple settings, but for most applications you will prefer [numpy](#).

Some specific modules of the standard library have grown old, and are now largely obsolete. While they are still maintained for the sake of backward compatibility, they are rarely used for new code and more modern and streamlined third-party packages are to be preferred. Two prominent examples are [loguru](#) for logging and [pytest](#) for unit testing, which largely replace the standard library modules [logging](#) and [unittest](#), respectively.

Some other, more specific applications are just not covered by the standard library, in which case you *have* to resort to a third-party package, unless you prefer implementing the functionality from scratch.

In all these cases, the Python Package Index, or [pypi](#), comes to the rescue. With almost a million different projects hosted, it has you covered, no matter what you need, although it is important to say that not all of them are actively developed and surely not all to the same standard.

Installing a package from pypi is as simple as

```
pip install package_name
```

or, depending on your system

```
python -m pip install package_name
```

3 Version control

All right: we know enough Python, now, to be able to do something sensible; but before we go full out and start code we need to talk about *version control*.

If you simply have no idea about what version control amounts to, congratulations: you are in the right place. If you safely practice version control, already, you are probably good to skip over this chapter. Anything in between is probably worth a cross check. If you are under the impression that creating a zip file every once in a while with a sequential number attached to it is enough, or you have developed a custom-made set of scripts for the purpose, you should probably pause for a second and re-think the whole thing through.

Fortunately, you live in a time when version control is a no-brainer. In the old days people used [CVS](#), and [RCS](#) before that. For a glimpse of time [SVN](#) experience some popularity, before people made up their mind and decided that distributed systems were the way to go. Then for a few years [mercurial](#) and [Git](#) competed with each other, before the latter took over the World. Nowadays it is fairly difficult to find any reasonable project not using Git for version control. That probably means something, and we might as well all stick with it. (By the way, Git was written by [Linus Torvalds](#), the mastermind beyond the Linux kernel.)

3.1 Version control basics

Version control: *(also known as revision control, source control, and source code management) is the software engineering practice of controlling, organizing, and tracking different versions in history of computer files—primarily source code text files, but generally any type of file.* (Verbatim from [Wikipedia](#).)

As you can see, version control is much more than just keeping track of which exact version of every file in your codebase you are using at any given time. Sure enough, version control allows you to freeze forever a specific configuration of your codebase, but the point is really to collect all the relevant metadata specific to any given *change*—what, when, by whom, and why?—in a form that can be effectively used to track the development process in a meaningful way.

As usual, let us start with some basic terminology.

Repository: the place where files current and historical data are stored.

Revision or version: the state at a point in time of the entire tree in the repository.

Clone: process of creating a repository containing the revisions from another repository.

Working copy: a local copy of files from a repository at a specific revision.

Checkout: the action of creating a local working copy from the repository.

Change or diff: a specific modification to a set of files under version control.

Commit: the process of writing the changes made in the working copy back to the repository.

Quite a few new things to commit to memory, isn't it? But don't worry, this is not nearly as bad as it seems. The main thing is: *all the code lives in a repository*. Not in an ordinary folder on your laptop or in a memory stick; not in your email attachments. In a repository. And, since we are at it: in a Git repository. Your responsibility in your day-to-day workflow as a software developer is to interact with the repository so that the latter can efficiently keep track of what's happening. It is as simple as that.

3.1.1 Centralized and distributed systems

In the old days version control was implemented *locally*. And yet some of the main ideas that are at the base of modern version control stem from these early systems—most notably the fact that when you change a file (and, quite possibly, only a few lines in a file) you don't actually have to keep a copy of each version of the file. If you store the *differences between revisions* in a database you are all set, as you can always recreate what any file looked like at any point in time, and potentially you save a lot of memory. That is a neat idea: keep track of *changesets*, not files.

The advent of computer networks brought about the *centralized version control systems* such as [CVS](#) and [SVN](#). This was by far the most popular model through most of the '90: you have a single, central server containing all the versioned files with whom an arbitrary number of clients can interact. The basic workflow is therefore:

- you check out a local working copy from the remote server;
- you modify the working copy;
- you commit the changes back to the repository.

If you think about, this makes a lot of sense. At any point in time there is a single, ultimate authority (the remote server) that everybody else has to conform to. If users A and B start working on the same file in their own local working copies, the rules are clear: if A commit their changes first, B will need to adapt to it because the server will not let them commit further changes unless all potential conflicts are resolved. The nice thing is that the process is beautifully *linear*. It is not by chance that Subversion used to assigns a sequential number to the repo at each commit.

Then *distributed version control systems* came about. A distributed system, as the name suggests, is one in which every client fully mirrors the repository, including its full history.

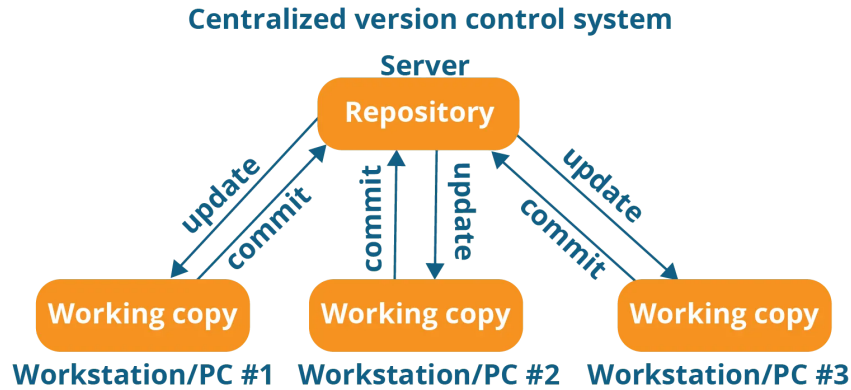


Figure 3.1: Schematic illustration of a centralized version control system.

All of a sudden there is no more a central server and constellation of clients, but rather a collection of machines containing exactly the same information. As you might imagine, this allows for a much richer variety of workflows. If you have the full repository on your laptop, for instance, you might imagine *committing changes* from your local copy to the repo while you are comfortably working on a airplane, disconnected from the network—how awesome is that? The fact is, if user A can do that, so can user B, and when they get off their respective planes and get back online, something new has happened: the repositories that were once identical are no more in synch. This is a big issue. Or not?

Well, it is and it isn't—Figure 3.2 shows a typical arrangement implementing a distributed version control system. Look carefully and don't be distracted by the fact that here too we have a server and a number of clients: while in principle all the repositories in a distributed server are on the same level, you do not typically want user A to be able to write directly on the machine owned by user B. From a security standpoint it is much simpler to have one server machine to which all users can write.

In practice this adds another layer to the workflow: you can commit changes to the local repository while seating on the plane, and you can later *push* the changes to the remote server that everybody else has access to. It is at the level of the push action that any possible conflict is merged.

This brings up an interesting aspect, though: when they get off the plane and back online, user A and user B can potentially want to push a series of commits to the repository that are interleaved in time, and no particular relation with each other. The concept of who is coming first and who is coming second, which is well defined in centralized system, is not directly applicable here. The workflow has become intrinsically *non linear*. This is why Git assigns a *hash*, and not a simple sequential integer, to each commit.

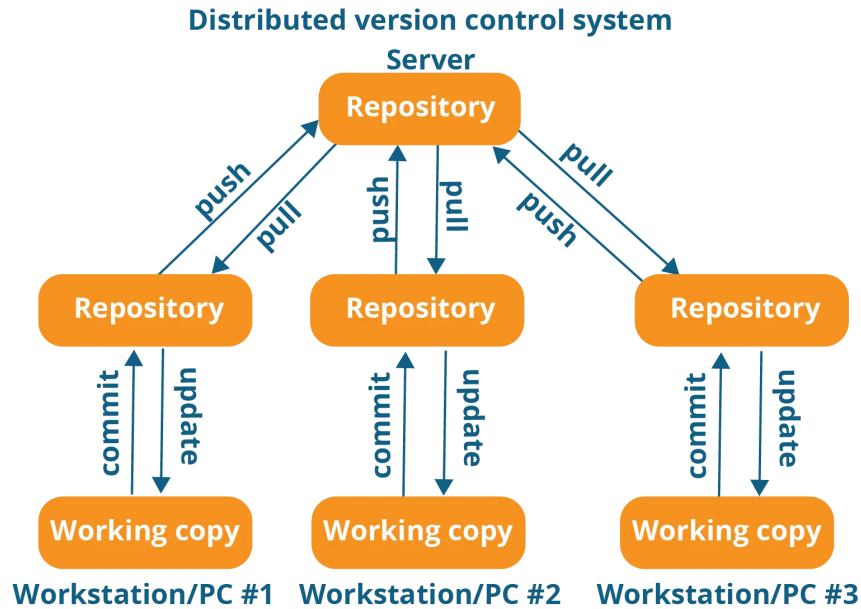


Figure 3.2: Schematic illustration of a distributed version control system.

i Note

Make sure you got this right before you move on. Say A and B use SVN to collaborate on a project, and they start at revision 113. A pushes a commit to the repository, which is bumped to version 114. When they try and push their commit, user B finds out that the repo has changed, and there is a conflict between their changes and the new tip. At this point B has to manually review the situation, and incorporate whatever changed in version 114 into their changeset. Now user B can commit their changes, and the repository is bumped to version 115. The centralized workflow is linear.

Now, let us suppose that A and B are using Git, instead. The point is, when A and B commit changes to the repository, these commits happen on the local copies. Neither one sees what the other is doing. There is no concept of before and after, just two histories that for a little while proceed on parallel path. When A and B get to push the changes to the server, it is not like either local repository is ahead of behind of the other—they are just *different*.

3.1.2 Versioning single files or the entire repository?

This might sound like a silly question, but old version control systems tracked modification on a file-by-file basis. This is how CVS worked, for instance—by assigning an independent

revision number to each single file. You would have, e.g., `file1` at version 1.3 and `file2` at version 2.13.

All modern control systems track a whole commit as a new revision, and *revisions are assigned to the entire repository*. When you think about, this actually makes a lot of sense. Sure enough, versioning single files can give you the cozy feeling that you are doing a better job, because it immediately tells you what changed, without the need to dig into the file. At a second thought, though, files are not generally independent from each other, and you really need to know the status of the entire repository to reliably predict the output of your code. Ultimately, the full changeset is what matters, and modern systems provide facilities to inspect changesets across different files.

i Note

Since we are at it, I have personally witnessed cases where repositories are used to store different versions of the same file alongside (with different suffixes), e.g., `my_file_v1`, `my_file_v2`, `my_file_v3` or, even worse, `my_file`, `my_file_new`, `my_file_newer`. When you do that, you are effectively borrowing somebody else's problem, trying to do the work that the version control system should be doing and, ultimately, interfering with it. Don't do it. When you modify a file and commit the changes, the old version is not gone. Should you need it, Git is there to serve it to you. Really.

For the same reason, resist the urge to copy and paste a block of code, comment the original (just in case you should need it in the future) and edit the copy. This will make diffs (i.e., visual comparisons across different versions) harder to understand, and unnecessarily so. Again: when you need to recover an older version of a file, Git has your back.

3.2 Hosting a Git repository

All right: we made up our mind and we are using Git for version control. Let us assume that we have installed Git on our computer. If you did read carefully, by now you will have realized that the next obvious question is: who is playing the role of the server? In other words: who is *hosting* our repository?

3.2.1 Self-hosting or not?

When it comes to where you physically host a repository, at the very top level you have two choices. If you have a machine that you control and to whom you can connect via a network, you can go ahead and place your repository right there. This is generally referred to as *self-hosting*.

Before you do that, though, think about for a second: do you really want to take on you all the hassle that comes with self-hosting? Manage the users, and setup the access permissions? Update the system as needed? Make periodic backups in case something goes wrong? No? That's what I thought.

This is another no-brainer. Unless you are some kind of 007 that needs complete privacy over your code (which is by definition not the case if you are developing free software), go for a fully-fledged hosting service. That will provide you not only the core functionality, but also all the ancillary tools that you need in you daily workflow, such as a nice web interface and an issue tracker.

3.2.2 GitHub

You might know, already: these days everybody is on [GitHub](#). Most of the big projects host their primary repository on GitHub—[Python](#), [numpy](#), [scipy](#) and [matplotlib](#) are just a few examples. (Well, not *all* of them. The Linux kernel lives on their own [servers](#), so does GNU.) Most of the small projects are developed on GitHub. You might just go with the flow and do that, too.

Note

Occasionally you hear people confusing Git with GitHub, so let's get that straight: Git is a free and open-source *program*, while GitHub is a proprietary *service* offering hosting of Git repositories. It is as simple as that.

Of course nothing is straightforward as it seems. Launched in 2008 by a small group of developers, GitHub has been thriving since, building upon the popularity of Git as a version control system. Ten years later, in 2018, it was acquired by Microsoft, with the widely publicized intent to have it continue to operate independently as a community-driven platform. (Ah ah.) As of 2025, GitHub does not have a CEO and is operating as part of the Microsoft AI division, instead. The increasingly pervasive emphasis on AI did not go un-noticed in the community and more than a few prominent actors have grown vocally critic against the new course of action, see e.g. [here](#) and [here](#).

Let's be honest: this is not anywhere close to a generalized rebellion and almost certainly GitHub is there to stay for the foreseeable future. Sadly: if you don't have an account, yet, you should probably get one.

3.2.3 Codeberg

On the bright side, [Codeberg](#) is an interesting reality that is slowly gaining traction. It is where most of the people fed up with GitHub are moving, and, for what it's worth, where these lecture notes are hosted.

Codeberg is run by a non-profit organization based in Germany and provides a reasonably complete experience with a look and feel similar to GitHub, only with no AI. (Well, I will be completely honest and confess that it's not yet at the same level, but it is definitely worth supporting because the underlying intent is admirable.)

Small piece of advice: if you create an account on GitHub, create one on Codeberg, too and experiment with it.

3.2.4 Digression: exchanging ssh keys

No matter which hosting service you use, creating your first repository is as simple as pressing a button and filling in a form with a handful of fields (name of the repository, visibility, license, main language).

Cloning a public repository on your computer is also largely trivial. You can clone the repo for these lecture notes, e.g., by doing

```
git clone ssh://git@codeberg.org/lbaldini/cmepda-basic.git
```

Note

We're not talking about what actually goes into the repo, right now, but [here](#) is a useful reference. We shall get back to it in due time.

The tricky part comes when you actually need to *push* changes to the repo, because that's where the server needs to authenticate you and make sure you have the proper write permission. So let's elaborate a little bit on that.

“How do I get the url to clone my favorite repo”, you are asking? Easily said: it's usually sitting somewhere on the top-right in the web interface of the repo itself. Github has a big green button called “<> Code”; Codeberg has a text box with a couple of buttons next to it. Both will give you the option of copying to the clipboard the clone link using either the [SSH](#) or the [HTTPS](#) communication protocol.

To make a long story short, HTTPS is ok for occasional cloning, I guess, but for regular development you definitely want to go for SSH. Citing the Codeberg [documentation](#) verbatim: *in comparison to using HTTPS, SSH keys provide a safer and passwordless way to authenticate with Codeberg. It is recommended to use one key per client. This means that if you access your Codeberg repository from your home PC, your laptop and your office PC you should generate separate keys for each machine.*

The only additional tiny barrier between yourself and the ultimate Git experience is the *ssh key exchange*. You don't need to be an expert in cryptography to use Git, but a short digression might be useful, at this point.

SSH, or Secure Shell, is a network protocol and a set of tools for securely accessing and controlling a remote computer over a network. Creating an SSH key pair means generating two mathematically linked keys:

- a private key (keep secret): stays on your machine, optionally protected by a passphrase.
- a public key (share freely): you upload it to the server or service (e.g., GitHub or Codeberg).

The two work together via public-key (asymmetric) crypto. During login, the server sends a challenge; your SSH client proves it has the private key by signing that challenge. The server verifies the signature using the public key it has on file—no password travels over the network.

How do you actually get to exchange the key with hosting service? There's plenty of documentation out there, including on [Codeberg](#) and [GitHub](#). As soon as you are able to push your first commit to the server, you are all set!

3.3 Daily workflow

We are now ready to discuss in some details how you should go about the actual day-to-day workflow when developing software under version control.

3.3.1 More nomenclature

Before we move on, we need to enlarge a little bit our VCS glossary.

Branches: alternative paths where more copies of the same files develop in different ways independently.

main: the unique line of development that is not a branch.

Merge: application of two sets of changes to a set of files.

Conflict: changes to the same file by two or more developers that the system is unable to reconcile.

3.3.2 Git overview

The easiest way to have a quick overview of the large inventory of Git commands is to look at the help.

```
lbaldini@nblbaldini:~/teaching/lbaldini/cmepda-basic$ git --help
usage: git [-v | --version] [-h | --help] [-C <path>] [-c <name>=<value>]
        [--exec-path[=<path>]] [--html-path] [--man-path] [--info-path]
        [-p | --paginate | -P | --no-pager] [--no-replace-objects] [--no-lazy-fetch]
        [--no-optional-locks] [--no-advice] [--bare] [--git-dir=<path>]
        [--work-tree=<path>] [--namespace=<name>] [--config-env=<name>=<envvar>]
        <command> [<args>]
```

These are common Git commands used in various situations:

start a working area (see also: `git help tutorial`)

<code>clone</code>	Clone a repository into a new directory
<code>init</code>	Create an empty Git repository or reinitialize an existing one

work on the current change (see also: `git help everyday`)

<code>add</code>	Add file contents to the index
<code>mv</code>	Move or rename a file, a directory, or a symlink
<code>restore</code>	Restore working tree files
<code>rm</code>	Remove files from the working tree and from the index

examine the history and state (see also: `git help revisions`)

<code>bisect</code>	Use binary search to find the commit that introduced a bug
<code>diff</code>	Show changes between commits, commit and working tree, etc
<code>grep</code>	Print lines matching a pattern
<code>log</code>	Show commit logs
<code>show</code>	Show various types of objects
<code>status</code>	Show the working tree status

grow, mark and tweak your common history

<code>backfill</code>	Download missing objects in a partial clone
<code>branch</code>	List, create, or delete branches
<code>commit</code>	Record changes to the repository
<code>history</code>	EXPERIMENTAL: Rewrite history
<code>merge</code>	Join two or more development histories together
<code>rebase</code>	Reapply commits on top of another base tip
<code>reset</code>	Set `HEAD` or the index to a known state
<code>switch</code>	Switch branches
<code>tag</code>	Create, list, delete or verify tags

collaborate (see also: `git help workflows`)

<code>fetch</code>	Download objects and refs from another repository
<code>pull</code>	Fetch from and integrate with another repository or a local branch

```
push          Update remote refs along with associated objects
```

```
'git help -a' and 'git help -g' list available subcommands and some
concept guides. See 'git help <command>' or 'git help <concept>'
to read about a specific subcommand or concept.
See 'git help git' for an overview of the system.
```

This a lot of stuff. The [Git cheat-sheet](#) provides an alternative, interesting cross section on the matter.

3.3.3 Top-level good habits

We are all set: we have all the tools that we need to start writing software. Shall we start right away to push commits to the server as if there was no tomorrow? Probably not—we should have a rough workflow in place.

Here a small set of sensible, top-level rules you should stick to:

- keep the main branch always working and functional—it should never break;
- use short-lived branches for individual tasks, such as fixing a bug or implementing a new feature;
- if you can break a task in smaller sub-tasks, by all means do it;
- commit coherent changes when possible, as opposed to incomplete stubs;
- always test before merging;
- push regularly.

3.3.4 Beginning your day

Let's dive into the gory details, and go through our full day of work, starting from the moment, early in the morning, when we get into our office and sit in front of the computer. First thing first, let us synchronize our local repository

```
git switch main
git pull --ff-only
git status
```

This moves you to main and update the branch, picking up changes that have happened since your last session. Most importantly, it only does so *assuming that the main can be fast-forwarded*, i.e., the local main and the remote main have not diverged—in which case a simple pull would silently create a merge commit. If the branches have diverged, Git just stops rather than making something up, and you can proceed to inspect the situation and choose the most appropriate course of action.

3.3.5 Getting started on the next, great feature

Now that we are up to date, we can start working on our next task, be it fixing a bug, improving the documentation, implementing a new feature, or anything else. Remember our good habits: make the task as small and self-contained as possible; try and break complex changes into smaller, simple ones—the latter are easier to review. Attack one problem at a time, try and isolate related changes in the smallest possible subset, and avoid as much as possible mixing unrelated changes.

The first step is to create a branch for your new endeavour; and, obviously switch to it. You achieve this with the commands

```
git branch name_of_the_new_branch
git switch name_of_the_new_branch
```

or, equivalently, with the more succinct version

```
git switch -c name_of_the_new_branch
```

This decouples you from the main branch, and from the work that your collaborators will be continue doing. The new branch can live for all the time that it takes for you to complete the task. When you're done, you can merge the branch back into the main, and we shall come back to that in a second.

(It goes without saying, if this is not a new thing and the development branch already exists, you just switch to it and pull the changes instead.)

Since we are at it, here is another piece of advice. *Pick a sensible name for your branches*, and keep in mind that *the branch name should express your intent*. Examples of terrible names:

- my_branch
- new_branch
- awesome_branch
- bug_fix

Examples of sensible names (mileage may vary, depending on the application)

- fix_issue_2376
- document_data_format
- poisson_likelihood

You can use a / as a separator and organize your branches with prefixes to indicate clearly whether we are talking about, e.g., a bug fix or a new features. Examples:

- fix/issue_2376

- docs/data_format

You get the point.

3.3.6 Doing the actual work

We are all set to start doing something useful. Again on the good habits: organize your work in small, testable steps—do not wait for the full task to be completed to commit changes, as that makes it more difficult to provide granular, informative commit messages. A commit need not represent a finished feature, but it should leave the repository internally coherent when possible.

Always inspect your local repository thoroughly before committing, in order to minimize the potential for mistakes. Two Git commands that you will find yourself typing often

```
git status
git diff
```

(We haven't had a chance to talk about unit tests, yet, but you should always run all the tests before committing changes. More on this later.)

In principle Git allows you to commit *all* the changes in your working copy at the same time

```
git commit -a -m "message"
```

but it is generally better to group modified files in small sets of related changes, each one with its own, informative commit message.

```
git add file1 file2 file3
git commit -m "message"
```

Note if you are committing a new file, you have to add it first.

What makes a good commit message? It might be more instructive start from what makes a bad commit message: “minor”, “fix”, “work”, “latest version” are all terrible comments, and they convey absolutely nothing about the *intent* of the commit. Please refrain from using them.

i Note

We already made the point that with a distributed version control system the workflow is inherently non linear, so that commits cannot be simply identified with a sequential integer. How does Git do that?

Well, look at [this one](#) for example. Its identifier, which you can tell from the url in the address bar of your browser, is 00df7eda11013a983d907985dd675873e01ffb70. What the hell is that? It is 160-bit hash (normally written as 40 hexadecimal characters) created from the metadata associated to the commit itself, such as the project name, the author of the change, the parent commit and the commit message. Clever, isn't it?

Push changes to the remote server frequently—at the very least at the end of each working session. All your work is tracked into the local version of the repository, but if your laptop breaks down or gets stolen and your last remote push is three months old, you are in trouble.

3.3.7 Merging changes

Sooner or later the time will come (hopefully) when your task is completed, your new feature is ready for prime time, and you want to move forward and merge the changes back into the main branch, or any other branch.

The basic rule, here is: *always merge changes via a pull request*. A pull request is basically a proposal to merge a set of changes from a source branch to a destination branch (typically the main). Pull requests can be open through the web interface, no matter whether you use GitHub or Codeberg, and the process should be largely self-explaining.

The pull request machinery is very effective in organizing the formal merge proposal into a set of useful things that provide full insight into what is being changed:

- a view of the complete diff;
- a place to describe the motivations for the changes;
- results from unit tests;
- a permanent link between discussion and implementation.

It doesn't matter if you work in a large team or by yourself, pull requests are the way to merge changes. If you work in a team, ideally every pull request should be reviewed by a fellow team member before being merged. It goes without saying, the smaller the pull request, the easier the review.

As soon as the pull request is merged, you can clean up things, switching back to main and deleting the development branch, which is no longer necessary. (Both GitHub and Codeberg will offer you a button to trigger the deletion.)

This completes our journey through the first successful implementation of a change. That was a lot. You will refine your own workflow as you move forward, but you can keep this tentative list of steps as a reference. There's still a couple of things we want to mention, but we are almost there.

3.3.8 Resolving conflicts

If your branch has no commits behind the main, or there are no conflicts with the main branch, merging a pull request is as easy as pushing a button.

Now imagine that the main has changed while you were working on your development branch—maybe another person has merged in their own bug fix or new feature. Your branch is now n commits ahead and m commits behind of the main. The first thing you do is *merging* the main into your branch. (Pay attention: not the other way around.)

```
# Check out your branch
git switch my-branch

# Always make sure you have no uncommitted changes
git status

# Get the latest remote state
git fetch origin

# Merge the latest main into your branch
git merge origin/main
```

This could go smooth, if you and your fellow developer who just beat you merging into the main branch happened to have touched different files, or even the same files in different places. Git is more than good at merging changes—it is almost miraculous: [here](#) is a story of an automatic merge that git managed to do from 66 different branches of the Linux kernel. (This is unusual, though. So much so that this one caught the attention of Linus Torvalds, no less.)

That all said, there will be cases when there isn't an obvious way to automatically merge things, and human intervention is needed. Maybe developers A and B just do not agree on how to do things, and started a commit war that a third person needs to stop. Git will be very clear about which files have conflicts and need to be fixed by hand: you will get a message on your terminal when you do the merge, and the problematic sections inside the files will be marked with exactly three marker lines:

```
<<<<<<< HEAD
your version
=====
incoming version
>>>>>>> main
```

Your text editor might even do something clever with the git annotation, and provide facilities to help you pick a course of action. Apply the necessary changes, remove the marker lines, and then commit and push the changes as usual.

3.3.9 Tagging the repository

We are all used to software versioning. The operating system on your computer or on your smartphone comes with a version. Every program that you use comes with a version and, e.g.,

```
git --version
```

will happily print on the screen the version of Git installed on your machine.

This naturally brings up the question of whether you should be versioning your code, and the answer is, generally speaking, yes.

From the standpoint of version control, versioning is achieved through *tagging*. Formally, a Git tag is a permanent name attached to a particular commit, and is customarily used to mark a snapshot of the repository that has some significance. The two commands

```
git tag -a v1.0.0 -m "message"
git push origin --tags
```

create an annotated tag in the local repository and push the tag(s) to the remote server.

All modern hosting services provide an ecosystem of *actions* that you can trigger upon specific events, such as Git tags, to produce a rich variety of effect—for instance, you could produce and archive release artifacts every time you tag the repo.

3.3.10 What shall I track in the first place?

Sooner or later, I guarantee, the moment will come when you will push by mistake to the remote server a 1.2 GB file happening to float around in your working copy for reason that you don't fully understand. That will be also the moment you will fully realize how good Git is at tracking stuff, as from that point on a fresh clone of the full repository will take forever. Sure enough you can do

```
git rm that_awfully_large_file
git add that_awfully_large_file
git commit -m "Remove huge file previously committed by mistake"
git push
```

and that will remove the file from all future versions of your branch, but not from the *history*.

(Now, if you are *really* motivated, there are nuclear weapons that you can use to sanitize a Git repository that you abused with a file that was too large, but this problem is relevant enough

that many repositories have pre-commit hooks setup that will happily refuse to accept commits that are too large. Period.)

This brings up the important question: what exactly shall I track in my repository?

Well... repositories are basically for source code, and source code comes in the form of text files. The vast majority of the content of your repository should be in text form. Dealing with text is where version control systems shine, as changes can be efficiently saved as diffs—remember? (On the other hand, when you put binary files in a repo, modifying them pretty much amounts to creating a new full copy every time.)

Of course not all the text must necessarily be source code. If you have text configuration files, that's totally legit. Documentation is also most welcome—there is hardly ever enough of that. If your tests require small mock files, be they text or binary, that is also fine. And that's pretty much it.

And is a cursory list of things that, most likely, you don't want to commit to a repository:

- any generated artifact or intermediate file that can be produced from other sources;
- temporary files produced by your operating system or any program—`DS_Store` is a classic;
- private tokens or credentials;
- large data files.

The last, as we said, is the one you should especially pay attention to.

3.4 Ancillary tools

Though Git provides all the core functionality that you need for version control, and GitHub and Codeberg run on top of Git, there are other important tools that you will be using if you start developing code.

3.4.1 Issue tracking

Any software project that is developed in a vaguely reasonable way will offer a place where you can file bugs or ask for features. [Here](#) is Python's. The thing is customarily called an *issue tracker*.

The nice thing is: both GitHub and Codeberg, if you want, provide you for free an issue tracker associated to each repository that you create. [Here](#) is the issue tracker for these lecture notes—use it without hesitation if you find a mistake or something might be explained better.

Get into the mood of using the issue tracker for your projects. It is definitely the right place to discuss changes before, while and after the fact. A few months or years down the road you will

be grateful to yourself for the fact that you didn't just fixed *that* bug—you planned the fix, discuss the possible solutions and describe the actual implementation.

3.4.2 Actions

Actions are basically tasks that get triggered by a specific event in a Git repository, e.g., a push on the main, the update of a pull request or the creation of a tag.

Typical use cases for actions include:

- running the unit tests in response to updates to a pull request;
- building the documentation when the main is tagged;
- publishing a package on PyPI when a new version is released.

From a practical standpoint, actions are built in the form of suitable yaml files that specify completely *what* is done and *when*. Sounds powerful? Well, actions *are* powerful, and now that you know they exist you can work your way through them.

3.4.3 Forking

If you look carefully the entry page for your repository in either GitHub or Codeberg, you might have noticed a small *Fork* icon on the top right. What is that for, you are wondering?

Fork: *a codebase that is created by duplicating an existing codebase and, generally, is subsequently modified independently of the original.* (Source: [Wikipedia](#).)

Why on Earth would you want to fork a repository? Didn't we say somewhere that duplicating code is bad? Well... generally yes, but not in this particular case.

Say you found an error in these notes—unlikely, I know. You might open an issue on the [issue tracker](#), so that the authors can pick it up and fix it. *Or*, you might be so kind to be willing to fix it yourself! The problem is: you don't have write access to the repository, and you cannot push changes.

One solution is to drop the owner of the repository a line asking to become a collaborator on the project. They might say yes or no. Granting write permissions on a repository is generally a big thing that requires trust. In any event, this is not the best course of action for a casual fix—collaborating is... well, for collaborating, as in “working together regularly”.

Forking is the perfect solution for this use case. You get a perfect replica of the original repository under your own Codeberg or GitHub account, and you can do whatever you want with it. Including fixing the error in these notes. If you fail to see where we are going with this, there's a very nice feature that the forking mechanism provides: *you can open a pull request to the original repository from your forked one*. How cool is that? Then the owner of the original repo can pick it up and merge it, if they please.

i Note

When you open a pull request to the original repository you have the option to “*Allow edits from maintainers on the PR*”—remember: the maintainers of the original repo own the repo itself, but the branch that you propose to merge in lives in your fork and is under your control. It is generally advisable to allow the maintainers edit the pull request branch, provided that you trust them, so that they can make edits as they see fit before the actual merge.

[Here](#) is a nice tutorial on the whole forking-branching-changing-making-a-pull-request thing.

4 Python packaging

This chapter is about how you should go about structuring the repository hosting your latest, awesome Python package. This will be a fairly short chapter—not because the issue is not important (in fact packaging is so important that Python even has its own dedicated [working group](#)) but because the most natural way to illustrate how to structure a repository is, well... through an actual repository.

And you are covered: [metarep](#) is there for you to look at, and it is designed to be a fellow complement to these notes. More specifically, the associated [documentation](#) will patiently walk you through all the practical aspects of making a package that can be installed with `pip` and placed on [PyPI](#).

If you want to share your package with the outside world, this is the standard way. As you will see, the bar to achieve that is surprisingly low, these days.

5 Algorithms (sort of)

Algorithm: (noun) *a list of instructions for solving a problem.* Source: the [Cambridge Dictionary](#), again.

Easy, isn't it? Here is one of the most popular words of the last decade, demystified. In this chapter we shall glance (and fairly quickly, too) through two very important and inter-related topics: algorithms and data structures. Using the right algorithm and/or data structure sometimes makes all the difference between being able to attack a problem in a practical time or not, so pay attention.

5.1 The Euclidean algorithm

If there is a *classic* algorithm that has passed the test of time, it is that due to Euclid to find the greatest common divisor (GCD) between two integers, so this is where we shall start from. Keep in mind, algorithms can be expressed in several different ways: flow-charts, pseudo-code, working code snippets—you name it. The main point is that the sequence of operation must be expressed *unambiguously*.

Let n and m be two positive integers. The following procedure will converge to the GCD between the two in a finite number of steps:

1. let r be the remainder of the integer division between n and m , or $r = n \bmod m$;
2. if $r = 0$, the m is the GCD, and we are done;
3. set $n \leftarrow m$, $m \leftarrow r$ and go back to 1.

(Here $n \leftarrow m$ means simply: *replace n with the value of m .*)

This is not a course in number theory, so we shall refrain from demonstrating that what we just said is actually true. Lazy as we are, we will just go ahead, implement the thing in Python and observe it in action.

```
def gcd(n: int, m: int) -> int:
    r = n % m
    while r != 0:
        n = m
        m = r
```

```

    r = n % m
    return m

print(gcd(49, 21))

```

7

Well, $49 = 7 \times 7$ and $21 = 7 \times 3$, so the answer is correct. In this particular case the algorithm completes in two iteration, and it is instructive to write down the values of all the relevant variables at each step.

Iteration	n	m	r
1	49	21	7
2	21	7	0

You see what's happening and why the thing works: at each step we are dealing with pairs of numbers that are smaller and smaller, until the remainder of the division becomes zero and we are done.

i Note

It is somewhat amusing to note that, had we started with $n = 21$ and $m = 49$ the first step of the algorithm would have simply swapped the two numbers, and we would have reached the same result, although with the need of one extra iteration.

5.1.1 How many iterations?

Think about it. *How many iteration does it take for the Euclidean algorithm to converge?* Well, this is an ill-posed question. I mean: give me two integers and I can have my program *count* the number of steps it take

```

def gcd_iterations(n: int, m: int) -> int:
    r = n % m
    k = 1
    while r != 0:
        n = m
        m = r
        r = n % m
        k += 1

```

```
    return k

print(gcd_iterations(144, 89))
print(gcd_iterations(144, 72))
```

```
10
1
```

but finding a closed-form expression that would magically calculate this number without running the algorithm from the beginning to the end seems out of the question. We can totally expect for it to increase as n increases, and yet we can get wildly different results for the same n if only we pick different values for m , as our small snippet illustrates.

At a second thought, there are a few legitimate questions that we can ask about our problem. If we fix n and we let m vary between 1 and n , what is the *average* number of steps that it takes for the algorithm to converge? What is the maximum? What is the minimum?

Let's make this clear up front: these are all difficult questions. (Well, except for the last: the minimum number is 1, which happens when m divides n .) Sure enough, we could wrap our small Python function into another loop and calculate all these quantities for any given n , and yet that would not provide the answer in the form of a closed-form expression $f(n)$.

As it turns out, the average value of steps that is required for the Euclidean algorithm to converge asymptotically scales as

$$\frac{12 \ln 2}{\pi^2} \ln n. \quad (5.1)$$

for large n . (And the worst case also scales as $\ln n$ for large n . In case you are wondering, that happens when m and n are two adjacent Fibonacci numbers. Don't ask me why.)

Think about for a second. These are very, very useful statements in practice. If I make n 1000 times larger, the problem is not really getting 1000 times more difficult. Both the average and the maximum number of steps only grow logarithmically. How do I tell this is the case, in practice?

```
print(gcd_iterations(196418, 121393))
```

```
25
```

(Spoiler alert, these are two adjacent Fibonacci numbers, so we are effectively dealing with the worst possible case.)

Again: this is a very nice way to characterize how difficult a given algorithm is from a purely computational standpoint as the input grows. If the number of steps to find the GCD increased linearly with n , we would live in a very different world, and one where computers would not work quite in the same way that we experience every day.

5.2 Asymptotic behavior and complexity

This is probably clear from the previous example, but attacking the asymptotic scaling of an algorithm in search of an explicit expression such as the one we have reported in the previous section is typically out of the question unless you do that as a day job. To this day, I have literally no idea where Equation 5.1 comes from. Luckily enough, most of the time you can derive useful scaling properties by means of much simpler arguments.

The *complexity* of an algorithm describes how the number of operations that are necessary for its completion grow as the size n of the input becomes larger and larger. (Depending on the specific, n might be the integer number we are trying to factorize, the length of the list we are processing, or the size of the file we are reading it—you get the idea.)

As we know, when taken at face value, this question is ill-defined; or, at the very least, it needs to be stated more precisely for us to answer, as the exact number of operations for completion, in general, does not depend only on the input size n , but also on which *specific* object of length n we are considering. (Think about: if my problem is searching for a specific value in a list of length n , I might be very likely and find it at the first step, or not find it at all.) As we shall see in a second, our additional qualifiers shall be: *we are interested in the average, minimum and maximum number of operation across all the possible input configuration, in the limit where n is large.*

Let us go back to a very simple example: find the largest element in a given list of integers of length n . Roughly speaking, a sensible search in Python might be coded as:

```
def find_max(values: list) -> int:
    maximum = values[0]
    for value in values[1:]:
        if value > maximum:
            maximum = value
    return maximum

print(find_max([1, 2, 5, 98, 3, 1672, 6, 34, 651]))
```

1672

Now, you might notice that we analyzed the Euclidean algorithm in terms of number of *steps*. When we started this new section, all of a sudden, we started taking about the number of *operations*; but what is an elementary operation? Well, as it turns out, we don't need to be too fussy about this; at least, not right now. The concept of elementary operation depends on the context and on the language, but for the purpose of our toy example we shall assume that assignments, list lookups and comparisons are all elementary operations on the same par. And the final return statement is one, too.

Note

Keep in mind: we are not trying to measure the precise completion time of our task—this will depend on many conditions outside our control, including which particular hardware we are using—only how this completion time scales with the size of the input. In other words, we don't care so much whether our task takes 2 or 4 seconds to get to the finish line. But what happens when we increase the size of the input by a factor of 10? Does the running time increase by a factor of 10, as well? Or 2? Or 100?

How many operations is our search algorithm performing, then? Well:

- one assignment and one list lookup at the beginning—this makes it 2;
- one lookup, one assignment and one comparison for each iteration in the for loop, i.e., $3(n - 1)$
- a variable number of assignments, between 0 and $(n - 1)$, in the `if` block;
- one final return instruction.

The grand total may vary between $3n$ and $4n - 1$, depending on the input. (If you look carefully, it is easy to see that the best and worst scenarios are those in which the list is sorted in descending or ascending order, respectively, as in these two special cases we never enter the `if`, or we do it every single time. Mileage may vary if the values in the list are randomly distributed.) And we now have a full answer to our original question of what the minimum, maximum and average number of operations are for our function to complete:

$$N_{\min} = 3n \quad N_{\max} = 4n - 1 \quad N_{\text{ave}} = \frac{7n - 1}{2}.$$

5.2.1 Big-O notation

This is the moment where we remind ourselves what exactly we are trying to accomplish: we want to gauge how the average (or minimum, or maximum) number of necessary operations scales, in relative terms, with the size n of the input, when the latter is large. In order to do that, we shall do two different things:

1. drop all the subdominant terms, if any, and only keep the one that grows faster with n ;
2. neglect the multiplicative factor in front of the leading term.

That is to say, $4n - 1$ is just like n —you drop the trailing 1 and ignore the 4; for the same reasons $3n$ is just like n ; $\frac{12 \ln 2}{\pi^2} \ln n$ is just like $\log n$. We have implicitly introduced the *big-O notation*, and from now on we shall say that the complexity of our search for the maximum in a list is $O(n)$, or *order* n , and the complexity of the Euclidean algorithm is $O(\log n)$, or *order* $\log n$.

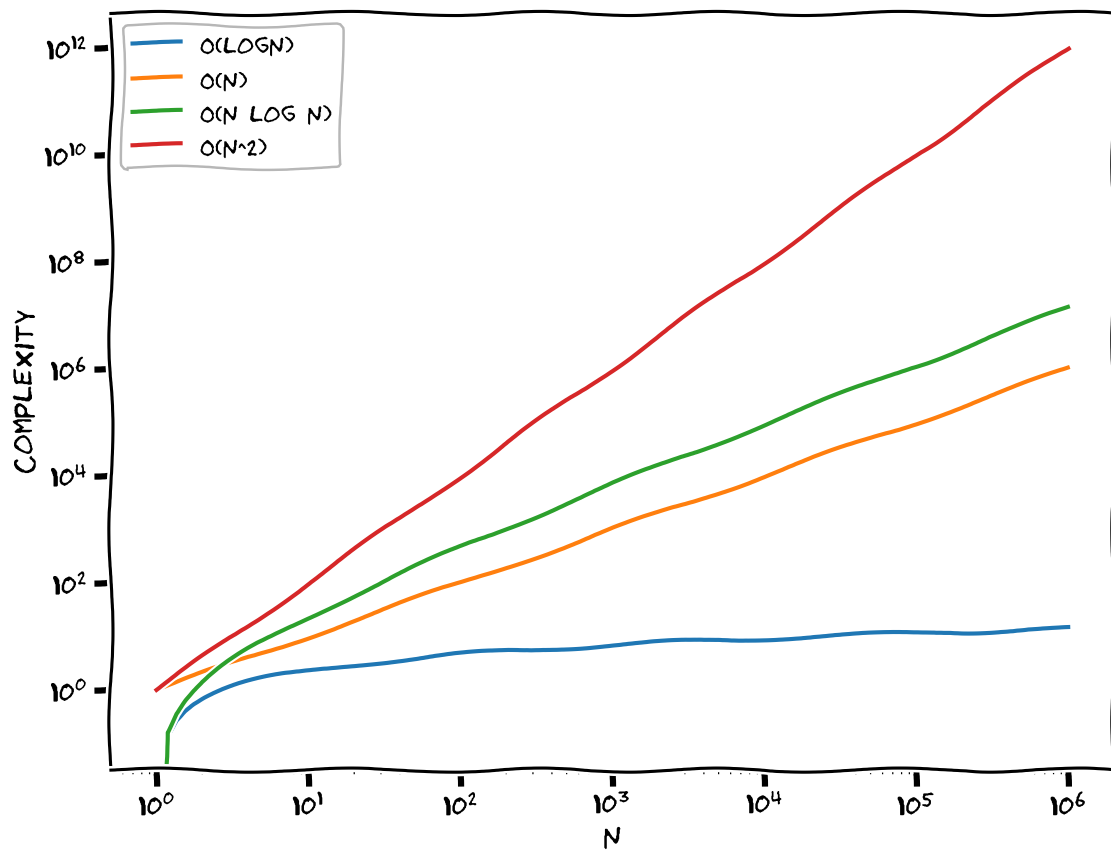


Figure 5.1: Relative scaling of the most important algorithmic asymptotic behaviors, in big-O notation.

Just so that you understand where we are heading with this, look at figure Figure 5.1 for a second. For an input size of a million (10^6), if you are able to beat down the complexity of a given task from $O(n^2)$ to $O(n \log n)$ —which for a large class of problems is possible, e.g., by just working in Fourier space—you are moving down on the far right of our plot from the red to the green curve, reducing the running time by a factor $\sim 10^5$. This might mean, e.g., that something that used to take a full day to run is now only taking one second. Not bad, eh?

And now, for something completely different, you might enjoy reading Knuth (1984). If it doesn't make you laugh, start over from the beginning of the chapter. Repeat until you laugh out loud.

5.2.2 Big-O in practice

The Python program for the search of a maximum in a list was fairly simple. If you are wondering how you would go about understanding the complexity of an arbitrary algorithm or actual piece of code you might be handed over, I hear you. It can be pretty hard to make it starting from the first principle.

One obvious thing that you can do is to simply use the brute force: implement the algorithm in your favorite language, run it on input data of different size, time the execution and make a scatter plot of the running time vs. input size—you should get something resembling Figure 5.1. (Use caution: results may vary from run to run, and you will need to do some averaging and keep track of the uncertainties but, that all said, the thing works as advertised.)

If you are really brave, you can do everything by pure analysis: go ahead, count the steps of your algorithm (or elementary operations in your program, whatever that means) for any given input, and evaluate the average, best and worst-case scenarios. In practice, though, that generally involves sophisticated mathematics—remember Equation 5.1.

Fascinating as mathematics is, however, you can often gauge useful information about the behavior of a piece of code by just looking at it. A simple loop over the input data, for instance, implies a complexity $O(n)$. How about two loops in sequence, one after the other? Remember: we are throwing away any multiplicative constant, so that's again $O(n)$. And how about two *nested* loops? Well, that brings us right into the land of $O(n^2)$ which, most likely, means you are not doing things properly. We shall get back to this in a moment, but you got the message.

5.3 Sequential vs. binary search

“Searching” is one of the words that appears next to “algorithm” more often, and this fundamental problem provides a natural setting to illustrate many of the topics we have glanced through in this chapter.

Say we want to write a function to search a given value in a list and return its index. The function should return -1 if the target value is not in the list. (For simplicity, we shall assume that the list is homogeneous and contains integers, but this is irrelevant for our argument.) One possible way to go about it is:

```
def sequential_search(values: list, target: int) -> int:
    for i, value in enumerate(values):
        if value == target:
            return i
    return -1

l = [1, 2, 4, 6, 12, 32, 99]
print(sequential_search(l, 4))
print(sequential_search(l, 3))
```

2
-1

What is the complexity of this *sequential search*? Easily said: there is one loop over the input, so $O(n)$. Note the function returns as soon as we find the target, so, generally speaking, the loop is not guaranteed to iterate over the entire list, but that is irrelevant for our purpose: even if we visit half of the list on average, this is still $O(n)$.

At a closer look, though, there is one important bit of information that we overlooked for this particular application: the input list that we passed to the function is *sorted* in ascending order. How can we turn that to our advantage?

```
def binary_search(values: list, target: int) -> int:
    left = 0
    right = len(values) - 1
    while left <= right:
        midpoint = (left + right) // 2
        value = values[midpoint]
        if value == target:
            return midpoint
        elif value < target:
            left = midpoint + 1
        else:
            right = midpoint - 1
    return -1

l = [1, 2, 4, 6, 12, 32, 99]
```

```
print(binary_search(1, 4))
print(binary_search(1, 3))
```

```
2
-1
```

How would you explain in words this kind of *binary search*? Well, we start from the entire list, find the midpoint, and see if our target is larger or smaller than the corresponding value in the list. In the first case we restrict the search to the rightmost half of the list, and in the second case we restrict it to the leftmost half. No matter what, we cut the number of elements we have to look at by a factor of 2 at each step, and we iterate until either we find the target, or we realize the latter was not there in the first place.

What is the maximum number of steps (ops: we switched from elementary operation to steps again, but by now you know they are essentially the same thing from the standpoint of the complexity) that the algorithm needs to get to the finish line? By definition we are done when the width of the search window is one, which means

$$\frac{n}{2^{N_{\max}}} \approx 1 \quad \text{or} \quad N_{\max} \approx \log_2 n$$

In the best case scenario we get the target at the first step. The average case, surprise surprise, is still $O(\log n)$. Which, again, is a *big* thing: go look at Figure 5.1 and compare $O(n)$ and $O(\log n)$ for large n ...

Before we move on, here is a slightly unrelated piece of advice. Binary search is a fairly standard algorithm and it is very unlikely that you will ever find yourself in need of implementing your very own custom copy. The Python standard library provides the [bisect](#) module

```
import bisect

def binary_search(values: list, target: int) -> int:
    # Check that the target value is not larger than all the elements,
    # as in that case bisect() would return an index outside the list.
    if target > values[-1]:
        return -1
    i = bisect.bisect_left(values, target)
    if values[i] == target:
        return i
    return -1

l = [1, 2, 4, 6, 12, 32, 99]
print(binary_search(l, 0))
```

```

print(binary_search(l, 1))
print(binary_search(l, 3))
print(binary_search(l, 4))
print(binary_search(l, 99))
print(binary_search(l, 100))

```

```

-1
0
-1
2
6
-1

```

and numpy provides the `searchsorted()` function that is doing pretty much the same thing

```

import numpy as np

def binary_search(values: list, target: int) -> int:
    # Check that the target value is not larger than all the elements,
    # as in that case bisect() would return an index outside the list.
    if target > values[-1]:
        return -1
    i = np.searchsorted(values, target)
    if values[i] == target:
        return i
    return -1

l = [1, 2, 4, 6, 12, 32, 99]
print(binary_search(l, 0))
print(binary_search(l, 1))
print(binary_search(l, 3))
print(binary_search(l, 4))
print(binary_search(l, 99))
print(binary_search(l, 100))

```

```

-1
0
-1
2
6
-1

```

That is, assuming that you remember that binary search exists when you need it, you are covered!

5.4 Sorting

And “sorting” is the other big thing, when it comes to algorithms. Challenge of the day: write a small Python function that sorts an existing list. (Not that it really matters, but again assume that the list only contains integers.)

When prompted with such a problem, the first thing that comes to mind might be something along the lines: well, I can create an empty list, loop over the original list and, for each element, find its place in the new (sorted) list, with a second loop. That is:

```
def sloppy_sort(values: list) -> list:
    sorted_list = []
    for value in values:
        for i, item in enumerate(sorted_list):
            if value <= item:
                sorted_list.insert(i, value)
                break
        else:
            # Executed if the loop finishes without `break`
            sorted_list.append(value)
    return sorted_list

l = [10, 1, 8, 5, 2, 7, 3, 9, 4, 6]
print(sloppy_sort(l))
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

This work as advertised. Unfortunately, this is also horrible in at least two different, unrelated ways. For one thing the complexity of this algorithm is quadratic (i.e., $O(n^2)$); you can tell that right away from the two nested loops. In addition, you need to allocate a list that will grow to occupy the same memory as the original one.

Note

`else` clauses on loops are a somewhat unusual Python feature that is described [here](#).

The first thing is easy to fix: we have neglected the fact that, at any point in the execution of the program, the new list is sorted, and for the purpose to find the index at which we would

insert the next item in the original list, we should do a binary search, which is $O(\log n)$ instead of $O(n)$.

```
import bisect

def slightly_better_sort(values: list) -> list:
    sorted_list = []
    for value in values:
        i = bisect.bisect_left(sorted_list, value)
        sorted_list.insert(i, value)
    return sorted_list

l = [10, 1, 8, 5, 2, 7, 3, 9, 4, 6]
print(slightly_better_sort(l))
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

You might be tempted to argue this is now $O(n \log n)$ in complexity, while, unfortunately, the cost of a list insertion is linear, and we are still where we started from: $O(n^2)$ and with a potentially big memory footprint. And this is where we stop guessing.

If you look at the [wikipedia page](#) for “Sorting algorithm”, you will find a long table of sorting algorithms available on the market. Although you are not an expert in algorithmic complexity by any reasonable metric, yet, you should be able to grasp the main take-away points from that table. In particular, the vast majority of the algorithms achieve the goal with a complexity of $O(n \log n)$.

Curiosities aside (if you have spent enough time on the wikipedia page you have probably stumbled across the neat concept of [Bogosort](#)) the main message is again: refrain from re-inventing the wheel. Python lists come with a `sort()`

```
l = [10, 1, 8, 5, 2, 7, 3, 9, 4, 6]
l.sort()
print(l)
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

that will happily sort *in place* with reasonable performance. Full disclaimer: this uses [Timsort](#) under the hood, in case you care.

And now, for something completely different, if you happen to be into sorting algorithms *and* Hungarian folk dance, you might enjoy this [article](#), along with the associated [videos](#).

5.5 Data structures

Data structures are often associated to algorithms in pedagogic resources, for the very good reason that in real life you often operate on complex structures, and at every step you need to know the cost of those operation in order not to be fooled. Take another look at the small snippet

```
def slightly_better_sort(values: list) -> list:
    sorted_list = []
    for value in values:
        i = bisect.bisect_left(sorted_list, value)
        sorted_list.insert(i, value)
    return sorted_list
```

Is this quadratic or not? Well, if it was only for the top-level loop and the binary search its complexity would be $O(n \log n)$, but, as we said, list insertion is an expensive (as in $O(n)$) operation in Python, and since n beats $\log n$ for large n , the answer is: yes, this thing is quadratic.

The next obvious question is: how do I know what is the complexity of a given operation on a given type? Well, that is a very broad question, and the answer depends a lot on the specifics of the context we are working in. If all we are talking about is Python native types, the [Time & Space Complexity Reference](#) is a good resource.

(On a related note, and sadly enough, the page is also yet another evidence of how the AI hype is destroying communities all over the web: *This wiki is in the process of being archived due to lack of usage and the resources necessary to serve it — predominately to bots, crawlers, and LLM companies.* But so goes life.)

5.5.1 Lists

Understanding the precise behavior of a given type requires a detailed knowledge of how the object is implemented internally, and we just don't have time for that—surely not for all the Python basic types. We shall take Python lists as an example, and see where this brings us.

In CPython, a list is basically a dynamic array of references to objects. Let's try and parse it. At the very fundamental level, what we are storing in the underlying array are *references* to the objects—think about addresses in memory where the actual objects are (independently) stored. (That is to say: if you ask Python what is the memory footprint for the list, it will only take into account the memory occupied by the references, not that necessary for the actual items.)

One of the key design elements is that, while lists can be heterogeneous, with each item in the list having its own different size, addresses in memory are all integers of the same size. In addition to this contiguous array of integers, CPython lists contain two additional fields, expressing the current *size* of the list and the *capacity* of the underlying array of references. The clever part is: the reserved capacity is generally larger than the size, so that appending a new element to the list is fast, up to the point where you saturate the capacity, and you have to resize the whole thing.

The following, admittedly cryptic, snippet might or might not help grasping how a Python list is implemented internally. Use it if it makes it any sense to you, or just skip over otherwise.

```
import ctypes

class PyListObject(ctypes.Structure):
    _fields_ = [
        ("ob_refcnt", ctypes.c_ssize_t),
        ("ob_type", ctypes.c_void_p),
        ("ob_size", ctypes.c_ssize_t),
        ("ob_item", ctypes.POINTER(ctypes.c_void_p)),
        ("allocated", ctypes.c_ssize_t),
    ]

values = [42, "hello", 3.5]

internal = PyListObject.from_address(id(values))

print(f"List address in memory: {hex(id(values))}")
print(f"Reference counter: {internal.ob_refcnt}")
print(f"List length: {internal.ob_size}")
print(f"List capacity: {internal.allocated}")

for index in range(internal.ob_size):
    address = internal.ob_item[index]
    value = ctypes.cast(address, ctypes.py_object).value
    print(f"[{index}] address={address:#x}, value={value!r}")
```

```
List address in memory: 0x7f01a6f69cc0
Reference counter: 1
List length: 3
List capacity: 4
[0] address=0x7f0235338110, value=42
[1] address=0x7f022030fb40, value='hello'
[2] address=0x7f021ae42b50, value=3.5
```

We are now ready to go through all the basic operations that Python lists support and see if we can make sense of their average and worst-case complexity:

- **get/set item:** since the references are organized in an array of contiguous memory slots of fixed size, the position of the appropriate reference can be easily calculated based on the index, and these are both $O(1)$;
- **get length:** the current length of the list, as we said, is stored as a separate field in the underlying data structure, so this is $O(1)$;
- **pop last:** this amounts to decrementing the size field by one unit (and, technically, we also have to return the ownership to the caller), which is also $O(1)$;
- **insert item:** inserting an item in the list at a given position requires shifting by one position to the right all the following references in the array, so inserting, as we said, is $O(n)$;
- **append item:** this one is interesting, because the behavior of the list relies on the fact that the capacity is deliberately larger than the size of the list (the details might depend on the Python version); if there is space left in the underlying array, appending an item is $O(1)$; when the size has grown to the full capacity of the array, a resize is necessary, which means that we have to create a new, larger array, copy all the stuff over, so that the worst case complexity is $O(n)$; it should be noted, however, that the worst case is comparatively rare, and the *amortized* worst case is still $O(1)$;
- **delete item:** just like inserting, deleting an item is $O(n)$ because, in order to keep all the references contiguous in the array, we have to shift by one position to the left all the references following the one that we have deleted;
- **sort:** this is using Timsort, which is $O(n \log n)$ in both the average and the worst case;
- **get minimum/maximum:** since lists are not explicitly keeping track of the maximum and minimum values, this requires a full iteration over the items, which is $O(n)$.

This was fairly quick, but again the [Time & Space Complexity Reference](#) is the place to start.

5.5.2 Dictionaries (aka hash tables)

We promised we would not go through all the Python native types and we shall hold on to that promise, but since they are ubiquitous in the language, we shall make an exception for dictionaries.

A dictionary is an associative container, uniquely mapping a set of keys to a corresponding set of values, that CPython implements in the form of a *hash table*—a peculiar data structure interesting on its own. The basic idea is as follow:

- we pre-allocate some space for holding references to the values; pretty much in the same spirit in which lists work, this space can grow or shrink as the dictionary evolves during the life cycle of our program;

- keys are mapped to indices in this target memory space through with a suitable function, that is customarily called the *hash function*—more about this in a second;
- when we need to access an item in the dictionary (be it for reading or writing) the hash function is applied to the key in order to calculate the address in the target space of the reference to the object itself.

The best part of hash tables is: getting, setting, inserting and deleting items all are $O(1)$ in the average case, as all them basically imply a single evaluation of the hash function.

Of course this is an oversimplification. The devil in the details, and there's a lot of details to be taken care of, here—devising the best hash function, handling collisions and optimizing the growing/shrinking strategy, just to name a few. The usual piece of advice applies here too: in the vast majority of cases you don't want to implement a custom hash table from scratch. But bear with me and let's continue this succinct overview.

First thing first, the hash function is supposed to be able to take literally anything as an input—a string, a number, you name it—and produce an integer withing a specified range in output. Generally speaking, there's a number of desirable properties that we would like a good hash function to have:

- be as fast as possible to calculate;
- be as uniform as possible in the output range;
- use all the available information in the key and send similar keys in very different hashes.

(The last two are meant to minimize collisions, i.e., different keys going in the same hash. It goes without saying: no matter how careful you are, if the index space is finite there will always be collisions to handle.)

How does hashing work in Python? Well, for one thing Python provides the `hash()` builtin, that you can use to experiment.

```
for item in (-2, -1, 1, 1., 1.000000000001, "Hello"):
    print(f"{item} -> {hash(item)}")
```

```
-2 -> -2
-1 -> -2
1 -> 1
1.0 -> 1
1.000000000001 -> 2306049
Hello -> -9045439524311981226
```

This is not nearly enough to reverse engineering the entire hashing infrastructure for native Python types, but it does provide some interesting clue as to what the basic design decisions for CPython were: integers, e.g., are hashed to themselves (except for -1 , which is reserved as

the C-level error return value), and numeric values that compare equal have the same hash value, even if they are of different types, as is the case for 1 and 1.0.

Can I hash literally any object in Python? Not quite—the following snippet, e.g., will raise an exception

```
l = [1, 2, 3]
print(hash(l))
```

```
TypeError: unhashable type: 'list'
```

```
-----
TypeError                                Traceback (most recent call last)
Cell In[14], line 2
      1 l = [1, 2, 3]
----> 2 print(hash(l))
TypeError: unhashable type: 'list'
```

Why is that? Well, as it turns out, you can only hash *immutable* types, which, if you think about for a second, makes a lot of sense: the content of a mutable type could change after insertion into a dictionary, invalidating the stored hash-table position! (And yes: you can only use immutable objects as keys of a Python dictionary.)

Now it is really time to move on, but before we do take a look at this last snippet and make sure you understand what is happening:

```
d = {}
d[3] = "Hi there!"
print(d)

d[3.] = "How are you?"
print(d)
```

```
{3: 'Hi there!'}
{3: 'How are you?'}
```

This is an odd corner of Python dictionaries—pay attention!

6 The Python scientific stack

Python is notorious for coming with *batteries included*. This is a humorous way of saying that the Python [standard library](#) is fairly extensive, and its capabilities are not limited to the bare minimum you would expect from a casual programming language. We have covered that in [Chapter 2](#).

Python is also, by its very own nature, a general-purpose programming language and, while the standard library allows you to send emails, or parsing all the most widely used text formats out there, it is not necessarily the best place to start if you crunch numbers as a day job. As a physicist and/or data scientist, you will need to dive into the Python scientific stack.

The Python scientific ecosystem is a thriving environment, with lots of interesting things happening every day—definitely something to stay tuned on. We shall only cover (and superficially, too) the *classical* packages:

- [NumPy](#): a numerical library offering a powerful n-dimensional array type, along with an extensive set of mathematical functions and facilities that inter-operate natively with it;
- [SciPy](#): the fundamental scientific package, building on top of NumPy and providing facilities for integration, optimization (aka fitting), interpolation, signal processing;
- [matplotlib](#): the standard package for visualization and plotting in Python;
- [pandas](#): a popular, high-level analysis tool—a must if you happen to work with excel files.

All of the above are sponsored by the [NumFocus](#) initiative, and you should take a look at the long list of [supported packages](#): chances are that you will find something that fits you.

6.1 NumPy

The fundamental package for scientific computing with Python.

At the very fundamental level, NumPy offers a new Python data type—a n-dimensional array:

```
import numpy as np

a = np.array([1., 2., 3.])
m = np.array([[1, 0], [0, 1]])
```

```
for item in a, m:
    print(item)
    print(f>Data type: {item.dtype}, shape: {item.shape}")
```

```
[1. 2. 3.]
Data type: float64, shape: (3,)
[[1 0]
 [0 1]]
Data type: int64, shape: (2, 2)
```

There is a number of convenient ways you can initialize a NumPy array, other than manually from Python iterables:

```
import numpy as np

print(f"zeros -> {np.zeros(5)}")
print(f"ones -> {np.ones(5)}")
print(f"full -> {np.full(5, 3.3)}")
print(f"linspace -> {np.linspace(0., 1., 5)}")
print(f"geomspace -> {np.geomspace(0.1, 1., 5)}")
print(f"logspace -> {np.logspace(-1, 0, 5)}")
print(f"arange -> {np.arange(0., 1.001, 0.2)}")
```

```
zeros -> [0. 0. 0. 0. 0.]
ones -> [1. 1. 1. 1. 1.]
full -> [3.3 3.3 3.3 3.3 3.3]
linspace -> [0.    0.25 0.5  0.75 1.   ]
geomspace -> [0.1      0.17782794 0.31622777 0.56234133 1.       ]
logspace -> [0.1      0.17782794 0.31622777 0.56234133 1.       ]
arange -> [0.  0.2 0.4 0.6 0.8 1. ]
```

Commit all of them to memory if you haven't done it already, because you will put each single one of them to good use, sooner or later. More details on the documentation page about [array creation](#).

6.1.1 Array and native Python types

Before we dig into the gory details of how you can use NumPy arrays in your daily workflow, it is useful to clarify some of the main differences with respect to the native Python data types.

When looked at superficially, NumPy arrays look a lot like Python lists:

```
import numpy as np

l = [1, 2, 3]
a = np.array([1, 2, 3])

print(l)
print(a)
```

```
[1, 2, 3]
[1 2 3]
```

Sure enough, they are both iterables, but that's pretty much it. For one thing Python lists are *heterogeneous*, while NumPy arrays are intrinsically *homogeneous* (more about types in a second). The semantic of the support that they offer to elementary arithmetic is completely different:

```
import numpy as np

l = [1, 2, 3]
a = np.array([1, 2, 3])

print(f"sum: {l + l} vs. {a + a}")
print(f"multiplication by a scalar: {l * 2} vs. {a * 2}")
```

```
sum: [1, 2, 3, 1, 2, 3] vs. [2 4 6]
multiplication by a scalar: [1, 2, 3, 1, 2, 3] vs. [2 4 6]
```

That is: for native Python lists *sum* means concatenation and *multiplication* by a scalar means simply repetition; in the case of NumPy arrays both operations are intended as *element by element*. Stop for a second and think about what is the most useful semantic for data analysis.

There are other notable differences, some of which we shall discuss in more details in the following of this chapter. Python lists and arrays have a completely different layout (and footprint) in memory and, as we shall see, they do operate at a fundamentally different speed. Arrays offer a much more powerful indexing and slicing. Arrays inter-operate natively with the NumPy and SciPy [mathematical functions](#).

But, before we move on and look into all of this, there is another odd corner that is worth pointing out—especially for those of you who learned (or are learning) Python as their first language. NumPy arrays always come with in a specific *type*—we know this already:

```
import numpy as np

a1 = np.linspace(0, 10, 11, dtype=int)
a2 = np.linspace(0, 10, 11, dtype=float)

print(a1, a1.dtype)
print(a2, a2.dtype)
```

```
[ 0  1  2  3  4  5  6  7  8  9 10] int64
[ 0.  1.  2.  3.  4.  5.  6.  7.  8.  9. 10.] float64
```

As you might notice, no matter how you specify the data type, since at the fundamental level NumPy is largely implemented in C, it is the underlying C data types that it uses—not Python’s. (Suggested [reading](#).) Now, for floating point numbers this does not make a whole lot of difference, but when you do operate on arrays of integers the consequences are dramatic, as you immediately lose the luxury of infinite-precision integer arithmetic that you are so used to:

```
import numpy as np

a = np.array([2, 4, 8, 16])

print(a.dtype)
print(a**16)
```

```
int64
[          65536      4294967296 281474976710656           0]
```

Make sure you understand this before you move on: you see the last zero in the output? The fact is: NumPy is storing the elements of our arrays as (signed) 64-bit integers, which means that the maximum value they can hold before overflow occurs is

$$2^{63} - 1 = 9223372036854775807$$

whereas the value we are trying to represent is too large:

$$16^{16} = (2^4)^{16} = 2^{64} = 18446744073709551616.$$

This happens in real life. Really. You are advised.

6.1.2 Indexing and slicing

This is covered in details [here](#). To first approximation, indexing works just as it does in ordinary Python lists, that is

```
import numpy as np

l = [1, 2, 3]
a = np.array(l)

print(l[0])
print(a[0])
```

```
1
1
```

Working with multidimensional arrays is fairly intuitive once you commit to memory that *NumPy uses C-order indexing, which means that the last index represents the most rapidly changing memory location*. For 2-dimensional arrays (i.e., ordinary matrices, or rank-2 tensors) this means that the first index indicates the rows, while the second runs over the columns:

```
import numpy as np

a = np.linspace(0, 15, 16, dtype=int).reshape((4, 4))

print(a)
print(a[0])
print(a[0][3], a[3][0])
print(a[0, 3], a[3, 0])
```

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]
 [12 13 14 15]]
[0 1 2 3]
3 12
3 12
```

You can slice arrays just as you do with any Python iterable, with the additional shortcuts that NumPy offers for addressing multi-dimensional arrays:

```

import numpy as np

a = np.linspace(0, 15, 16, dtype=int)

print(a)
print(a[2:6])
print(a[0:16:2])

a = a.reshape((4, 4))

print(a)
print(a[1:3, 1:3])

```

```

[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15]
[2 3 4 5]
[ 0  2  4  6  8 10 12 14]
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]
 [12 13 14 15]]
[[ 5  6]
 [ 9 10]]

```

And you can index numpy arrays with either integer or boolean arrays, which, as we shall see in a moment, is *really* handy in many situation.

```

import numpy as np

a = np.linspace(0, 15, 16, dtype=int)
b = np.array([1, 3, 5, 7, 11])
c = np.array([True, False] * 8)

print(a)
print(a[b])
print(a[c])

```

```

[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15]
[ 1  3  5  7 11]
[ 0  2  4  6  8 10 12 14]

```

Note

A boolean array used for the purpose of indexing another array is customarily called a *mask*. As we shall see in a second, this is important when vectorizing problems, as this trick allows to emulate what we would achieve with conditional statements in a normal loop.

As we said, all the gory details are in the documentation page about [indexing](#), and it doesn't make sense to repeat everything here.

6.1.3 Broadcasting

This is covered in details [here](#). As we said, arithmetic operations on NumPy arrays are generally to be intended as *element-wise*, that is

```
import numpy as np

a = np.array([1., 2., 3.])
b = np.array([4., 5., 6.])

print(a + b)
print(a * b)
```

```
[5.  7.  9.]
[ 4. 10. 18.]
```

Interestingly, arithmetic operations in NumPy are not limited to arrays with the same exact shape. Look at the following examples:

```
import numpy as np

c = np.pi
a1 = np.array([1, 2, 3, 4])
a2 = np.linspace(0, 15, 16, dtype=int).reshape((4, 4))

print(c)
print(a1)
print(a2)

print(a1 * c)
print(a2 * c)
print(a1 * a2)
```

```

3.141592653589793
[1 2 3 4]
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]
 [12 13 14 15]]
[ 3.14159265  6.28318531  9.42477796 12.56637061]
[[ 0.          3.14159265  6.28318531  9.42477796]
 [12.56637061 15.70796327 18.84955592 21.99114858]
 [25.13274123 28.27433388 31.41592654 34.55751919]
 [37.69911184 40.8407045  43.98229715 47.1238898 ]]
[[ 0  2  6 12]
 [ 4 10 18 28]
 [ 8 18 30 44]
 [12 26 42 60]]

```

This behavior goes under the name of *broadcasting*. NumPy broadcasting rules are explained [here](#) and we will not overlap with that. In the simplest cases the thing is fairly intuitive, but you should take your time to master it, because broadcasting is central to *vectorization*, as we shall see in the next section.

Before we move on, though, it is worth mentioning that this very basic idea of element-wise operations is not limited to elementary arithmetic, and covers all the mathematical functions defined in NumPy and derived scientific packages:

```

import numpy as np

a = np.linspace(0.1, 1., 10)

print(a)
print(np.log10(a))
print(np.exp(a))
print(np.sin(a))

```

```

[0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1. ]
[-1.          -0.69897    -0.52287875 -0.39794001 -0.30103    -0.22184875
 -0.15490196 -0.09691001 -0.04575749  0.          ]
[1.10517092 1.22140276 1.34985881 1.4918247  1.64872127 1.8221188
 2.01375271 2.2254093 2.45960311 2.71828183]
[0.09983342 0.19866933 0.29552021 0.38941834 0.47942554 0.56464247
 0.64421769 0.71735609 0.78332691 0.84147098]

```

Make sure you fully understand that, in order for this to work, you really need mathematical functions that are designed for the purpose—the Python standard library is not helping, here

```
import math

import numpy as np

a = np.linspace(0.1, 1., 10)

print(math.log10(a))
```

TypeError: only length-1 arrays can be converted to Python scalars

```
-----
TypeError                                Traceback (most recent call last)
Cell In[14], line 7
      3 import numpy as np
      5 a = np.linspace(0.1, 1., 10)
----> 7 print(math.log10(a))
TypeError: only length-1 arrays can be converted to Python scalars
```

But don't worry: if you need mathematical functions, SciPy has [your back](#).

6.1.4 Vectorization

If there is one thing that Python is not famous for, that is speed. Quite the contrary, Python has a fairly consolidated name for being sloooooow.

On the face of it, you shouldn't be surprised. We have been bragging all along about all the nice features that Python offers—the fact that you don't have to declare variables, or that containers can be heterogeneous, infinite-precision integer arithmetic, you name it. Since, as Richard Feynman used to say, there is no free lunch in quantum electrodynamics, all these good things come at a price. And the price, in this case, is called speed.

Do I really care, you ask? Well, it depends. If you are parsing a text file or fetching a web page, probably not. If a task takes 2 ms to complete, and you are not doing it 1000 times a second, beating down the execution time by a factor of ten wouldn't really make a difference. As a human, all you notice is that you press enter and you are immediately done in both cases.

If, on the other hand, you are performing a CPU-intensive processing on a TB of data, then probably yes—speed matters. If you can beat down the execution time from 20 minutes to 10 seconds, that is a game changer—you don't have to go have a coffee in between two consecutive runs, and since too much coffee is bad for you, that is a net positive. Also, you have a much shorter development cycle, which will help you in the long run.

(Before you move one, make sure you have fully digested *both* paragraphs above. You don't need to optimize *everything*. So much so that the saying goes: *premature optimization is the root of all evil*. We shall get back to that in Chapter 11.)

Fortunately, when you do need to optimize, NumPy is one of the things that might come to rescue. Why is that? Well, NumPy is written in C as a Python extension, and the underlying functions are highly optimized to crunch numbers. In other words, when you perform array operations using NumPy, you are actually executing optimized C code. How cool is that?

Let's see that in action. As you probably know, every programming language comes with a pseudo random number generator (PRNG), which is the physicist's best friend when it comes to setup simulations. (PRNGs typically generate numbers equi-distributed between 0 and 1, and then there are well-defined techniques to transform them into arbitrary distributions.) Python is no exception, as the standard library provides the `random` module; on top of that, NumPy offers its own `PRNG`.

Which one shall we use? Well, let us assume we want to generate 1 million random numbers, and let's check out both!

```
1  import random
2  import time
3  import numpy as np
4
5  n = 1000000
6
7  # The slow way: explicit for loop in Python.
8  t0 = time.time()
9  x = []
10 for i in range(n):
11     x.append(random.random())
12 dt = time.time() - t0
13 print("Elapsed time: %.4f s" % dt)
14
15 # The quick way: using numpy.
16 t0 = time.time()
17 x = np.random.random(size=n)
18 dt = time.time() - t0
19 print("Elapsed time: %.4f s" % dt)
```

Elapsed time: 0.0663 s

Elapsed time: 0.0087 s

i Note

Note the two block of code are (roughly) equivalent from a functional point of view, but surely not on a number by number basis. For one thing the pure-Python version creates a *list* of random numbers, while the NumPy version creates an *array* (of the same length). That said, Python uses the [Mersenne Twister](#) as the core generator, while NumPy uses by default [PCG-64](#), a 128-bit implementation of O’Neill’s permutation congruential generator. And, even if they used the same underlying algorithm, you would have to set the seed explicitly to get the same sequence. None of this is really relevant to the point we are trying to make.

All right. The main take-away message here is: NumPy generates the same amount of random numbers as plain Python in about a tenth of the time—not bad. Let’s be very explicit: the fact that in Python you need a for loop (lines 9–11) while the rough NumPy equivalent is a one-liner (line 17) does not mean that there is no loop in the second version. The loop is *implicit* and, as we said, it happens at the speed of C. Much faster than Python.

Which immediately brings to the corollary: *avoid for loops in pure Python when crunching numbers*. It’s ok to use a for loop if you have to iterate over the lines of a small text file, of the files in a folder on your machine. Not so much over millions and millions of numbers. You get the difference.

Let us push this a little bit farther in the context of a toy Monte Carlo simulation of a gas detector. More specifically, let’s say we have a beam of $E_0 = 6$ keV monochromatic X-rays impinging on a gas cell where the average energy necessary to create an electron-ion pair by ionization is $W_i = 30$ eV. Every time a photon undergoes photoelectric absorption, a photoelectron is emitted, and the latter ionizes the gas creating $E_0/W_i = 200$ primary electrons on average (we shall assume this is Poisson distributed), each of which is in turn *multiplied* through a avalanche-like process with a mean gain $G = 100$, with an underlying exponential distribution. The total number of ionization pairs is our measure of the energy, and we are interested in measuring the energy resolution of the detector. (Note this is the sum of 200 exponential random variables which, by virtue of the central limit theorem, is approximately gaussian distributed. I will leave it to you to calculate the expected mean and variance.)

i Note

A little piece of advice before we actually move forward with our plan. If you happen to do *real* work on gas detectors, be advised we are making at least to big mistakes as far as the underlying physics goes, here. For one thing, the primary ionization is not Poisson distributed—the phenomenon is under-dispersed and the corresponding variance is smaller than what you would expect based on the Poisson distribution by a factor F called the [Fano Factor](#). Additionally, in a realistic gas detector, the gain distribution is more complex than an exponential, and is often parametrized as a Polya function. But

this is all about explaining vectorization, not doing real simulations.

How would we go about simulating this process in Python? Well, here is something that might do the trick:

```
import random
import time

import numpy as np
from matplotlib import pyplot as plt

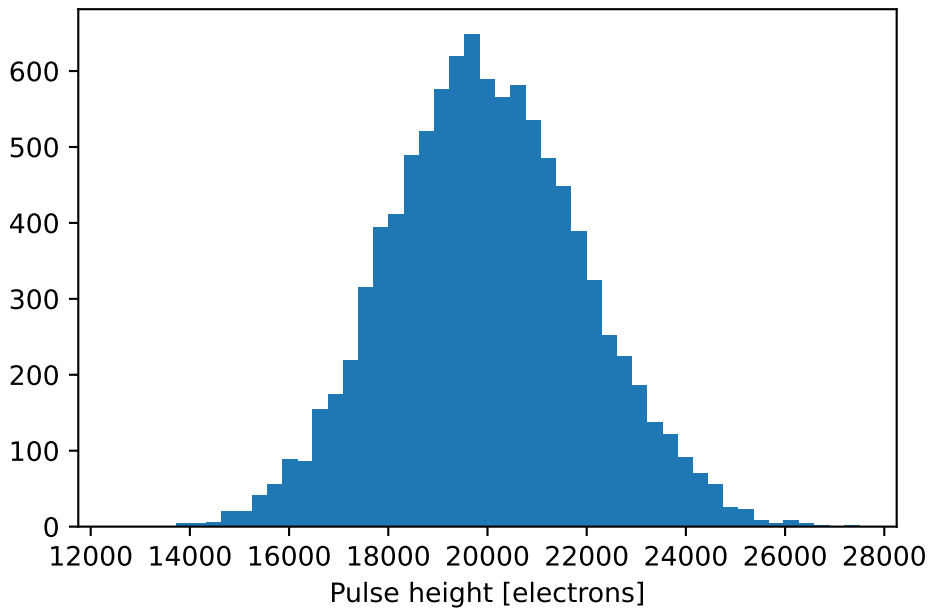
num_photons = 10000
E_0 = 6000. # eV
W_i = 30. # eV
G = 100.

pha = []
t0 = time.time()

# Outer loop over the impinging X-rays.
for i in range(num_photons):
    # Initialize the total number of the electron-ion pairs that are created
    # by ionization for this particular event.
    total_ionization = 0
    # Extract the number of primary electrons created by the photoelectron.
    # Too bad the random module in the standard lib does not provide the
    # Poisson distribution, and we do have to resort to numpy...
    num primaries = np.random.poisson(E_0 / W_i)
    # Inner loop over the primary electrons...
    for j in range(num primaries):
        # For each primary electron, extract the number of electron-ion pairs
        # generated in the avalanche.
        total_ionization += random.expovariate(1. / G)
    pha.append(total_ionization)

dt = time.time() - t0
plt.hist(pha, bins=np.linspace(12500., 27500., 50))
plt.xlabel("Pulse height [electrons]")
print(f"Simulation completed in {dt:.3f} s")
```

Simulation completed in 0.291 s



Now, can I speed things up using NumPy? You bet you can!

```
import time

import numpy as np
from matplotlib import pyplot as plt

rng = np.random.default_rng()

num_photons = 10000
E_0 = 6000. # eV
W_i = 30. # eV
G = 100.

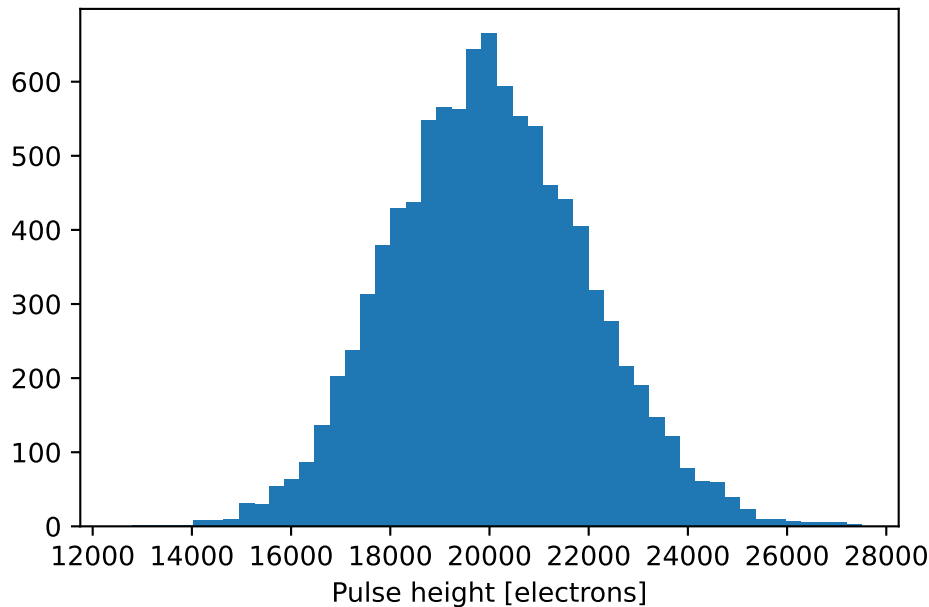
t0 = time.time()

# Create an array with the number of primaries for each single X-ray.
num_primaries = rng.poisson(E_0 / W_i, size=num_photons)
# Populate an array with as many copies of the event id as the corresponding
# primary electrons
event_id = np.repeat(np.arange(num_photons), num_primaries)
# Extract the gain for each single primary electron.
event_gain = rng.exponential(scale=G, size=event_id.size)
```

```
# Sum all the ionization pairs, on an event-by-event basis, to get the "energy".
pha = np.bincount(event_id, weights=event_gain, minlength=num_photons)

dt = time.time() - t0
plt.hist(pha, bins=np.linspace(12500., 27500., 50))
plt.xlabel("Pulse height [electrons]")
print(f"Simulation completed in {dt:.3f} s")
```

Simulation completed in 0.014 s



Let's take a breath. The histograms look similar to each other, which is good. (As I said: I will leave it to you to assess whether the mean and the standard deviations look reasonable, based on our initial assumptions.) The numpy version is roughly a factor of 20 faster—this is also good news.

But the one thing you should be *really* be paying attention to is the fundamentally different way the problem is cast in the two cases. In the pure-Python version we have an outer loop over the events, and we process each one before we move to the next; in the second version we perform each physical step of the simulation (i.e., extract the number of primary electrons, extract the avalanche size) *for all the events at the same time*.

This latter approach is sometimes called *vectorization*, and you readily see the pros and the cons. The pure-Python version is easier to read, as it proceeds in the same temporal order that it would in real life. In the vectorized version we are mixing things up—in this particular case

first comes the primary ionization for all the X-rays, then the avalanche for all the primary electrons. The flow is not nearly as natural to follow, and it also implies a larger memory footprint, because we have to keep track of larger arrays. The advantage: we can delegate all the (now implicit) loops to NumPy and they happen at the speed of C.

Vectorization with NumPy is a difficult language to learn. At the very fundamental level the basic rules are:

- all the loops are implicit, and you keep everything in memory;
- conditional branches in the control flow are achieved via masks;
- in many cases you have to resort to (more or less) obscure NumPy functions such as `bincount` to achieve what you need.

Mastering this language is one of the avenues to be able to do a sensible simulation in Python.

6.2 SciPy

Fundamental algorithms for scientific computing in Python .

We are going very fast on this one. Among the various things that SciPy has to offer we list:

- `scipy.integrate`: integration;
- `scipy.interpolate`: interpolation;
- `scipy.optimize`: optimization (including curve fitting);
- `scipy.stats`: statistical functions.

6.2.1 Interpolating splines

Splines are interesting objects, and not nearly as widely known as they deserve, which is why we are taking a small detour, here. (When we have a series of point, we physicist have a tendency to insist fitting a function through them, where in some cases interpolating is really the right thing to do.)

Technically speaking, *splines* are functions passing through a series of given points, defined piecewise by polynomials of fixes degree k . ($k = 3$ is a fairly popular choice, but values between 1 and 5 are used.)

If you think about for a second, for n point you have kn parameters, which gives you enough freedom to:

- have the spline pass exactly through every point;
- have the first $k - 1$ derivatives continuous everywhere.

All the coefficients of the polynomials are calculated once and forever at creation time, based on the coordinates of the control points.

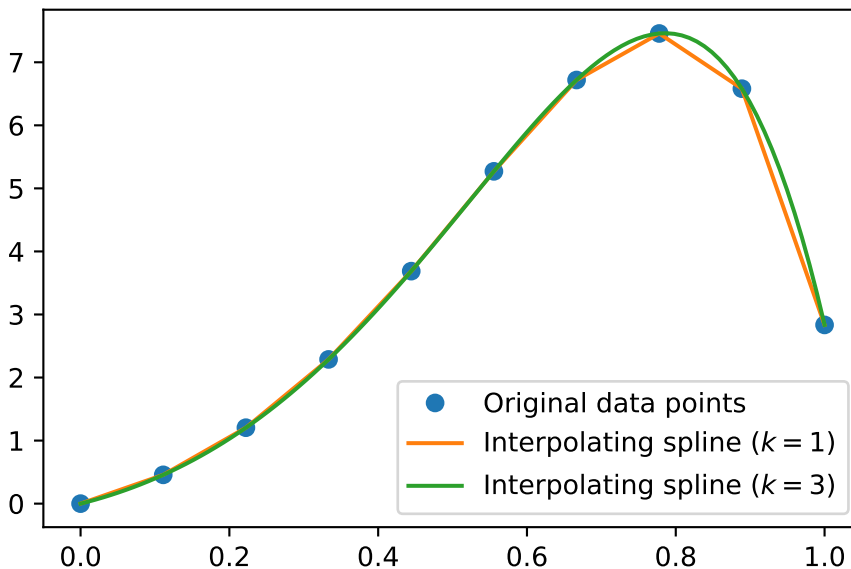
Making a spline using SciPy is as easy as

```
import numpy as np
from scipy.interpolate import InterpolatedUnivariateSpline

x = np.linspace(0., 1., 10)
y = np.exp(3. * x) * np.sin(3. * x)

s1 = InterpolatedUnivariateSpline(x, y, k=1)
s3 = InterpolatedUnivariateSpline(x, y, k=3)

xgrid = np.linspace(0., 1., 100)
plt.plot(x, y, "o", label="Original data points")
plt.plot(xgrid, s1(xgrid), label="Interpolating spline ($k = 1$)")
plt.plot(xgrid, s3(xgrid), label="Interpolating spline ($k = 3$)")
plt.legend()
```



For $k = 1$ you get a broken line, whose first derivatives are discontinuous at the control points; for $k = 3$ you get a so-called *cubic spline*, which is well behaved in most situations. If all you care is literally getting a smooth curve that goes through a series of point, with no underlying physics or insight, splines are often superior to polynomial interpolation or curve fitting.

There is a few more extra goodies that you get for free when you buy into splines. Evaluation is fairly inexpensive: all you need to do is find the two control points that include the x value

where you want to evaluate, and since the input array of x coordinates is required to be sorted, this can be done via a binary search with a complexity $O(\log n)$. In addition, since this is all as simple as polynomials, derivatives and integrals are easy:

```
import numpy as np
from scipy.interpolate import InterpolatedUnivariateSpline

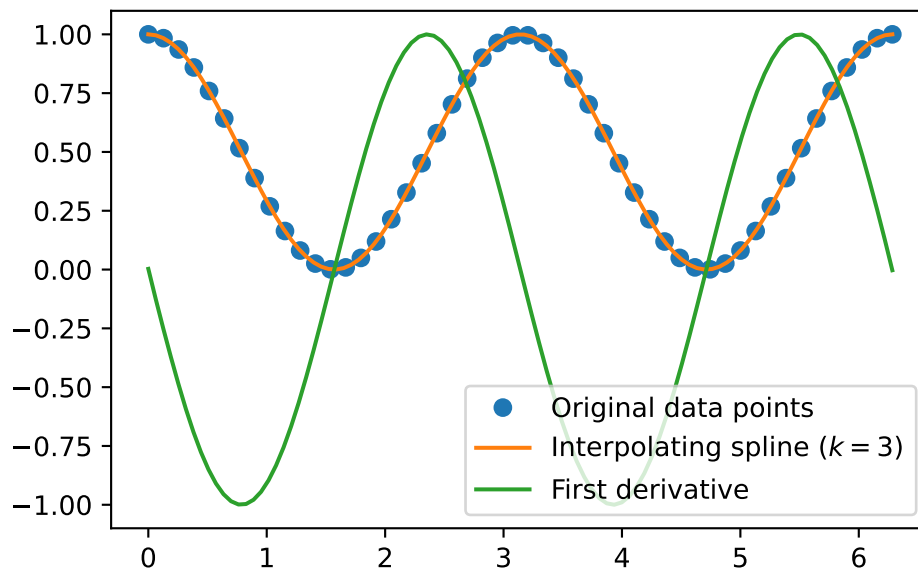
x = np.linspace(0., 2. * np.pi, 50)
y = np.cos(x)**2

s = InterpolatedUnivariateSpline(x, y, k=3)

xgrid = np.linspace(0., 2. * np.pi, 100)
plt.plot(x, y, "o", label="Original data points")
plt.plot(xgrid, s(xgrid), label="Interpolating spline ($k = 3$)")
plt.plot(xgrid, s(xgrid, 1), label="First derivative")
plt.legend()

print(s.integral(0., 2. * np.pi)) # <- this should be approximately pi
```

3.1416006818156785



Really: take a few minutes to dive into spline, because they might come a time in your life when they are the right tool to use.

6.3 pandas

A fast, powerful, flexible and easy to use open source data analysis and manipulation tool.

And we are at the end of the chapter. Not before we mention pandas, though.

Chances are that, sooner or later, you will have to deal with spreadsheets for a particular task. (It might be an administrative task, as opposed to one related to pure research, but keep in mind these will come too.) When that happen, pandas `read_excel()` function is there for you. Basic usage is as simple as

```
import pandas as pd

df = pd.read_excel("data.xlsx")
```

and this will give you a pandas `DataFrame` object.

We don't have time to drag this any longer, but pandas is definitely another keyword that you want to commit to memory and have it at hand in case of need.

7 Object oriented programming

Warning

The material is being migrated—see [here](#) and [here](#)

8 Advanced Python

Warning

The material is being migrated—see [here](#) and [here](#)

8.1 Error handling

8.2 Iterators and iterable

8.3 Generators

8.4 Decorators

8.5 Miscellanea

8.5.1 Dataclasses

9 Testing

How do I make sure that my awesome, glorious program is correct? Good question, and the short answer is: in real life you don't. Especially if your code is (even moderately) complex—not even mentioning whether it is asynchronous or is meant to communicate with hardware.

For compiled high-level languages, at the very least, the compiler makes it for a first barrier for all obvious (and a good deal of less-than-obvious) mistakes—if you get a type wrong, just to name one, it will happily refuse to move forward. Unfortunately Python is interpreted and this kind of safeguard is just not there: the interpreter will exit upon an syntax error, but that's pretty much it. All the rest is caught at runtime. If you are in production, that simply means *too late*.

Just so that we do not focus too much on the negative side of things, this is not to say that there is nothing you can do. In fact, besides paying attention, there are at least two things that you can do, even in interpreted languages:

1. unit testing; and
2. static analysis.

Generally speaking, physicists hate both, and profoundly too. But if you are writing code, they should come right next to version control in your work-flow toolbox. This is exactly what this chapter is about.

9.1 Unit testing

Unit testing is about testing that your code does the right thing in a well defined set of specific conditions where you know the answer in advance. Let us illustrate this in a trivial case:

```
def square(x):
    """Toy function returning the square of x.
    """
    return x**2.

def test():
    """Dumb test battery.
    """
```

```
assert square(2.) == 4.
assert square(2) == 4.
assert square(-2.) == 4.
print("All tests passed!")

if __name__ == "__main__":
    test()
```

All tests passed!

This makes it for a sensible structure, at least in principle: here is my small Python module, exporting the function `square()`, and a small test harness that I can execute by simply invoking the file from the terminal. I have a few, representative cases covered, here: positive and negative floats, as well as an integer. (It goes without saying, we can always add more.) If everything goes well, I get a nice “All tests passed!” in response.

Note

In Python, the keyword `assert` checks that a condition is true, and raise an `AssertionError` exception otherwise. (You will learn what is an exception in Chapter 8.) Note assertions can be disabled by running Python with the `-O` (for *optimized*) command-line switch, so you should be careful what you use the `assert` for.

9.1.1 Unit testing basics

We are on the right track, but not quite there, yet. In order to make unit testing really effective we need a sensible set of design rules and good practices, as well as some logistical infrastructure to cover the practical side of things.

First and foremost, *break up your program in small, isolated pieces (e.g., functions or methods), each encapsulating a well-defined and (if at all possible) simple functionality, and test each piece separately*. In practice, this is accomplished by means of a sensible hierarchy of functions and classes—which is often the hardest thing to get right when you structure your code. If you make sure that each single piece behaves as expected in the largest possible set of external conditions (e.g., with different input arguments) then you add confidence that the whole thing behaves as intended too. This is generally much simpler than testing the whole program at once—which you will have to do, too.

No matter what you do, if your code is actively used, you can be sure that it will change with time—in response to bugs that users will find, or new features they will request, or even

in response to the natural evolution of the underlying language. If your codebase is (even moderately) complex, each change, no matter how small it is, will have a huge phase space for breaking things in unrelated places, through ramifications that you did not anticipate. If you have an extensive suite of unit tests, chances that you catch any possible problem early increase. You cannot be 100% sure, but, generically speaking, your confidence increases with the coverage of your unit tests.

What does make a unit test a *good* unit test? Well, it is hard to tell on purely speculative grounds, but there are some common features that you typically would like to have:

- test a very specific behavior with a binary output—a test should pass or fail;
- run as quickly as possible, since in the long run you will want *many* of them;
- be deterministic, i.e., produce the same result every time, even when you need a PRNG;
- cover normal cases, edge cases, and errors; and as many as possible.

Reality is, physicists tend to work with a development model that one might call physicist-driven development (PDD), i.e.:

1. write a whole bunch of code that does something—it doesn't matter how;
2. swear you will come back to it tomorrow and write the unit tests;
3. add more code, doing yet more things, and more complicated.

Professional programmers, on the other hand, are more prone to what generally goes under the name of test-driven development (TDD), which sounds more like:

1. write an empty placeholder for your new function;
2. write all the unit tests that you can imagine—they will all fail, initially;
3. implement your function and tweak it until all the tests pass.

I'll let you pick which one makes more sense. TDD is not mandatory, and there are alternative, viable ways to do business. PDD is a recipe for disasters. Period.

9.1.2 Unit testing in practice

Back to the logistical side of things. The snippet of code we have shown at the beginning of this section, if admirable in spirit, comes with a number of shortcomings. Most notably, everything happens manually: you have to run the script yourself, and you have to inspect the output yourself. As your code grows in complexity, this is just not sustainable. *Unit testing should happen automatically, all the time, and somebody or something should come get you out of bed whenever a test fails.*

The Python standard library provides a module called `unittest`. Just like the `logging` module, it is one of the few modules in the standard lib that did not withstand the test of time and, while kept for backward compatibility, it is rarely used for new projects.

If you are starting fresh, you might as well opt for the third-party package [pytest](#), as everybody else is doing these days. From a practical standpoint, this means creating a folder in your repository called `tests` and move all your test code in a series of dedicated Python modules in there, which each module being a series of loose function whose name starts with `test_`

```
from mymodule import square

def test_square():
    """Unit test for the square function.
    """
    assert square(2.) == 4.
    assert square(2) == 4.
    assert square(-2.) == 4.
```

You might have noticed that we have removed the final `print()` statement: there is no need for it, since it is now `pytest` that is responsible for collecting and running the tests, and examining the output. How? Simple:

`pytest`

I know: it sounds magic, but it is really that simple. All you have to do is concentrate on writing the tests, and `pytest` will take care of all the bookkeeping.

Ah, one last thing: you are welcome to trigger the tests manually, but, as we said, unit tests should run automatically no matter what you do. This is sometimes referred to as *continuous integration*, and the standard way to achieve that in 2026 is through GitHub or Codeberg *actions*. The basic idea is that under specific conditions (e.g., when you push a change on the main branch, or on a branch with a pull request) a complete run of the unit tests is triggered and you get an email if something goes wrong.

If you want to really see how all these things play out in practice, go ahead and look (again) at [metarep](#)—make sure to hit the links to the [unit tests](#) and the [GitHub actions](#).

How do you integrate all of this in your daily workflow? Easy! Did you just find a bug in your code? Make sure you add a unit test along with the fix, so that you will never be hurt again by *that* particular bug. (There will be others, of course.) Are you adding a new feature? Make sure the new code is covered by unit tests.

9.2 Static code analysis

By its very nature, as we said, Python will show you all the errors at runtime. Now, say you have a bug in a part of the code that is exercised very rarely, and not covered by unit tests: the

very first time that you *do* exercise that particular path, the Python interpreter might crash, or happily do something that is not what you intended it to do. It might take years for even realizing that there was a bug in the first place.

If you think about, many common mistakes can be found by just looking at the code and without running it. And in fact you might argue that all of them can, at least in principle, provided that your sight is sharp enough. The point is: a good part of it can be done programmatically. Generally, *a program will not understand your program, but a program can be designed to spot specific kinds of errors and inconsistencies*. This is customarily called *static code analysis*; [pylint](#) and [ruff](#) are popular tools doing just that.

Note

For good or bad, this is becoming more and more uncertain as the AI revolution unfolds and LLMs become better and better at understanding code and finding mistakes. While this is true in many different areas, the implications vary. Unit tests will presumably continue to be relevant for the foreseeable future: you might have a horde of agents writing unit tests for you, but you will always need unit tests—which means that you will presumably have to understand what a unit test is. On the other hand, *deterministic* code analysis might fall out of fashion in favor of AI-driven code review, and this section of the notes might become obsolete. We'll see.

Take the following snippet, for example.

```
x = 1.
y = 2.
very_uncommon_condition = False
if very_uncommon_condition:
    print(x + z)
else:
    print(x + y)
```

3.0

See anything wrong? If we entered the first branch of the `if` statement we would have crashed because we mis-typed `y` as `z`.

Let's feed this into `pylint`, and see what happens

```
[lbaldini@nbbaldini ~]$ pylint snippet.py
***** Module snippets.linting1
snippet.py:1:0: C0111: Missing module docstring (missing-docstring)
```

```

snippet.py:1:0: C0103: Constant name "x" doesn't conform to UPPER_CASE naming style (invalid
name)
snippet.py:2:0: C0103: Constant name "y" doesn't conform to UPPER_CASE naming style (invalid
name)
snippet.py:3:0: C0103: Constant name "very_uncommon_condition" doesn't conform to UPPER_CASE
name)
snippet.py:5:14: E0602: Undefined variable 'z' (undefined-variable)
-----
Your code has been rated at -5.00/10 (previous run: -5.00/10, +0.00)

```

For one thing I did not follow [PEP8](#) and got some unfortunate variable name. This is excusable, but the worst (or best, depending on how you look at it) part is that `pylint` did find the typo! It also assigned a score (−5 out of 10, for the record) to the overall coding.

We are not going to drag this any longer, but you should consider using static code analysis routinely as you program in Python. Static analysis tools tend to be quite verbose, it is true, to the point where they can become annoying; in addition, they try and enforce many different (good) things at once:

- formal correctness;
- efficiency;
- avoiding anti-patterns;
- style guides.

But they are also typically highly customizable, and you can can mute errors you don't care about with only a few lines in your `pyproject.toml` file. (See [here](#) for an example.) Including some static analysis in your continuous integration, for instance, is often a good idea. Up to you to strike the right balance.

10 Documentation

If you look at the documentation of any reasonably maintained package, you will notice that, among other things, it covers the application programming interfaces, or APIs. That is to say, it provides the full description of the classes and functions that the package exposes—not only a top-level summary of how they are intended to be used, but also all the gory details of the arguments and return values of functions and class methods.

Take the documentation of our favorite function `scipy.curve_fit()`, for instance; you have, in order:

- the full signature of the function;
- a short, top-level description;
- a detailed description of all the arguments that can be passed to the function;
- a detailed description of the return values;
- a detailed description of the exceptions that can be raised;
- additional notes and reference.

Lots of stuff, eh?

Now, one thing that you might start wondering is: how do people keep the code and the documentation in synch? Do they physically live in separate files? And: if I ever change the signature of my function in the code, how do I make sure I remember to update the documentation? Because if the documentation starts diverging away from the underlying code, that is pretty bad—wrong documentation is definitely worse than no documentation at all.

The main message in this chapter is: documentation should live as close as possible to the code it describes—possibly in the very same file—and the documentation process should be automated wherever it makes sense. *Use one single source of truth and avoid copy and paste at all costs.*

i Note

Simple exercise: click on [this link](#) to the `scipy.curve_fit()` documentation and take a good look. Next to the signature of the function you will find another link to the [source code](#) for the same function. Open the latter in another tab and switch back and forth between the two. Anything that captures your attention?

10.1 Sphinx

Different languages have different toolchains of choice, when it comes to documentation. For Python, everybody uses [Sphinx](#). And, in this context, everybody really means *everybody*: [Python](#), [numpy](#), [scipy](#), [matplotlib](#).

When you think about documenting a software project, there are actually two equally important parts to it, namely:

- descriptive text explaining the purpose and usage of the project (this might include, among other things, a user guide, examples, tutorials and so on and so forth);
- technical documentation (i.e., the APIs), describing the inner workings of the code.

Just to put things in context, the reference page for the [optimize](#) module in SciPy is a good example of the first, while the documentation of the [curve_fit\(\)](#) function, that we now know by heart, belongs to the second category.

Sphinx handles both things equally well—the first is expressed in the form of a series of .rst markup files, while the second is extracted from the *docstrings* in the Python code. The whole thing is driven via a python configuration file that you can create with [sphinx-quickstart](#) utility.

10.1.1 General documentation

Descriptive documentation goes into dedicated .rst markup files. (As a matter of fact you will need at least a `index.rst` file that acts as the main entry point for the whole thing. It goes without saying, you can have as many .rst files as you see fit.)

There is nothing particular to say here, except for the fact that Sphinx uses [reStructuredText](#) as the particular markup language. The documentation will get you started with the basic syntax.

Sphinx come with an extensive set of built-in [extension](#) that do lots of interesting and useful things. At the very least you will want to use

- [apidoc](#) (generate API documentation from Python packages);
- [autodoc](#) (include documentation from docstrings);
- [napoleon](#) (support for the NumPy and google style docstring).

And of course there are also [contributed extensions](#) doing even more amazing thing. You are definitely covered.

10.1.2 Documenting the interfaces

And now about documenting the interfaces. This is typically handled through proper *doc-strings*—properly formatted pieces of text delimited by triple quotes that accompany classes and functions.

We are not going into all the gory details, but basically if instead of coding your functions like this:

```
def square(x):  
    return x**2.
```

you do something along the lines of

```
import numpy as np  
from numpy.typing import ArrayLike, NDArray  
  
def square(x: ArrayLike) -> float | NDArray:  
    """Return the square of a number or an array.  
  
    By virtue of the ``2.`` rather than ``2`` in the exponent, the  
    result is always floating point, even when the input is integer.  
  
    Parameters  
    -----  
    x : array_like  
        Value or values to square.  
  
    Returns  
    -----  
    array_like  
        Squared value or values.  
  
    Examples  
    -----  
    >>> square(2)  
    4.0  
    >>> square(np.array([1, 2, 3]))  
    array([1., 4., 9.])  
    """  
    return x**2.
```

that will go a long way toward a good documentation for your package. You can dive into the Sphinx documentation and find your way have it pick up your docstrings.

Note this is using the [NumPy style](#) docstrings. Alternative possibilities include the [google style](#) and the plain [Sphinx style](#).

10.1.3 Compiling the documentation

Once you have *written* the source for the documentation all you need to do is *compile*. If you have run [sphinx-quickstart](#) to start with, you should have a `Makefile`, along with a `make.bat` Windows batch file with all the proper targets. You should be able to compile the static html, e.g., by just doing

```
cd docs
make html
```

10.2 Making the documentation available.

Final step: compiling the documentation locally, per se, is not very useful—if not while you are writing it. In terms of your users what you care most is for them to be able to see it.

Nowadays the standard way to make the documentation available is through a [GitHub action](#). You might have one, e.g., that compiles and deploys the html documentation each time you tag your package.

10.3 A concrete example

And now, if you want to understand how all these things play out in practice, go ahead and look (again) at [metarep](#)—make sure to hit the link to the [documentation](#).

There is a lot of overlap with this chapter, but the two really serve different purposes, and together they will hopefully get you up and running.

11 Optimization

 Warning

The material is being migrated—see [here](#)

A Floating-point arithmetic

Let's start off with a deliberate incitement

```
print(0.1 + 0.2 == 0.3)
```

False

What the heck? Did we just discovered a major bug in one of the most widely used programming languages on the face of the Earth or, rather, we are missing something obvious?

I hate to break it to you, but Python floating-point arithmetic is just fine. The thing is, floating point arithmetic per se is interesting, and we take this chance to review a few facts that are relevant to all of us who happen to work with numbers.

A.1 Base 2 numeral system

You might have heard that virtually all digital computers work internally in base 2—the *flip-flop* being the basic building block underlying our modern digital ecosystem, everything inside the computer is a long stream of carefully placed zeros and ones. If we want to understand floating-point arithmetic in a digital computer, we have to learn how to count in base 2, first.

When we write the number 2.25 in the ordinary decimal positional notation, what we actually mean is

$$2.25 = 2 + 2 \times 10^{-1} + 5 \times 10^{-2}.$$

We can generalize our positional representation to an arbitrary base b by saying that

$$(c_m \dots c_2 c_1 c_0 . c_{-1} c_{-2} \dots)_b = c_m \times b^m + \dots + c_2 \times b^2 + c_1 \times b + c_0 + c_{-1} b^{-1} + c_{-2} b^{-2} \dots \quad (\text{A.1})$$

How would we go about writing 2.25 in base 2? Well, it is not *too* difficult to convince ourselves that

$$(10.01)_2 \triangleq 0b10.01 = 1 \times 2 + 0 \times 1 + 0 \times 2^{-1} + 1 \times 2^{-2} = 2 + \frac{1}{4} = 2.25.$$

That was easy. How about 0.1? We can try and approximate our target from above and from below, and see if we can get *exactly* to the point. Unfortunately that is not the case

Table A.1

0	0/2	0b0.0	0.1	0b0.1	1/2	0.5
0	0/4	0b0.00	0.1	0b0.01	1/4	0.25
0	0/8	0b0.000	0.1	0b0.001	1/8	0.125
0.0625	1/16	0b0.0001	0.1	0b0.0010	2/16	0.125
0.09375	3/32	0b0.00011	0.1	0b0.00100	4/32	0.125
0.09375	6/64	0b0.000110	0.1	0b0.000111	7/64	0.109375
0.09375	12/128	0b0.0001100	0.1	0b0.0001101	13/128	0.101562
0.0976562	25/256	0b0.00011001	0.1	0b0.00011010	26/256	0.101562
0.0996094	51/512	0b0.000110011	0.1	0b0.000110100	52/512	0.101562

We can continue as much as we want, but 0.1 does not have a finite binary representation—it is actually periodic in base 2

$$0.1 = 0b0.00011001100110011001100110011001100110011001100110011001100110011 \dots = 0b0.000\overline{11}$$

That is hardly surprising: many rational numbers do not have a finite decimal representation—take for example

$$\frac{1}{3} = 0.\overline{3}$$

In a sense, we could argue that in any base b most rational numbers lack a finite representation—exactly which one clearly depends on b .

A.2 Floating-point representation

The representation of floating-point numbers in a binary digital computer, nowadays, is pretty much everywhere conforming to the IEEE 754 standard (Cowlshaw 2008), coming in two different flavors at 32 (single precision) and 64 bits (double precision). Python internally uses the latter by default.

More precisely, a floating point number is represented in the computer in the form

$$x = (-1)^s \times m \times 2^{e-b} \quad \text{with} \quad 1 \leq m < 2. \quad (\text{A.2})$$

Moving from the most to the least significant bit:

- s is the *sign*—the most significant bit is 0 if the number is positive, and 1 if it's negative;

- e is the *exponent*, for which the standard reserves 8 of the 32 bits in single precision and 11 of the 64 bits in double precision; b , the *bias*, is an additive, positive constant that amounts to 127 in single precision and 1023 in double precision—its purpose is to be able to represent numbers smaller than 1 without the need of a sign for the exponent;
- m is the *mantissa*, and it is represented as the sequence of binary digits after the separator (assuming that the integer part is 1), taking up 23 bits in single precision and 52 in double precision.

A couple of technical details. The values $e = 0$ and $e = 2^8 - 1 = 255$ (in single precision) or $e = 2^{11} - 1 = 2047$ (in double precision) are *reserved* (more about this in a second). Therefore for *normal* numbers e ranges between 1 and 254 (in single precision) or 2046 (in double precision). Conversely, $e - b$ goes between -126 and 127 in single precision and -1022 a 1023 in double precision.

For the mantissa, we have effectively one extra bit (that is, 24 in single precision and 53 in double precision) coming from the fact that a leading one is assumed for normal numbers.

And now the two special cases:

- when $e = 0$ the number is treated as a *subnormal* number, where the exponent is by definition $1 - b$ (-126 in single precision and -1022 in double precision) but the leading bit of the mantissa is assumed to be 0 instead of 1 (which is to say that $m < 1$);
- $e = 255$ (in single precision) or $e = 2047$ signal an infinity.

A.2.1 A simple example

That was a fairly terse introduction to the IEEE 754 standard. You don't have to commit to memory the whole thing, but you should at least remember the general form Equation A.2

Now, how about we fully work out a simple example, say $x = 2.25$? First thing first, we need to write it in the form of the product of a mantissa $1 < m < 2$ times a power of 2. That would be

$$x = 2.25 = 1.125 \times 2$$

If we assume double precision, we have

$$s \equiv 0$$

$$e - b = 1 \implies e = 1024 = 0b100000000000$$

$$m = 1.125 = 0b(1).00100$$

Note we have indicated all the 11 bits for the exponent and all the 52 bits for the mantissa, including the trailing zeroes, and the MSB of the mantissa in parenthesis. If you wonder why the mantissa reads like that, keep in mind that

$$1.125 = 1 + \frac{1}{8},$$

if that makes sense.

And we are officially ready to assemble the full, glorious sequence of 64 bits that would represent $x = 2.25$ in the memory of our computer. That would be

$$x \rightarrow 0b0 \overbrace{0010000000}^{e-b} \overbrace{00100}^{m \text{ (MSB omitted)}}$$

A.3 Back to the problem

What is this offering, exactly, to the solution of our initial dilemma? Well, the point is: many of the numbers that we can represent exactly in base 10 cannot be represented with a finite number of digits in base 2, and this is exactly where things get potentially confusing.

The three numbers that we have started with, 0.1, 0.2 and 0.3, all look harmless because they are easy to represent in base 10. Unfortunately, none of the fractions

$$\frac{1}{10}, \frac{1}{5} \text{ e } \frac{3}{10}$$

can be represented exactly in base 2, because all of them have the prime factor 5 at the denominator. In other words, within a binary digital computer, we can only store an approximation, in the form of a fraction where the denominator is a power of 2.

Which fraction exactly? Well, fortunately the Python method `as_integer_ratio()` is designed to tell us exactly that

```
for x in (0.1, 0.2, 0.3):
    numerator, denominator = x.as_integer_ratio()
    print(f"{x} -> {numerator} / {denominator} = {x:.20f}")
```

```
0.1 -> 3602879701896397 / 36028797018963968 = 0.10000000000000000555
0.2 -> 3602879701896397 / 18014398509481984 = 0.20000000000000001110
0.3 -> 5404319552844595 / 18014398509481984 = 0.29999999999999998890
```

And now we start to see the solution: when we write 0.1 in our text editor, what we are committing to the memory is in fact a slightly different number—very close, but not quite the same. No wonder things do not add up as we naively expected

```
x = (0.1 + 0.2)
numerator, denominator = x.as_integer_ratio()
print(f"(0.1 + 0.2) -> {numerator} / {denominator} = {x:.20f}")
```

$(0.1 + 0.2) \rightarrow 1351079888211149 / 4503599627370496 = 0.30000000000000004441$

See? It's not like we are off by much. We start seeing a discrepancy around the seventeenth decimal place, which in most cases we would hardly care about. And yet the two things, when compared bit by bit, are not identical.

A.4 Properties of floating-point arithmetic

Good: we have learned a very fundamental thing: *floating-point arithmetic on a digital computer is, by its very nature, intrinsically inexact*. There is really nothing we can do about its—it is one of the facts of life that we have to accept and learn to live with.

A.4.1 Range and resolution

The available *range* for a floating-point number is primarily dictated by the number of bits allocated for the exponents. In the double-precision IEEE 754 standard

- the mantissa ranges from 1 to 2 (excluded);
- the exponent ranges from -1022 to 1023

and *normal* numbers span the range between

$$1 \times 2^{-1022} \approx 2.2 \times 10^{-308} \quad \text{and} \quad (2 - 2^{-52}) \times 2^{1023} \approx 2 \times 2^{1023} \approx 1.8 \times 10^{308}$$

Note

For completeness: when the exponent takes its smallest possible value, which would be technically -1023 , the IEEE standard switches to the *sub-normal* representation, where the exponent is still -1022 , and the most significant bit of the mantissa is assumed to be 0 instead of 1. This is a completely different regime, where the relative precision is not constant anymore, and the range is extended toward the zero, down to

$$2^{-52} \times 2^{-1022} = 2^{-1074} \approx 4.9 \times 10^{-324}$$

Back to the Python prompt, this reads

```
print(1.e-324)
print(5.e-324)
print(1.5e308)
print(2.e308)
```

```
0.0
5e-324
1.5e+308
inf
```

(Watch out: you can trigger an overflow quite easily, if you are not careful. $171!$ is too big to be represented, and so is 144^{144} and e^{710} .)

Clearly, floating-point arithmetic is designed to operate at constant *relative precision*, determined by the number of bits allocated for the mantissa. With 53 effective bits we have

$$2^{53} \approx 9 \times 10^{15}$$

possible values in the interval $[1, 2]$, or about 16 significant digits. The maximum (relative) rounding error is 2^{-53} which is sometimes called *half spacing*. Things work out pretty good when we multiply and divide, not necessarily so much when we subtract.

Since the relative precision is constant, the physical spacing between consecutive floating point number is not. More precisely, the latter is approximately proportional to the value we are representing. For numbers between 1 e 2 the spacing is 2^{-52} ; between 2 e 4 it becomes 2^{-51} and so on and so forth: it doubles every time you cross a power of 2. The `math` module of the standard library provides the `ulp` (*unit in the last place*) function which, for a given float x returns the absolute distance to the one immediately above that can be represented

```
import math

for x in (1., 1.e10, 1.e20, 1.e50, 1.e100):
    print(math.ulp(x))
```

```
2.220446049250313e-16
1.9073486328125e-06
16384.0
2.076918743413931e+34
1.942668892225729e+84
```

By the time we reach $2^{52} \approx 4.5 \times 10^{15}$ the unit in the last place becomes greater than 1. Not a big deal, per se, but if for a moment we wanted to treat the Avogadro number as an integer, we would have a hard time adding a single molecule to a mole of gas, as the ulp is of the order of 70 millions:

```
import math

N = 6.02e23

print(math.ulp(N))
print(N == N + 1)
```

67108864.0
True

Let's wrap this up. With 11 bits for the exponent and $52 + 1$ bits for the mantissa, the double-precision IEEE standard provides a dynamic range of about 616 orders of magnitude and a precision of 16 digits. This is not terrible, considering that there are about 42 orders of magnitude between the proton radius and that of the visible universe, and that 16 significant digits are enough for basically all the precision measurements available.

A.4.2 More on arithmetics

Fun fact: some of the properties of elementary arithmetics that we are taught in elementary school, such as the commutative, associative and distributive properties of some operations, cannot be taken for granted when dealing with floating-point numbers in a digital computer. At this point, this should be hardly a surprise, but just for the reference

```
a = 0.2
b = 0.1
c = 0.3

print((a + b) - c == a + (b - c))
print(a * (b + c) == a * b + a * c)
```

False
False

(Think about: if it is true that we are making approximations at every step in the way, then it is all but unexpected that the final answer might depend on the exact sequence of operations.)

Most of the times we are dealing with *small* differences, at the level of the machine precision, but there are situations that can be engineered to make everything blow up:

```
a = 1.e16
b = -1.e16
c = 1.

print((a + b) + c)
print(a + (b + c))
```

```
1.0
0.0
```

(This brings us back to the example of the Avogadro number: summing very large numbers to much smaller numbers is bad!)

A.4.3 Floating-point comparisons

By now we know that the very idea of *comparing* floating-point number is ill-defined, and yet you see all the time programs like

```
a = 0.1
b = 0.2
c = 0.3

if a + b <= c:
    print("win!")
else:
    print("lose...")
```

```
lose...
```

Now, if you think about this in terms of the left-hand side of the inequality being a generic operation on random numbers reasonably distributed over an interval much larger than the machine precision, you will get the expected answer most of the time. (And yet you cannot rely on that being true.)

Luckily, both the Python standard library and `numpy` offer facilities to handle comparisons between floats in a more rigorous manner.

```
import math
import numpy as np

a = 0.1
b = 0.2
c = 0.3

print(math.isclose(a + b, c))
print(np.allclose(a + b, c))
```

True
True

A.5 What did we learn?

Well, that the floating point arithmetic is intrinsically inexact, and you do have to pay attention. The IEEE 754 standard is also *very* well designed, and things *usually* work out fine—just not all the time.

Pay extra attention to:

- addition and subtractions between numbers differing by many orders of magnitude;
- subtractions between numbers very close to each other;
- long chains of multiplications and divisions, with potential for underflow or overflow at some intermediate point;
- coherent accumulation of small errors within loops (if you have 10 minutes to spare, take a look at [this report](#)).

Consider yourself advised!

B Assignments

B.1 Version control—assignment 1

—Goal

Create a pull request on the [repository](#) for these lecture notes that is worth merging.

(Please refrain from rewriting the thing from scratch. It is ok to use an LLM, if you really insist.)

—Specifications

- Create an account on [Codeberg](#), if you don't have one, already.
- Fork the [repository](#) for these lecture notes.
- Find a *weakness* in the notes—I know, this one is hard. Anything will do: a typo in the text, a bug in one of the code snippets, a conceptual error, a link to an resource that you deem useful, a small additional piece of text that you think would be beneficial.
- Create a new branch in your working copy. (Use a meaningful name for the branch.)
- Edit your working copy as you see fit, and commit and push your changes.
- Create a pull request against the original repository.
- Wait for the pull request to be reviewed and merged. (There might be discussion on the pull request—please pay attention.)

B.2 Basic Python—assignment 1

Download a book (not covered by copyright) in plain-text format, e.g., from [gutenberg](#).

(If you have a hard time picking one, we suggest the English translation of “[The Republic](#)” by Plato.)

—Goal

Write a Python program that prints the relative frequency of each letter of the alphabet (without distinguishing between lower and upper case) in the book.

—Specifications

- The program should have a `--help` option summarizing the usage.

- The program should accept the path to the input file from the command line.
- The program should print out the total elapsed time.
- The program should have an option to display a histogram of the frequencies.
- [Optional] The program should have an option to skip the parts of the text that do not pertain to the book (e.g., preamble and license).
- [Optional] The program should have an option to print out the basic book stats (e.g., number of characters, number of words, number of lines).

B.3 Basic Python—assignment 2

—Goal

Write a program to explore the properties of a few elementary particles. The program must contain a base class `Particle` and two child classes, `Proton` and `Alpha`, that inherit from it.

—Specifications

- Instances of the class `Particle` must be initialized with their mass, charge, and name (all members should be read-only).
- The class constructor must also accept (optionally) and store one and only one of the following quantities: energy, momentum, beta or gamma.
- Whatever the choice, the user should be able to read and set any of these quantities using just the `.` (dot) operator, e.g., `print(my_particle.energy)`, `my_particle.beta = 0.5`.
- Attempts to set non physical values should be rejected.
- The `Particle` class must have a method to print the particle information in a nicely formatted fashion.
- The child classes `Alpha` and `Protons` must use class attributes to store their mass, charge and name.

B.4 Basic Python—assignment 3

—Goal

Create a `ProbabilityDensityFunction` class that is capable of throwing pseudo-random numbers with an arbitrary distribution.

(In practice, start with something easy, like a triangular distribution—the initial debug will be easier if you know exactly what to expect.)

—Specifications

- The signature of the constructor should be `__init__(self, x, y)`, where `x` and `y` are two numpy arrays sampling the pdf on a grid of values, that you will use to build a spline.
- [Optional] Add more arguments to the constructor to control the creation of the spline (e.g., the order of the latter).
- The class should be able to evaluate itself on a generic point or array of points.
- The class should be able to calculate the probability for the random variable to be included in a generic interval.
- The class should be able to throw random numbers according to the distribution that it represents.
- [Optional] How many random numbers do you have to throw to hit the numerical inaccuracy of your generator?

B.5 Packaging—assignment 1

—Goal

Clean up your `ProbabilityDensityFunction` class and create a fully-fledged Python package.

(The layout of the package might draw inspiration from [metarep](#).)

—Specifications

- The package should have adequate unit tests.
- The package should have continuous integration.
- The package should have adequate documentation, and publicly accessible from the web.
- [Optional] Add a rudimentary release manager to automate the tagging process.

B.6 Advanced Python—assignment 1

—Goal

Write a class to handle a sequence of voltage measurements at different times.

—Specifications

- The class name must be `VoltageData`.
- The class must be initialized with two generic iterables of the same length holding the numerical values of times and voltages.
- Alternatively the class can be initialized from a text file.
- The class must expose two attributes: `timestamps` and `voltages`, each returning a numpy array (of type `numpy.float64`) with the corresponding quantity.

- The values should be accessible with the familiar square parenthesis syntax: the first index must refer to the entry, the second selects time (0) or voltage (1). Slicing must also work.
- Calling the `len()` function on a class instance must return the number of entries.
- The class must be iterable: at each iteration, a numpy array of two values (time and voltage) corresponding to an entry in the file must be returned.
- The `print()` function must work on class instances. The output must show one entry (time and voltage), as well as the entry index, per line.
- The class must also have a debug representation, printing just the values row by row.
- The class must be callable, returning an interpolated value of the tension at a given time.
- The class must have a `plot()` method that plots data using matplotlib. The plot function must accept an `ax` argument, so that the user can select the axes where the plot is added (with a new figure as default). The user must also be able to pass other plot options as usual.
- [Optional] Support a third optional column for the voltage errors.
- [Optional] Add unit tests.

C Contributing

We are more than happy to accept suggestions for improving the material. If you (as in: a human being) found a typo, a conceptual mistake, or simply think that something could be explained better, by all means reach out opening an issue on the [issue tracker](#) of this project or, even better, making a pull request.

(Did I say *pull request*? Look [here](#)!)

You will be automatically assigned a seat in the hall of fame of the contributors at the end of the book in exchange—how cool is that?

C.1 Note for agents

Teaching and learning at the University level is really about in-depth discussions between human beings, and we shall happily close any issue and/or reject any pull request that was automatically generated by an agent, or otherwise produced using generative AI.

Agents are great, everybody knows. But if we really wanted to hunt down the last typo in these notes throwing a LLM at them, we could have easily done it ourselves. (We actually *do* it ourselves, every once in a while.)

As much as we understand why you might want to bump up the number of pull requests closed on your github stats, if your agent found this repo it probably means that it did process thousands and thousands of other repos and, honestly, there are better ways to use our precious energy.

No offense intended

D Acknowledgements

And here is a list, in alphabetical order, of all the persons who helped make the world a better place by reporting legit errors or suggesting improvements through our [issue tracker](#).

- Roberta Fallarino
- Leonardo Guadagnini
- Àlex Guitart
- Giordano Notarnicola
- Marco Orsi
- Ettore Olivieri

E Use of AI for these notes

The material upon which these notes are based predates the general availability of generative AI and LLMs, and the original design has been largely retained in the rewrite. No generative AI has been used for the choice of the topics, the logical organization of the material, nor for proofreading or otherwise refining the text—not a single paragraph in these notes has been automatically generated.

We sporadically used commercially available LLMs for brainstorming about very specific technical issues and for bibliographical research, as well as for setting up and developing the structure of the repository and explore the features that [Quarto](#) offers.

The vast majority of the code snippets are retained from the original material, and are completely human-generated. There is a few exceptions, most notably that in Section [5.5.1](#), where snippets were drafted by generative AI, and then refined by hand.

In the process of brainstorming with LLMs, we learned genuinely new things, such as the [else clauses on loops](#).

F Release history

0.9.2 (2026-10-09)

- A couple of edits and clarifications ([PR 44](#)).
- Fixed two errors in the appendix about the IEEE 754 standard ([PR 43](#)).
- Typo fixed ([PR 39](#)).

0.9.1 (2026-10-05)

- Fix for [issue 37](#) ([PR 38](#)).

0.9.0 (2026-10-04)

- Bug fix in one of the code snippets about sorting, see [issue 31](#).
- New small section about forking ([PR 33](#)).
- Assignments added ([PR 34](#)).
- New small section about resolving conflicts ([PR 35](#)).
- New appendix about the use of AI in these notes ([PR 36](#)).

0.8.1 (2026-10-02)

- Cover page added ([PR 29](#)).

0.8.0 (2026-09-30)

- New chapter about testing ([PR 27](#)).

0.7.0 (2026-09-30)

- New chapter about documentation ([PR 26](#)).

0.6.0 (2026-09-30)

- New chapter about the Python scientific stack ([PR 24](#)).

0.5.0 (2026-09-23)

- Overall index structure slightly revised.
- Broken link to the CCBY badge fixed.
- New chapter about algorithms and data structures ([PR 19](#)).
- New short chapter about Python packaging ([PR 17](#)).
- A few typos fixed ([PR 16](#) and [PR 18](#)).

0.4.0 (2026-09-21)

Versions prior to this one were not tracked—sorry.

Cowlshaw, Mike, ed. 2008. “IEEE Standard for Floating-Point Arithmetic.” Standard IEEE Std 754-2008. New York, NY, USA: IEEE Computer Society. <https://web.archive.org/web/20160806053349/http://www.csee.umbc.edu/~tsimo1/CMSC455/IEEE-754-2008.pdf>.

Knuth, Donald Ervin. 1984. “The Complexity of Songs.” *Communications of the ACM* 27 (4): 344–46. <https://doi.org/10.1145/358027.358042>.