Chapter 2

Python programming for physicists

Our first item of business is to learn how to write computer programs in the Python programming language. Python is easy to learn, simple to use, and enormously powerful. It has facilities and features for performing tasks of many kinds. You can do art or engineering in Python, surf the web or calculate your taxes, write words or write music, make a movie or make the next billion-dollar Internet start-up. We will not attempt to learn about all of Python's features, however, but restrict ourselves to those that are most useful for doing physics calculations. We will learn about the core structure of the language first—how to put together the instructions that make up a program—but we will also learn about some of the powerful features that can make the life of a computational physicist easier, such as features for doing matrix calculations and features for making graphs and computer graphics. Some other features of Python that are more specialized, but still occasionally useful for physicists, will not be covered here, but there is excellent documentation available on the web, so if there is something you want to do and it is not in this book, you can probably find it online. A good place to start when looking for information about Python is the official Python website at `www.python.org`.

2.1 Getting started

A Python program consists of a list of instructions, resembling a mixture of English words and mathematics and collectively referred to as *code*. We will see exactly what form the instructions take in a moment, but first we need to know how and where to enter them into the computer.

When you are programming in Python you typically work in a *development environment*, which takes the form of a window or windows on your computer that show the program you are working on and allow you to enter or edit lines of code and run your program. There are several different development environments available for use with Python. Here we describe two of the most common ones, the basic environment called IDLE and the more sophisticated Jupyter. (If you already know you want to use Jupyter, you can skip the next section on IDLE and go straight to

Section 2.1.2 on page 11.)

IDLE and Jupyter are by no means the only development environments for Python. If you are comfortable with computers and enjoy trying things out, there are a wide range of others, mostly available for free, with names like PyCharm, Spyder, PyDev, Wing, and more. You can also use general-purpose coding environments like Visual Studio to write Python programs. Feel free to experiment and see what works for you. Either IDLE or Jupyter can do everything we will need for the material in this book, but nothing in the book will depend on what development environment you use. As far as the programming and the physics go, they are all equivalent.

2.1.1 IDLE

IDLE[1] is a simple development environment that comes with the Python language. If you have Python installed on your computer then you probably have IDLE installed as well. If not, it is available as a free download from the web. How you start IDLE depends on what kind of computer you have, but most commonly you click on an icon on the desktop or under the start menu on a PC, or in the dock or the applications folder on a Mac. If you wish, you can now start IDLE running on your computer and follow along with the developments in this section step by step.

The first thing that happens when you start IDLE is that a window appears on the computer screen. This is the *Python shell window*. It will have some text in it, looking something like this:

```
Python 3.12 (main, Feb 4 2025)
Type "help" for more information.
>>>
```

This tells you what version of Python you are running (your version may be different from the one above), along with some other information, followed by the symbol ">>>", which is a prompt: it tells you that the computer is ready for you to type something in. When you see this prompt you can type any command in the Python language at the keyboard and the computer will carry out that command immediately. This can be a useful way to quickly try out individual Python commands but it is not the main way that we will use Python. Normally, we want to type in an entire Python program at once, consisting of many commands one after another, then run the whole program together. To do this, go to the top of the window, where you will see a set of menu headings. Click on the "File" menu and select "New Window". This will create a second window on the screen, this one completely empty. This is an *editor window*. It behaves differently from the Python shell window. You type a

[1] IDLE stands for "Integrated Development Environment" (sort of). The name is also a joke, the Python language itself being named, allegedly, after the influential British comedy troupe *Monty Python*, one of whose members was the comedian Eric Idle.

complete program into this window, usually consisting of many lines. You can edit it, add things, delete things, and so forth, in a manner similar to the way one works with a word processor. The menus at the top of the window provide a range of word-processor style features, such as cut and paste, and when you are finished writing your program you can save your work just as you would with a word processor document. Then you can run your complete program, the whole thing, by clicking on the "Run" menu at the top of the editor window and selecting "Run Module" (or you can press the F5 function key, which is quicker).

To get the hang of how it works, try the following quick exercise. Open up an editor window if you didn't already (by selecting "New Window" from the "File" menu) and type the following two-line program into the window, just as it appears here:

```
x = 1
print(x)
```

(If it is not obvious what this is meant to do, it will be soon.) Now save your program by selecting "Save" from the "File" menu at the top of the editor window and typing in a name.[2] The names of all Python programs must end with ".py", so a suitable name might be "example.py" or something similar. (If you do not give your program a name ending in ".py" then the computer will not know it is a Python program and will not handle it properly when you try to load it again—you will probably find that such a program will not even run at all, so the ".py" is important.)

Once you have saved your program, run it by selecting "Run module" from the "Run" menu. When you do this the program will start running, and any output it produces—anything it says or does or prints out—will appear in the Python shell window (the other window, the one that appeared first). In this case you should see something like this in the Python shell window:

```
1
>>>
```

The only result of this small program is that the computer prints out the number "1" on the screen. (It's the value of the variable x in the program—see Section 2.2.1 below.) The number is followed by a prompt ">>>" again, which tells you that the computer is done running your program and is ready to do something else.

It is always a good idea to save your programs, as here, when they are finished

[2]Note that you can have several windows open at once, including the Python shell window and one or more editor windows, and that each window has its own "File" menu with its own "Save" item. When you click on one of these to save, IDLE saves the contents of the corresponding window and that window only. Thus if you want to save a program you must be careful to use the "File" menu for the window containing the program, rather than for any other window. If you click on the menu for the shell window, for instance, IDLE will save the contents of the shell window, not your program, which is probably not what you wanted.

and ready to run. If you forget to do it, IDLE will ask you if you want to save before it runs your program.

2.1.2 Jupyter

Jupyter is a more advanced development environment for Python programming that allows you not only to enter and run Python code but also to save the output from the code, and to interleave the code with text and graphics, providing commentary, analysis, or documentation. Figure 2.1 shows a screenshot of a Jupyter session, called a *notebook*, with the code of a program, its output, and some text all visible.

Jupyter is free and once installed it runs in your web browser, appearing in a standard window or tab within the browser. There is also a free web version of Jupyter, created by Google and called *Colab*, which provides all the benefits of Jupyter without requiring you to install anything at all—you just open the web page and start programming. You can find Colab at `https://colab.research.google.com`.

If you wish, you can start Jupyter running on your computer now and follow along with the developments in this section step by step.

When you first start up Jupyter, you will see a file browser window with a list of

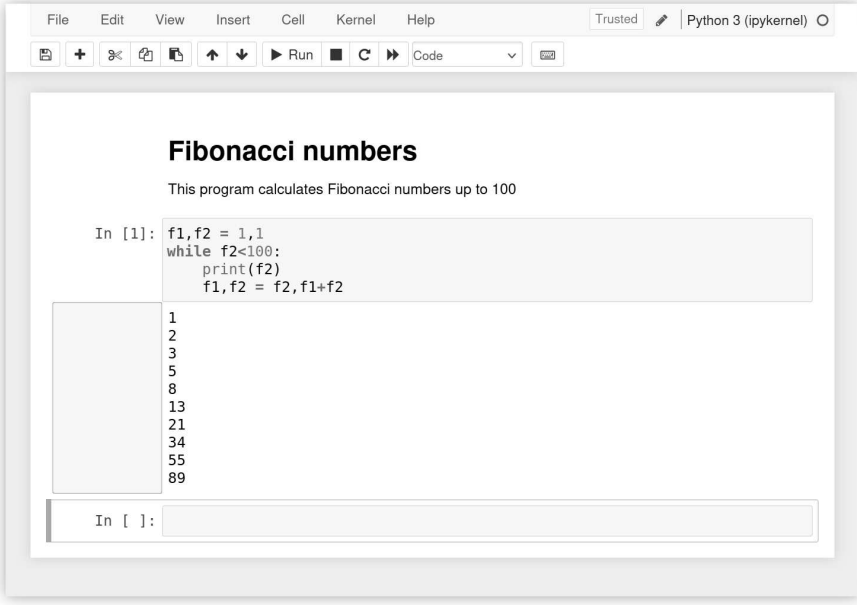


Figure 2.1: A Jupyter notebook running in a browser window. Jupyter notebooks allow you to mix text, code, and program output, as seen here, as well as graphics and images, to make a document that combines programs, documentation, results, and notes in a single place.

the files in your current folder. Over to the right you will see a menu button labeled with the word "New," and if you click there it will give you the option to start a new Python notebook (probably labeled "Python 3" if you are using the common version 3 of the Python language). Once you start a notebook you will see a new page with a box that looks like this:

```
In [ ]:
```

This is a Jupyter *cell*. You can click on it and type in Python code. For example, try clicking on the cell and entering the following short program, exactly as it appears here:

```
In [ ]: x = 1
        print(x)
```

Once you have typed in the program, you can run it by clicking the "Run" button at the top of the screen (or by pressing Shift-Enter, which is often quicker). When you do this the program will start running and any output it produces—anything it says or does or prints out—will appear immediately below it. In this case you will see something like this:

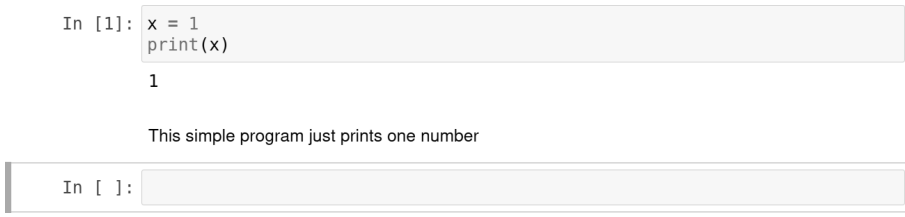
```
In [1]: x = 1
        print(x)
        1

In [ ]:
```

The only result of this small program is that the computer has printed out the number "1." (It's the value of the variable x in the program.) Now you can go back and make additions or changes to the program and run it again if you wish, editing it as many times as you like until it gives the results you want. When you are finished, or at any point along the way, you can save your entire Jupyter notebook by clicking "Save" under the "File" menu.

As mentioned above, Jupyter also allows you to add text to your notebook, documenting or commenting on your code. To do this, go to the pull-down menu that says "Code" and change it to "Markdown". This will change the current cell to a text cell, into which you can now type ordinary text—anything you like:

```
        This simple program just prints one number
```

When you "run" the cell, the text will be added to your document, so the complete notebook now looks like this:



Text entered in this way has no effect on your code—its only purpose is to provide notes and context for human readers. A text cell can also contain HTML markup language, which allows you to specify emphasis, colors, fonts, and other stylistic features, and mathematics, which is written using LaTeX mathematical notation.[3]

Jupyter allows you to have multiple cells containing code or text in the same notebook, a collection of programs together in one document. For instance, if you were working through the examples and exercises in this chapter you might want to put all your code and results in one document. The code in each code cell can be run individually by clicking on the cell and then clicking "Run."

A word of caution is in order here, however: the portions of code in different cells of a Jupyter notebook are not completely separate programs. Although each cell can be run on its own, it shares variables and functions previously defined in other cells. (If you don't know what variables and functions are we will learn about them shortly.) Thus, for example, if you execute a line "`x = 1`" that creates a variable `x` in one cell, then all other cells will henceforth also have that variable. The line "`print(x)`" in another cell would print the value "`1`" even if the variable `x` is not defined in that cell. This feature can be convenient when developing linked sets of programs, allowing one to avoid duplicating code multiple times in different programs, but it can also produce unexpected results if you are unwary. For example, you might forget that you defined a variable in one cell and use it inadvertently in another. If you want to avoid this, you can include the special command

```
%reset
```

at the start of a code cell, which causes Jupyter to erase all existing variables and functions and start from scratch whenever you run that cell. It will first ask you if you are certain you want to erase the variables and you have to say yes to proceed, which you might consider a useful failsafe or an irritating interruption depending on your point of view. If you want to skip the question you can use the command

```
%reset -f
```

which will erase all variables without any cautionary prompt.

[3] If you are not familiar with HTML or LaTeX notation there are many good introductions available online.

Finally, we should mention that some of Python's graphics features, which we will use for visualizing results of our calculations, do not work perfectly with Jupyter. In particular, for programs using the `turtle` and `qdraw` graphics packages, graphics will appear when the program is first run but will fail to appear if the program is run a second time. This is a known issue with Jupyter which, at the time of writing, has unfortunately not been fixed. A simple workaround is to restart the Python "kernel" before running the program a second time, which can be done by clicking on the "Kernel" menu at the top of the Jupyter window.

2.2 BASIC PROGRAMMING

A program is a list of instructions, or *statements*, which under normal circumstances the computer carries out, or *executes*, in the order they appear in the program. Individual statements do things like performing arithmetic, asking for input from the user of the program, or printing out results. The following sections introduce the various types of statements in the Python language one by one.

2.2.1 VARIABLES AND ASSIGNMENTS

Quantities of interest in a program—which in physics usually means numbers, or sets of numbers like vectors or matrices—are represented by *variables*, which play roughly the same role as they do in ordinary algebra. Our first example of a program statement in Python is this:

```
x = 1
```

This is an *assignment statement*. It tells the computer that there is a variable called `x` and we are assigning it the value 1. You can think of the variable as a box that stores a value for you, and you can come back and retrieve that value at any later time, or change it to a different value. We will use variables extensively in our computer programs to represent physical quantities like positions, velocities, forces, fields, voltages, probabilities, and wavefunctions.

In normal algebra variable names are usually just a single letter like `x`, but in Python (and most other programming languages) they don't have to be—they can be two, three, or more letters, or entire words. Variable names in Python can be as long as you like and can contain both letters and numbers, as well as the underscore symbol "_", but they cannot start with a number, or contain any other symbols, or spaces. Thus `x` and `Physics_101` are fine names for variables, but `4Score&7Years` is not (because it starts with a number, and also because it contains an `&`). Upper- and lower-case letters are distinct from one another, meaning that `x` and `X` are two different variables which can have different values.[4]

[4] Also variables cannot have names that are "reserved words" in Python. Reserved words are the

Many of the programs you will write will contain large numbers of variables representing the values of different things and keeping them straight in your head can be a challenge. It is a very good idea—one that is guaranteed to save you time and effort in the long run—to give your variables meaningful names that describe what they represent. If you have a variable that represents the energy of a system, for instance, you might call it `energy`. If you have a variable that represents the velocity of an object you could call it `velocity`. For more complex concepts, you can make use of the underscore symbol "_" to create variable names with more than one word, like `maximum_energy` or `angular_velocity`. There will also be times when single-letter variable names are appropriate. If you need variables to represent the x and y positions of an object, for instance, then by all means call them `x` and `y`. And there is no reason why you cannot call your velocity variable simply `v` if that seems natural to you. But whatever you do, choose names that help you remember what the variables represent.

2.2.2 Variable types

Variables come in several types. Variables of different types store different kinds of quantities. The main types we will use for our physics calculations are integer, floating-point, and complex variables.

- **Integer**: Integer variables can take integer values and integer values only, such as 1, 0, or −286784. Both positive and negative values are allowed, but not fractional values like 1.5.
- **Floating-point**: A floating-point variable, or "float" for short, can take real, or floating-point, values such as 3.14159, -6.63×10^{-34}, or 1.0. Note that a floating-point variable can take an integer value like 1.0 (which after all is also a real number), by contrast with integer variables which cannot take noninteger values.
- **Complex**: A complex variable can take a complex value, such as 1 + 2j or −3.5 − 0.4j. Notice that in Python the unit imaginary number is called j, not i. (Despite this, we will use i in some of the mathematical formulas we derive in this book, since it is the common notation among physicists. Just remember that when you translate your formulas into computer programs you must use j instead.)

You might be asking yourself what these different types mean. What does it mean that a variable has a particular type? Why do we need different types? Couldn't all values, including integers and real numbers, be represented with complex variables, so that we only need one type of variable? In principle they could, but there are significant advantages to having the different types. The values of the variables in

words used in programming statements and include "`for`", "`if`", and "`while`". (We will see the special uses of each of these words in Python programming later in the chapter.)

a program are stored by the computer in its memory, and it takes twice as much memory to store a complex number as it does a float, because the computer has to store both the real and imaginary parts. Even if the imaginary part is zero (so that the number is actually real), the computer still takes up memory space storing that zero. This may not seem like a big issue given the huge amounts of memory computers have these days, but in many physics programs we need to store enormous numbers of variables—millions or billions of them—in which case memory space can become a limiting factor.

Moreover, calculations with complex numbers take longer to complete, because the computer has to calculate both the real and imaginary parts. Again, even if the imaginary part is zero, the computer still has to do the calculation, so it takes longer either way. Many of our physics programs will involve millions or billions of operations. Big physics calculations can take days or weeks to run, so the speed of individual mathematical operations can have a big effect. Of course, if we really need to work with complex numbers then we will have to use complex variables, but if our numbers are real then it is better to use a floating-point variable.

Similar considerations apply to floating-point variables and integers. If the numbers we are working with are genuinely noninteger real numbers, then we should use floating-point variables to represent them. But if we know that the numbers are integers then using integer variables is usually faster and takes up less memory space.

Moreover, integer variables are in some cases actually more accurate than floating-point variables. As we will see in Section 4.2, floating-point calculations on computers are not infinitely accurate. Just as on a hand-held calculator, computer calculations are only accurate to a certain number of significant figures (typically about 16 on modern computers). That means that the value 1 assigned to a floating-point variable may actually be stored on the computer as 0.9999999999999999. In many cases the difference will not matter much, but what happens, for instance, if something special is supposed to take place in your program if, and only if, the number is less than 1? In that case, the difference between 1 and 0.9999999999999999 could be crucially important. Numerous bugs and problems in computer programs have arisen because of exactly this kind of issue—experiments have failed and spacecraft have crashed. Luckily there is a simple way to avoid it. If the quantity you're dealing with is genuinely an integer, then store it in an integer variable. That way you know that 1 means 1. Integer variables are not accurate to just 16 significant figures: they are perfectly accurate. They represent the exact integer you assign to them, nothing more and nothing less. If you say "`x = 1`", then indeed `x` is equal to 1.

This is an important lesson, and one that is often missed when people first start programming: if you have an integer quantity, use an integer variable. In quantum mechanics most quantum numbers are integers. The number of atoms in a gas is an integer. So is the number of planets in the solar system or the number of stars in the galaxy. Coordinates on lattices in solid-state physics are often integers. Dates

are integers. The population of the world is an integer. If you were representing any of these quantities in a program it would in most cases be best to use an integer variable. More generally, whenever you create a variable to represent a quantity in one of your programs, think about what type of value that quantity will take and choose the type of variable to match it.

How do you tell the computer what type you want a variable to be? The name of the variable is no help. A variable called `x` could be an integer or it could be a complex variable.

The type of a variable is set by the value that we give it. Thus for instance if we say "`x = 1`" then `x` will be an integer variable, because we have given it an integer value. If we say "`x = 1.5`" on the other hand then it will be a float. If we say "`x = 1+2j`" it will be complex.[5] Very large floating-point or complex values can be specified using scientific notation, in the form "`x = 1.2e34`" (which means 1.2×10^{34}) or "`x = 1e-12 + 2.3e45j`" (which means $10^{-12} + 2.3 \times 10^{45}$j).

The type of a variable can change as a Python program runs. For example, suppose we have the following two lines one after the other in our program:

```
x = 1
x = 1.5
```

If we run this program then after the first line is executed by the computer `x` will be an integer variable with value 1. But immediately after that the computer will execute the second line and `x` will become a float with value 1.5. Its type has changed from integer to float.[6]

However, although you *can* change the types of variables in this way, it doesn't mean you should. It is considered poor programming to use the same variable as two different types in a single program, because it makes the program significantly more difficult to follow and increases the chance that you may make a mistake in your programming. If `x` is an integer in some parts of the program and a float in others then it becomes difficult to remember which it is and confusion can ensue. A good programmer, therefore, will use a given variable to store only one type of quantity in a given program. If you need a variable to store another type, use a different variable with a different name. Thus, in a well written program, the type of a variable will be set the first time it is given a value and will remain the same for the rest of the program.

---

[5] Notice that when specifying complex values we say `1+2j`, not `1+2*j`. The latter means "one plus two times the variable *j*", not the complex number 1 + 2i.

[6] If you have previously programmed in one of the *static-typed* languages, such as C, C++, Fortran, or Java, then you will be used to creating variables with a declaration such as "`int i`" which means "I'm going to be using an integer variable called `i`." In such languages the types of variables are fixed once they are declared and cannot change. There is no equivalent declaration in Python. Variables in Python are created when you first use them, with types which are deduced from the values they are given and which may change when they are given new values.

This doesn't quite tell the whole story, however, because as we've said a floating-point variable can also take an integer value. There will be times when we wish to give a variable an integer value, like 1, but nonetheless have that variable be a float. There is no contradiction in this, but how do we tell the computer that this is what we want? If we simply say "`x = 1`" then, as we have seen, `x` will be an integer variable.

There are two simple ways to do what we want here. The first is to specify a value that has an explicit decimal point in it, as in "`x = 1.0`". The decimal point is a signal to the computer that this is a floating-point value (even though, mathematically speaking, 1 is of course an integer) and the computer knows in this situation to make the variable `x` a float. Thus "`x = 1.0`" specifies a floating-point variable called `x` with the value 1.

An alternative way to achieve the same thing is to write "`x = float(1)`", which tells the computer to take the value 1 and convert it into a floating-point value before assigning it to the variable `x`. This makes `x` a float.

A similar issue can arise with complex variables. There will be times when we want to create a variable of complex type, but we want to give it a purely real value. If we just say "`x = 1.5`" then `x` will be a real, floating-point variable, which is not what we want. So instead we say "`x = 1.5 + 0j`", which tells the computer that we intend `x` to be complex. Alternatively, we can write "`x = complex(1.5)`", which achieves the same thing.

There is one further type of variable, the *string*, which is often used in Python programming in general, but which comes up only rarely in physics programming, which is why we have not mentioned it so far. A string variable stores text in the form of strings of letters, punctuation, symbols, digits, and so forth. To indicate a string value one uses quotation marks, like this:[7]

```
x = "This is a string"
```

This statement would create a variable `x` of string type with the value "`This is a string`". Any character can appear in a string, including numerical digits. Thus one is allowed to say, for example, `x = "1.234"`, which creates a string variable `x` with the value "`1.234`". It is crucial to understand that this is not the same as a floating-point variable with the value 1.234. A floating-point variable contains a number, the computer knows it's a number, and, as we will shortly see, one can do arithmetic with that number or use it as the starting point for some more complicated mathematical calculation. A string variable with the value "`1.234`" does not represent a number. The value "`1.234`" is, as far as the computer is concerned, just a string of symbols in a row. The symbols happen to be digits in this case (and a decimal point) but they could just as easily be letters or spaces or punctuation. If you try to do arithmetic

[7] In Python you can use either single or double quotes to indicate a string value: `'string'` or `"string"`. We use double quotes for compatibility with other programming languages, where this is the common standard, but you will see single quotes used in many places also.

with a string variable, even one that appears to contain a number, the computer will most likely either complain or give you something entirely unexpected. We will not have much need for string variables in this book and they will as a result appear only rather rarely. One place they do appear, however, is in the following section on output and input.

In all of the programming we have seen so far you are free to put spaces between parts of a Python statement. For example, "x=1" and "x = 1" do exactly the same thing—the spaces have no effect. Spaces do, however, much improve the readability of a program. When we start writing more complicated statements in the following sections, we will find it very helpful to add some spaces here and there. There are a few places where one cannot add extra spaces, the most important being at the beginning of a line, before the start of a statement. As we will see in Section 2.3.1, inserting extra spaces at the beginning of a line does have an effect on the way a program works, so, unless you know what you are doing, you should avoid putting spaces at the beginning of lines.

You can also include blank lines between statements in a program, at any point and as many as you like. This can be useful for separating logically distinct parts of a program from one another, again making the program easier to understand. We will use this trick many times in the programs in this book to improve their readability.

2.2.3 Output and input statements

We have so far seen one example of a program statement, the assignment statement, as in "x = 1". The next types of statements we will examine are statements for output and input of data in Python programs. We have already seen an example of a basic output statement, the "print" statement. In Section 2.1 we gave this very short example program:

```
x = 1
print(x)
```

The first line of this program we understand: it creates an integer variable called x and gives it the value 1. The second statement tells the computer to "print" the value of x on the screen of the computer. Note that it is the *value* of the variable x that is printed, not the letter "x". The value of the variable in this case is 1, so this short program will result in the computer printing a "1" on the screen, as we saw on page 10.

The print statement always prints the current value of the variable at the moment the statement is executed. Thus consider this program:

```
x = 1
print(x)
x = 2
print(x)
```

First the variable x is set to 1 and its value is printed out, resulting in a 1 on the screen as before. Then the value of x is changed to 2 and the value is printed again, which produces a 2 on the screen. Overall we get this:

```
1
2
```

Thus the two print statements, although they look identical, produce different results in this case. Note also that each print statement starts its printing on a new line on the screen.

The print statement can be used to print out more than one thing on a line. Consider this program:

```
x = 1
y = 2
print(x,y)
```

which produces this result:

```
1 2
```

Note now the two variables in the print statement are separated by a comma. When their values are printed out, however, they are printed with a space between them (not a comma).

We can also print out words, like this:

```
x = 1
y = 2
print("The value of x is",x,"and the value of y is",y)
```

which produces this on the screen:

```
The value of x is 1 and the value of y is 2
```

Adding a few words to your program like this can make its output easier to read and understand. You can also have print statements that print out only words, as in `print("The results are as follows")` or `print("End of program")`.

The print statement can also print out the values of floating-point and complex variables. For instance, we can write

```
x = 1.5
z = 2+3j
print(x,z)
```

and we get

```
1.5 (2+3j)
```

In general, a print statement can include any series of items separated by commas, including variables or text in quotation marks, and the computer will print out the appropriate things in order, with spaces in between.[8] Occasionally you may want to print things with something other than spaces in between, in which case you can write the following:

```
print(x,z,sep="...")
```

which would print

```
1.5...(2+3j)
```

The code `sep="..."` tells the computer to use whatever appears between the quotation marks as a separator between values—three dots in this case, but you could use any letters, numbers, or symbols you like. You can also have no separator between values at all by writing `print(x,z,sep="")` with nothing between the quotation marks, which in the present case would give

```
1.5(2+3j)
```

You can also use the print statement without any items at all between the parentheses, as in "`print()`". (Note that the parentheses are still required, even though they empty.) This statement just prints a blank line, which can sometimes be useful for making the output of your program more readable. For instance, if you are printing a large number of results at once it can be helpful to break them into blocks with blank lines.

Input statements are only a little more complicated. The basic form of an input statement in Python is like this:

```
x = input("Enter the value of x: ")
```

When the computer executes this statement it does two things. First, the statement acts something like a print statement and prints out the quantity, if any, inside the parentheses.[9] So in this case the computer would print the words "`Enter the value`

[8] The print statement differs between Python version 3 and earlier versions. In earlier versions there were no parentheses around the items to be printed—you would just write "`print x`". If you are using an earlier version of Python with this book then you will have to remember to omit the parentheses from your print statements. Alternatively, if you are using version 2.6 or later (but not version 3) then you can make the print statement behave as it does in version 3 by including the statement `from __future__ import print_function` at the start of your program. (Note that there are two underscore symbols before the word "`future`" and two after it.) See Appendix D for further discussion of the differences between Python versions.

[9] It doesn't act exactly like a print statement however, since it can only print a single quantity, such as a string of text in quotes (as here) or a variable, where the print statement can print many quantities in a row.

`of x: `". If there is nothing inside the parentheses, as in "`x = input()`", then the computer prints nothing, but the parentheses are still required nonetheless.

Next the computer will stop and wait for the user to type a value on the keyboard. It will wait patiently until the user types something and then the value that the user types is assigned to the variable `x`. However, there is a catch: the value entered is always interpreted as a *string* value, even if you type a number.[10] (We encountered strings previously in Section 2.2.2.) Thus consider this simple two-line program:

```
x = input("Enter the value of x: ")
print("The value of x is",x)
```

This does nothing more than collect a value from the user then print it out again. If we run this program it might look something like the following:

```
Enter the value of x: 1.5
The value of x is 1.5
```

This looks reasonable. But we could also do the following:

```
Enter the value of x: Hello
The value of x is Hello
```

As you can see "value" is interpreted rather loosely. As far as the computer is concerned, anything you type in is a string, so it doesn't care whether you enter digits, letters, a complete word, or several words. Anything is fine.

For physics calculations, however, we usually want to enter numbers, and have them interpreted correctly as numbers, not strings. Luckily it is straightforward to convert a string into a number. The following will do it:

```
temp = input("Enter the value of x: ")
x = float(temp)
print("The value of x is",x)
```

This program receives a string input from the user and assigns it to the temporary variable `temp`, which will be a string-type variable. Then the statement "`x = float(temp)`" converts the string value to a floating-point value, which is then as-

[10] Input statements are another thing that changed between versions 2 and 3 of Python. In version 2 and earlier, the value generated by an input statement would have the same type as whatever the user entered. If the user entered an integer, the input statement would give an integer value. If the user entered a float it would give a float, and so forth. However, this was considered confusing, because it meant that if you then assigned that value to a variable (as in the program above) there would be no way to know in advance what the type of the variable would be—the type would depend on what the user entered at the keyboard. So in version 3 of Python the behavior was changed to its present form in which the input is always interpreted as a string. If you are using a version of Python earlier than version 3 and you want to reproduce the behavior of version 3 then you can write "`x = raw_input()`". The function `raw_input` in earlier versions is the equivalent of `input` in version 3.

signed to the variable `x`, and this is the value that is printed out. One can also convert string values to integers or complex numbers with statements of the form "`x = int(temp)`" or "`x = complex(temp)`".

In fact, one does not have to use a temporary variable. The code above can be expressed more succinctly like this:

```
x = float(input("Enter the value of x: "))
print("The value of x is",x)
```

which takes the string value given by `input`, converts it to a float, and assigns it directly to the variable `x`. We will use this trick many times in this book.

In order for the program above to work, the value the user types must be one that makes sense as a floating-point value, otherwise the computer will complain. Thus, for instance, the following is fine:

```
Enter the value of x: 1.5
The value of x is 1.5
```

But suppose we do this:

```
Enter the value of x: Hello
ValueError: invalid literal for float(): Hello
```

This is our first example of an *error message*. The computer, in opaque technical language, is complaining that we have given it an incorrect value.

It is normal to make a few mistakes when writing or using computer programs, and you will soon become accustomed to the occasional error message (if you are not already). Working out what these messages mean is one of the tricks of the trade—they are often not entirely transparent.

2.2.4 Arithmetic

So far our programs have done very little, certainly nothing that would be of much use for physics. But we can make them much more useful by adding some arithmetic into the mix.

In most places where you can use a single variable in Python you can also use a mathematical expression, like "`x+y`". Thus you can write "`print(x)`" but you can also write "`print(x+y)`" and the computer will calculate the sum of `x` and `y` for you and print out the result. The basic mathematical operations—addition, subtraction, etc.—are written as follows:

`x+y`	addition
`x-y`	subtraction
`x*y`	multiplication
`x/y`	division
`x**y`	raising x to the power of y

Notice that we use the asterisk symbol "`*`" for multiplication and the slash symbol "`/`" for division, because there is no × or ÷ symbol on a standard computer keyboard.

Two more obscure, but still useful operations, are integer division and the modulo operation:

`x//y` the integer part of `x` divided by `y`, meaning `x` is divided by `y` and the result is rounded down to the nearest integer. For instance, `14//3` gives 4 and `-14//3` gives −5.

`x%y` `x` modulo `y`, which means the remainder after `x` is divided by `y`. For instance, `14%3` gives 2, because 14 divided by 3 gives 4-remainder-2. This also works for non-integers: `1.5%0.4` gives 0.3, because 1.5 is 3 × 0.4, remainder 0.3. (There is, however, no modulo operation for complex numbers.) The modulo operation is particularly useful for telling when one number is divisible by another—the value of `n%m` will be zero if `n` is divisible by `m`. Thus, for instance, `n%2` is zero if `n` is even (and one if `n` is odd).

There are also a handful of other mathematical operations available in Python, but they are more obscure and rarely used.[11]

An important rule about arithmetic in Python is that the type of result a calculation gives depends on the type of values that go into it. Consider, for example, this statement

```
x = a + b
```

If `a` and `b` are variables of the same type—integer, float, complex—then when they are added together the result will also have the same type and this will be the type of variable `x`. So if `a` is 1.5 and `b` is 2.4, meaning that they are both floats, then `x` will be a float with value 3.9. Note when adding floats like this that even if the end result of the calculation is a whole number, the variable `x` will still be floating point: if `a` is 1.5 and `b` is 2.5, then the result of adding them together is 4, but `x` will still be a floating-point variable with value 4.0 because `a` and `b` are floating point.

If `a` and `b` are of different types, then the end result has the more general of the two types that went into it. This means that if you add a float and an integer, for example, the end result will be a float. If you add a float and a complex number, the end result will be complex.

The same rules apply to subtraction, multiplication, integer division, and the

[11] Such as:

`x|y` bitwise (binary) OR of two integers
`x&y` bitwise (binary) AND of two integers
`x^y` bitwise (binary) XOR of two integers
`x>>y` shift the bits of integer `x` rightwards `y` places
`x<<y` shift the bits of integer `x` leftwards `y` places

modulo operation: the type of the end result is the same as the starting types, or the more general type if there are two different starting types. The division operation, however—ordinary non-integer division denoted by "`/`"—is different. It follows basically the same rules except that it never gives an integer result. Division only ever gives floating-point or complex values. This is necessary because you can divide one integer by another and get a non-integer result (such as $3 \div 2 = 1.5$ for example), so it would not make sense to have integer starting values always give an integer final result.[12] Thus if you divide any combination of integers or floats by one another you will always get a floating-point value. If you start with one or more complex numbers then you will get a complex value at the end.

You can combine several mathematical operations together to make a more complicated expression, like `x+2*y-z/3`. When you do this the operations obey rules similar to those of normal algebra. Multiplications and divisions are performed before additions and subtractions. If there are several multiplications or divisions in a row they are carried out in order from left to right. Powers are calculated before anything else. Thus

`x+2*y`	is equivalent to	$x + 2y$
`x-y/2`	is equivalent to	$x - \frac{1}{2}y$
`3*x**2`	is equivalent to	$3x^2$
`x/2*y`	is equivalent to	$\frac{1}{2}xy$

You can also use parentheses `()` in your algebraic expressions, just as you would in normal algebra, to mark things that should be evaluated as a unit, as in `2*(x+y)`. However, only round parentheses `()` can be used for this purpose, not square brackets `[]` or braces `{}`. Parentheses within parentheses are fine, as in `x = 2*(x+3*(y-z))`.

You can also add spaces between the parts of a mathematical expression to make it easier to read. The spaces do not affect the value of the expression. So "`x=2*(a+b)`" and "`x = 2 * ( a + b )`" do the same thing. Thus the following are allowed statements in Python

```
x = a + b/c
x = (a + b)/c
x = a + 2*b - 0.5*(1.618**c + 2/7)
```

[12] This is another respect in which version 3 of Python differs from earlier versions. In version 2 and earlier all operations gave results of the same type that went into them, including division. This, however, caused confusion for exactly the reason given here: if you divided 3 by 2, for instance, the result had to be an integer, so the computer rounded it down from 1.5 to 1. Because of the difficulties this caused, the language was changed in version 3 to give the current more sensible behavior. You can still get the old behavior of dividing then rounding down using the integer divide operation `//`. Thus `3//2` gives 1 in all versions of Python. If you are using Python version 2 (technically, version 2.1 or later) and want the newer behavior of the divide operation, you can achieve it by including the statement "`from __future__ import division`" at the start of your program. The differences between Python versions are discussed in more detail in Appendix D.

On the other hand, the following will *not* work:

```
2*x = y
```

You might expect that this would result in the value of x being set to half the value of y, but it's not so. In fact, if you write this line in a program the computer will stop when it gets to it and print the cryptic error message "`SyntaxError: can't assign to operator`" because it doesn't know what to do. The problem is that Python does not know how to solve equations for you by rearranging them. It only knows about the simplest forms of equations, such as "`x = y/2`". If an equation needs to be rearranged to give the value of x then you have to do the rearranging for yourself. Python will do basic sums for you, but its knowledge of math is very limited.

To be more precise, statements like "`x = a + b/c`" in Python are not technically equations at all, in the mathematical sense. They are assignments. When it sees a statement like this, what your computer actually does is very simple-minded. It first examines the right-hand side of the equals sign and evaluates whatever expression it finds there, using the current values of any variables involved. When it is finished working out the value of the whole expression, and only then, it takes that value and assigns it to the variable on the left of the equals sign. In practice, this means that assignment statements in Python sometimes behave like ordinary equations, but sometimes they don't. A simple statement like "`x = 1`" does exactly what you would think, but what about this statement:

```
x = x + 1
```

This does not make sense, under any circumstances, as a mathematical equation. There is no way that x can ever be equal to $x + 1$. It would imply that $0 = 1$. But this statement makes perfect sense in Python. Suppose the value of x is currently 1. When the statement above is executed by the computer it first evaluates the expression on the right-hand side, which is `x + 1` and therefore has the value $1 + 1 = 2$. Then it assigns this value to the variable on the left-hand side, which just happens in this case to be the same variable x. So x now gets a new value 2. In fact, no matter what value of x we start with, this statement will always end up giving x a new value that is 1 greater. So this statement has the simple (but potentially very useful) effect of increasing the value of x by one.

Thus consider the following lines:

```
x = 3
print(x)
x = x**2 - 2
print(x)
```

What will happen when the computer executes these lines? The first two are straightforward enough: the variable x gets the value 3 and then the 3 gets printed out. But

then what? The third line says "`x = x**2 - 2`" which in normal mathematical notation would be $x = x^2 - 2$, which is a quadratic equation with solutions $x = 2$ and $x = -1$. However, the computer will not set `x` equal to either of these values. Instead it will evaluate the right-hand side of the equals sign and get $x^2 - 2 = 3^2 - 2 = 7$ and then set `x` to this new value. Then the last line of the program will print out "7".

Thus the computer does not necessarily do what one might think it would, based on one's experience with normal mathematics. The computer will not solve equations for x or any other variable. It will not do your algebra for you—it's not that smart.

Another set of useful tricks are the Python *modifiers*, which allow you to make changes to a variable as follows:

`x += 1`	add 1 to `x` (i.e., make `x` bigger by 1)
`x -= 4`	subtract 4 from `x`
`x *= -2.6`	multiply `x` by −2.6
`x /= 5*y`	divide `x` by 5 times `y`
`x //= 3.4`	divide `x` by 3.4 and round down to an integer

As we have seen, you can achieve the same result as these modifiers with statements like "`x = x + 1`", but the modifiers are more succinct. Some people also prefer them precisely because "`x = x + 1`" looks like bad algebra and can be confusing.

Finally in this section, a nice feature of Python, not available in most other computer languages, is the ability to assign the values of two variables with a single statement. For instance, we can write

```
x,y = 1,2.5
```

which is equivalent to the two statements

```
x = 1
y = 2.5
```

One can assign three or more variables in the same way, listing them and their assigned values with commas in between.

A more sophisticated example is

```
x,y = 2*z+1,(x+y)/3
```

An important point to appreciate is that, like all other assignment statements, this one evaluates the whole of the right-hand side of the equals sign before assigning values to the variables on the left. Thus in this example the computer will calculate both of the values `2*z+1` and `(x+y)/3` from the current `x`, `y`, and `z`, before assigning those values to `x` and `y`.

One purpose for which this type of multiple assignment is commonly used is to interchange the values of two variables. If we want to swap the values of `x` and `y` we can write:

```
x,y = y,x
```

and the two will be exchanged. In most other computer languages such swaps are more complicated, requiring the use of an additional temporary variable.

EXAMPLE 2.1: A BALL DROPPED FROM A TOWER

Let us use what we have learned to solve a first physics problem. This is a very simple problem, one we could easily do on paper, but we will move onto more complex ones shortly.

The problem is as follows. A ball is dropped from the top of a tower of height h. It has initial velocity zero and accelerates downward under gravity. The challenge is to write a program that asks the user to enter the height in meters of the tower and a time interval t in seconds, then prints on the screen the height of the ball above the ground at time t after it is dropped, ignoring air resistance.

The steps involved are the following. First, we use `input` statements to get the values of h and t from the user. Second, we calculate how far the ball falls in the given time, using the standard kinematic formula $s = \frac{1}{2}gt^2$, where $g = 9.81\,\mathrm{ms}^{-2}$ is the acceleration due to gravity. Third, we print the height above the ground at time t, which is equal to the total height of the tower minus this value, or $h - s$.

Here is what the program looks like, all four lines of it:[13]

File: `dropped.py`

```
h = float(input("Enter the height of the tower: "))
t = float(input("Enter the time interval: "))
s = 9.81*t**2/2
print("The height of the ball is",h-s,"meters")
```

Let us use this program to calculate the height of a ball dropped from a 100 m high tower after 1 second and after 5 seconds. Running the program twice in succession we find the following:

```
Enter the height of the tower: 100
Enter the time interval: 1
The height of the ball is 95.095 meters

Enter the height of the tower: 100
Enter the time interval: 5
The height of the ball is -22.625 meters
```

[13] Many of the example programs in this book are also available online for you to download. The programs, along with various other useful resources, are packaged together in a single zip file which can be downloaded from `https://www.umich.edu/~mejn/cpresources.zip`. Throughout the book, a name printed in the margin next to a program, such as "`dropped.py`" above, indicates that the complete program can be found, under that name, in these online resources. Any mention of programs or data in the "online resources" also refers to the same download.

Note that the result is negative in the second case, which means that the ball would have fallen below ground level if that were possible, although in practice the ball would hit the ground first. Thus a negative value indicates that the ball hits the ground before time t.

Before we leave this example, here is a suggestion for a possible improvement to the program. At present we perform the calculation of the distance traveled with the single line "`s = 9.81*t**2/2`", which includes the constant 9.81 representing the acceleration due to gravity. When we do physics calculations on paper, however, we normally do not write out the values of constants in full like this. Normally we would write $s = \frac{1}{2}gt^2$, with the understanding that g represents the acceleration. We do this primarily because it is easier to read and understand. A single symbol g is easier to read than a row of digits, and moreover the use of the standard letter g reminds us that the quantity we are talking about is the gravitational acceleration, rather than some other constant that happens to have value 9.81. Especially in the case of constants that have many digits, such as $\pi = 3.14159265\ldots$, the use of symbols rather than digits makes life a lot easier.

The same is also true of computer programs. You can make your programs easier to read and understand by using symbols for constants instead of writing the values out in full. This is easy to do—just create a variable to represent the constant, like this:

```
g = 9.81
s = g*t**2/2
```

You only have to create the variable `g` once in your program (usually somewhere near the beginning) and then you can use it as many times as you like thereafter.[14] Doing this also has the advantage of decreasing the chances that you will make a typographical error in the value of a constant. If you have to type out many digits every time you need a particular constant, odds are you are going to make a mistake at some point. If you have a variable representing the constant then you know the value will be right every time you use it, as long as you typed it correctly when you first created the variable.[15]

Using variables to represent constants in this way is one example of a programming trick that improves your programs even though it does not change the way they actually work. Instead it improves readability and reliability, which can be al-

[14] In some computer languages, such as C, there are separate entities called "variables" and "constants," a constant being like a variable except that its value can be set only once in a program and is fixed thereafter. There is no such thing in Python, however; there are only variables.

[15] There exists a Python module called `scipy.constants`, part of the larger `scipy` package, that defines values for a wide range of physical constants, so that you don't have to. It includes the acceleration due to gravity g, as well as the electronic charge, Planck's constant, the speed of light, and many more. We will not use this package in our programs in this book, but it may be worth a look if you use such constants often. We discuss the use of Python packages in Section 2.2.5.

most as important as writing a correct program. We will see other examples of such tricks later.

Exercise 2.1: Another ball dropped from a tower

A ball is again dropped from a tower of height h with initial velocity zero. Write a program that asks the user to enter the height in meters of the tower and then calculates and prints the time the ball takes until it hits the ground, ignoring air resistance. Use your program to calculate the time for a ball dropped from a 100 m high tower.

Exercise 2.2: Altitude of a satellite

A satellite is to be launched into a circular orbit around the Earth so that it orbits the planet once every T seconds.

a) Show that the altitude h above the Earth's surface that the satellite must have is

$$h = \left(\frac{GMT^2}{4\pi^2}\right)^{1/3} - R,$$

where $G = 6.67 \times 10^{-11}\,\mathrm{m^3\,kg^{-1}\,s^{-2}}$ is Newton's gravitational constant, $M = 5.97 \times 10^{24}$ kg is the mass of the Earth, and $R = 6371$ km is its radius.

b) Write a program that asks the user to enter the desired value of T and then calculates and prints out the correct altitude in meters.

c) Use your program to calculate the altitudes of satellites that orbit the Earth once a day (so-called geosynchronous orbit), once every 90 minutes, and once every 45 minutes. What do you conclude from the last of these calculations?

d) Technically a geosynchronous satellite is one that orbits the Earth once per *sidereal day*, which is 23.93 hours, not 24 hours. Why is this? And how much difference will it make to the altitude of the satellite?

2.2.5 FUNCTIONS, PACKAGES, AND MODULES

There are many operations one might want to perform in a program that are more complicated than simple arithmetic, such as multiplying matrices, calculating a logarithm, or making a graph. Python comes with facilities for doing each of these and many other common tasks easily and quickly. These facilities are divided into *packages*—collections of related useful things—and each package has a name by which you can refer to it. For instance, all of the standard mathematical functions, such as logarithm and square root, are contained in a package called `math`. Before you can use any of these functions you have to tell the computer that you want to. For example, to tell the computer you want to use the log function, you would add the following line to your program:

```
from math import log
```

This tells the computer to "import" the logarithm function from the `math` package, which means that it copies the code defining the function from where it is stored (usually on the hard drive of your computer) into the computer's memory, ready for use by your program. You need to import each function you use only once per program: once the function has been imported it continues to be available until the program ends.[16] You must import the function before the first time you use it in a calculation and it is good practice to put the "from" statement at the very start of the program, which guarantees that it occurs before the first use of the function and also makes it easy to find when you are working on your code. As we write more complicated programs, there will often be situations where we need to import many different functions into a single program with many different from statements, and keeping those statements together in a tidy block at the start of the code will make things much easier.

Once you have imported the log function you can then use it in a calculation like this:

```
x = log(2.5)
```

which will calculate the (natural) logarithm of 2.5 and set the variable `x` equal to the result. Note that the argument of the logarithm, the number 2.5 in this case, goes in parentheses. If you omit the parentheses the computer will complain. (Also if you use the log function without first importing it from the `math` package the computer will complain.)

The `math` package contains a good selection of the most commonly used mathematical functions, including the following:

`log`	natural logarithm
`log10`	log base 10
`exp`	exponential
`sin`, `cos`, `tan`	sine, cosine, tangent (argument in radians)
`asin`, `acos`, `atan`	arcsine, arccosine, arctangent (in radians)
`sinh`, `cosh`, `tanh`	hyperbolic sine, cosine, tangent
`sqrt`	positive square root

Note that the trigonometric functions work with angles specified in radians, not degrees. The exponential and square root functions may seem redundant, since one can calculate both exponentials and square roots by taking powers. For instance, `x**0.5` would give the square root of `x`. Because of the way the computer calculates powers and roots, however, using the functions above is usually quicker and more accurate.

[16] If you are programming in a Jupyter notebook or in Colab, functions need be imported only once per notebook: functions imported by one code cell are automatically available to all other cells.

The `math` package also contains a number of less common functions, such as the Gaussian error function and the gamma function, as well as two objects that are not functions at all but constants, namely e and π, which are denoted `e` and `pi`. This program, for instance, calculates the value of π^2:

```
from math import pi
print(pi**2)
```

which prints `9.869604401089358` (which is roughly the right answer). Note that there are no parentheses after the "`pi`" when we use it in the print statement, because it is not a function. It is just a variable called `pi` with value 3.14159...

The functions in the `math` package do not work with complex numbers and the computer will give an error message if you try, but there is another package called `cmath` that contains versions of most of the same functions that do work with complex numbers, plus a few additional functions that are specific to complex arithmetic.

In some cases you may find you want to use more than one function from the same package in a program. You can import two different functions—say the log and exponential functions—with two statements like this:

```
from math import log
from math import exp
```

but a more succinct way to do it is to use a single statement like this:

```
from math import log,exp
```

You can import a list as long as you like from a single package in this way:

```
from math import log,exp,sin,cos,sqrt,pi,e
```

You can also import *all* of the functions in a package with a statement of the form

```
from math import *
```

The `*` here means "everything". In most cases, however, we advise against using this import-everything form because it can give rise to some unexpected behaviors (for instance, if, unbeknownst to you, a package contains a function with the same name as one of your variables, causing a clash between the two). It is usually better to explicitly import only those functions you actually need to use.[17]

[17] A particular problem occurs when an imported package contains a function with the same name as a previously existing function. In such a case the newly imported one will supersede the previous one, which may not always be what you want. For instance, the packages `math` and `cmath` contain many functions with the same names, such as `sqrt`. But the `sqrt` function in `cmath` works with complex numbers and the one in `math` does not. If one did "`from cmath import *`" followed by "`from math import *`", one would end up with the version of `sqrt` that works only with real numbers. And if one then attempted to calculate the square root of a complex number, one would get an error message.

There is, however, another way to import the entire contents of a package in Python which avoids these pitfalls and can sometimes be useful. The statement

```
import math
```

imports the entire `math` package in one step. Subsequently, if we want to use, say, the logarithm function, we write

```
x = math.log(2.5)
```

Note how we now specify the name of the imported package "`math`", followed by a period, followed by the function name. If we wanted to take a square root we would say

```
x = math.sqrt(2.5)
```

and so forth. Thus this form simplifies the *importing* of functions from packages, at the expense of making the *use* of those functions a bit more complicated. In some cases this is a worthwhile tradeoff, and we will use it occasionally in this book.

A variant of the same trick, which can simplify life sometimes, is the statement

```
import math as mt
```

This defines "`mt`" as an alias for the `math` package, so that we can use "`mt`" anywhere we would previously have used "`math`", as in `x = mt.log(2.5)` or `x = mt.sqrt(2.5)`. In this case, doing so would not actually save us much effort—we have to type two letters "`mt`" instead of four—but some packages have much longer names, in which case this can be a useful trick.

Finally in this section, some large packages are for convenience split into smaller subpackages, called *modules*. A module within a larger package is referred to as `packagename.modulename`. For example, as we will see shortly, there are a large number of useful mathematical facilities available in the package called `numpy`, including facilities for linear algebra and Fourier transforms, each in their own module within the larger package. Thus the linear algebra module is called `numpy.linalg` and the Fourier transform module is called `numpy.fft` (for "fast Fourier transform"). We import a function from a module thus:

```
from numpy.linalg import inv
```

This would import the `inv` function, which calculates the inverse of a matrix. Alternatively we could import the entire linear algebra module thus:

```
import numpy.linalg as la
```

Then we would refer to the matrix inversion function as `la.inv`.

Smaller packages, like the `math` package, have no submodules, in which case one could, arguably, say that the entire package is also a module, and in such cases the words package and module are often used interchangeably.

EXAMPLE 2.2: CONVERTING POLAR COORDINATES

Suppose the position of a point in two-dimensional space is given to us in polar coordinates r, θ and we want to convert it to Cartesian coordinates x, y. How would we write a program to do this? The appropriate steps are:

1. Get the user to enter the values of r and θ.
2. Convert those values to Cartesian coordinates using the standard formulas

$$x = r\cos\theta, \qquad y = r\sin\theta.$$

3. Print out the results.

Since the formulas involve the mathematical functions sin and cos, we are going to have to import those functions from the `math` package. Also, the sine and cosine functions in Python (and in most other computer languages) take arguments in radians. If we want to be able to enter the angle θ in degrees then we are going to have to convert from degrees to radians, which means multiplying by π and dividing by 180.

Thus our program might look something like this:

File: `polar.py`

```
from math import sin,cos,pi

r = float(input("Enter r: "))
d = float(input("Enter theta in degrees: "))

theta = d*pi/180
x = r*cos(theta)
y = r*sin(theta)

print("x =",x," y =",y)
```

Take a moment to read through this complete program and make sure you understand what each line is doing. If we run the program, it will do something like the following:

```
Enter r: 2
Enter theta in degrees: 60
x = 1.0  y = 1.73205080757
```

Try it for yourself.

2.2.6 Other packages

There are an extraordinary number of packages available in Python for performing almost any computational task imaginable. There are packages for making graphics and playing sounds, packages for data handling and statistics, packages for linear algebra and calculus and trigonometry. Here are a few packages that are useful for computational physics:

`math`	Basic mathematical functions for real numbers
`cmath`	Basic mathematical functions for complex numbers
`numpy`	Arrays, vectors, and matrices
`scipy`	Basic scientific tools, like special functions and statistics
`matplotlib`	Graph drawing
`collections`	Data structures, such as queues and default-dictionaries
`csv`	Reading and writing data files in the common CSV format
`pandas`	Spreadsheet-style data processing

We will not go into these packages in detail here, but we will introduce them as needed in the following chapters.

2.2.7 Built-in functions

There are a small number of functions in Python, called *built-in functions*, which do not come from any package. These functions are always available to you in every program; you do not have to import them. We have in fact seen several examples of built-in functions already. For instance, we saw the `float` function, which takes a number and converts it to floating point (if it is not floating point already):

```
x = float(1)
```

There are similar functions `int` and `complex` that convert to integers and complex numbers. Another example of a built-in function, one we have not seen previously, is the `abs` function, which returns the absolute value of a number, or the modulus in the case of a complex number. Thus, `abs(-2)` returns the integer value 2 and `abs(3+4j)` returns the floating-point value 5.0.

Earlier we also used the built-in functions `input` and `print`, which are not mathematical functions in the usual sense of taking a number as argument and performing a calculation on it, but as far as the computer is concerned they are still functions. Consider, for instance, the statement

```
x = input("Enter the value of x: ")
```

Here the `input` function takes as argument the string "`Enter the value of x: `", prints it out, waits for the user to type something in response, then sets `x` equal to that something.

The print function is slightly different. When we say

```
print(x)
```

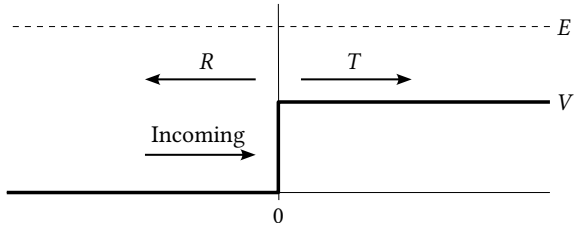
print is a function, but it is not here generating a value the way the log or input functions do. It does something with its argument x, namely printing it out on the screen, but it does not generate a value. This differs from the functions we are used to in mathematics, which always generate a value, but it is nonetheless allowed in Python. Sometimes you just want a function to do something but it doesn't need to generate a value.

Exercise 2.3: Write a program to perform the inverse operation to that of Example 2.2. That is, ask the user for the Cartesian coordinates x, y of a point in two-dimensional space, then calculate and print the corresponding polar coordinates, with the angle θ in degrees.

Exercise 2.4: A spaceship travels from Earth in a straight line at relativistic speed v to another planet x light years away. Write a program to ask the user for the value of x and the speed v as a fraction of the speed of light c, then print out the time in years that the spaceship takes to reach its destination (a) in the rest frame of an observer on Earth and (b) as perceived by a passenger on board the ship. Use your program to calculate the answers for a planet 10 light years away with $v = 0.99c$.

Exercise 2.5: Quantum potential step

A well-known quantum mechanics problem involves a particle of mass m that encounters a one-dimensional potential step, like this:



The particle with initial kinetic energy E and wavevector $k_1 = \sqrt{2mE}/\hbar$ enters from the left and encounters a sudden jump in potential energy of height V at position $x = 0$. By solving the Schrödinger equation, one can show that when $E > V$ the particle may either (a) pass the step, in which case it has a lower kinetic energy of $E - V$ on the other side and a correspondingly smaller wavevector of $k_2 = \sqrt{2m(E-V)}/\hbar$, or (b) it may be reflected, keeping all of its kinetic energy and an unchanged wavevector but moving in the opposite direction. The probabilities T and R for transmission and reflection are given by

$$T = \frac{4k_1k_2}{(k_1+k_2)^2}, \qquad R = \left(\frac{k_1-k_2}{k_1+k_2}\right)^2.$$

Suppose we have a particle with mass equal to the electron mass $m = 9.11 \times 10^{-31}$ kg and energy 10 eV encountering a potential step of height 9 eV. Write a Python program to compute and print out the transmission and reflection probabilities using the formulas above.

Exercise 2.6: Planetary orbits

The orbit in space of one body around another, such as a planet around the Sun, need not be circular. In general it takes the form of an ellipse, with the body sometimes closer in and sometimes further out. If you are given the distance ℓ_1 of closest approach that a planet makes to the Sun, also called its *perihelion*, and its linear velocity v_1 at perihelion, then any other property of the orbit can be calculated as follows.

a) Kepler's second law tells us that the distance ℓ_2 and velocity v_2 of the planet at its most distant point, or *aphelion*, satisfy $\ell_2 v_2 = \ell_1 v_1$. At the same time the total energy, kinetic plus gravitational, of a planet with velocity v and distance r from the Sun is given by

$$E = \tfrac{1}{2}mv^2 - G\frac{mM}{r},$$

where m is the planet's mass, $M = 1.9891 \times 10^{30}$ kg is the mass of the Sun, and $G = 6.6738 \times 10^{-11}$ m^3 kg^{-1} s^{-2} is Newton's gravitational constant. Given that energy must be conserved, show that v_2 is the smaller root of the quadratic equation

$$v_2^2 - \frac{2GM}{v_1\ell_1}v_2 - \left[v_1^2 - \frac{2GM}{\ell_1}\right] = 0.$$

Once we have v_2 we can calculate ℓ_2 using the relation $\ell_2 = \ell_1 v_1 / v_2$.

b) Given the values of v_1, ℓ_1, and ℓ_2, other parameters of the orbit are given by simple formulas that can be derived from Kepler's laws and the fact that the orbit is an ellipse:

$$\begin{aligned}
\text{Semi-major axis:} \quad & a = \tfrac{1}{2}(\ell_1 + \ell_2), \\
\text{Semi-minor axis:} \quad & b = \sqrt{\ell_1 \ell_2}\,, \\
\text{Orbital period:} \quad & T = \frac{2\pi ab}{\ell_1 v_1}\,, \\
\text{Orbital eccentricity:} \quad & e = \frac{\ell_2 - \ell_1}{\ell_2 + \ell_1}.
\end{aligned}$$

Write a program that asks the user to enter the distance to the Sun and velocity at perihelion, then calculates and prints the quantities ℓ_2, v_2, T, and e.

c) Test your program by having it calculate the properties of the orbits of the Earth (for which $\ell_1 = 1.4710 \times 10^{11}$ m and $v_1 = 3.0287 \times 10^4$ m s^{-1}) and Halley's comet ($\ell_1 = 8.7830 \times 10^{10}$ m and $v_1 = 5.4529 \times 10^4$ m s^{-1}). Among other things, you should find that the orbital period of the Earth is one year and that of Halley's comet is about 76 years.

2.2.8 Comment statements

This is a good time to mention another important feature of Python (and every other computer language), namely *comments*. In Python any program line that starts with a hash mark "#" is ignored completely by the computer. You can type anything you like on the line following a hash mark and it will have no effect:

```
# Hello!  Hi there!  This line does nothing at all.
```

Such lines are called *comments*. Comments make no difference whatsoever to the way a program runs, but they can be very useful nonetheless. You can use comment lines to leave reminders for yourself in your programs, saying what particular parts of the program do, what quantities are represented by which variables, changes that you mean to make later to the program, things you are not sure about, and so forth. Here, for instance, is a version of the polar coordinates program from Example 2.2, with comments added to explain what is happening:

File: polar.py

```
from math import sin,cos,pi

# Ask the user for the values of the radius and angle
r = float(input("Enter r: "))
d = float(input("Enter theta in degrees: "))

# Convert the angle to radians
theta = d*pi/180

# Calculate the equivalent Cartesian coordinates
x = r*cos(theta)
y = r*sin(theta)

# Print out the results
print("x =",x," y =",y)
```

This version of the program will perform identically to the original version on page 34, but it is easier to read and understand.

Comments may seem unnecessary for short programs like this one, but when you move on to creating larger programs that perform complex physics calculations you will find them very useful for reminding yourself of how things work. When you are writing a program you may think you remember how everything works and there is no need to add comments, but when you return to the same program again a week later after spending the intervening time on something else you will find it's a different story—you can't remember how anything works or why you did things this way or that, and you will be very glad if you scattered a few helpful pointers in comment lines around the program.

Comments become even more important if someone else other than you needs to understand a program you have written, for instance if you are working as part of a team that is developing a large program together. Understanding how other people's programs work can be tough at the best of times, and you will make your collaborators' lives a lot easier if you include some explanatory comments as you go along.

Comments do not have to start at the beginning of a line. Python ignores any

portion of a line that follows a hash mark, whether the hash mark is at the beginning or not. Thus you can write things like this:

```
theta = d*pi/180     # Convert the angle to radians
```

and the computer will perform the calculation $\theta = d\pi/180$ at the beginning of the line but completely ignore the hash mark and the text at the end. This is a useful trick when you intend that a comment should refer to a specific single line of code only.

2.3 Controlling programs with "if" and "while"

The programs we have seen so far are all very linear. They march from one statement to the next, from beginning to end of the program, then they stop. An important feature of computers is their ability to break this linear flow, to jump around the program, execute some lines but not others, or make decisions about what to do next based on given criteria. In this section we will see how this is done in the Python language.

2.3.1 The if statement

It will happen often in our computer programs that we want to do something only if a certain condition is met—only if $n = 0$ perhaps, or if $x > \frac{1}{2}$. We can do this using an *if statement*. Consider the following example:

```
x = int(input("Enter a whole number no greater than ten: "))
if x>10:
    print("You entered a number greater than ten")
    print("Let me fix that for you")
    x = 10
print("Your number is",x)
```

If we run this program and type in "5", we get:

```
Enter a whole number no greater than ten: 5
Your number is 5
```

But if we break the rules and enter 11, we get:

```
Enter a whole number no greater than ten: 11
You entered a number greater than ten
Let me fix that for you
Your number is 10
```

This behavior is achieved using an if statement—the second line in the program above—which tests the value of the variable `x` to see if it is greater than ten. Note

the structure of the if statement: there is the "if" part itself, which consists of the word `if` followed by the condition you are checking. In this case the condition is that $x > 10$. The condition is followed by a colon, and following that are one or more lines that tell the computer what to do if the condition is satisfied. In our program there are three of these lines, the first two printing out messages and the third fixing the value of `x`. Note that these three lines are *indented*—they start with a few spaces so that the text is shifted over a bit from the left-hand edge. This is how we tell the program which instructions are "part of the if." The indented instructions will be executed only if the condition in the if statement is met, i.e., only if $x > 10$ in this case. Whether or not the condition is met, the computer then moves on to the next line of the program, which prints the value of `x`.

In Section 1 we saw that you are free to add spaces between the parts of a Python statement to make it more readable, as in "`x = 1`", and that such spaces will have no effect on the operation of the program. Here we see an exception to that rule: spaces at the beginning of lines do have an effect with an if statement. For this reason one should be careful about putting spaces at the beginning of lines—they should be added only when they are needed, as here, and not otherwise.

A question that people sometimes ask is, "How many spaces should I put at the start of a line when I am indenting it?" The answer is that you can use any number you like. Python considers any number of spaces, from one upward, to constitute an indentation. The only rule is that once you start the indentation, every line has to have the same number of spaces. You cannot vary the number from one line to the next.

Even though you are free to choose the number of spaces you use for an indentation, it has however become standard over the years among most Python programmers to use *four* spaces, and this is the number you will see used in almost all programs, including the programs in this book. In fact, most Python development environments automatically insert spaces for you when they see an if statement, and they typically insert four. (Jupyter provides the option to use either two or four spaces, but the default is four.)

There are various different types of conditions one can use in an if statement. Here are some examples:

`if x==1:`	Check if $x = 1$. Note the double equals sign.
`if x>1:`	Check if $x > 1$
`if x>=1:`	Check if $x \geq 1$
`if x<1:`	Check if $x < 1$
`if x<=1:`	Check if $x \leq 1$
`if x!=1:`	Check if $x \neq 1$

Note particularly the double equals sign in the first example. It is one of the most common programming errors that people make in Python to use a single equals sign in an if statement instead of a double one. If you do this, you will get an error message when you try to run your program.

You can also combine two conditions in a single if statement, like this:

```
if x>10 or x<1:
    print("Your number is either too big or too small.")
```

You can use "and" in a similar way:

```
if x<=10 and x>=1:
    print("Your number is just right.")
```

You can combine more than two criteria on a line as well—as many as you like.

Two useful further elaborations of the if statement are `else` and `elif`:

```
if x>10:
    print("Your number is greater than ten.")
else:
    print("Your number is fine.  Nothing to see here.")
```

This prints different messages depending on whether `x` is greater than 10 or not. Note that the `else` line, like the original `if`, is not indented and has a colon at the end. It is followed by one or more indented lines, the indentation indicating that the lines are "inside" the else clause.

An even more elaborate example is the following:

```
if x>10:
    print("Your number is greater than ten.")
elif x>9:
    print("Your number is OK, but you're cutting it close.")
else:
    print("Your number is fine.  Move along.")
```

The word `elif` is short for "else if." If the first criterion is not met it tells the computer to try a different one. Note that we can use both `elif` and `else` one after the other, as here—if neither of the conditions specified in the `if` and `elif` clauses is satisfied then the computer moves on to the `else` clause. You can also have more than one `elif`, indeed you can have as many as you like, each one testing a different condition if the previous one was not satisfied.

2.3.2 The while statement

A useful variation on the if statement is the *while statement.* It looks and behaves similarly to the if statement:

```
x = int(input("Enter a whole number no greater than ten: "))
while x>10:
    print("This is greater than ten.  Please try again.")
    x = int(input("Enter a whole number no greater than ten: "))
print("Your number is",x)
```

As with the if statement, the while statement checks if the condition given is met (in this case if $x > 10$). If it is, it executes the indented block of code immediately following; if not, it skips the block. However (and this is the important difference), if the condition is met and the block is executed, the program then loops back from the end of the block to the beginning and checks the condition again. If the condition is still true, then the indented lines will be executed again. And it will go on looping around like this, repeatedly checking the condition and executing the indented code until the condition is finally false. (And if it is never false, then the loop goes on forever.[18]) Thus, if we were to run the snippet of code above, we would get something like this:

```
Enter a whole number no greater than ten: 11
This is greater than ten.  Please try again.
Enter a whole number no greater than ten: 57
This is greater than ten.  Please try again.
Enter a whole number no greater than ten: 100
This is greater than ten.  Please try again.
Enter a whole number no greater than ten: 5
Your number is 5
```

The computer keeps on going around the loop, asking for a number until it gets what it wants. This construct—sometimes also called a *while loop*—is commonly used in this way to ensure that some condition is met in a program or to keep on performing an operation until a certain point or situation is reached.

As with the if statement, we can specify two or more criteria in a single while statement using "and" or "or". The while statement can also be followed by an else statement, which is executed once (and once only) if and when the condition in the while statement fails. (This type of else statement is primarily used in combination with the break statement described in the next section.) There is no equivalent of `elif` for a while loop, but there are two other useful statements that modify its behavior, `break` and `continue`.

[18] If you accidentally create a program with a loop that goes on forever then you will need to know how to stop the program. In IDLE just closing the window where the program is running does the trick. In Jupyter you can click on the run button again to stop the program.

2.3.3 Break and continue

Two useful refinements of the while statement are the break and continue statements. The break statement allows us to break out of a loop even if the condition in the while statement is not met. For instance,

```
while x>10:
    print("This is greater than ten.  Please try again.")
    x = int(input("Enter a whole number no greater than ten: "))
    if x==111:
        break
```

This loop will continue looping until you enter a number not greater than 10, *except* if you enter the number 111, in which case it will give up and proceed with the rest of the program.

If the while loop is followed by an else statement, the else statement is *not* executed after a break. This allows you to create a program that does different things if the while loop finishes normally (and executes the else statement) or via a `break` (in which case the else statement is skipped).

This example also illustrates another new concept: it contains an if statement *inside* a while loop. This is allowed in Python and used often. In the programming jargon we say the if statement is *nested* inside the while loop. While loops nested inside if statements are also allowed, or ifs within ifs, or whiles within whiles. And it doesn't have to stop at just two levels. Any number of statements within statements is allowed. For some of the more complicated calculations in this book we will see examples nested four or five levels deep. In the example above, note how the break statement is doubly indented from the left margin—it is indented by an extra four spaces, for a total of eight, to indicate that it is part of a statement-within-a-statement.[19]

The continue statement is similar to the break statement, but with one important difference. Saying `continue` anywhere in a loop will make the program skip the rest of the indented code in the while loop, but instead of getting on with the rest of the program as `break` does, it then goes back to the beginning of the loop, checks the condition in the while statement again, and goes around the loop again if the condition is met. In other words, the continue statement abandons the current iteration of the loop and starts a new one, but does not abandon looping altogether.

Example 2.3: Even and odd numbers

Suppose we want to write a program that takes as input a single integer and prints out the word "even" if the number is even, and "odd" if the number is odd. We can

[19] We will come across some examples in this book where we have a loop nested inside another loop and then a break statement inside the inner loop. In that case the break statement breaks out of the inner loop only, and not the outer one.

do this by making use of the fact that n modulo 2 is zero if (and only if) n is even. Recalling that n modulo 2 is written as `n%2` in Python, here is how the program would go:

```
n = int(input("Enter an integer: "))
if n%2==0:
    print("even")
else:
    print("odd")
```

Now suppose we want a program that asks for two integers, one even and one odd—in either order—and keeps on asking until it gets what it wants. We could do this by checking all of the various combinations of even and odd, but a simpler approach is to notice that if we have one even and one odd number then their sum is odd; otherwise it is even. Thus our program might look like this:

File: `evenodd.py`

```
print("Enter two integers, one even, one odd")
m = int(input("Enter the first integer: "))
n = int(input("Enter the second integer: "))
while (m+n)%2==0:
    print("One must be even and the other odd.")
    m = int(input("Enter the first integer: "))
    n = int(input("Enter the second integer: "))
print("The numbers you chose are",m,"and",n)
```

Note how the while loop checks to see if $m + n$ is *even*. If it is, then the numbers you entered must be wrong—either both are even or both are odd—so the program asks for another pair, and it keeps on doing this until it gets what it wants.

As before, take a moment to look over this program and make sure you understand what each line does and how the program works.

Example 2.4: The Fibonacci numbers

The Fibonacci numbers are the sequence of integers in which each is the sum of the previous two, with the first two numbers being 1 and 1. Thus the first few members of the sequence are 1, 1, 2, 3, 5, 8, 13, 21. Suppose we want to calculate the Fibonacci numbers up to 1000. This would be a laborious task for a human, but it is straightforward for a computer program. All the program needs to do is keep a record of the most recent two numbers in the sequence, add them together to calculate the next number, then keep on repeating for as long as the numbers are less than 1000. Here is a program to do it:

```
f1 = 1
f2 = 1
while f1<=1000:
```

```
    print(f1)
    fnext = f1 + f2
    f1 = f2
    f2 = fnext
```

Observe how the program works. The variables `f1` and `f2` store the two most recent numbers of the sequence. If `f1` is less than 1000, we print it out, then calculate the next number by summing `f1` and `f2` and store the result in the variable `fnext`. Then we update the values of `f1` and `f2` and go around the loop again. The process continues until the value of `f1` exceeds 1000, then stops.

This program works fine, but here is a neater way to solve the same problem using the "multiple assignment" feature of Python discussed in Section 2.2.4:

File: `fibonacci.py`

```
f1,f2 = 1,1
while f1<=1000:
    print(f1)
    f1,f2 = f2,f1+f2
```

If we run this program, we get the following:

```
1
1
2
3
5
8
13
21
34
55
89
144
233
377
610
987
```

Indeed, the computer will happily print out the Fibonacci numbers up to a billion or more in just a second or two. Try it if you like.

Exercise 2.7: Catalan numbers

The Catalan numbers C_n are a sequence of integers 1, 1, 2, 5, 14, 42, 132... that play important roles in quantum mechanics and the theory of disordered systems. (They were central to

Eugene Wigner's proof of the so-called semicircle law.) They are given by

$$C_0 = 1, \qquad C_{n+1} = \frac{4n+2}{n+2} C_n.$$

Write a program that prints in increasing order all Catalan numbers less than or equal to one billion.

2.4 LISTS AND ARRAYS

We have seen how to work with integer, real, and complex numbers in Python and how to use variables to store those numbers. All the variables we have seen so far, however, represent only a single number—a single integer, real, or complex value. But in physics it is common for a variable to represent several numbers at once. We might use a vector $\mathbf{r}$, for instance, to represent the position of a point in three-dimensional space, meaning that the single symbol $\mathbf{r}$ actually corresponds to three real numbers (x, y, z). Similarly, a matrix, again usually denoted by just a single symbol, can represent an entire grid of numbers, $m \times n$ of them, where m and n could be as large as we like. There are also many cases where we have a set of numbers that we would like to treat as a single entity even if they do not form a vector or matrix. We might, for instance, do an experiment in the lab and make a hundred measurements of some quantity. Rather than give a different name to each one—a, b, c, and so forth—it makes sense to denote them by a_1, a_2, a_3, and then to consider them collectively as a set $A = \{a_i\}$, a single entity made up of a hundred numbers.

Situations like these are so common that Python provides standard features, called *containers*, for storing collections of numbers. There are several kinds of containers. In this section we look at the most common types: lists and arrays.

2.4.1 LISTS

The most basic type of container in Python is the *list*. A list, as the name suggests, is a list of quantities, one after another. In all the examples in this book the quantities will be numbers of some kind—integers, floats, and so forth—although any type of quantity that Python knows about is allowed in a list, such as strings for example.[20]

The quantities in a list, which are called its *elements*, do not all have to be of the same type. You can have an integer, followed by a float, followed by a complex number if you want. In most of the cases we will deal with, however, the elements will all be of the same type—all integers, say, or all floats—because this is what physics calculations usually demand. Thus, for instance, in the example above where we make a hundred measurements of a quantity in the lab and we want to represent

[20] If you have programmed in another computer language, then you may be familiar with arrays, which are similar to lists but not exactly the same. Python has both lists and arrays and both have their uses in physics calculations. We study arrays in Section 2.4.2.

them on the computer, we could use a list one hundred elements long and all the elements would presumably be of the same type (probably floats) because they all represent measurements of the same thing.

A list in Python is written like this: `[3, 0, 0, -7, 24]`. The elements of the list are enclosed in square brackets and separated by commas. The elements of this particular list are all integers, but they could be anything. Another example of a list might be `[1, 2.5, 3+4.6j]` which has three elements of different types, one integer, one real, and one complex.

A list can be assigned to a variable:

```
r = [1, 1, 2, 3, 5, 8, 13, 21]
```

Previously in this chapter all variables have represented just single numbers, but here we see that a variable can also represent a list of numbers. You can print a list variable, just as you can any other variable, and the computer will print out the entire list. If we run this program:

```
r = [1, 1, 2, 3, 5, 8, 13, 21]
print(r)
```

we get this:

```
[1, 1, 2, 3, 5, 8, 13, 21]
```

The quantities that make up the elements of a list can be specified using other variables, like this:

```
x = 1.0
y = 1.5
z = -2.2
r = [x, y, z]
```

This will create a three-element list with the value `[1.0, 1.5, -2.2]`. It is important to bear in mind, in this case, what happens when Python encounters an assignment statement like `r = [x, y, z]`. Remember that in such situations Python first evaluates the expression on the right-hand side, which gives `[1.0, 1.5, -2.2]` in this case, then assigns that value to the variable on the left. Thus the end result is that `r` is equal to `[1.0, 1.5, -2.2]`. It is a common error to think of `r` as being equal to `[x, y, z]` so that if, say, the value of `x` is changed later in the program the value of `r` will change as well. This is incorrect. The value of `r` will get set to `[1.0, 1.5, -2.2]` and will not change later if `x` is changed. If you want to change the value of `r` you have to explicitly assign a new value to it, with another statement like `r = [x, y, z]`.

The elements of lists can also be calculated from entire mathematical expressions, like this:

```
r = [2*x, x+y, z/sqrt(x**2+y**2)]
```

The computer will evaluate all the expressions on the right-hand side, then create a list from the values it calculated.

Once we have created a list we probably want to do some calculations with the elements it contains. The individual elements in a list `r` are denoted `r[0]`, `r[1]`, `r[2]`, and so forth. That is they are numbered in order, from beginning to end of the list, the numbers go in square brackets after the variable name, and crucially the numbers start from *zero*, not one. This may seem odd—it's not the way we usually do things in physics or in everyday life—and it takes a little getting used to. However, it turns out, as we will see, to be more convenient in a lot of situations than starting from one.

The individual elements, such as `r[0]`, behave like single variables and you can use them in the same way you would an ordinary variable. Thus, here is a short program that calculates and prints out the length of a vector in three dimensions:

```
from math import sqrt
r = [1.0, 1.5, -2.2]
length = sqrt( r[0]**2 + r[1]**2 + r[2]**2 )
print(length)
```

The first line imports the square root function from the `math` package, which we need for the calculation. The second line creates the vector, in the form of a three-element list. The third line is the one that does the actual calculation. It takes each of the three elements of the vector, which are denoted `r[0]`, `r[1]`, and `r[2]`, squares them, and adds them together. Then it takes the square root of the result, which by Pythagoras' theorem gives us the length of the vector. The final line prints out the length. If we run this program it prints

```
2.8442925306655784
```

which is the correct answer (to 17 significant figures).

We can change the values of individual elements of a list at any time, like this:

```
r = [1.0, 1.5, -2.2]
r[1] = 3.5
print(r)
```

The first line will create a list with three elements. The second then changes the value of element 1, which is the middle of the three elements since they are numbered starting from zero. So if we run the program it prints out this:

```
[1.0, 3.5, -2.2]
```

A powerful and useful feature of Python is its ability to perform operations on entire lists at once. For instance, it sometimes happens that we want to know the sum of the values in a list. Python contains a built-in function called `sum` that can calculate such sums in a single line, thus:

```
r = [1.0, 1.5, -2.2]
total = sum(r)
print(total)
```

The first line here creates a three-element list and the second calculates the sum of its elements. The final line prints out the result, and if we run the program we get this:

```
0.3
```

Other useful built-in functions include `max` and `min`, which give the largest and smallest values in a list respectively, and `len`, which calculates the number of elements in a list. Applied to the list `r` above, for instance, `max(r)` would give 1.5 and `min(r)` would give −2.2, while `len(r)` would give 3. Thus, for example, one can calculate the mean of the values in a list like this:

```
r = [1.0, 1.5, -2.2]
mean = sum(r)/len(r)
print(mean)
```

The second line here sums the elements in the list and then divides by the number of elements to give the mean value. In this case, the calculation would give a mean of 0.1.

Another feature of lists in Python, one that we will use often, is the ability to add elements to an already existing list. Suppose we have a list called `r` and we want to add a new element to the end of the list with value, say, 6.1. We can do this with the statement

```
r.append(6.1)
```

This slightly odd-looking statement is a little different in form from ones we have seen previously.[21] It consists of the name of our list, which is `r`, followed by a dot (i.e., a period), then "`append(6.1)`". Its effect is to add a new element to the end of the list with the given value, which is 6.1 in this case. The value can also be specified using a variable or a mathematical expression, thus:

```
r = [1.0, 1.5, -2.2]
x = 0.8
r.append(2*x+1)
print(r)
```

[21] This is an example of Python's object-oriented programming features. The function `append` is technically a "method" that belongs to the list "object" `r`. The function does not exist as an entity in its own right, only as a subpart of the list object. We will not dig into Python's object-oriented features in this book, since they are of relatively little use for the type of physics programming we will be doing. For software developers engaged in large-scale commercial or group programming projects, however, they can be invaluable.

If we run this program we get

```
[1.0, 1.5, -2.2, 2.6]
```

Note how the computer has calculated the value of `2*x+1` to be 2.6, then added that value to the end of the list.

A particularly useful trick that we will employ frequently in this book is the following. We create an *empty* list, a list with no elements in it at all, then add elements to it one by one as we learn of or calculate their values. A list created in this way can grow as large as we like (within limitations set by the amount of memory the computer has to store the list).

To create an empty list we say

```
r = []
```

This creates a list called `r` with no elements. Even though it has no elements in it, the list still exists. It's like an empty set in mathematics—it exists as an object, but it doesn't contain anything (yet). Now we can add elements thus:

```
r.append(1.0)
r.append(1.5)
r.append(-2.2)
print(r)
```

which produces

```
[1.0, 1.5, -2.2]
```

We will, for instance, use this technique to make graphs in Section 3.1. Note that you must create the empty list first before adding elements. You cannot add elements to a list until it has been created—the computer will give an error message if you try.

We can also remove a value from the end of a list by saying `r.pop()`:

```
r = [1.0, 1.5, -2.2, 2.6]
r.pop()
print(r)
```

which gives

```
[1.0, 1.5, -2.2]
```

And we can remove a value from anywhere in a list by saying `r.pop(n)`, where `n` is the number of the element you want to remove.[22] Bear in mind that the elements are

[22]However, removing an element from the middle (or the beginning) of a list is a slow operation because the computer then has to move all the elements above that down one place to fill the gap. For a long list this can take a long time and slow down your program, so you should avoid doing it if possible.

numbered from zero, so if you want to remove the first item from a list you would say `r.pop(0)`.

2.4.2 Arrays

As we have seen, a list in Python is an ordered set of values, such as a set of integers or a set of floats. There is another object in Python that is somewhat similar: an *array*. An array is also an ordered set of values, but there are some important differences between lists and arrays:

1. The number of elements in an array is fixed. You cannot add elements to an array once it is created, or remove them.
2. The elements of an array must all be of the same type, such as all floats or all integers. You cannot mix elements of different types in the same array and you cannot change the type of the elements once an array is created.

Lists, as we have seen, have neither of these restrictions and, on the face of it, these seem like significant drawbacks of the array. Why would we ever use an array if lists are more flexible? The answer is that arrays have several significant advantages over lists as well:

3. Arrays can be two-dimensional, like matrices in algebra. That is, rather than just a one-dimensional row of elements, we can have a grid of them. Indeed, arrays can in principle have any number of dimensions, including three or more, although we will not use dimensions above two in this book. Lists, by contrast, are always just one-dimensional.
4. Arrays behave roughly like vectors or matrices: you can do arithmetic with them, such as adding them together, and you will get the result you expect. This is not true with lists. If you try to do arithmetic with a list you will either get an error message, or you will not get the result you expect.
5. Arrays work faster than lists. Especially if you have a very large array with many elements, then calculations may be significantly faster using an array.

In physics it often happens that we are working with a fixed number of elements all of the same type, as when we are working with matrices or vectors, for instance. In that case, arrays are clearly the tool of choice: the fact that we cannot add or remove elements is immaterial if we never need to do such a thing, and the superior speed of arrays and their flexibility in other respects can make a significant difference to our programs. We will use arrays extensively in this book.

Before you use an array you need to create it, meaning you need to tell the computer how many elements it will have and of what type. Python provides functions

(On the other hand, if it doesn't matter to you what order the elements of a list appear in, then you can effectively remove any element rapidly by first setting it equal to the last element in the list, then removing the last element.) There is also another, less commonly used container in Python called a *deque*, that allows one to quickly add or remove elements from either the beginning or the end (but not the middle). We will not used deques in this book however.

that allow you do this in several different ways. These functions are all found in the package `numpy`. (The name is short for "numerical Python.")

In the simplest case, we can create a one-dimensional array with n elements, all of which are initially equal to zero, using the function `zeros` from the `numpy` package. The function takes two arguments. The first is the number of elements the array is to have and the second is the type of the elements, such as `int`, `float`, or `complex`. For instance, to create a new array with four floating-point elements we would do the following:

```
from numpy import zeros
a = zeros(4,float)
print(a)
```

In this example the new array is denoted `a`. When we run the program the array is printed out as follows:

```
[ 0.  0.  0.  0.]
```

Note that arrays are printed out slightly differently from lists—there are no commas between the elements, only spaces.

We can use the same approach to create an array of ten integers with the statement "`a = zeros(10,int)`" or an array of a hundred complex numbers with the statement "`a = zeros(100,complex)`". The size of the arrays you can create is limited only by the computer memory available to hold them. Modern computers can hold arrays with hundreds of millions or even billions of elements.

To create a two-dimensional floating-point array with m rows and n columns, you say "`zeros([m,n],float)`", so

```
a = zeros([3,4],float)
print(a)
```

produces

```
[[ 0.  0.  0.  0.]
 [ 0.  0.  0.  0.]
 [ 0.  0.  0.  0.]]
```

Note that the first argument of the `zeros` function in this case is itself a *list* (that's why it is enclosed in brackets `[...]`), whose elements give the size of the array along each dimension. We could create a three-dimensional array by giving a three-element list (and so on for higher dimensions).

There is a similar function in `numpy` called `ones` that creates an array with all elements equal to one. The form of the function is exactly the same as for the function `zeros`. Only the values in the array are different.

On the other hand, if we are going to change the values in an array immediately after we create it, then it doesn't make sense to have the computer set all of them to

zero (or one). Setting them to zero takes some time, time that is wasted if you don't need the zeros. In that case you can use a different function, `empty`, again from the package `numpy`, to create an empty array:

```
from numpy import empty
a = empty(4,float)
```

This creates an array of four "empty" floating-point elements. In practice the elements are not actually empty. Instead they contain whatever numbers happened to be littered around the computer's memory at the time the array is created. The computer just leaves those values as they are and doesn't waste any time changing them. You can also create empty integer or complex arrays by saying `int` or `complex` instead of `float`.

A different way to create an array is to take a list and convert it into an array, which you can do with the function `array` from the package `numpy`. For instance we can say:

```
from numpy import array
r = [1.0, 1.5, -2.2]
a = array(r,float)
```

which will create an array of three floating-point elements, with values 1.0, 1.5, and −2.2. If the elements of the list (or some of them) are not already floats, they will be converted to floats.[23] You can also create integer or complex arrays in the same fashion, and the list elements will be converted to the appropriate type if necessary.[24]

The second and third lines above can conveniently be combined into one, like this:

```
a = array([1.0,1.5,-2.2],float)
```

This is a quick and easy way to create a new array with predetermined values in its elements. We will use this trick frequently.

We can also create two-dimensional arrays with specified values. To do this we again use the `array` function, but now the argument we give it is a *list of lists*, which gives the elements of the array row by row. For example,

[23] Though it is not something we will often need to do, you can also convert an array into a list using the built-in function `list` by writing `r = list(a)`. Note that you do not specify a type for the list, because lists don't have types. The types of the elements in the list will just be the same as the types of the elements in the array.

[24] Two caveats apply here. (1) If you create an integer array from a list that has any floating-point elements, the fractional part of the floating-point elements (i.e., the part after the decimal point) will be thrown away. (2) If you try to create a floating-point or integer array from a list containing complex values you will get an error message. This is not allowed.

```
a = array([[1,2,3],[4,5,6]],int)
print(a)
```

This creates a two-dimensional array of integers and prints it out:

```
[[ 1  2  3]
 [ 4  5  6]]
```

The list of lists must have the same number of elements for each row of the array (three in this case) or the computer will complain.

We can refer to individual elements of an array in a manner similar to the way we refer to the elements of a list. For a one-dimensional array we write `a[0]`, `a[1]`, and so forth. Note, as with lists, that the numbering of the elements starts at zero, not one. We can also set individual elements equal to new values thus:

```
a[2] = 4
```

Note, however, that, since the elements of an array are of a particular type (which cannot be changed after the array is created), any value you specify will be converted to that type. If you give an integer value for a floating-point array element, it will be converted to floating-point. If you give a floating-point value for an integer array, it will be converted to an integer, discarding the part after the decimal point, if any. (And if you try to assign a complex value to an integer or floating-point array you will get an error message—this is not allowed.)

For two-dimensional arrays we use two indices, separated by commas, to denote the individual elements, as in `a[2,4]`, with counting again starting at zero for both indices. Thus, for example

```
from numpy import zeros
a = zeros([2,2],int)
a[0,1] = 1
a[1,0] = -1
print(a)
```

would produce the output

```
[[ 0  1]
 [-1  0]]
```

Note that when Python prints a two-dimensional array it observes the convention of standard matrix arithmetic that the first index of a two-dimensional array denotes the row of the array element and the second denotes the column.

2.4.3 Reading an array from a file

Another, somewhat different, way to create an array is to read a set of values from a computer file, which we can do with the function `loadtxt` from the package `numpy`. Suppose we have a text file that contains the following string of numbers, on consecutive lines:

```
1.0
1.5
-2.2
2.6
```

and suppose that this file is called `values.txt` on the computer. Then we can do the following:

```
from numpy import loadtxt
a = loadtxt("values.txt",float)
print(a)
```

When we run this program, we get the following printed on the screen:

```
[ 1.0  1.5 -2.2  2.6]
```

As you can see, the computer has read the numbers in the file and put them in a floating-point array of the appropriate length. (For this to work the file `values.txt` has to be in the same folder or directory on the computer as your Python program.[25])

We can use the same trick to read a two-dimensional grid of values and put them in a two-dimensional array. If the file `values.txt` contained the following:

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

then the exact same program above would create a two-dimensional 3×4 array of floats with the appropriate values in it.

The `loadtxt` function is a very useful one for physics calculations. It happens often that we have a file or files containing numbers we need for a calculation. They might be data from an experiment, for example, or numbers calculated by another computer program. We can use `loadtxt` to transfer those numbers into an array so that we can perform calculations on them.

[25] You can also give a full path name for the file, specifying explicitly the folder as well as the file name, in which case the file can be in any folder.

2.4.4 ARITHMETIC WITH ARRAYS

As with lists, the individual elements of an array behave like ordinary variables, and we can do arithmetic with them in the usual way. We can write things like

```
a[0] = a[1] + 1
```

or

```
x = a[2]**2 - 2*a[3]/y
```

But we can also do arithmetic with entire arrays at once, a powerful feature that can be very useful in physics calculations. In general, when doing arithmetic with whole arrays the rule is that whatever arithmetic operation you specify is done independently to each element of the array or arrays involved. Consider this short program:

```
from numpy import array
a = array([1,2,3,4],int)
b = 2*a
print(b)
```

When we run this program it prints

```
[2 4 6 8]
```

As you can see, when we multiply the array `a` by 2 the computer simply multiplies each individual element by 2. A similar thing happens if you divide. Note that when we run this program, the computer creates a new array `b` holding the results of the multiplication. This is another way to create arrays, different from the methods we mentioned before. We do not have to create the array `b` explicitly, using for instance the `empty` function. When we perform a calculation with arrays, Python will automatically create a new array for us to hold the results.

If you add or subtract two separate arrays, the computer will add or subtract each element separately, so that

```
a = array([1,2,3,4],int)
b = array([2,4,6,8],int)
print(a+b)
```

results in

```
[3 6 9 12]
```

(For this to work, the arrays must have the same size. If they do not, the computer will complain.)

All of these operations give the same result as the equivalent mathematical operations on vectors in normal algebra, which makes arrays well suited to representing

vectors in physics calculations.[26] If we represent a vector using an array, then arithmetic operations such as multiplying or dividing by a scalar or adding or subtracting vectors can be written just as they would in normal mathematics. You can also add or subtract a scalar quantity to or from an array, which the computer interprets to mean it should add that quantity to every element. So

```
a = array([1,2,3,4],int)
print(a+1)
```

results in

```
[2 3 4 5]
```

However, if we multiply two arrays together the outcome is perhaps not exactly what you would expect—you do not get the vector (dot) product of the two. If we do this:

```
a = array([1,2,3,4],int)
b = array([2,4,6,8],int)
print(a*b)
```

we get

```
[2 8 18 32]
```

What has the computer done? It has multiplied the two arrays together element by corresponding element. The first elements of the two arrays are multiplied together, then the second elements, and so on. This is logical in a sense—it is the exact equivalent of what happens when you add. The prescribed operation, multiplication in this case, is performed independently on each element. (Division works similarly.) Occasionally this may be what you want the computer to do, but more often in physics calculations we want the true vector dot product of our arrays. In that case we can calculate the product using the function `dot` from the package `numpy`:

```
from numpy import array,dot
a = array([1,2,3,4],int)
b = array([2,4,6,8],int)
print(dot(a,b))
```

When we run this program, it prints

```
60
```

which is the correct value of the dot product.

[26] The same operations, by contrast, do not work with lists, so lists are less good for storing vector values.

All of the operations above also work with two-dimensional arrays, which makes such arrays convenient for storing matrices. Multiplying and dividing by scalars as well as addition and subtraction of two-dimensional arrays all work as in standard matrix algebra. Multiplication will multiply element by element, which is usually not what you want, but the `dot` function calculates the standard matrix product. Consider, for example, this matrix calculation:

$$\begin{pmatrix}1&3\\2&4\end{pmatrix}\begin{pmatrix}4&-2\\-3&1\end{pmatrix}+2\begin{pmatrix}1&2\\2&1\end{pmatrix}=\begin{pmatrix}-3&5\\0&2\end{pmatrix}$$

In Python we would do this as follows:

```
a = array([[1,3],[2,4]],int)
b = array([[4,-2],[-3,1]],int)
c = array([[1,2],[2,1]],int)
print(dot(a,b)+2*c)
```

You can also multiply matrices and vectors together. If `v` is a one-dimensional array then `dot(a,v)` treats it as a column vector and multiplies it on the left by the matrix `a`, while `dot(v,a)` treats it as a row vector and multiplies on the right by `a`. Python is intelligent enough to know the difference between row and column vectors, and between left- and right-multiplication, and to choose the right operation in each case.

Functions can be applied to arrays in much the same way as to lists. The built-in functions `sum`, `min`, `max`, and `len` described in Section 2.4.1 can be applied to one-dimensional arrays to calculate sums of elements, minimum and maximum values, and the number of elements. Applying functions to arrays with two or more dimensions produces more erratic results. For instance, the `len` function applied to a two-dimensional array returns the number of rows in the array and the functions `max` and `min` produce only error messages. However, the `numpy` package contains functions that perform similar duties and work more predictably with two-dimensional arrays, such as functions `min` and `max` that find minimum and maximum values. In place of the `len` function, there are two different features, called `size` and `shape`. Consider this example:

```
a = array([[1,2,3],[4,5,6]],int)
print(a.size)
print(a.shape)
```

which produces

```
6
(2, 3)
```

Thus, `a.size` tells you the total number of elements in all rows and columns of the array `a` (which is roughly the equivalent of the `len` function for lists and one-dimensional arrays), and `a.shape` returns a list giving the dimensions of the array.

(Technically it is a "tuple" not a list, but for our purposes it is roughly the same thing. You can say `n = a.shape`, and then `n[0]` is the number of rows of `a` and `n[1]` is the number of columns.) For one-dimensional arrays there is no difference between `size` and `shape`. They both give the total number of elements.

There are a number of other functions in the `numpy` package that are useful for performing calculations with arrays. The full list can be found in the online documentation at `www.scipy.org`.

Example 2.5: Average of a set of values in a file

Suppose we have a set of numbers stored in a file `values.txt` and we want to calculate their mean. Even if we don't know how many numbers there are we can do the calculation quite easily:

```
from numpy import loadtxt
values = loadtxt("values.txt",float)
mean = sum(values)/len(values)
print(mean)
```

The first line imports the `loadtxt` function and the second uses it to read the values in the file and put them in an array called `values`. The third line calculates the mean as the sum of the values divided by the number of values and the fourth prints out the result.

Now suppose we want to calculate the mean-square value. To do this, we first need to calculate the squares of the individual values, which we can do by multiplying the array `values` by itself. Recall, that the product of two arrays in Python multiplies together each pair of corresponding elements, so `values*values` is an array with elements equal to the squares of the original values. (We could also write `values**2`, which would produce the same result.) Then we can use the function `sum` to add up the squares. Thus our program might look like this:

```
from numpy import loadtxt
values = loadtxt("values.txt",float)
mean = sum(values*values)/len(values)
print(mean)
```

On the other hand, suppose we want to calculate the *geometric* mean of our set of numbers. (We will assume our numbers are all positive, since one cannot take the geometric mean of negative numbers.) The geometric mean of a set of n values x_i is defined to be the nth root of their product, thus:

$$\overline{x} = \left(\prod_{i=1}^{n} x_i \right)^{1/n}. \tag{2.1}$$

Taking natural logs of both sides we get

$$\ln \overline{x} = \ln\left(\prod_{i=1}^{n} x_i\right)^{1/n} = \frac{1}{n}\sum_{i=1}^{n} \ln x_i \tag{2.2}$$

or

$$\overline{x} = \exp\left(\frac{1}{n}\sum_{i=1}^{n} \ln x_i\right). \tag{2.3}$$

In other words, the geometric mean is the exponential of the arithmetic mean of the logarithms. To write a program to calculate this, we need one new thing: the `numpy` package contains its own `log` function that will calculate the logs of all the elements of an array. Thus we can write

```
from numpy import loadtxt,log
from math import exp
values = loadtxt("values.txt",float)
geometric = exp(sum(log(values))/len(values))
print(geometric)
```

The log function here calculates all the logarithms in a single step, then we take their average and calculate the exponential of the result, which gives us our geometric mean.

Finally in this section, here is a word of warning. Consider the following program:

```
from numpy import array
a = array([1,1],int)
b = a
a[0] = 2
print(a)
print(b)
```

Take a look at this program and work out for yourself what you think it will print. If we actually run it (and you can try this for yourself) it prints the following:

```
[2 1]
[2 1]
```

This may not be what you were expecting. Reading the program, it looks like array `a` should be equal to `[2,1]` and `b` should be equal to `[1,1]` when the program ends, but the output of the program appears to indicate that both are equal to `[2,1]`. What has happened?

The answer lies in the line "`b = a`" in the program. In Python, direct assignment of arrays in this way, setting the value of one array equal to another, does not work as you might expect. You might imagine that "`b = a`" would cause Python to create

a new array b holding a copy of the numbers in the array a, but this is not what happens. In fact, all that "b = a" does is that it declares "b" to be a new name for the array previously called "a". That is, "a" and "b" now both refer to the same array of numbers, stored somewhere in the memory of the computer. If we change the value of an element in array a, as we do in the program above, then we also change the same element of array b, because a and b are really just the same array.[27]

This is a tricky point, one that can catch you out if you are not aware of it. You can do all sorts of arithmetic operations with arrays and they will work just fine, but this one operation, setting an array equal to another array, does not work the way you expect it to.

Why does Python do this? At first sight it seems peculiar, annoying even, but there is a good reason for it. Arrays can be very large, with millions or even billions of elements. So if a statement like "b = a" caused the computer to create a new array b that was a complete copy of the array a, it might have to copy very many numbers in the process, potentially using a lot of time and memory space. But in many cases it is not actually necessary to make a copy of the array. Particularly if you are interested only in reading the numbers in an array, not in changing them, then it does not matter whether a and b are separate arrays that happen to contain the same values or are actually just two names for the same array—everything will work the same either way. Creating a new name for an old array is normally far faster than making a copy of the entire contents, so, in the interests of efficiency, this is what Python does.

Of course there are times when you really do want to make a new copy of an array, so Python also provides a way of doing this. To make a copy of an array a we can use the function copy from the numpy package thus:

```
from numpy import copy
b = copy(a)
```

This will create a new array b whose elements are an exact copy of those of array a. If we were to use this line, instead of the line "b = a", in the program above, then run the program, it would print this:

```
[2 1]
[1 1]
```

which is now the "correct" answer.

[27] If you have worked with the programming languages C or C++ you may find this behavior familiar, since those languages treat arrays the same way. In C, the statement "b = a", where a and b are arrays, also merely creates a new name for the array a, not a new array.

Exercise 2.8: Suppose arrays a and b are defined as follows:

```
from numpy import array
a = array([1,2,3,4],int)
b = array([2,4,6,8],int)
```

What will the computer print upon executing the following lines? (Try to work out the answer before testing it on the computer.)

a) `print(b/a+1)`
b) `print(b/(a+1))`
c) `print(1/a)`

2.4.5 SLICING

Here is another useful trick, called *slicing*, which works with both arrays and lists. Suppose we have a list `r`. Then `r[m:n]` is another list composed of a subset of the elements of `r`, starting with element *m* and going up to *but not including* element *n*. Here is an example:

```
r = [1, 3, 5, 7, 9, 11, 13, 15]
s = r[2:5]
print(s)
```

which produces

```
[5, 7, 9]
```

Observe what has happened. The variable `s` is a new list, which is a sublist of `r` consisting of elements 2, 3, and 4 of `r`, but not element 5. Since the numbering of elements starts at zero, not one, element 2 is actually the third element of the list, which is the 5, and elements 3 and 4 are the 7 and 9. So `s` has three elements equal to 5, 7, and 9.

Slicing is useful in many physics calculations, particularly, as we will see, in matrix calculations, calculations on lattices, and in the solution of differential equations. There are a number of variants on the basic slicing formula above. You can write `r[2:]`, which means all elements of the list from element 2 up to the end of the list, or `r[:5]`, which means all elements from the start of the list up to, but not including, element 5. And `r[:]` with no numbers at all means all elements from the beginning to the end of the list, i.e., the entire list. This last is not very useful—if we want to refer to the whole list we can just say `r`. We get the same thing, for example, whether we write `print(r[:])` or `print(r)`. However, we will see a use for this form in a moment.

There is also a three-index version of slicing which takes the form `r[m:n:k]`. This version creates a new list with elements drawn from `r`, starting at element *m* and going up to but not including element *n*, but now increasing in steps of *k* elements

at a time. For example `s = r[1:6:2]` would give a list composed of elements 1, 3, and 5 of `r`, which would be `[3, 7, 11]` if `r` is the same as the example above. The third index of the slice can also be negative, in which the elements are drawn from `r` in reverse order. For example `r[5:2:-1]` produces a list consisting of elements 5, 4, and 3 from the original `r` (but not element 2), which would give `[11, 9, 7]` in this case. This also provides a quick way to reverse the order of the elements in a list. Writing `s = r[::-1]` gives you a new list `s` containing all the elements of `r` in reverse order.

Slicing can also be applied to arrays, giving you a new array of the same type as the one you started with and containing a subset of its elements. For example:

```
from numpy import array
a = array([2,4,6,8,10,12,14,16],int)
b = a[3:6]
print(b)
```

which prints

```
[ 8 10 12]
```

You can also write `a[3:]`, or `a[:6]`, or `a[:]`, as with lists, or the three-index version `a[3:6:2]`.

Slicing works with two-dimensional arrays as well. For instance, `a[2,3:6]` gives you a one-dimensional array with three elements equal to `a[2,3]`, `a[2,4]`, and `a[2,5]`, while `a[2:4,3:6]` gives you a two-dimensional array of size 2×3 with values drawn from the appropriate subblock of `a`, starting at `a[2,3]`. And `a[2,:]` gives you the whole of row 2 of array `a`, which means the third row since the numbering starts at zero, while `a[:,3]` gives you the whole of column 3, which is the fourth column. These forms will be particularly useful to us for doing vector and matrix arithmetic.

2.4.6 Set, dicts, and other containers

We have discussed two types of Python containers in detail, lists and arrays, and these are the only types we will use in this book. There are, however, a number of others, which we describe briefly in this section and which can be useful in certain circumstances.

A *set* is a Python container that stores an unordered collection of unique values, akin to a set in mathematics. A set could for instance contain the integer values $\{1, 4, -3\}$. In addition to integers a set can also contain floats, complex values, strings, or almost any other Python quantity, including entire lists, arrays, or even other sets. The contents of a set are unordered, which means that there is no first or last element and the elements do not come in any particular order. The elements are also unique, meaning that a value can appear at most once in the set. So there cannot be two 1s for example. If you try and add two 1s, only the first will be accepted and the attempted

addition of the second will have no effect. Sets can be created in a variety of ways, but we can, for example, write `s = set([1,4,-3])`, which takes the list `[1,4,-3]` and converts it into a set in much the same way that the `array` function converts a list into an array. Note, however, that, unlike the `array` function, the `set` function does not need to be imported from a package. It is a built-in function, always available in Python. (See Section 2.2.7 for discussion of built-in functions.)

A *dict*—short for "dictionary"—is a Python container that behaves like a more flexible kind of list or array. Consider the following example:

```
d = dict()
d[4] = 10
d[2] = 7.7
d[-3] = 5+3j
```

This code first creates an empty dict using the built-in `dict` function, then adds three elements to it. Elements can take (almost) any value we care to give them, and each element is identified by an index, technically called a *key*, which is given in square brackets after the dict name, in a manner similar to a list or array, as in `d[4]` or `d[-3]`. Important points to notice about dicts are:

1. The elements of a dict, like those of a list, can have any type of value, including integer, floating-point, or complex values, or strings. They can also be lists, arrays, sets, or even other dicts. The elements of a dict do not all have to be of the same type.
2. Unlike a list or array, the elements of a dict can have any index—the indices do not need to start at zero or run consecutively. Elements of a dict are only created when you give them a value. If you ask for an element that has not yet been created you will get an error message. For instance, `print(d[0])` would give an error in the example above.
3. The index or key of an element does not need to be an integer. It can take almost any value we like, including floats, strings, and others. For instance, we are allowed to write `d[3.14] = "xyz"` or `d["abc"] = 2`.

Sets and dicts are powerful data structures. While they will not be necessary for the calculations we do in this book, they do find use in computational physics and in many other applications, and you may encounter them when working with Python, so they are worth knowing about.

Other Python containers include the *deque*, a type of double-ended list that allows one to quickly add or remove elements from either the start or the end of the list, and the *defaultdict*, a variant of a dict that defines a default value for elements that have not been explicitly created. These, however, are relative rarities in Python code, and particularly in computational physics, and we will not consider them further in this book.

2.5 For loops

In Section 2.3.2 we saw a way to make a program loop repeatedly around a given section of code using a while statement. In practice, however, while statements are used only rather rarely. There is another, much more commonly used loop construction in the Python language, the *for loop*. A for loop is a loop that runs through the elements of a container, such as a list or array, in turn. Consider this short example:

```
r = [1, 3, 5]
for n in r:
    print(n)
    print(2*n)
print("Finished")
```

If we run this program it prints the following:

```
1
2
3
6
5
10
Finished
```

What's happening here is as follows. The program first creates the list `r`, then the `for` statement sets `n` equal to each value in the list in turn. For each value the computer carries out the steps in the following two lines, printing out n and $2n$, then loops back around to the `for` statement again and sets `n` to the next value in the list. Note that the two print statements are indented, in a manner similar to the if and while statements we saw earlier. This is how we tell the program which instructions are "in the loop." Only the indented instructions will be executed each time around the loop. When the program has worked its way through all the values in the list, it stops looping and moves on to the next line of the program, which in this case is a third print statement which prints the word "Finished." In this example the computer will go around the loop three times, since there are three elements in list `r`.

The same construction works with arrays as well—you can use a for loop to go through the elements of a (one-dimensional) array in turn.[28] Also the statements `break` and `continue` (see Section 2.3.3) can be used with for loops the same way they are used with while loops: `break` ends the loop and moves to the next statement after the loop; `continue` abandons the current iteration of the loop and moves on to the next iteration. And you can add an `else` statement at the end of a for loop, which operates in the same way as it does with a while loop: the code following the else

[28] For loops also work with sets, dicts, deques, and other more exotic containers (see Section 2.4.6 on page 63), although we will not use any of these in this book.

statement is executed after the for loop ends, but only if it ends normally. If the loop is aborted prematurely using `break` then the else statement is skipped.

The most common use of a for loop is simply to run through a piece of code a specified number of times, such as ten, say, or a million. To achieve this, Python provides a special built-in function called `range`, which creates a list of a given length, usually for use with a for loop. For example `range(5)` returns a list `[0, 1, 2, 3, 4]`—that is, a list of consecutive integers, starting at zero and going up to, but not including, 5. Note that this means the list contains exactly five elements but does not include the number 5 itself.[29] Thus

```
r = range(5)
for n in r:
    print("Hello again")
```

produces the following output

```
Hello again
Hello again
Hello again
Hello again
Hello again
```

The for loop gives `n` each of the values in `r` in turn, of which there are five, and for each of them it prints out the words "Hello again". So the end result is that the computer prints out the same message five times. In this case we are not actually interested in the values `r` contains, only the fact that there are five of them—they merely provide a convenient tool that allows us to run around the same piece of code a given number of times.

A more interesting use of the `range` function is the following:

```
r = range(5)
for n in r:
    print(n**2)
```

[29] Technically, `range` produces not a list but an *iterator*, a specialized object that contains the elements of the range in order. If you actually wanted to produce a list using `range` then you could write "`list(range(5))`", which would create an iterator and then convert it to a list. In practice, however, we need to do this very rarely, and never in this book—the main use of the `range` function is in for loops and you can use an iterator directly in a for loop without converting it into a list first.

The difference between an iterator and a list is that the values in an iterator are not stored in memory the way the values in a list are, but are instead calculated on the fly when they are needed, which saves memory space. In versions of Python prior to version 3, the `range` function produced a list, not an iterator, but both lists and iterators give the same results when used in for loops, so the loops in this book will work without modification with either version 2 or version 3 of Python. For further discussion of this point, and of iterators in general, see Appendix D starting on page 580.

Now we are making use of the actual values `r` contains, printing out the square of each one in turn:

```
0
1
4
9
16
```

In both of these examples we used a variable `r` to store the results of the `range` function, but it is not necessary to do this. Often one takes a shortcut and just writes

```
for n in range(5):
    print(n**2)
```

which achieves the same result with less fuss. This is probably the most common form of the for loop and we will see many loops of this form throughout this book.

There are a number of useful variants of the `range` function, as follows:

```
range(5)          gives   [0, 1, 2, 3, 4]
range(2,8)        gives   [2, 3, 4, 5, 6, 7]
range(2,20,3)     gives   [2, 5, 8, 11, 14, 17]
range(20,2,-3)    gives   [20, 17, 14, 11, 8, 5]
```

When there are two arguments to the function it generates integer values that run from the first up to, but not including, the second. When there are three arguments, the values run from the first up to but not including the second, *in steps of* the third. Thus in the third example above the values increase in steps of 3. In the fourth example, which has a negative argument, the values *decrease* in steps of 3. Note that in each case the values returned by the function do not include the value at the end of the given range—the first value in the range is always included; the last never is.

Thus, for example, we can print out the first ten powers of two with the following lines:

```
for n in range(1,11):
    print(2**n)
```

Note how the upper limit of the range is given as 11. This program will print out the powers 2, 4, 8, 16, and so forth up to 1024. It stops at 2^{10}, not 2^{11}, because `range` always excludes the last value.

A further point to notice about the `range` function is that all its arguments must be integers. The function will not work if you give it non-integer arguments, such as floats, and you will get an error message if you try. It is particularly important to remember this when the arguments are calculated from the values of other variables. This short program, for example, will not work:

```
p = 10
q = 2
for n in range(p/q):
    print(n)
```

You might imagine these lines would print out the integers from zero to four, but if you try it you will just get an error message because, as discussed in Section 2.2.4, the division operation returns a floating-point value, even if the result of the division is, mathematically speaking, an integer. Thus the quantity "p/q" in the program above is a floating-point quantity equal to 5.0 and is not allowed as an argument of the range function. We can fix this problem by using integer division instead:

```
for n in range(p//q):
    print(n)
```

This will now work as expected. (See Section 2.2.4, page 24 for a discussion of integer division.)

Another useful function is arange from the numpy package, which is similar to range but generates arrays, rather than lists or iterators[30] and moreover works with floating-point arguments as well as integer ones. For example, arange(1,8,2) gives a one-dimensional array of integers [1,3,5,7], while arange(1.0,8.0,2.0) gives an array of floating-point values [1.0,3.0,5.0,7.0] and arange(2.0,2.8,0.2) gives [2.0,2.2,2.4,2.6]. As with range, the arange function can be used with one, two, or three arguments, and does the equivalent thing to range in each case.

Another similar function is the function linspace, also from the numpy package, which generates an array with a given number of floating-point values between given limits. For instance, linspace(2.0,2.8,5) divides the interval from 2.0 to 2.8 into 5 values, creating an array with floating-point elements [2.0,2.2,2.4,2.6,2.8]. Similarly, linspace(2.0,2.8,3) would create an array with elements [2.0,2.4,2.8]. Note that, unlike both range and arange, linspace includes the last point in the range. Also note that although linspace can take either integer or floating-point arguments, it always generates floating-point values, even when the arguments are integers.

EXAMPLE 2.6: PERFORMING A SUM

It happens often in physics calculations that we need to evaluate a sum. If we have the values of the terms in the sum stored in a list or array then we can calculate the sum using the built-in function sum described in Section 2.4.1. In more complicated

[30] The function arange generates an actual array, calculating all the values and storing them in the computer's memory. This can cause problems if you generate a very large array because the computer can run out of memory, crashing your program, an issue that does not arise with the iterators generated by the range function. For instance, arange(20000000000) will produce an error message on most computers, while the equivalent expression with range will not. See Appendix D for more discussion of this point.

situations, however, it is often more convenient to use a for loop. Suppose, for instance, that we want to know the value of the sum $s = \sum_{k=1}^{100}(1/k)$. The standard way to program this is as follows:

1. First create a variable to hold the value of the sum, and initially set it to zero. As above, we will call the variable `s`, and we want it to be a floating-point variable, so we will write "`s = 0.0`".
2. Now use a for loop to take the variable `k` through all values from 1 to 100. For each value, calculate `1/k` and add it to the variable `s`.
3. When the for loop ends the variable `s` will contain the value of the complete sum.

Thus our program looks like this:

```
s = 0.0
for k in range(1,101):
    s += 1/k
print(s)
```

Note how we use `range(1,101)` so that the values of `k` start at 1 and end at 100. We also used the "+=" modifier, which adds to a variable as described in Section 2.2.4. If we run this program it prints the value of the sum thus:

```
5.187377517639621
```

As another example, suppose we have a set of real values stored in a computer file called `values.txt` and we want to compute and print the sum of their squares. We could achieve this as follows:

```
from numpy import loadtxt
values = loadtxt("values.txt",float)
s = 0.0
for x in values:
    s += x**2
print(s)
```

Here we have used the function `loadtxt` from Section 2.4.3 to read the values in the file and put them in an array called `values`. Note also how this example does not use the `range` function, but simply goes through the list of values directly.

For loops and the `sum` function give us two different ways to compute sums of quantities. It is not uncommon for there to be more than one way to achieve a given goal in a computer program, and in particular it is often the case that one can use either a for loop or a function to perform the same calculation. In general for loops are more flexible, but functions are often faster and can save significant amounts of time if you are dealing with large arrays. Thus both approaches have their advantages. Part of the art of good computer programming is learning which approach is best in which situation.

EXAMPLE 2.7: FINDING THE LARGEST NUMBER IN A LIST

Another operation that comes up frequently in computational physics is finding the largest or smallest element in a list or array. There exist two built-in functions, `max` and `min`, that can perform these operations, but again one can also use a for loop, and often the latter approach is more flexible.

To find the largest element in a list using a for loop we would create a variable to hold the largest value—let's call it `largest`—and initially give it a value less than or equal to the smallest value a list element can have. For instance, if our list contains only positive values, then we could safely set `largest` to zero, knowing that no list element will be smaller than this. Then we run through the elements in our list in turn and check each one against the current value of `largest`. If an element is larger than the current value, it becomes the new largest value. The code looks like this:

```
from numpy import loadtxt
values = loadtxt("values.txt",float)
largest = 0.0
for x in values:
    if x>largest:
        largest = x
print(largest)
```

When the loop ends the variable `largest` is equal to the largest element in the list. We can use a similar procedure to find the smallest element also.

EXAMPLE 2.8: EMISSION LINES OF HYDROGEN

Let us revisit an example we saw in Chapter 1. On page 6 we gave a program for calculating the wavelengths of emission lines in the spectrum of the hydrogen atom, based on the Rydberg formula

$$\frac{1}{\lambda} = R\left(\frac{1}{m^2} - \frac{1}{n^2}\right). \tag{2.4}$$

Our program looked like this:

```
R = 1.097e-2
for m in [1,2,3]:
    print("Series for m =",m)
    for k in [1,2,3,4,5]:
        n = m + k
        invlambda = R*(1/m**2-1/n**2)
        print("  ",1/invlambda,"nm")
```

Based on what we have learned we can now understand how this program works. It uses two nested for loops—a loop within another loop—with the code in the inner

loop doubly indented. We discussed nesting previously in Section 2.3.3. The first for loop takes the integer variable `m` through the values 1, 2, 3. Then for each value of `m`, the second, inner loop takes `k` though the values 1, 2, 3, 4, 5, adds those values to `m` to calculate `n` and then applies the Rydberg formula. The end result is that the program prints out a wavelength for each combination of values of m and n, which is what we want.

This program works fine, but knowing what we do now, we can write a simpler version by making use of the `range` function, thus:

File: `rydberg.py`

```
R = 1.097e-2
for m in range(1,4):
    print("Series for m =",m)
    for n in range(m+1,m+6):
        invlambda = R*(1/m**2-1/n**2)
        print("  ",1/invlambda,"nm")
```

Note how we were able to eliminate the variable `k` in this version by specifying a range for `n` that depends directly on the value of `m`.

Exercise 2.9: The semi-empirical mass formula

In nuclear physics, the semi-empirical mass formula is a formula for calculating the approximate nuclear binding energy B of an atomic nucleus with atomic number Z and mass number A:

$$B = a_1 A - a_2 A^{2/3} - a_3 \frac{Z^2}{A^{1/3}} - a_4 \frac{(A-2Z)^2}{A} + \frac{a_5}{A^{1/2}},$$

where, in units of millions of electron volts, the constants are $a_1 = 15.8$, $a_2 = 18.3$, $a_3 = 0.714$, $a_4 = 23.2$, and

$$a_5 = \begin{cases} 0 & \text{if } A \text{ is odd,} \\ 12.0 & \text{if } A \text{ and } Z \text{ are both even,} \\ -12.0 & \text{if } A \text{ is even and } Z \text{ is odd.} \end{cases}$$

a) Write a program that takes as its input the values of A and Z, and prints out the binding energy for the corresponding atom. Use your program to find the binding energy of an atom with $A = 58$ and $Z = 28$. (Hint: The correct answer is around 500 MeV.)

b) Modify your program to print out not the total binding energy B, but the binding energy per nucleon, which is B/A.

c) Now modify your program so that it takes as input just a single value of the atomic number Z and then goes through all values of A from $A = Z$ to $A = 3Z$, to find the one that has the largest binding energy per nucleon. This is the most stable nucleus with the given atomic number. Have your program print out the value of A for this most stable nucleus and the value of the binding energy per nucleon.

d) Modify your program once more so that, instead of taking Z as input, it runs through all values of Z from 1 to 100 and prints out the most stable value of A for each one, along with the corresponding binding energy per nucleon. At what value of Z does the overall maximum binding energy per nucleon occur? (The true answer, in real life, is $Z = 28$, which is nickel.)

The nucleus with the maximum binding energy per nucleon is the most stable in the sense that creating any other state with the same number of nucleons, for instance by fissioning into smaller nuclei, would require the input of energy, and hence is never going to happen spontaneously.

2.6 USER-DEFINED FUNCTIONS

We saw in Section 2.2.5 how to use functions, such as `log` or `sqrt`, to do mathematics in our programs, and Python comes with a broad array of functions for performing all kinds of calculations. There are many situations in computational physics, however, where we need a specialized function to perform a particular calculation and Python allows you to define your own functions in such cases.

Suppose, for example, we are performing a calculation that requires us to calculate the factorials of integers. Recall that the factorial $n!$ of a positive integer n is defined as the product of all integers from 1 to n. We can calculate such a product in Python with a loop like this:

```
f = 1.0
for k in range(1,n+1):
    f *= k
```

When the loop finishes, the variable `f` will be equal to the factorial we want.[31]

If our calculation requires us to calculate factorials many times in various different parts of the program we could use a loop, as above, each time, but this would get tedious quickly and would increase the chances that we make an error. A more convenient approach is to define our own function to calculate the factorial, which we do like this:

```
def factorial(n):
    f = 1.0
    for k in range(1,n+1):
        f *= k
    return f
```

This definition consists of the word `def`, followed by the name we give our function, its argument in parentheses, and a colon. Then the rest of the lines contain the code

[31] We have chosen to make `f` a floating-point variable in this example, even though the factorial is an integer. We could use an integer variable, but factorials can be very large and for such large numbers floating-point calculations are usually faster in Python.

that performs the calculation of the function. Note how these lines are indented, in a manner similar to the if statements and for loops of previous sections. This indentation tells Python which lines are part of the function and where the function ends. The last line of the function consists of the word `return` followed by the value that is to be returned as the result of the function.

Once we have this definition, we can, anywhere later in the program, say

```
a = factorial(10)
```

or

```
b = factorial(r+2*s)
```

and the program will calculate the factorial of the appropriate number. In effect what happens when we write "`a = factorial(10)`"—when the function is *called*—is that the program jumps to the definition of the function (the part starting with `def` above), sets `n = 10`, and then runs through the instructions in the function. When it gets to the final line "`return f`" it jumps back to where it came from and the value of the `factorial` function is set equal to whatever quantity appeared after the word `return`—which is the final value of the variable `f` in this case. The net effect is that we calculate the factorial of 10 and set the variable `a` equal to the result. User-defined functions allow us to encapsulate complex calculations inside a single function definition and can make programs much easier to write and to read. We will see many uses for them in this book.[32]

An important point to note is that any variables created inside the definition of a function exist only inside that function. Such variables are called *local variables*. For instance the variables `f` and `k` in the factorial function above are local variables. This means we can use them only when we are inside the function and they disappear when we leave. Thus, for example, you could print the value of the variable `k` just fine if you put the print statement inside the function, but if you were to try to print the variable anywhere outside the function then you would get an error message telling you that no such variable exists.[33] Note, however, that the reverse is not true—you can use a variable inside a function that is defined outside it. The arguments of functions are local variables too, so the variable `n` in our factorial function does not exist outside the function. You are allowed to change the argument variable inside the function—it does not have to keep the value it arrived with—but if you do change it your changes will have no effect on anything outside the function.

[32]While the factorial is a nice example to illustrate user-defined function, note that there is actually a function in the `math` package, called `factorial`, that calculates factorials, so in practice it would usually be simpler to use that function than to write your own.

[33]To make things more complicated, you can separately define a variable called `k` outside the function and then you are allowed to print that variable (or do any other operation with it), but in that case it is a *different* variable—now you have two variables called `k` that have separate values and which value you get depends on whether you are inside the function or not.

User-defined functions can have more than one argument. As an example, suppose that the location of a point is specified in cylindrical coordinates r, θ, z, and we want to know the distance d between the point and the origin. The simplest way to do the calculation is to convert r and θ to Cartesian coordinates first, then apply Pythagoras' theorem to calculate d:

$$x = r\cos\theta, \qquad y = r\sin\theta, \qquad d = \sqrt{x^2 + y^2 + z^2}. \tag{2.5}$$

If we find ourselves having to do such a conversion many times within a program we might want to define a function to do it. Here is a suitable function in Python:

```
def distance(r,theta,z):
    x = r*cos(theta)
    y = r*sin(theta)
    d = sqrt(x**2+y**2+z**2)
    return d
```

(This assumes that we have already imported the functions `sin`, `cos`, and `sqrt` from the `math` package.)

Note how the function takes three arguments. When we call the function we must supply it with three values and they must come in the same order—r, θ, z—that they occur in the definition of the function. Thus if we say

```
D = distance(2.0,0.1,-1.5)
```

the program will calculate the distance for $r = 2$, $\theta = 0.1$, and $z = -1.5$. (If we give the wrong number of arguments for the function—one, or two, or four—we will get an error message.)

The values of function arguments can be of any type that Python knows about, including integers and real and complex numbers, but also including lists, arrays, or other objects. This allows us, for example, to create functions that perform operations on vectors or matrices stored in arrays. We will see examples of such functions when we look at linear algebra methods in Chapter 6.

The value returned by a function can also be of any type, including integer, real, complex, or a list or array. Using lists or arrays allows us to return more than one value if want to, or to return a vector or matrix. For instance, we might write a function to convert from polar coordinates to Cartesian coordinates like this:

```
def cartesian(r,theta):
   x = r*cos(theta)
   y = r*sin(theta)
   position = [x,y]
   return position
```

This function takes a pair of values r, θ and returns a two-element list containing the corresponding values of x and y. In fact, we could combine the two final lines here into one and say simply

```
return [x,y]
```

Or we could return x and y in the form of a two-element array by saying

```
return array([x,y],float)
```

An alternative way to return multiple values from a function is to use the "multiple assignment" feature of Python, which we examined in Section 2.2.4. We saw there that one can write statements of the form "`x,y = a,b`" which will simultaneously set `x = a` and `y = b`. The equivalent maneuver with a user-defined function is to write

```
def f(z):
    # Some calculations here...
    return a,b
```

which will make the function return the values of `a` and `b` both. To call such a function we write something like

```
x,y = f(1)
```

and the two returned values will get assigned to the variables `x` and `y`. One can also specify three or more returned values in this fashion, and the individual values themselves can again be lists, arrays, or other objects, in addition to single numbers, which allows functions to return very complex sets of values when necessary.

User-defined functions can also return no value at all—it is permitted for functions to end without a return statement. The body of the function is marked by indenting the lines of code and the function ends when the indentation does, whether or not there is a return statement. If the function ends without a return statement then the program will jump back to wherever it came from, to the statement where it was called, but without giving a value. Why would you want to do this? In fact there are many cases where this is a useful thing to do. For example, suppose you have a program that uses three-element arrays to hold vectors and you find that you frequently want to print out the values of those vectors. You could write something like

```
print("(",r[0],r[1],r[2],")")
```

every time you want to print a vector, but this is difficult to read and prone to typing errors. A better way to do it would be to define a function that prints a vector, like this:

```
def print_vector(r):
    print("(",r[0],r[1],r[2],")")
```

Then when you want to print a vector you simply say "`print_vector(r)`" and the computer handles the rest. Note how, when calling a function that returns no value you simply give the name of the function. One just says "`print_vector(r)`", and not "`x = print_vector(r)`" or something like that. This is different from the functions we are used to in mathematics, which always return a value. Perhaps a better name for functions like this would be "user-defined statements" or something similar, but by convention they are still called functions in Python.[34]

A `return` statement in a user-defined function can also occur in the middle of the function—it does not have to be at the end. And there does not have to be just one `return` statement. For example, the following function takes two arguments, x_1 and x_2, and returns +1, −1, or 0 depending on whether x_1 is greater than, less than, or equal to x_2, respectively:

```
def compare(x1,x2):
    if x1>x2:
        return 1
    if x1<x2:
        return -1
    return 0
```

The code defining a function—the code starting with the word `def`—can occur anywhere in a program, except that it must occur before the first time you use the function. It is good programming style to put all your function definitions (you will often have more than one) at or near the beginning of your programs. This guarantees that they come before their first use, and also makes them easier to find if you want to look them up or change them later.

One more trick is worth mentioning. The functions you define do not have to be in the same file on your computer as the rest of the program you are writing. You can, for example, place the definition for a function called `myfunction` in a separate file called `mydefinitions.py`. You can put the definitions for many different functions in the same file if you want. Then, when you want to use a function in a program, you say

```
from mydefinitions import myfunction
```

This tells Python to look in the file `mydefinitions.py` for the definition of `myfunction` and magically that function will now become available in your program. This is a very convenient feature if you have a function that you need to use in many differ-

[34] We have already seen one other example of a function with no return value, the standard `print` function itself.

ent programs: you need write the function only once and store it in a file, then you can import it into as many other programs as you like.

As you will no doubt have realized, this is what is happening when we say things like "`from math import sqrt`" in a program. Someone wrote a function called `sqrt` that calculates square roots and placed it in a file so that you can import it when you need it. The `math` package in Python is nothing other than a large collection of function definitions for useful mathematical functions, gathered together in one file.[35]

Example 2.9: Prime factors and prime numbers

Suppose we have an integer n and we want to know its prime factors. The prime factors can be calculated relatively easily by dividing repeatedly by all integers from 2 up to n and checking to see if the remainder is zero. Recall that the remainder after division can be calculated in Python using the modulo operation "`%`". Here is a function that takes an integer `n` as argument and returns a list of its prime factors:

```
def factors(n):
    factorlist = []
    k = 2
    while k<=n:
        while n%k==0:
            factorlist.append(k)
            n //= k
        k += 1
    return factorlist
```

This is a slightly tricky piece of code—make sure you understand how it does the calculation. Note how we have used the integer division operation "`//`" to perform the divisions, which ensures that the result returned is another integer. (Remember that the ordinary division operation "`/`" produces a float even when the numbers being divided are integers.) Note also how we change the value of the variable `n` (which is the argument of the function) inside the function. This is allowed: the argument variable behaves like any other variable and can be modified, although it is a local variable, so it exists only inside the function and gets erased when the function ends.

Now if we say "`print(factors(17556))`", the computer prints out the list of factors "`[2, 2, 3, 7, 11, 19]`". On the other hand, if we specify a prime number in the argument, such as "`print(factors(23))`", we get back "`[23]`"—the only prime factor of a prime number is itself. We can use this fact to make a program that prints

[35] In fact the functions in the `math` package are not written in Python—they are written in the C programming language, and one has to do some additional trickery to make these C functions work in Python, but the same basic principle still applies.

out the prime numbers up to any limit we choose by checking to see if they have only a single prime factor:

```
for n in range(2,10000):
    if len(factors(n))==1:
        print(n)
```

Run this program, and in a matter of seconds we have a list of the primes up to 10 000. (This is not a very efficient way of calculating primes—see Exercise 2.11 on page 80 for a faster way of doing it.)

2.6.1 RECURSION

Another useful feature of user-defined functions is *recursion*, the ability of a function to call itself. For example, consider the following definition of the factorial $n!$ of a positive integer n:

$$n! = \begin{cases} 1 & \text{if } n = 1, \\ n \times (n-1)! & \text{if } n > 1. \end{cases}$$

This constitutes a complete definition of the factorial which allows us to calculate the value of $n!$ for any positive integer. We can employ this definition directly to create an alternative Python function for factorials, like this:

```
def factorial(n):
    if n==1:
        return 1
    else:
        return n*factorial(n-1)
```

Note how, if n is not equal to 1, the function calls itself to calculate the factorial of $n-1$. This is recursion. If we now say "`print(factorial(5))`" the computer will correctly print the answer 120.

Why does this work? Consider what happens when we call the function for different values of n. If we call `factorial(1)` then it's simple: the function just returns the answer 1. If we call `factorial(2)`, then the function calls itself to calculate `factorial(1)`, which as we have said returns 1, then it multiplies that value by $n = 2$ to get $2! = 2 \times 1 = 2$, which is the correct result. If we call `factorial(3)`, then the function calls itself to calculate `factorial(2)`, which correctly returns 2 as we have said, then multiplies that value by 3 to get $3! = 3 \times 2 = 6$, which again is the correct result. And so forth. If the function can correctly calculate `factorial(n-1)`, then it can also correctly calculate `factorial(n)`, just by multiplying by an extra factor of n. Hence it can correctly calculate $n!$ for any n. (You can create a formal proof of correctness using mathematical induction, but it is not really necessary: it is already clear from the argument above why the program works.)

Thus, we have seen two different ways of calculating factorials, either directly or using recursion. In most cases, if a quantity can be calculated *without* recursion, then it will be faster to do so, and we normally recommend taking this route if possible. For this reason recursion is something of a specialized technique that finds only occasional use. However, there are some calculations that are essentially impossible (or at least much more difficult) without recursion, and for these it is a useful tool. We will see some examples later in this book.

Example 2.10: The Catalan numbers

The Catalan numbers C_n are a sequence of integers 1, 1, 2, 5, 14, 42, 132... that play important roles in quantum mechanics and the theory of disordered systems. We encountered them previously in Exercise 2.7 on page 45. With just a little rearrangement, the definition given there can be rewritten in the form

$$C_n = \begin{cases} 1 & \text{if } n = 0, \\ \dfrac{4n-2}{n+1} C_{n-1} & \text{if } n > 0. \end{cases}$$

This allows us to write a very simple Python function to calculate the Catalan numbers as follows:

```
def C(n):
    if n==0:
        return 1
    else:
        return (4*n-2)*C(n-1)//(n+1)
```

Notice how we use integer division to make sure the results are always integers. We can use this function to calculate the 100th Catalan number C_{100} thus:

```
print(C(100))
```

which prints

```
896519947090131496687170070074100632420837521538745909320
```

Exercise 2.10: Binomial coefficients

The binomial coefficient $\binom{n}{k}$ is an integer equal to

$$\binom{n}{k} = \frac{n!}{k!(n-k)!} = \frac{n \times (n-1) \times (n-2) \times \ldots \times (n-k+1)}{1 \times 2 \times \ldots \times k}$$

when $k \geq 1$, or $\binom{n}{0} = 1$ when $k = 0$.

a) Write a user-defined function `binomial(n,k)` that calculates the binomial coefficient for given integers n and k. Make sure your function returns the answer in the form of an integer (not a float) and gives the correct value of 1 for the case $k = 0$.

b) Using your function, write a program to print out the first 20 lines of "Pascal's triangle." The nth line of Pascal's triangle contains $n + 1$ numbers, which are the coefficients $\binom{n}{0}$, $\binom{n}{1}$, and so on up to $\binom{n}{n}$. Thus the first few lines are

1 1
1 2 1
1 3 3 1
1 4 6 4 1

c) The probability that an unbiased coin, tossed n times, will come up heads k times is $\binom{n}{k}/2^n$. Write a program to calculate (i) the total probability that a coin tossed 100 times comes up heads exactly 60 times, and (ii) the probability that it comes up heads 60 or more times.

Exercise 2.11: Prime numbers

The program in Example 2.9 is not a very efficient way of calculating prime numbers: it checks each number to see if it is divisible by any number less than it. We can write a much faster program for prime numbers by making use of the following observations:

a) A number n is prime if it has no prime factors less than n. Hence we need only check if it is divisible by other primes.

b) If a number n is non-prime, having a factor r, then $n = rs$, where s is also a factor. If $r \geq \sqrt{n}$ then $n = rs \geq \sqrt{n}s$, which implies that $s \leq \sqrt{n}$. In other words, any non-prime must have factors, and hence also prime factors, less than or equal to $\sqrt{n}$. Thus to determine if a number is prime we only have to check its prime factors up to and including $\sqrt{n}$. If there are none then the number is prime.

c) If we find even a single prime factor less than $\sqrt{n}$ then we know that the number is non-prime, and hence there is no need to check any further—we can abandon this number and move on to something else.

Write a Python program that prints out all the primes up to ten thousand. Create a list to store the primes, which starts out with just the one prime number 2 in it. Then for each number n from 3 to 10 000 check whether the number is divisible by any of the primes in the list up to and including $\sqrt{n}$. As soon as you find a single prime factor you can stop checking the rest of them—you know n is not a prime. If you find no prime factors $\sqrt{n}$ or less, then n is prime and you should add it to the list. You can print out the list all in one go at the end of the program, or you can print out the individual numbers as you find them.

Exercise 2.12: Greatest common divisor

Euclid showed that the greatest common divisor $g(m, n)$ of two nonnegative integers m and n satisfies

$$g(m,n) = \begin{cases} m & \text{if } n = 0, \\ g(n, m \bmod n) & \text{if } n > 0. \end{cases}$$

Write a Python function `g(m,n)` that employs recursion to calculate the greatest common divisor of m and n using this formula. Use your function to calculate and print the greatest common divisor of 108 and 192.

2.7 Good programming style

When writing a program to solve a physics problem there are, usually, many ways to do it, many programs that will give you the solution you are looking for. For instance, you can use different names for your variables, use either lists or arrays for storing sets of numbers, break up the code by using user-defined functions to do some operations, and so forth. Although all of these approaches may ultimately give the same answer, not all of them are equally satisfactory. There are well written programs and poorly written ones. A well written program will, as far as possible, have a simple structure, be easy to read and understand, and, ideally, run fast. A poorly written one may be convoluted or unnecessarily long, difficult to follow, or may run slowly. Making programs easy to read is a particularly important—and often overlooked—goal. An easy-to-read program makes it easier to find problems, easier to modify the code, and easier for other people to understand how things work.

Good programming is, to some extent, a matter of experience, and you will quickly get the hang of it as you start to write programs. But here are a few general rules of thumb that may help.

1. **Use meaningful variable names.** Give your variables names that help you remember what they represent. The names don't have to be long. In fact, very long names are usually harder to read. But choose your names sensibly. Use `E` for energy and `t` for time. Use full words where appropriate or even pairs of words to spell out what a variable represents, like `mass` or `angular_momentum`. If you are writing a program to calculate the value of a mathematical formula, give your variables the same names as in the formula. If variables are called x and β in the formula, call them `x` and `beta` in the program.
2. **Use the right types of variables.** Use integer variables to represent quantities that actually are integers, like vector indices or quantum numbers. Use floats and complex variables for quantities that really are real or complex numbers.
3. **Include comments in your programs.** Leave comments in the code to remind yourself what particular variables mean, what calculations are being performed in different sections of the code, what arguments functions require, and so forth. It is amazing how you can come back to a program you wrote only a week ago and not remember how it works. You will thank yourself later if you include comments. And comments are even more important if you are writing programs that other people will have to read and understand. It is frustrating to be the person who has to fix or modify someone else's code if they neglected to include any comments to explain how it works.
4. **Import functions first.** If you are importing functions from packages, put your import statements at the start of your program. This makes them easy to find if you need to check them or add to them, and it ensures that you import functions before the first time they are used.

5. **Give your constants names.** If there are constants in your program, such as the number of atoms N in a gas or the mass m of a particle, create suitably named variables at the beginning of your program to represent these quantities, then use those variables wherever those quantities appear in your program. This makes formulas easier to read and understand and it allows you to later change the value of a constant by changing only a single line at the beginning of the program, even if the constant appears many times throughout your calculations. Thus, for example, you might have a line "`A = 58`" that sets the atomic mass of an atom for a calculation at the beginning of the program, then you would use `A` everywhere else in the program that you need to refer to the atomic mass. If you later want to perform the same calculation for atomic mass 59, you need only change the single line at the beginning to "`A = 59`". Most physics programs have a section near the beginning (usually right after the import statements) that defines all the constants and parameters of the program, making them easy to find when you need to change their values.
6. **Employ user-defined functions, where appropriate.** User-defined functions can usefully encapsulate repeated operations, especially complicated operations, and can greatly increase the legibility of your code. Avoid overusing them, however: simple operations, ones that can be represented by just a line or two of code, are often better left in the main body of the program. It makes the flow of the calculation easier to follow and may also make the program faster, since there is a (small) time cost to using any function. Normally you should put your function definitions at the start of your program, probably after imports and constant definitions. This ensures that each function definition appears before the first use of the function and that the definitions can be easily found and modified when necessary.
7. **Print out partial results and updates throughout your program.** Large computational physics calculations can take a long time—minutes, hours, or even days. You will find it helpful to include print statements in your program that print updates about where the program has got to or partial results from the calculations, so you know how the program is progressing. It is difficult to tell whether a calculation is working correctly if the computer simply sits silently, saying nothing, for hours on end.

   Thus, for example, if there is a for loop in your program that repeats many times, it could be useful to include code like this at the beginning of the loop:

```
for n in range(1000000):
    if n%1000==0:
        print("Step",n)
```

   These lines will cause the program to print out what step it has reached every time `n` is exactly divisible by 1000, i.e., every thousandth step. So it will print:

```
Step 0
Step 1000
Step 2000
Step 3000
```

and so forth as it goes along.

8. **Lay out your programs clearly.** You can add spaces or blank lines in most places within a Python program without affecting the operation of the program and doing so can improve readability. Make use of blank lines to split code into logical blocks. Make use of spaces to divide up complicated algebraic expressions or particularly long program lines.

   You can also split long program lines into more than one line if necessary. If you place a backslash symbol "\" at the end of a line it tells the computer that the following line is a continuation of the current one, rather than a new line in its own right. Thus, for instance you can write:

   ```
   energy = mass*(vx**2 + vy**2 + vz**2)/2 + mass*g*z \
            + moment_of_inertia*omega**2/2
   ```

   and the computer will interpret this as a single formula. If a program line is very long indeed you can spread it over three or more lines with backslashes at the end of each one, except the last.[36]

9. **Don't make your programs unnecessarily complicated.** A short simple program is enormously preferable to a long involved one. If the job can be done in ten or twenty lines, then it is probably worth doing it that way—the code will be easier to understand, for you or anyone else, and if there are mistakes in the program it will be easier to work out where they lie.

Good programming, like good science, is a matter of creativity as well as technical skill. As you gain more experience with programming you will no doubt develop your own programming style and learn to write code in a way that makes sense to you and others, creating programs that achieve your scientific goals quickly and elegantly.

[36] Under certain circumstances, you do not need to use a backslash. If a line does not make sense on its own but it does make sense when the following line is interpreted as a continuation, then Python will automatically assume the continuation even if there is no backslash character. This, however, is a complicated rule to remember, and there are no adverse consequences to using a backslash even when it is not strictly needed, so in most cases it is simpler just to use the backslash and not worry about the rules.

Chapter summary

- Python is a modern programming language that is powerful, widely used, free, and well suited to computational physics. Python programs consist of a sequence of **statements**, normally carried out in order one after another, that specify elementary operations.
- Python programming is typically performed in a **development environment**, a program that allows you to enter, edit, and run programs. Two examples are **IDLE**, a simple environment that runs on any computer, and **Jupyter**, which runs in your web browser. **Colab** is a version of Jupyter that runs entirely on the web and does not require any software installation.
- Numerical values in programs are represented by **variables**, which play a similar role to variables in algebra. Python variables come in several basic types: **integer** (called `int` within programs), **floating-point** (`float`), **complex numbers** (`complex`), and **strings** (`str`). The type of a variable is not normally specified explicitly; it is determined by the value you give the variable.
- Basic arithmetic in Python looks similar to conventional algebra, with operations such as `+`, `-`, `*`, `/`, and functions such as `sin`, `cos`, `log`, and `sqrt`. A few functions are **built in** to Python and always available, but most, including most mathematical functions, must be **imported** from **packages** before use.
- **Comments** allow the programmer to leave textual messages within a program to document what it is doing or how it works. Comments in Python are indicated by the hash character "#".
- **If** and **while** statements allow sections of code to be executed if a variable takes a certain value or some similar logical condition applies. The section of code to be executed is denoted by **indentation**.
- Collections of numbers, such as vectors or matrices, can be stored in containers of various kinds. **Lists**, as the name suggests, are lists of values, one after another. Lists can contain a mix of different types of values—integers, floats, etc.—and can grow or shrink with the addition and removal of elements.
- **Arrays** are similar to lists but can only contain a single type of value and their length is fixed and cannot change. On the other hand, arrays can be two-dimensional (to represent matrices), and they allow fast vector and matrix arithmetic operations.
- Other containers include **sets**, which are unordered collections of values, and **dictionaries** or dicts, in which each value is referred to by a unique index.
- **For loops** allow one to execute a section of code repeatedly. For loops in Python work by iterating through the items in a container such as a list or array. The

number of times the section of code is executed is equal to the number of elements in the list or array. Once again the section is indicated by indentation.

- **User-defined functions** provide a way to add new functions to Python programs, including mathematical functions but also functions that perform complex program operations. A **recursive** function is one that calls itself, a useful feature that allows one to express certain operations more simply than one otherwise would be able to.