Idiomatic Python 1

That which you inherit from your forefathers,
you must make your own in order to truly possess it.

Johann Wolfgang von Goethe

This chapter is *not* intended as an introduction to programming in general or to programming with Python. A tutorial on the Python programming language can be found in the online supplement to this book; if you're still learning what variables, loops, and functions are, we recommend you go to our tutorial (see Appendix A) before proceeding with the rest of this chapter. You might also want to have a look at the (official) Python Tutorial at `www.python.org`. Another introduction to Python, with nice material on visualization, is Ref. [41]. Programming, like most other activities, is something you learn by doing. Thus, you should always try out programming-related material as you read it: *there is no royal road to programming*. Even if you have solid programming skills but no familiarity with Python, we recommend you work your way through one of the above resources, to familiarize yourself with the basic syntax. In what follows, we will take it for granted that you have worked through our tutorial and have modified the different examples to carry out further tasks. This includes solving many of the programming problems we pose there.

What this chapter *does* provide is a quick summary of Python features, with an emphasis on those which the reader is more likely not to have encountered in the past. In other words, even if you are already familiar with the Python programming language, you will most likely still benefit from reading this short chapter. Observe that the title at the top of this page is *Idiomatic* Python: this refers to coding in a *Pythonic* manner. The motive is not to proselytize but, rather, to let the reader work with the language (i.e., not against it); we aim to show how to write Python code that feels "natural". If this book was using, say, Go or Julia instead of Python, we would still be making the same point: one should try to do the best job possible with the tools at one's disposal. As noted in the Preface, the use of idioms allows us to write shorter codes in the rest of the book, thereby emphasizing the numerical method over programming details; this is not merely an aesthetic concern but a question of cognitive consonance.

At a more mundane level, this chapter contains all the Python-related reference material we will need in this volume: reserved words, library functions, tables, and figures. Keeping the present chapter short is intended to help you when you're working through the following chapters and need to quickly look something up.

1.1 Why Python?

Since computational physics is a fun subject, it is only appropriate that the programming involved should also be as pleasant as possible. In this book, we use Python 3, a popular, open-source programming language that has been described as "pseudocode that executes". Python is especially nice in that it doesn't require lots of boilerplate code, making it easy to write new programs; this is great from a pedagogical perspective, since it allows a beginner to start using the language without having to first study lengthy volumes. Importantly, Python's syntax is reasonably simple and leads to very readable code. Even so, Python is very expressive, allowing you to do more in a single line than is possible in many other languages. Furthermore, Python is cross-platform, providing a similar experience on Windows and Unix-like systems. Finally, Python comes with "batteries included": its standard library allows you to do a lot of useful work, without having to implement basic things (e.g., sorting a list of numbers) yourself.

In addition to the functionality contained in core Python and in the standard library, Python is associated with a wider ecosystem, which includes libraries like Matplotlib, used to visualize data. Another member of the Python ecosystem, especially relevant to us, is the NumPy library (NumPy stands for "Numerical Python"); containing numerical arrays and several related functions, NumPy is one of the main reasons Python is so attractive for computational work. Another fundamental library is SciPy ("Scientific Python"), which provides many routines that carry out tasks like numerical integration and optimization in an efficient manner. A pedagogical choice we have made in this book is to start out with standard Python, use it for a few chapters, and only then turn to the `numpy` library; this is done in order to help students who are new to Python (or to programming in general) effectively distinguish between Python lists and `numpy` arrays. The latter are then used in the context of linear algebra (chapter 4), where they are indispensable, both in terms of expressiveness and in terms of efficiency.

Speaking of which, it's worth noting at the outset that, since our programs are intended to be easy to read, in some cases we have to sacrifice efficiency.[1] Our implementations are intended to be pedagogical, i.e., they are meant to teach you how and why a given numerical method works; thus, we almost never employ NumPy or SciPy functionality (other than `numpy` arrays), but produce our own functions, instead. We make some comments on alternative implementations here and there, but the general assumption is that you will be able to write your own codes using different approaches (or programming languages) once you've understood the underlying numerical method. If all you are interested in is a quick calculation, then Python along with its ecosystem is likely going to be your one-stop shop. As your work becomes more computationally challenging, you may need to switch to a compiled language; most work on supercomputers is carried out using languages like Fortran or C++ (or sometimes even C). Of course, even if you need to produce a hyper-efficient code for your research, the insight you may gain from building a prototype in Python could be invaluable; similarly, you could write most of your code in Python and re-express a few

[1] Thus, we do not talk about things like Python's Global Interpreter Lock, cache misses, page faults, and so on.

performance-critical components using a compiled language. We hope that the lessons you pick up here (both on the numerical methods and on programming in general) will serve you well if you need to employ another environment in the future.

The decision to focus on Python (and NumPy) idioms is coupled to the aforementioned points on Python's expressiveness and readability: idiomatic code makes it easier to conquer the complexity that arises when developing software. (Of course, it does require you to first become comfortable with the idioms.) That being said, our presentation will be *selective*; Python has many other features that we will not go into. Most notably, we don't discuss how to define classes of your own or how to handle exceptions; the list of omitted features is actually very long.[2] While many features we leave out are very important, discussing them would interfere with the learning process for students who are still mastering the basics of programming. Even so, we do introduce topics that haven't often made it into computational-science texts (e.g., list comprehensions, dictionaries, for-else, array manipulation via slicing and `@`) and use them repeatedly in the rest of the book.

When deemed necessary, we point to further functionality in Python. For more, have a look at the bibliography and at The Python Language Reference (as well as The Python Standard Library Reference). Once you have mastered the basics of core Python, you may find books like Ref. [75] and Ref. [84] a worthwhile investment. On the wider theme of developing good programming skills, volumes like Ref. [65] can be enriching, as is also true of any book written by Brian Kernighan. Here we provide only the briefest of summaries.

1.2 Code Quality

We will not be too strict in this book about coding guidelines. Issues like code layout can be important, but most of the programs we will write are so short that this won't matter too much. If you'd like to learn more about this topic, your first point of reference should be PEP 8 – Style Guide for Python Code. Often more important than issues of code layout[3] are questions about how you write and check your programs. Here is some general advice:

- **Code readability matters** Make sure to target your program to humans, not the computer. This means that you should avoid using "clever" tricks. Thus, you should use good variable names and write comments that add value (instead of repeating the code). The human code reader that will benefit from this is first and foremost yourself, when you come back to your programs some months later.
- **Be careful, not swift, when coding** Debugging is typically more difficult than coding itself. Instead of spending two minutes writing a program that doesn't work and then requires you to spend two hours fixing it up, try to spend 10 minutes on designing the code and then carefully converting your ideas into program lines. It doesn't hurt to also use Python interactively (while building the program file) to test out components of the code one-by-one or to fuse different parts together.

[2] For example: decorators, coroutines, or type hints.
[3] You should look up the term "bikeshedding".

- **Untested code is wrong code** Make sure your program is working correctly. If you have an example where you already know the answer, make sure your code gives that answer. Manually step through a number of cases (i.e., mentally, or on paper, do the calculations the program is supposed to carry out). This, combined with judiciously placed print-outs of intermediate variables, can go a long way toward ensuring that everything is as it should be. When modifying your program, ensure it still gives the original answer when you specialize the problem to the one you started with.
- **Write functions that do one thing well** Instead of carrying out a bunch of unrelated operations in sequence, you should structure your code so that it makes use of well-named (and well-thought-out) functions that do one thing and do it well. You should break down the tasks to be carried out and logically separate those into distinct functions. If you design these well, in the future you will be able to modify your programs to carry out much more challenging tasks, by only adding a few lines of new code (instead of having to change dozens of lines in an existing "spaghetti" code).
- **Use trusted libraries** In most of this book we are "reinventing the wheel", because we want to understand how things work (or don't work). Later in life, you should not have to always use "hand-made" versions of standard algorithms. As mentioned, there exist good (widely employed and tested) libraries like `numpy` that you should learn to make use of. The same thing holds, obviously, for the standard Python library: you should generally employ its features instead of "rolling your own".

One could add (much) more advice along these lines. Since our scope here is much more limited, we conclude by pointing out that in the Python ecosystem (or around it) there's extensive infrastructure [82] to carry out version control (e.g., `git`), testing (e.g., `doctest` and `unittest`), as well as debugging (e.g., `pdb`), program profiling and optimization, among other things. You should also have a look at the `pylint` tool.

1.3 Summary of Python Features

1.3.1 Basics

Python can be used interactively: this is when you see the Python prompt >>>, also known as a chevron. You don't need to use Python interactively: like other programming languages, the most common way of writing and running programs is to store the code in a file. Linear combinations of these two ways of using Python are also available, fusing interactive sessions and program files. In any case, your program is always executed by the Python interpreter. Appendix A points you in the direction of tools you could employ.

Like other languages (e.g., C or Fortran), Python employs variables, which can be integers, complex numbers, etc. Unlike those languages, Python is a dynamically typed language, so variables get their type from their value, e.g., `x = 0.5` creates a floating-point variable (a "float"). It may help you to think of Python values as being produced first and labels being attached to them after that. Numbers like `0.5` or strings like `"Hello"`,

are known as *literals*. If you wish to print the value of a variable, you use the `print()` built-in function, i.e., `print(x)`. Further functionality is available in the form of standard-library modules, e.g., you can `import` the `sqrt` function that is to be found in the `math` module. Users can define their own modules: we will do so repeatedly. You can carry out arithmetic with variables, e.g., `x**y` raises `x` to the `y`-th power or `x//y` does "floor division". It's usually a good idea to group related operations using parentheses. Python also supports augmented assignment, e.g., `x += 1` or even multiple assignment, e.g., `x, y = 0.5, "Hello"`. This gives rise to a nifty way to swap two variables: `x, y = y, x`.

Comments are an important feature of programming languages: they are text that is ignored by the computer but can be very helpful to humans reading the code. That human may be yourself in a few months, at which point you may have forgotten the purpose or details of the code you're inspecting. Python allows you to write both single-line comments, via `#`, or docstrings (short for "documentation strings"), via the use of triple quotation marks. Crucially, we don't include explanatory comments in our code examples, since this is a book which explicitly discusses programming features in the main text. That being said, in your own codes (which are not embedded in a book discussing them) you should always include comments.

1.3.2 Control Flow

Control flow refers to programming constructs where not every line of code gets executed in order. A classic example is conditional execution via the `if` statement:

```
>>> if x!=0:
...       print("x is non-zero")
```

Indentation is important in Python: the line after `if` is indented, reflecting the fact that it belongs to the corresponding scenario. Similarly, the colon, `:`, at the end of the line containing the `if` is also syntactically important. If you wanted to take care of other possibilities, you could use another indented block starting with `else:` or `elif x==0:`. In the case of boolean variables, a common idiom is to write: `if flag:` instead of `if flag==True:`.

Another concept in control flow is the loop, i.e., the repetition of a code block. You can do this via `while`, which is typically used when you don't know ahead of time how many iterations you are going to need, e.g., `while x>0:`. Like conditional expressions, a `while` loop tests a condition; it then keeps repeating the body of the loop until the condition is no longer true, in which case the body of the block is jumped over and execution resumes from the following (non-indented) line. We sometimes like to be able to break out of a loop: if a condition in the middle of the loop body is met, then: a) if we use `break` we will proceed to the first statement *after* the loop, or b) if we use `continue` we skip not the entire loop, but the rest of the loop body *for the present iteration*.

A third control-flow construct is a `for` loop: this arises when you need to repeat a certain action a fixed number of times. For example, by saying `for i in range(3):` you will repeat whatever follows (and is indented) three times. Like C, *Python uses 0-indexing*, meaning that the indices go as 0, 1, 2 in this case. In general, `range(n)` gives the integers

from `0` to `n-1` and, similarly, `range(m, n, i)` gives the integers from `m` to `n-1` in steps of `i`. Above, we mentioned how to use `print()` to produce output; this can be placed inside a loop to print out many numbers, each on a separate line; if you want to place all the output on the same line you do:

```
>>> for i in range(1,15,2):
...       print(0.01*i, end=" ")
```

that is, we've said `end=" "` after passing in the argument we wish to print. As we'll discuss in the following section, Python's `for` loop is incredibly versatile.

1.3.3 Data Structures

Python supports container entities, called data structures; we will mainly be using lists.

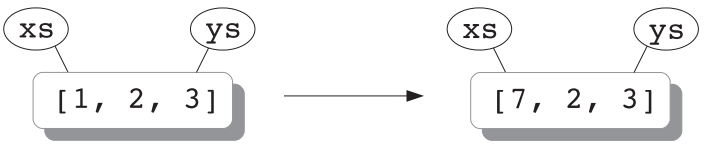
Lists A list is a container of elements; it can can grow when you need it to. Elements can have different types. You use square brackets and comma-separated elements when creating a list, e.g., `zs = [5, 1+2j, -2.0]`. You also use square brackets when indexing into a list, e.g., `zs[0]` is the first element and `zs[-1]` the last one. Lists are mutable sequences, meaning we can change their elements, e.g., `zs[1] = 9`, or introduce new elements, via `append()`. The combination of `for` loops and `range()` provides us with a powerful way to populate a list. For example:

```
>>> xs = []
>>> for i in range(20):
...       xs.append(0.1*i)
```

where we started with an empty list. In the following section, we'll see a more idiomatic way of accomplishing the same task. You can concatenate two lists via the addition operator, e.g., `zs = xs + ys`; the logical consequence of this is the idiom whereby a list can be populated with several (identical) elements using a one-liner, `xs = 10*[0]`. There are several built-in functions (applicable to lists) that often come in handy, most notably `sum()` and `len()`.

Python supports a feature called slicing, which allows us to take a slice out of an existing list. Slicing, like indexing, uses square brackets: the difference is that slicing uses two integers, with a colon in between, e.g., `ws[2:5]` gives you the elements `ws[2]` up to (but not including) the element `ws[5]`. Slicing obeys convenient defaults, in that we can omit one of the integers in `ws[m:n]` without adverse consequences. Omitting the first index is interpreted as using a first index of 0, and omitting the second index is interpreted as using a second index equal to the number of elements. You can also include a third index: in `ws[m:n:i]` we go in steps of `i`. Note that list slicing uses colons, whereas the arguments of `range()` are comma separated. Except for that, the pattern of `start`, `end`, `stride` is the same.

We are now in a position to discuss how copying works. In Python a new list, which is



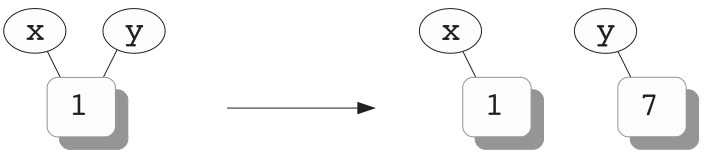
Labelling and modifying a mutable object (in this case, a list) Fig. 1.1

assigned to be equal to an old list, is simply the old list by another name. This is illustrated in Fig. 1.1, which corresponds to the three steps `xs = [1,2,3]`, followed by `ys = xs`, and then `ys[0] = 7`. In other words, in Python we're not really dealing with variables, but with *labels* attached to values, since `xs` and `ys` are just different names for the same entity. When we type `ys[0] = 7` we are not creating a new value, simply modifying the underlying entity that both the `xs` and `ys` labels are attached to. Incidentally, things are different for simpler variables, e.g., `x=1; y=x; y=7; print(x)` prints 1 since 7 is a new value, not a modification of the value `x` is attached to. This is illustrated in Fig. 1.2, where we see that, while initially both variable names were labelling the same value, when we type `y=7` we create a new value (since the number 7 is a new entity, not a modification of the number 1) and then attach the `y` label to it.

Crucially, *when you slice you get a new list*, meaning that if you give a new name to a slice of a list and then modify that, then the original list is unaffected. For example, `xs = [1,2,3]`, followed by `ys = xs[1:]`, and then `ys[0] = 7` does not affect `xs`. This fact (namely, that slices don't provide views on the original list but can be manipulated separately) can be combined with another nice feature (namely, that when slicing one can actually omit both indices) to create a copy of the entire list, e.g., `ys = xs[:]`. This is a shallow copy, so if you need a deep copy, you should use the function `deepcopy()` from the standard module `copy`; the difference is immaterial here.

Tuples Tuples can be (somewhat unfairly) described as immutable lists. They are sequences that can neither change nor grow. They are defined using parentheses instead of square brackets, e.g., `xs = (1,2,3)`, but you can even omit the parentheses, `xs = 1,2,3`. Tuple elements are accessed the same way that list elements are, namely with square brackets, e.g., `xs[2]`.

Strings Strings can also be viewed as sequences, e.g., if `name = "Mary"` then `name[-1]` is the character `'y'`. Note that you can use either single or double quotation marks. Like tuples, strings are immutable. As with tuples, we can use + to concatenate two strings. A



Labelling immutable objects (in this case, integers) Fig. 1.2

useful function that acts on strings is `format()`: it uses *positional* arguments, numbered starting from 0, within curly braces. For example:

```
>>> x, y = 3.1, -2.5
>>> "{0} {1}".format(x, y)
'3.1 -2.5'
```

The overall structure is string-dot-format-arguments. This can lead to powerful ways of formatting strings, e.g.,

```
>>> "{0:1.15f} {1}".format(x, y)
'3.100000000000000 -2.5'
```

Here we also introduced a colon, this time followed by `1.15f`, where `1` gives the number of digits before the decimal point, `15` gives the number of digits after the decimal point, and `f` is a type specifier (that leads to the result shown for floats).

Dictionaries Python also supports dictionaries, which are called associative arrays in computer science (they're called maps in C++). You can think of dictionaries as being similar to lists or tuples, but instead of being limited to integer indices, with a dictionary you can use strings or floats as *keys*. In other words, dictionaries contain key and value pairs. The syntax for creating them involves curly braces (compare with square brackets for lists and parentheses for tuples), with the key-value pair being separated by a colon. For example, `htow = {1.41: 31.3, 1.45: 36.7, 1.48: 42.4}` is a dictionary associating heights to weights. In this case both the keys and the values are floats. We access a dictionary value (for a specific key) by using the name of the dictionary, square brackets, and the key we're interested in: this returns the value associated with that key, e.g., `htow[1.45]`. In other words, indexing uses square brackets for lists, tuples, strings, and dictionaries. If the specific key is not present, then we get an error. Note, however, that accessing a key that is not present *and then assigning* actually works: this is a standard way key:value pairs are introduced into a dictionary, e.g., `htow[1.43] = 32.9`.

1.3.4 User-Defined Functions

If our programs simply carried out a bunch of operations in sequence, inside several loops, their logic would soon become unwieldy. Instead, we are able to group together logically related operations and create what are called user-defined functions: just as in our earlier section on control flow, this refers to lines of code that are not necessarily executed in the order in which they appear inside the program file. For example, while the `math` module contains a function called `exp()`, we could create our own function called, say, `nexp()`, which, e.g., uses a different algorithm to get to the answer. The way we introduce our own functions is via the `def` keyword, along with a function name and a colon at the end of the line, as well as (the by now expected) indentation of the code block that follows. Here's a function that computes the sum from 1 up to some integer:

```
>>> def sumofints(nmax):
...     val = sum(range(1,nmax+1))
...     return val
```

We are taking in the integer up to which we're summing as a parameter. We then ensure that `range()` goes up to (but not including) `nmax+1` (i.e., it includes `nmax`). We split the body of the function into two lines: first we define a new variable and then we *return* it, though we could have simply used a single line, `return sum(range(1,nmax+1))`. This function can be called (in the rest of the program) by saying `x = sumofints(42)`.

The function we just defined took in one parameter and returned one value. It could have, instead, taken in no parameters, e.g., summing the integers up to some constant; we would then call it by `x = sumofints()`. Similarly, it could have printed out the result, inside the function body, instead of returning it to the external world; in that case, where no `return` statement was used, the `x` in `x = sumofints(42)` would have the value `None`. Analogously, we could be dealing with several input parameters, or several return values, expressed by `def sumofints(nmin,nmax):`, or `return val1, val2`, respectively. The latter case is implicitly making use of a tuple.

We say that a variable that's either a parameter of a function or is defined inside the function is *local* to that function. If you're familiar with the terminology other languages use (pass-by-value or pass-by-reference), then note that Python employs *pass-by-assignment*, which for immutable objects behaves like pass-by-value (you *can't* change what's outside) and for mutable objects behaves like pass-by-reference (you *can* change what's outside), if you're not re-assigning. It's often a bad idea to change the external world from inside a function: it's best simply to return a value that contains what you need to communicate to the external world. This can become wasteful, but here we opt for conceptual clarity, always returning values without changing the external world. This is a style inspired by *functional programming*, which aims at avoiding *side effects*, i.e., changes that are not visible in the return value. (Unless you're a purist, input/output is fine.) Python also supports *nested functions* and *closures*: though we won't use these, it's good to know they exist. On a related note, Python contains the keywords `global` and `nonlocal` as well as function one-liners via `lambda`, but we won't be using them.

A related feature of Python is the ability to provide default parameter values:

```
>>> def cosder(x, h=0.01):
...     return (cos(x+h) - cos(x))/h
```

You can call this function with either `cosder(0.)` or `cosder(0., 0.001)`; in the former case, `h` has the value 0.01. Basically, the second argument here is *optional*. As a matter of good practice, you should make sure to always use immutable default parameter values. Finally, note that in Python one has the ability to define a function that deals with an indefinite number of positional or keyword arguments. The syntax for this is `*args` and `**kwargs`, but a detailed discussion would take us too far afield.

A pleasant feature of Python is that *functions are first-class objects*. As a result, we

can pass them in as arguments to other functions; for example, instead of hard-coding the cosine as in our previous function, we could say:

```
>>> def der(f, x, h=0.01):
...     return (f(x+h) - f(x))/h
```

which is called by passing in as the first argument the function of your choice, e.g., `der(sin, 0., 0.05)`. Note how `f` is a regular parameter, but is used inside the function the same way we use functions (by passing arguments to them inside parentheses). We passed in the name of the function, `sin`, as the first argument and the `x` as the second argument.[4] As a rule of thumb, you should pass a function in as an argument if you foresee that you might be passing in another function in its place in the future. If you basically expect to always keep carrying out the same task, there's no need to add yet another parameter to your function definition. Incidentally, we really meant it when we said that in Python functions are first-class objects. You could even have a list whose elements are functions, e.g., `funcs = [sumofints, cos]`. Similarly, a problem explores a dictionary that contains functions as values (or keys).

1.4 Core-Python Idioms

We are now in a position to discuss Pythonic idioms: these are syntactic features that allow us to perform tasks more straightforwardly than would have been possible with the syntax introduced above. Using such alternative syntax to make the code more concise and expressive helps write new programs, but also makes the lives of future readers easier. Of course, you do have to exercise your judgement.[5]

1.4.1 List Comprehensions

At the start of section 1.3.3, we saw how to populate a list: start with an empty one and use `append()` inside a `for` loop to add the elements you need. List comprehensions (often shortened to *listcomps*) provide us with another way of setting up lists. The earlier example can be replaced by `xs = [0.1*i for i in range(20)]`. This is much more compact (one line vs three). Note that when using a list comprehension the loop that steps through the elements of some other sequence (in this case, the result of stepping through `range()`) is placed *inside* the list we are creating! This particular syntax is at first sight a bit unusual, but very convenient and strongly recommended.

It's a worthwhile exercise to replace hand-rolled versions of code using listcomps. For example, if you need a new list whose elements are two times the value of each element in `xs`, you should *not* say `ys = 2*xs`: as mentioned earlier, this concatenates the two lists, which is not what we are after. Instead, what *does* work is `ys = [2*x for x in xs]`.

[4] This means that we did *not* pass in `sin()` or `sin(x)`, as those wouldn't work.
[5] As Emerson put it in his 1841 essay on *Self-Reliance*, "A foolish consistency is the hobgoblin of little minds".