Idiomatic Python

1

> It's not always about speculating; at some point one must think about practice.
>
> Immanuel Kant

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`. Reference [41] is a readable book-length introduction to Python (also available online); Ref. [71] is another introduction, with nice material on visualization. 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, Julia or Rust 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. Before summarizing Python features, we make some big-picture comments on the choice of language, as well as on programming in general.

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; that, combined with the fact that one can use Python interactively, make 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/unrelated 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 hyperefficient code for your research, the insight you may gain from building a prototype in Python could

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

be invaluable; similarly, you could write most of your code in Python and re-express a few 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.

We sometimes 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've mastered the basics of core Python, you may find books like Ref. [120] and Ref. [132] a worthwhile investment. On the wider theme of developing good programming skills, volumes like Ref. [103] 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] Discussion of which, more often than not, sheds light on the narcissism of minor differences ("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 [128] 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-based indexing* (which we will shorten to "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 subsection, 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.

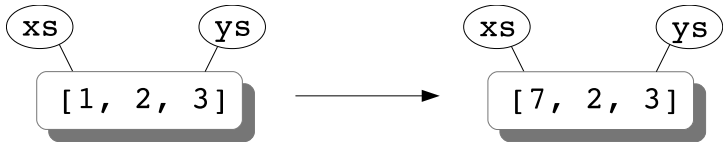
Lists A list is a container of elements; it can can grow when you need it to. Elements can have different types. You use square brackets and comma-separated elements when creating a list, e.g., `zs = [5, 1+2j, -2.0]`. You also use square brackets when indexing into a list, e.g., `zs[0]` is the first 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 `append()` 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



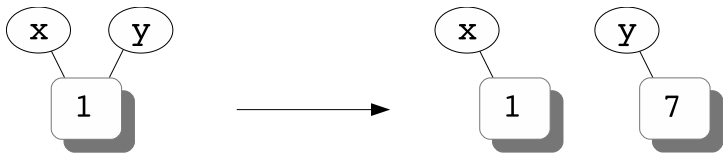
Labelling and modifying a mutable object (in this case, a list) Fig. 1.1

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, `compexp()` as in section 2.4.4, 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*. On a related note, Python contains the keywords `global` and `nonlocal` as well as function one-liners via `lambda`; some of these features are briefly touched upon in problem 1.4.

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, problem 1.2 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 us 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]`. More

[4] This means that we did *not* pass in `sin()` or `sin(x)`, as those wouldn't work.

[5] "A foolish consistency is the hobgoblin of little minds" (Ralph Waldo Emerson, *Self-Reliance*).

generally, if you need to apply a function to every element of a list, you could simply do so on the fly: `ws = [f(x) for x in xs]`. Another powerful feature lets you "prune" a list as you're creating it, e.g., `zs = [2*x for x in xs if x>0.3]`; this doubles an element only if that element is greater than 0.3 (otherwise it doesn't even introduce it).

1.4.2 Iterating Idiomatically

Our earlier example, `ys = [2*x for x in xs]`, is an instance of a significant syntactic feature: a `for` loop is not limited to iterating through a collection of integers in fixed steps (as in our earlier `for i in range(20)`) but can iterate through the list elements themselves *directly*.[6] This is a general aspect of iterating in Python, a topic we now turn to; the following advice applies to all loops (i.e., not only to listcomps).

One list Assuming the list `xs` already exists, you may be tempted to iterate through its elements via something like `for i in range(len(xs)):` then in the loop body you would get a specific element by indexing, i.e., `xs[i]`. The Pythonic alternative is to step through the elements themselves, i.e., `for x in xs:` and then simply use `x` in the loop body. Instead of iterating through an index, which is what the error-prone syntax `range(len(xs))` is doing, this uses Python's `in` to iterate directly through the list elements.

Sometimes, you need to iterate through the elements of a list `xs` in reverse: the old-school (C-influenced) way to do this is `for i in range(len(xs)-1, -1, -1):`, followed by indexing, i.e., `xs[i]`. This works, but all those `-1`'s don't make for light reading; instead, use Python's built-in `reversed()` function, saying `for x in reversed(xs):` and then using `x` directly. A final use case: you often need access to both the index showing an element's place in the list, and the element itself. The "traditional" solution would involve `for i in range(len(xs)):` followed by using `i` and `xs[i]` in the loop body. The Pythonic alternative is to use the built-in `enumerate()` function, `for i, x in enumerate(xs):` and then use `i` and `x` directly; this is more readable and less error-prone.

Two lists We sometimes want to iterate through two lists, `xs` and `ys`, in parallel. You should be getting the hang of things by now; the unPythonic way to do this would be `for i in range(len(xs)):`, followed by using `xs[i]` and `ys[i]` in the loop body. The Pythonic solution is to say `for x, y in zip(xs,ys):` and then use `x` and `y` directly. The `zip()` built-in function creates an iterable entity consisting of tuples, fusing the 0th element in `xs` with the 0th element in `ys`, the 1st element in `xs` with the 1st element[7] in `ys`, and so on.

Dictionaries We can use `for` to iterate Pythonically through more than just lists; in the tutorial you have learned that it also works for lines in a file. Similarly, we can iterate through the *keys* of a dictionary; for our earlier height-to-weight dictionary, you could say `for h in htow:` and then use `h` and `htow[h]` in the loop body. An even more Pythonic way of doing the same thing uses the `items()` method of dictionaries to produce all the key-value pairs: `for h,w in htow.items():` lets you use `h` and `w` inside the loop body.

[6] In other words, Python's `for` is similar to the `foreach` that some other languages have.
[7] In English we say "first", "second", etc. We'll use numbers, e.g., 0th, when using the 0-indexing convention.

Code 1.1 forelse.py

```
def look(target,names):
      for name in names:
            if name==target:
                  val = name
                  break
      else:
            val = None
      return val

names = ["Alice", "Bob", "Eve"]
print(look("Eve", names))
print(look("Jack", names))
```

For-else In this section we've spent some time discussing the line where `for` appears. We now turn to a way of using `for` loops that is different altogether: similarly to `if` statements, you can follow a `for` loop by an `else` (!). This is somewhat counterintuitive, but can be very helpful. The way this works is that the `for` loop is run as usual: if no `break` is encountered during execution of the `for` block, then control proceeds to the `else` block. If a `break` is encountered during execution of the `for` block, then the `else` block is not run. (Try re-reading the last two sentences after you study code 1.1.)

The `else` in a `for` loop is nice when we are looking for an item within a sequence and we need to do one thing if we find it and a different thing if we don't. An example is given in code 1.1, the first full program in the book. We list such codes in boxes with the filename at the top; we strongly recommend you download these Python codes and use them in parallel to reading the main text;[8] you would run this by typing `python forelse.py` or something along those lines; note that, unlike other languages you may have used in the past, there is no *compilation* stage. This specific code uses several of the Python features mentioned in this chapter: it starts with defining a function, `look()`, which we will discuss in more detail below. The main program uses a listcomp to produce a list of strings and then calls the `look()` function twice, passing in a different first argument each time; we don't even need a variable to hold the first argument(s), using string literal(s) directly. Since we are no longer using Python interactively, we employ `print()` to ensure that the output is printed on the screen (instead of being evaluated and then thrown away).

Turning to the `look()` function itself, it takes in two parameters: one is (supposed to be) a target string and the other one is a list of strings. The latter could be very long, e.g., the first names listed in a phone book. Note that there are three levels of indentation involved here: the function body gets indented, the `for` loop body gets indented, and then the conditional

[8] "One cannot so well grasp a thing and make it one's own, when it has been learned from someone else, as when one has discovered it oneself" (René Descartes, *Discourse on the Method*, part VI).

expression body also gets indented. Incidentally, our `for` loop is iterating through the list elements directly, as per our earlier admonition, instead of using error-prone indices. You should spend some time ensuring that you understand what's going on. Crucially, the `else` is indented at the level of the `for`, not at the level of the `if`. We also took the opportunity to employ another idiom mentioned above: `None` is used to denote the absence of a value. The two possibilities that are at play (target in sequence or target not in sequence) are probed by the two function calls in the main program. When the target is found, a `break` is executed, so the `else` is not run. When the target is not found, the `else` is run. It might help you to think of this `else` as being equivalent to `nobreak`: the code in that block is only executed when no `break` is encountered in the main part of the `for` loop. (Of course, this is only a mnemonic, since `nobreak` is not a reserved word in Python.) We will use the for-else idiom repeatedly in this volume (especially in chapter 5), whenever we are faced with an iterative task which may plausibly fail, in which case we wish to communicate that fact to the rest of the program. Since even some expert users are uncomfortable with the for-else idiom, problem 1.5 gives you a tour of the alternatives.

1.5 Basic Plotting with `matplotlib`

We will now visualize relationships between numbers via `matplotlib`, a plotting library (i.e., not part of core Python) which can produce quality figures: all the plots in this book were created using `matplotlib`. Inside the `matplotlib` package is the `matplotlib.pyplot` module, which is used to produce figures in a MATLAB-like environment.

A simple example is given in code 1.2. This starts by importing `matplotlib.pyplot` in the (standard) way which allows us to use it below without repeated typing of unnecessary characters. We then define a function, `plotex()`, that takes care of the plotting, whereas the main program simply introduces four list comprehensions and then calls our function. The listcomps also employ idiomatic iteration, in the spirit of applying what you learned in the previous section. If you're still a beginner, you may be wondering why we defined a Python function in this code. An important design principle in computer science goes by the name of *separation of concerns* (or sometimes *information hiding* or *encapsulation*): each aspect of the program should be handled separately. In our case, this means that each component of our task should be handled in a separate function.

Let's discuss this function in more detail. Its parameters are (meant to be) four lists, namely two pairs of x_i and y_i values. The function body starts by using `xlabel()` and `ylabel()` to provide labels for the x and y axes. It then creates individual curves/sets of points by using `matplotlib`'s function `plot()`, passing in the x-axis values as the first argument and the y-axis values as the second argument. The third positional argument to `plot()` is the *format string*: this corresponds to the color and point/line type. In the first case, we used `r` for red and `-` for a solid line. Of course, figures in this book are in black and white, but you can produce the color version using the corresponding Python code. In order to help you interpret this and other format strings, we list allowed colors and some

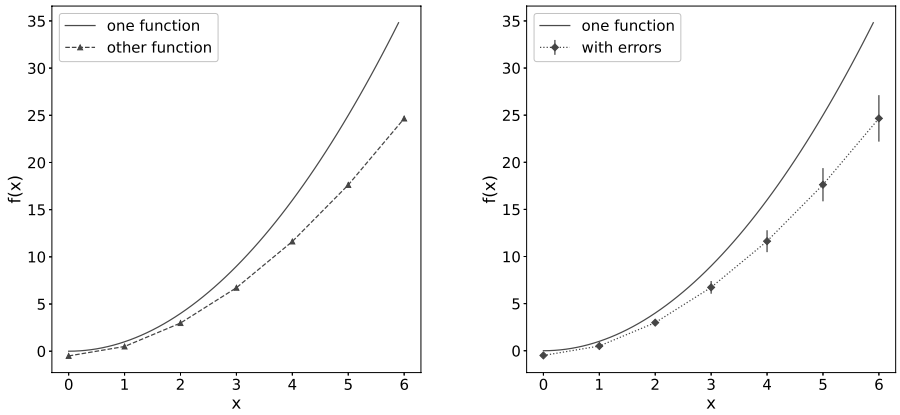
Code 1.2 plotex.py

```
import matplotlib.pyplot as plt

def plotex(cxs,cys,dxs,dys):
      plt.xlabel('x', fontsize=20)
      plt.ylabel('f(x)', fontsize=20)
      plt.plot(cxs, cys, 'r-', label='one function')
      plt.plot(dxs, dys, 'b--^', label='other function')
      plt.legend()
      plt.show()

cxs = [0.1*i for i in range(60)]
cys = [x**2 for x in cxs]
dxs = [i for i in range(7)]
dys = [x**1.8 - 0.5 for x in dxs]

plotex(cxs, cys, dxs, dys)
```

of the most important line styles/markers in table 1.1. The fourth argument to `plot()` is a keyword argument containing the label corresponding to the curve. In the second call to `plot()` we pass in a different format string and label (and, obviously, different lists); observe that we used two style options in the format string: `--` to denote a dashed line and `^` to denote the points with a triangle marker. The function concludes by calling `legend()`, which is responsible for making the legend appear, and `show()`, which makes the plot actually appear on our screen.

Table 1.1 Color, line styles, and markers in `matplotlib`

Character	Color	Character	Description
'b'	blue	'-'	solid line style
'g'	green	'--'	dashed line style
'r'	red	'-.'	dash-dot line style
'c'	cyan	':'	dotted line style
'm'	magenta	'o'	circle marker
'y'	yellow	's'	square marker
'k'	black	'D'	diamond marker
'w'	white	'^'	triangle-up marker



Examples of figures produced using `matplotlib` **Fig. 1.3**

The result of running this program is in the left panel of Fig. 1.3. A scenario that pops up very often in practice involves plotting points with error bars:

```
dyerrs = [0.1*y for y in dys]
plt.errorbar(dxs, dys, dyerrs, fmt='b:D', label='with errors')
```

where we have called `errorbar()` to plot the points with error bars: the three positional arguments here are the x values, the y values, and the errors in the y values. After that, we pass in the format string using the keyword argument `fmt` and the label as usual. We thereby produce the right panel of Fig. 1.3.

We could fine-tune almost all aspects of our plots, including basic things like line width, font size, and so on. For example, we could get TeX-like equations by putting dollar signs inside our string, e.g., 'x_i' appears as x_i. We could control which values are displayed via `xlim()` and `ylim()`, we could employ a log-scale for one or both of the axes (using `xscale()` or `yscale()`), and much more. The online documentation can help you go beyond these basic features. Finally, we note that instead of providing `matplotlib` with Python lists as input, you could be using NumPy arrays; this is the topic we now turn to.

1.6 NumPy Idioms

NumPy arrays are not used in chapters 2 and 3 so, if you are still new to Python, you should focus on mastering Python lists: how to grow them, modify them, etc. Thus, you should *skip this section and the corresponding NumPy tutorial for now* and come back when you are about to start reading chapter 4. If you're feeling brave you can keep reading, but know that it is important to distinguish between Python lists and NumPy arrays.

In our list-based codes, we typically carry out the same operation over and over again, a fixed number of times, e.g., `contribs = [w*f(x) for w,x in zip(ws,xs)]`; this list com-

Table 1.2 Commonly used numpy data types

Type	Variants
Integer	`int8`, `int16`, `int32`, `int64`
Float	`float16`, `float32`, `float64`, `longdouble`
Complex	`complex64`, `complex128`, `complex256`

prehension uses the `zip()` built-in to step through two lists in parallel. Similarly, our Python lists are almost always "homogeneous", in the sense that they contain elements of only one type. This raises the natural question: wouldn't it make more sense to carry out such tasks using a homogeneous, fixed-length container? This is precisely what the Numerical Python (NumPy) array object does: as a result, it is fast and space-efficient. It also allows us to avoid, for the most part, having to write loops: even in our listcomps, which are more concise than standard loops, there is a need to explicitly step through each element one by one. Numerical Python arrays often obviate such syntax, letting us carry out mathematical operations on entire blocks of data in one step.[9] The standard convention is to import `numpy` with a shortened name: `import numpy as np`.

One-dimensional arrays One-dimensional (1d) arrays are direct replacements for lists. The easiest way to make an array is via the `array()` function, which takes in a sequence and returns an array containing the data that was passed in, e.g., `ys = np.array(contribs)`; the `array()` function is part of `numpy`, so we had to say `np.array()` to access it. There is both an `array()` function and an array object involved here: the former created the latter. Printing out an array, the commas are stripped, so you can focus on the numbers. If you want to see how many elements are in the array `ys`, use the `size` attribute, via `ys.size`. Remember: NumPy arrays are fixed-length, so the total number of elements cannot change. Another very useful attribute arrays have is `dtype`, namely the data type of the elements; table 1.2 collects several important data types. When creating an array, the data type can also be explicitly provided, e.g., `zs = np.array([5, 8], dtype=np.float32)`.

NumPy contains several handy functions that help you produce pre-populated arrays, e.g., `np.zeros(5)`, `np.ones(4)`, or `np.arange(6)`. The latter also works with float arguments, however, as we will learn in the following chapter, this invites trouble. For example, `np.arange(1.5,1.75,0.05)` and `np.arange(1.5,1.8,0.05)` behave quite differently. Instead, use the `linspace()` function to get a specified number of evenly spaced elements over a specified interval, e.g., `np.linspace(1.5,1.75,6)` or `np.linspace(1.5,1.8,7)`. There also exists a function called `logspace()`, which produces a logarithmically spaced grid of points.

Indexing for arrays works as for lists, namely with square brackets, e.g., `zs[2]`. Slicing appears, at first sight, to also be identical to how slicing of lists works. However, there is a crucial difference, namely that *array slices are views on the original array*; let's revisit our

[9] From here onward, we will not keep referring to these new containers as `numpy` arrays: they'll simply be called arrays, the same way the core-Python lists are simply called lists.

earlier example. For arrays, `xs = np.array([1,2,3])`, followed by `ys = xs[1:]`, and then `ys[0] = 7` *does* affect `xs`. NumPy arrays are efficient, eliminating the need to copy data: of course, one should always be mindful that different slices all refer to the same underlying array. For the few cases where you do need a true copy, you can use `np.copy()`, e.g., `ys = np.copy(xs[1:])` followed by `ys[0] = 7` does not affect `xs`. We could, just as well, copy the entire array over, without impacting the original. In what follows, we frequently make copies of arrays inside functions, in the spirit of impacting the external world only through the return value of a function.

Another difference between lists and arrays has to with *broadcasting*. Qualitatively, this means that NumPy often knows how to handle entities whose dimensionalities don't quite match. For example, `xs = np.zeros(5)` followed by `xs[:] = 7`, leads to an array of five sevens. Remember, since array slices are views on the original array, `xs[:]` cannot be used to create a copy of the array, but it *can* be used to broadcast one number onto many slots; this syntax is known as an "everything slice". Without it, `xs = 7` leads to `xs` becoming an integer (number) variable, not an array, which isn't what we want.

The real strength of arrays is that they let you carry out operations such as `xs + ys`: if these were lists, you would be concatenating them, but for arrays addition is interpreted as an elementwise operation (i.e., each pair of elements is added together).[10] If you wanted to do the same thing with lists you would need a listcomp and `zip()`. This ability to carry out such batch operations on all the elements of an array (or two) at once is often called *vectorization*. You could also use other operations, e.g., to sum the results of pairwise multiplication you simply say `np.sum(xs*ys)`. This is simply evaluating the scalar product of two vectors, in one short expression. Equally interesting is the ability NumPy has to also combine an array with a scalar: this follows from the aforementioned *broadcasting*, whereby NumPy knows how to interpret entities with different (but compatible) shapes. For example, `2*xs` doubles each array element; this is very different from what we saw in the case of lists. You can think of what's happening here as the value `2` being "stretched" into an array with the same size as `xs` and then an elementwise multiplication taking place. Needless to say, you could carry out several other operations with scalars, e.g., `1/xs`. Such combinations of broadcasting (whereby you can carry out operations between scalars and arrays) and vectorization (whereby you write one expression but the calculation is carried out for all elements) can be hard to grasp at first, but are very powerful (both expressive and efficient) once you get used to them.

Another very useful function, `np.where()`, helps you find specific indices where a condition is met, e.g., `np.where(2 == xs)` returns a tuple of arrays, so we would be interested in its 0th element. NumPy also contains several functions that look similar to corresponding `math` functions, e.g., `np.sqrt()`; these are are designed to take in entire arrays so they are almost always faster. NumPy also has functions that take in an array and return a scalar, like the `np.sum()` we encountered above. Another very helpful function is `np.argmax()`, which returns the index of the maximum value. You can also use NumPy's functionality to create your own functions that can handle arrays. For example, our earlier `contribs = [w*f(x) for w,x in zip(ws,xs)]` can be condensed into the much cleaner `contribs =`

[10] Conversely, if you wish to concatenate two arrays you cannot use addition; use `np.concatenate()`.

Table 1.3 Important attributes of numpy arrays

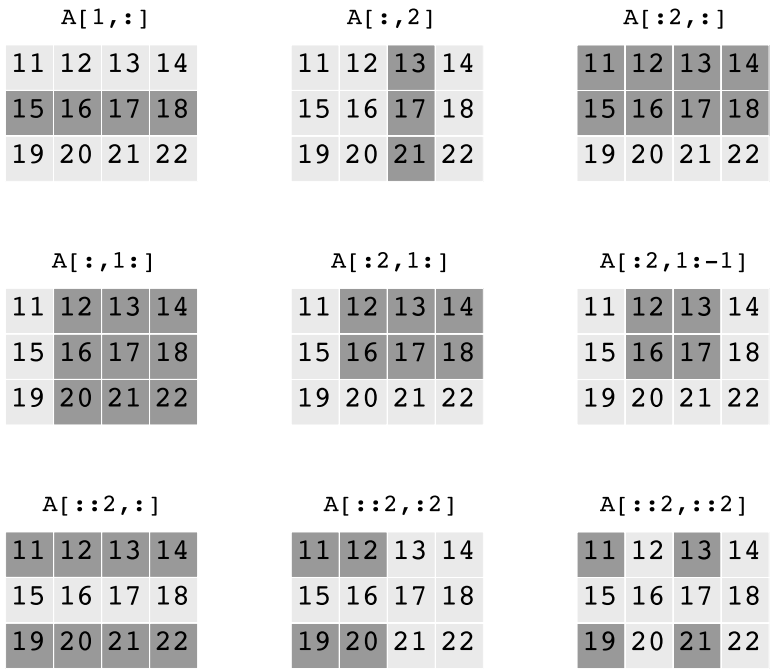
Attribute	Description
`dtype`	Data type of array elements
`ndim`	Number of dimensions of array
`shape`	Tuple with number of elements for each dimension
`size`	Total number of elements in array

`ws*fa(xs)`, where `fa()` is designed to take in arrays (so, e.g., it uses `np.sqrt()` instead of `math.sqrt()`).

Two-dimensional arrays In core Python, matrices can be represented by lists of lists which are, however, quite clunky (as you'll further experience in the following section). In Python a list-of-lists is introduced by, e.g., `LL = [[11,12], [13,14], [15,16]]`. Just like for one-dimensional arrays, we can say `A = np.array(LL)` to produce a two-dimensional (2d) array that contains the elements in `LL`. If you type `print(A)` the Python interpreter knows how to strip the commas and split the output over three rows, making it easy to see that it's a two-dimensional entity, similar to a mathematical matrix.

Much of what we saw on creating one-dimensional arrays carries over to the two-dimensional case. For example, `A.size` is 6: this is the total number of elements, including both rows and columns. Another attribute is `ndim`, which tells us the dimensionality of the array: `A.ndim` is 2 for our example. The number of dimensions can be thought of as the number of distinct "axes" according to which we are listing elements. Incidentally, our terminology may be confusing to those coming from a linear-algebra background. In linear algebra, we say that a matrix $\mathbf{A}$ with m rows and n columns has "dimensions" m and n, or sometimes that it has dimensions $m \times n$. In contradistinction to this, the NumPy convention is to refer to an array like `A` as having "dimension 2", since it's made up of rows and columns (i.e., how many elements are in each row and in each column doesn't matter). There exists yet another attribute, `shape`, which returns a tuple containing the number of elements in each dimension, e.g., `A.shape` is `(3, 2)`. Table 1.3 collects the attributes we'll need. You can also create two-dimensional arrays by passing a tuple with the desired shape to the appropriate function, e.g., `np.zeros((3,2))` or `np.ones((4,6))`. Another function helps you make a square identity matrix via, e.g., `np.identity(4)`. All of these functions produce floats, by default.

If you want to produce an array starting from a hand-rolled list, but wish to avoid the Python list-of-lists syntax (with double square brackets, as well as several commas), you can start from a one-dimensional list, which is then converted into a one-dimensional array, which in its turn is re-shaped into a two-dimensional array via the `reshape()` function, e.g., `A = np.array([11,12,13,14,15,16]).reshape(3,2)`. Here's another example: `A = np.arange(11,23).reshape(3,4)`. Observe how conveniently we've obviated the multitude of square brackets and commas of `LL`.



Examples of slicing a 3×4 two-dimensional array

Fig. 1.4

It is now time to see one of the nicer features of two-dimensional arrays in NumPy: the intuitive way to access elements. To access a specific element of `LL`, you have to say, e.g., `LL[2][1]`. Recall that we are using Python 0-indexing, so the rows are numbered 0th, 1st, 2nd, and analogously for the columns. This double pair of brackets, separating the numbers from each other, is quite cumbersome and also different from how matrix notation is usually done, i.e., A_{ij} or $A_{i,j}$. Thus, it comes as a relief to see that NumPy array elements can be indexed simply by using only one pair of square brackets and a comma-separated pair of numbers, e.g., `A[2, 1]`.

Slicing is equally intuitive, e.g., `A[1,:]` picks a specific row and uses an everything-slice for the columns, and therefore leads to that entire row. Figure 1.4 shows a few other examples: the highlighting shows the elements that are chosen by the slice shown at the top of each matrix. The most interesting of these is probably `A[:2,1:-1]`, which employs the Python convention of using `-1` to denote the last element, in order to slice the "middle" columns: `1:-1` avoids the first/0th column and the last column. We also show a few examples that employ a stride when slicing. To summarize, when we use two numbers to index (such as `A[2,1]`), we go from a two-dimensional array to a number. When we use one number to index/slice (such as `A[:,2]`), we go from a two-dimensional array to a one-dimensional array. When we use slices (such as `A[::2,:]`), we get collections of rows, columns, individual elements, etc. We can combine what we just learned about slicing two-dimensional arrays with what we already know about NumPy broadcasting. For

example, `A[:,:] = 7` overwrites the entire matrix and you could do something analogous for selected rows, columns, etc.

Similarly to the one-dimensional case, *vectorized* operations for two-dimensional arrays allow us to, say, add together (elementwise) two square matrices, `A + B`. On the other hand, a simple multiplication is *also* carried out elementwise, i.e., each element in `A` is multiplied by the corresponding element in `B` when saying `A*B`. This may be unexpected behavior, if you were looking for a matrix multiplication. To repeat, *array operations are always carried out elementwise*, so when you multiply two two-dimensional arrays you get the *Hadamard product*, $(\mathbf{A} \odot \mathbf{B})_{ij} = A_{ij}B_{ij}$. The matrix multiplication that you know from linear algebra follows from a different formula, namely $(\mathbf{AB})_{ij} = \sum_k A_{ik}B_{kj}$, see Eq. (C.10). From Python 3.5 and onward[11] you can multiply two matrices using the `@` infix operator, i.e., `A@B`. In older versions you had to say `np.dot(A,B)`, which was not as intuitive as one would have liked. There are many other operations we could carry out, e.g., we can multiply together a two-dimensional and a one-dimensional matrix, `A@xs` or `xs@A`. It's easy to come up with more convoluted examples; here's another one: `xs@A@ys` is much easier to write (and read) than `np.dot(xs,np.dot(A,ys))`. Problem 1.8 studies the dot product of two one-dimensional arrays in detail: the main takeaway is that we compute it via `xs@ys`.[12] In a math course, to take the dot product (also known as the *inner product*) of two vectors you had to first take the transpose of the first vector (to have the dimensions match); conveniently, NumPy takes care of all of that for us, allowing us to simply say `xs@ys`.[13] Finally, *broadcasting* with two-dimensional arrays works just as you'd expect, e.g., `A/2` halves all the matrix elements.

NumPy contains several other important functions, with self-explanatory names. Since we just mentioned the (potential) need to take the transpose: NumPy has a function called `transpose()`; we prefer to access the transpose as an array attribute, i.e., `A.T`. You may recall our singing the praises of the `for` loop in Python; well, another handy idiom involves iterating over rows via `for row in A:` or over columns via `for column in A.T:`; marvel at how expressive both these options are. Iterating over columns will come in very handy in section 4.4.4 when we will be handling eigenvectors of a matrix.

In linear algebra the product of an $n \times 1$ column vector and a $1 \times n$ row vector produces an $n \times n$ matrix. If you try to do this "naively" in NumPy using 1d arrays, you will be disappointed: both `xs@ys` and `xs@(ys.T)` give a number (the same one). Inspect the `shape` attribute of `ys` and of `ys.T` to see what's going on. What you really need is the function `np.outer()`, which computes the outer product, e.g., `np.outer(xs, ys)`.

All the NumPy functions we mentioned earlier, such as `np.sqrt()`, also work on two-dimensional arrays, and the same holds for subtraction, powers, etc. Intriguingly, functions like `np.sum()`, `np.amin()`, and `np.amax()` also take in an (optional) argument `axis`: if you set `axis = 0` you operate across rows (column-wise) and `axis = 1` operates across columns (row-wise). Similarly, we can create our own user-defined functions that know how to handle an entire two-dimensional array/matrix at once. There are also several handy functions that are intuitively easier to grasp for the case of two-dimensional ar-

[11] Python 3.5 was already 8 years old when the present textbook came out.

[12] Make sure not to get confused by the fact that `A@B` produces a matrix but `xs@ys` produces a number.

[13] Similarly, when multiplying a vector by a matrix, `xs@A` is just as simple to write as `A@xs`.

rays, e.g., `np.diag()` to get the diagonal elements, or `np.tril()` to get the lower triangle of an array and `np.triu()` to get the upper triangle. In chapter 8 we will also need to use `np.meshgrid()`, which takes in two coordinate vectors and returns coordinate matrices. Another helpful function is `np.fill_diagonal()` which allows you to efficiently update the diagonal of a matrix you've already created. Finally, `np.trace()` often comes in handy.

The default storage-format for a NumPy array in memory is *row major*: as a beginner, you shouldn't worry about this too much, but it means that rows are stored in order, one after the other. This is the same format used by the C programming language. If you wish to use, instead, Fortran's column-major format, presumably in order to interoperate with code in that language, you simply pass in an appropriate keyword argument when creating the array. If you structure your code in the "natural" way, i.e., first looping through rows and then through columns, all should be well, so you can ignore this paragraph.

1.7 Project: Visualizing Electric Fields

Each chapter concludes with a physical application of techniques introduced up to that point. Since this is only the first chapter, we haven't covered any numerical methods yet. Even so, we can already start to look at some physics that is not so accessible without a computer by using `matplotlib` for more than just basic plotting. We will visualize a vector field, i.e., draw field lines for the electric field produced by several point charges.

1.7.1 Electric Field of a Distribution of Point Charges

Very briefly, let us recall *Coulomb's law*: the force on a test charge Q located at point P (at the position $\mathbf{r}$), coming from a single point charge q_0 located at $\mathbf{r}_0$ is given by:

$$\mathbf{F}_0 = k\frac{q_0 Q}{(\mathbf{r}-\mathbf{r}_0)^2}\frac{\mathbf{r}-\mathbf{r}_0}{|\mathbf{r}-\mathbf{r}_0|} \tag{1.1}$$

where Coulomb's constant is $k = 1/(4\pi\epsilon_0)$ in SI units (and ϵ_0 is the permittivity of free space). The force is proportional to the product of the two charges, inversely proportional to the square of the distance between the two charges, and points along the line from charge q_0 to charge Q. The electric field is then the ratio of the force $\mathbf{F}_0$ with the test charge Q in the limit where the magnitude of the test charge goes to zero. In practice, this gives us:

$$\mathbf{E}_0(\mathbf{r}) = kq_0\frac{\mathbf{r}-\mathbf{r}_0}{|\mathbf{r}-\mathbf{r}_0|^3} \tag{1.2}$$

where we cancelled out the Q and also took the opportunity to combine the two denominators. This is the electric field at the location $\mathbf{r}$ due to the point charge q_0 at $\mathbf{r}_0$.

If we were faced with more than one point charge, we could apply the *principle of superposition*: the total force on Q is made up of the vector sum of the individual forces acting on Q. As a result, if we were dealing with the n point charges $q_0, q_1, \ldots, q_{n-1}$ located at $\mathbf{r}_0, \mathbf{r}_1, \ldots, \mathbf{r}_{n-1}$ (respectively) then the situation would be that shown in Fig. 1.5. Our figure is in two dimensions for ease of viewing, but the formalism applies equally well to three dimensions. The total electric field at the location $\mathbf{r}$ is:

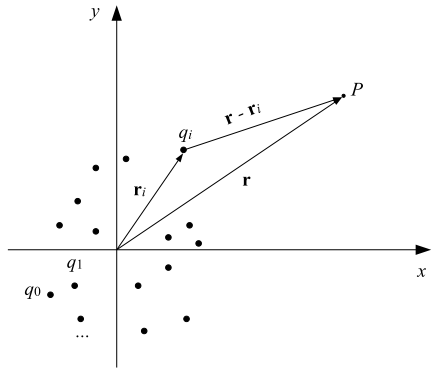


Fig. 1.5 Physical configuration made up of n point charges

$$\mathbf{E}(\mathbf{r}) = \sum_{i=0}^{n-1} \mathbf{E}_i(\mathbf{r}) = \sum_{i=0}^{n-1} kq_i \frac{\mathbf{r} - \mathbf{r}_i}{|\mathbf{r} - \mathbf{r}_i|^3} \tag{1.3}$$

namely, a sum of the individual electric field contributions, $\mathbf{E}_i(\mathbf{r})$. Note that you can consider this total electric field at any point in space, $\mathbf{r}$. Note, also, that the electric field is a vector quantity: at any point in space this $\mathbf{E}$ has a magnitude and a direction. One way of visualizing vector fields consists of drawing *field lines*, namely imaginary curves that help us keep track of the direction of the field. More specifically, the tangent of a field line at a given point gives us the direction of the electric field at that point. Field lines do not cross; they start at positive charges ("sources") and end at negative charges ("sinks").

1.7.2 Plotting Field Lines

We will plot the electric field lines in Python; while more sophisticated ways of visualizing a vector field exist (e.g., line integral convolution), what we describe below should be enough to give you a qualitative feel for things. While plotting functions (or even libraries) tend to change much faster than other aspects of the programming infrastructure, the principles discussed apply no matter what the specific implementation looks like.

We are faced with two tasks: first, we need to produce the electric field (vector) at several points near the charges as per Eq. (1.3) and, second, we need to plot the field lines in such a way that we can physically interpret what is happening. As in the previous code, we make a Python function for each task. For simplicity, we start from a problem with only two point charges (of equal magnitude and opposite sign). Also, we restrict ourselves to two dimensions (the Cartesian x and y).

Code 1.3 is a Python implementation, where Coulomb's constant is divided out for simplicity. We start by importing `numpy` and `matplotlib`, since the heavy lifting will be done

vectorfield.py **Code 1.3**

```
import numpy as np
import matplotlib.pyplot as plt
from math import sqrt
from copy import deepcopy

def makefield(xs, ys):
      qtopos = {1: (-1,0), -1: (1,0)}
      n = len(xs)
      Exs = [[0. for k in range(n)] for j in range(n)]
      Eys = deepcopy(Exs)
      for j,x in enumerate(xs):
            for k,y in enumerate(ys):
                  for q,pos in qtopos.items():
                        posx, posy = pos
                        R = sqrt((x - posx)**2 + (y - posy)**2)
                        Exs[k][j] += q*(x - posx)/R**3
                        Eys[k][j] += q*(y - posy)/R**3
      return Exs, Eys

def plotfield(boxl,n):
      xs = [-boxl + i*2*boxl/(n-1) for i in range(n)]
      ys = xs[:]
      Exs, Eys = makefield(xs, ys)
      xs=np.array(xs); ys=np.array(ys)
      Exs=np.array(Exs); Eys=np.array(Eys)
      plt.streamplot(xs, ys, Exs, Eys, density=1.5, color='m')
      plt.xlabel('$x$')
      plt.ylabel('$y$')
      plt.show()

plotfield(2.,20)
```

by the function `streamplot()`, which expects NumPy arrays as input. We also import the square root and the `deepcopy()` function, which can create a distinct list-of-lists.

The function `makefield()` takes in two lists, `xs` and `ys`, corresponding to the coordinates at which we wish to evaluate the electric field (x and y together make up $\mathbf{r}$). We also need some way of storing the $\mathbf{r}_i$ at which the point charges are located. We have opted to store these in a dictionary, which maps from charge q_i to position $\mathbf{r}_i$—take some time to consider

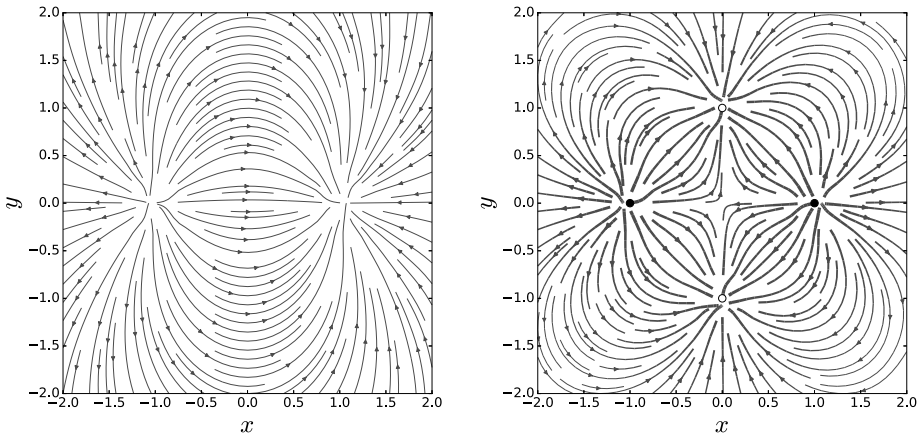
alternative ("manual") implementations. For each position $\mathbf{r}$ we need to evaluate $\mathbf{E}(\mathbf{r})$: in two dimensions, this is made up of $E_x(\mathbf{r})$ and $E_y(\mathbf{r})$, namely the two Cartesian components of the total electric field. Focusing on only one of these for the moment, say $E_x(\mathbf{r})$, we realize that we need to store its value for any possible $\mathbf{r}$, i.e., for any possible x and y values. We decide to use a list-of-lists, produced by a nested list comprehension. We then create another list-of-lists, for $E_y(\mathbf{r})$. We need to map out (i.e., store) the value of the x and y components of the total electric field, at all the desired values of the vector $\mathbf{r}$, namely, on a two-dimensional grid made up of `xs` and `ys`. This entails computing the electric field (contribution from a given point charge q_i) at all possible y's for a given x, and then iterating over all possible x's. We also need to iterate over our point charges q_i and their locations $\mathbf{r}_i$ (i.e., the different terms in the sum of Eq. (1.3)); we do this by saying `for q, pos in qtopos.items():` at which point we unpack `pos` into `posx` and `posy`.

We thus end up with three nested loops: one over possible x values, one over possible y values, and one over i. All three of these are written idiomatically, employing `items()` and `enumerate()`. The latter was used to ensure that we won't only have access to the x and the y values, which are needed for the right-hand side of Eq. (1.3), but also to two indices (j and k) that will help us store the electric-field components in the appropriate list-of-lists entry, e.g., `Exs[k][j]`.[14] This storing is carried out after defining a helper variable to keep track of the vector magnitude that appears in the denominator in Eq. (1.3). You should think about the += a little bit: since the left-hand side is for given j and k, the summation is carried out only when we iterate over the q_i (and $\mathbf{r}_i$). Incidentally, our idiomatic iteration over the point charges means that we don't even need an explicit i index.

Our second function, `plotfield()`, is where we build our two-dimensional grid for the `xs` and `ys`.[15] We take in as parameters the length L and the number of points n we wish to use in each dimension and create our `xs` using a list comprehension; all we're doing is picking x's from $-L$ to L. We then create a copy of `xs` and name it `ys`. After this, we call our very own `makefield()` to produce the two lists-of-lists containing $E_x(\mathbf{r})$ and $E_y(\mathbf{r})$ for many different choices of $\mathbf{r}$. The core of the present function consists of a call to `matplotlib`'s function `streamplot()`; this expects NumPy arrays instead of Python lists, so we convert everything over. If you skipped section 1.6, as you were instructed to do, you should relax: this call to `np.array()` is all you need to know for now (and until chapter 4). We also pass in to `streamplot()` two (optional) arguments, to ensure that we have a larger density of field lines and to choose the color. Most importantly, `streamplot()` knows how to take in `Exs` and `Eys` and output a plot containing curves with arrows, exactly like what we are trying to do. We also introduce x and y labels, using dollar signs to make the symbols look nicer.

The result of running this code is shown in the left panel of Fig. 1.6. Despite the fact that the charges are not represented by a symbol in this plot, you can easily tell that you are dealing with a positive charge on the left and a negative charge on the right. At this point, we realize that a proper graphic representation of field lines also has another feature: the density of field lines should correspond to the strength of the field (i.e., its magnitude). Our

[14] The confusing index order follows `streamplot()`'s documentation (see also page 630).
[15] This would all be much easier with two-dimensional arrays, but you skipped that section.



Visualizing the electric fields resulting from two and four point charges **Fig. 1.6**

figure has discarded that information: the `density` argument we passed in had a constant value. This is a limitation of the `streamplot()` function.

There is a way to represent both the direction (as we already did) and the strength of the field using `streamplot()`, using the optional `linewidth` parameter. The argument passed in to `linewidth` can be a two-dimensional NumPy array, which keeps track of the strength at each point on the grid; it's probably better to pass in, instead, the logarithm of the magnitude at each point (possibly also using an offset). We show the result of extending our code to also include line width in the right panel of Fig. 1.6, where a stronger field is shown using a thicker line. This clearly shows that the field strength is larger near the charges (and in between them) than it is as you go far away from them. To make things interesting, this shows a different situation than the previous plot did: we are now dealing with four charges (two positive and two negative, all of equal magnitude). We also took the opportunity to employ symbols representing the position of the point charges themselves.

Problems

1.1 Study the following program, meant to evaluate the factorial of a positive integer:

```
def fact(n):
      return 1 if n==0 else n*fact(n-1)
print(fact(10))
```

This uses Python's version of the *ternary operator*. Crucially, it also uses *recursion*: we are writing the solution to our problem in terms of the solution of a smaller version of the same problem. The function then calls itself repeatedly. At some point we reach the *base case*, where the answer can be directly given. Recursion is helpful when the problem you are solving is amenable to a "divide-and-conquer" approach,

as we will see in section 6.4.3.3. For the example above, recursion is quite unnecessary: write a Python function that evaluates the factorial *iteratively*.

1.2 Produce a function, `descr()`, which describes properties of the function and argument value that are passed in to it. Define it using `def descr(f, x):`.

(a) Write the function body; this should separately print out `f`, `x`, and `f(x)`. Now call it repeatedly, for a number of user-defined and Python standard library functions. Notice that when you print out `f` the output is not human friendly.
(b) Pass in as the first argument not the function but a string containing the function name. You now need a mechanism that converts the string you passed in to a function (since you still need to print out `f(x)`). Knowing the list of possibilities, create a dictionary that uses strings as keys and functions as values.
(c) Modify the previous program so that the first argument that is passed in to `descr()` is a function, not a string. To produce the same output, you must also modify the body of the function since your dictionary will now be different.

1.3 Iterate through a list `xs` in reverse, printing out both the (decreasing) index and each element itself. Come up with both Pythonic and unPythonic solutions.

1.4 Imagine you have access to two Python functions, `fa()` and `fb()`, with one parameter each.[16] You now wish to apply the function `der()` that was defined in section 1.3.4 to a linear combination of `fa()` and `fb()`, namely `a*fa(x) + b*fb(x)`.

(a) Your initial reaction might be to define a third function, `fc()`; but what if you don't actually know the values of the `a` and `b` coefficients until you're already inside some other Python function, say, `caller()`? Based on what we've introduced in the main text, one idea is to define `fc()` to take three parameters (`x`, `a`, and `b`). This would then not interoperate with `der()`, which expects as its first argument a function taking in a single parameter. Try this.
(b) Your next instinct may be to modify the definition of `der()`: sometimes that's OK, but often it's just asking for trouble. What you're really after is a throwaway function that knows about the values of variables defined in the same scope. Do so via an anonymous function (via Python's `lambda` keyword) right where you need it. Look up the documentation and implement this solution.
(c) You may next be tempted to actually *name* your lambda function and then use it in `der()`. Try this. Unfortunately, this is explicitly discouraged in PEP 8.[17]
(d) Instead, define a *nested function* `func()` inside `caller()`, the latter calling `der()`.
(e) Use a *closure*: `func()` is nested inside `caller()`, the latter returning `func()`; recall that in Python you use the function name without the parentheses.

1.5 We will now elaborate on the for-else idiom introduced in `forelse.py`.

(a) First, we realize that our `look()` function, while helpful from a pedagogical perspective, is somewhat pointless: you didn't really need to write a function to

[16] The same idea applies to the two functions `sumofints()` and `cos()` that we encountered in the main text.
[17] It's easy to see why: if you feel the need to name your... anonymous function, then perhaps you shouldn't be using an anonymous function in the first place.

check for membership in a list. Directly use Python's `in` keyword to test whether or not a given string is to be found in a given list of strings.

(b) Now make `look()` slightly more useful by having it return not only the value (i.e., the string) but also the index where that value was found.

(c) Experiment with making `look()` shorter, by avoiding the for-else idiom altogether. First, do this the simplest way, i.e., by providing the default option `val = None` *before* you enter the loop. Second, avoid using `break` altogether by opting for two exit points in your function, i.e., two separate `return` statements.[18]

(d) The for-else idiom is actually better than the alternatives when exceptions (which we don't discuss elsewhere) are involved. Check the documentation to see how you can `raise` a `ValueError` in the `else` block of your `for` loop.

1.6 The following is implicitly defining a recurrence relation:

```
f0,f1 = 0,1
for i in range(n-1):
      f0,f1 = f1,f0+2*f1
```

We will now produce increasingly fancier versions of this code snippet.

(a) Define a function that takes in the cardinal number n and returns the corresponding latest value following the above recurrence relation. In other words, for $n = 0$ you should get 0, for $n = 1$ you should get 1, for $n = 2$ you should get 2, for $n = 3$ you should get 5, and so on.

(b) Define a *recursive* function taking in the cardinal number n and returning the corresponding latest value. The interface of the function will be identical to that of the previous part (the implementation will be different).

(c) Define a similar function that is more efficient. Outside the function, define a dictionary `ntoval = {0:0, 1:1}`. Inside the function, you should check to see if the n that was passed in exists as a key in `ntoval`: if it does, then simply return the corresponding value; if it doesn't, then carry out the necessary computation and augment the dictionary with a new key-value pair.

(d) If you take separation of concerns seriously, you may be feeling uncomfortable about accessing and modifying `ntoval` inside your function (since it is not being passed in as a parameter). Write a new function that looks like the one in the previous part, but takes in two parameters: `n` and `ntoval`.

(e) While part (d) respects separation of concerns, unfortunately it is not actually efficient. Write a similar function which uses a *mutable default parameter value*, i.e., it is defined by saying `def f5(n, ntoval = {0:0, 1:1}):`.

Test all five functions with $n = 8$: each of them should return 408. The functions in parts (c) and (e) should be efficient in the sense that if you now call them with, say, $n = 6$ they won't need to recompute the answer since they have already done so.

[18] Elsewhere in this book we always employ a single exit point, to make our codes easier to reason about.

1.7 This problem studies the quantity $(1 + 1/n)^n$ where $n = 10^1, 10^2, \ldots, 10^7$. Print out a nicely formatted table where the three columns are: (a) the value of n, (b) the quantity of interest computed with single-precision floating-point numbers, (c) the same quantity but now using doubles. You will need to use NumPy to get the singles to work. Keep in mind that the numbers shown in the second and third columns should be different (if they aren't, you're doing it wrong).

1.8 Investigate the relative efficiency of multiplying two one-dimensional NumPy arrays, `as` and `bs`; these should be large and with non-constant content. Do this in four distinct ways: (a) `sum(as*bs)`, (b) `np.sum(as*bs)`, (c) `np.dot(as,bs)`, and (d) `as@bs`. You may wish to use the `default_timer()` function from the `timeit` module. To produce meaningful timing results, repeat such calculations thousands of times (at least).

1.9 Take two matrices, $\mathbf{A}$ and $\mathbf{B}$, which are $n \times m'$ and $m' \times m$, respectively. Implement matrix multiplication, without relying on `numpy`'s `@` or `dot()` as applied to matrices.

(a) Write the most obvious implementation you can think of, which takes in $\mathbf{A}$ and $\mathbf{B}$ and returns a new $\mathbf{C}$. Use three loops.
(b) Write a function without the third loop by applying `@` to vectors.
(c) Write a third function that takes in $\mathbf{A}$ and $\mathbf{B}$ and returns $\mathbf{C}$, but this time these are lists-of-lists, instead of arrays.
(d) Test the above three functions by employing specific examples for $\mathbf{A}$ and $\mathbf{B}$, say 3×4 and 4×2, respectively.

1.10 Write your own functions that implement functionality similar to: (a) `np.argmax()`, (b) `np.where()`, and (c) `np.all()`, where the input will be a one-dimensional NumPy array. Note that `np.where()` is equivalent to `np.nonzero()` for this case.

1.11 Rewrite code 5.9, i.e., `action.py`, such that:

(a) The two lines involving `arr[1:-1]` now use an explicit loop and index.
(b) The calls to `np.fill_diagonal()` are replaced by explicit loop(s) and indices.

Compare the expressiveness (and total line-count) in your code vs that in `action.py`.

1.12 [$\mathcal{P}$] We now help you produce the right panel of Fig. 1.6:

(a) First try to produce the curves themselves. You'll need to appropriately place two positive and two negative charges and plot the resulting field lines. (Remember that dictionaries don't let you use duplicate keys, for good reason.)
(b) Introduce line width, by producing a list of lists containing the logarithm of the square root of the sum of the squares of the components of the electric field at that point, i.e., $\log\left[\sqrt{(E_x(\mathbf{r}))^2 + (E_y(\mathbf{r}))^2}\right]$ at each $\mathbf{r}$.
(c) Figure out how to add circles/dots (of color corresponding to the charge sign).
(d) Re-do this plot for the case of four positive (and equal) charges.

1.13 [$\mathcal{P}$] Problem 1.12 was made easier by the fact that you knew what the answer had to look like. You now need to produce a plot for the field lines (including line width) for the case of four alternating charges on a line. Place all four charges on the x axis, and

give their x_i positions the values -1, -0.5, 0.5, and 1. The leftmost charge should be positive and the next one negative, and so on (they are all of equal magnitude).

1.14 [$\mathcal{P}$] Study methane *isotherms* with the van der Waals equation of state, Eq. (5.2).

(a) Plot 40 isotherms (i.e., constant-T curves, showing P vs v), where T goes from 162 to 210 K, v goes from $1.5b$ to $9b$ and the curves look smooth.

(b) If you solved the previous part correctly, you should barely be able to tell the different curves apart. Beautify your plot by employing an automated color map.

1.15 [$\mathcal{P}$] We address the problem of *wave interference*, by repurposing `vectorfield.py`. Work in two dimensions (just like in section 1.7) and assume that the combined effect of two objects dropped in a water basin can be modelled by:

$$W(\mathbf{r}) = A\sin(k|\mathbf{r}-\mathbf{r}_0|) + A\sin(k|\mathbf{r}-\mathbf{r}_1|) \tag{1.4}$$

You can think of this as an updated version of Eq. (1.3), where we are dealing with the linear addition of sinusoidal waves, instead of electric-field contributions. For simplicity, take $A = 1$, $k = 2\pi/0.3$, $\mathbf{r}_0 = (-1\ \ 0)^T$, and $\mathbf{r}_1 = (1\ \ 0)^T$. Since we are now faced with a density plot, you should employ (not `streamplot()` but) `imshow()` from Matplotlib; in each of x and y your plot should extend from -4 to $+4$.

1.16 [$\mathcal{P}$] We will now examine the simple (yet non-linear) pendulum on the *phase plane*. The total energy of a pendulum of mass m and length l is:

$$E = \frac{1}{2}ml^2\dot{\theta}^2 + mgl(1-\cos\theta) \tag{1.5}$$

where θ is the angle from the vertical. Your task is to create a plot of $\dot{\theta}$ vs θ, where you will show the contours of constant E using Matplotlib's `contour()`; feel free to repurpose `vectorfield.py`. It's best to vary θ from -2π to $+2\pi$; you may also have to play around with the `levels` parameter; take $m = 1$ kg, $l = 1$ m, and $g = 9.8$ m/s^2. Physically interpret the different regions you encounter; roughly speaking, you should be seeing open curves[19] (above and below) and eye-shaped areas (near the middle). The boundary between the two regions is known as a *separatrix*.

1.17 [$\mathcal{P}$] We will now discuss how to visualize *binary stars*. While stars in well-detached binaries will have spherical shapes, stars in close binaries will experience tidal distortions and will have nearly ellipsoidal shapes. The *Roche model* was introduced to account for either possibility: it studies the total gravitational potential for two masses that are in circular orbit (about their barycenter). Switching to a coordinate frame that eliminates the circular orbit of the two masses (a co-rotating/"synodic" frame), the (dimensionless) Roche potential at any point (x, y, z) can be written as:

$$\Phi = \frac{m_0}{\sqrt{(x-m_1)^2+y^2+z^2}} + \frac{m_1}{\sqrt{(x+m_0)^2+y^2+z^2}} + \frac{x^2+y^2}{2} \tag{1.6}$$

The (dimensionless) masses can be written in terms of the mass ratio $q = m_0/m_1 \leq 1$ ($m_0 = q/(1+q)$ and $m_1 = 1/(1+q)$). We focus on the orbital plane ($z = 0$).

[19] The open curves have constant amplitude as θ is increased. We will see a different scenario in problem 8.43.

(a) Repurpose `vectorfield.py`, using Matplotlib's `contour()`, to visualize the curves of constant gravitational potential for the case of $q = 0.4$. Pass in an appropriate `levels` *list* to make your figure look good. (Note that a star's surface *is* an equipotential surface.) The characteristic ∞ shape you find in the middle is made up of two *Roche lobes*; a Roche lobe tells you the maximum volume that a star in a binary can occupy without losing gravitational control of its constituents.

(b) Another way you can visualize the Roche potential of Eq. (1.6) is via a surface plot. You can create such a figure via Matplotlib's `plot_surface()`. Go over its documentation, which will explain that you will first need to pass in `projection='3d'` when creating your axes. Make sure you can easily identify the different ridges and wells corresponding to the two stars as well as the area around them (e.g., via the use of `set_zlim()`).

1.18 [$\mathcal{P}$] The problem of a single particle in a one-dimensional lattice is a staple of introductory courses on solid-state physics; we will see how to tackle it for the case of a general potential in problem 4.56 (after learning about linear-algebra techniques). Here, we address a specific case, that of the *Kronig–Penney model*, involving an infinite periodic array of potential barriers (or, viewed from another perspective, of wells) of rectangular shape. Writing down the wave function within the barrier (of width $L - M$) and separately that within the well (of width M), and then imposing continuity of the wave function and of its derivative, one arrives at the condition:

$$\begin{aligned} \cos(KL) &= \cos(k_0 M)\cosh[k_1(L-M)] + \frac{k_1^2 - k_0^2}{2k_0 k_1}\sin(k_0 M)\sinh[k_1(L-M)] \\ k_0 &= \sqrt{2mE/\hbar^2}, \qquad k_1 = \sqrt{2m(V_0 - E)/\hbar^2} \end{aligned} \tag{1.7}$$

V_0 is the barrier height, E the energy, and K the wave number in *Bloch's theorem*:

$$\psi(x + L) = e^{iKL}\psi(x) \tag{1.8}$$

This wave number obeys $-\pi < KL < \pi$. Equation (1.7) implicitly provides us with the energy-dispersion relation, i.e., $E(K)$. However, we don't know how to solve complicated equations yet (we tackle this challenge in problem 5.45, after learning about root-finding). Our approach here will be to: hand-pick a value of E, compute the right-hand side of the equation, and check to see if its absolute value is larger than unity; if so, then there is no solution for K. If, however, the absolute value of the right-hand side is smaller than unity, then we can find the value of K by taking the inverse cosine and dividing by L. (Since the cosine is an even function, you need to consider both K and $-K$ as equally acceptable solutions.) Take $L = 1, M = 0.4, \hbar^2/m = 1$ and study five thousand E values from (roughly) 0 to 110. Plot E vs K for the two cases of: (a) $V_0 = 0$ and (b) $V_0 = 35$. Do yourself a favor and employ the special functions contained in the complex-number-aware module `cmath` (i.e., not in `math`). Your plot for $V_0 = 0$ should be a parabola, which corresponds to a plane-wave dispersion; of course, your results will lie in $-\pi < KL < \pi$, so you will find a "folded" version of the parabola. Your plot for $V_0 = 35$ should exhibit *band gaps*, resulting from the fact that there are energy regions for which no solution exists.