كليـة الحاسبات والمعلومات

Computer Applications

in

Physics (II)

CONTENTS

Preface **x**

1 Introduction **1**

2 Python programming for physicists **9**
2.1 Getting started 9
2.2 Basic programming 12
2.2.1 Variables and assignments 12
2.2.2 Variable types 14
2.2.3 Output and input statements 18
2.2.4 Arithmetic 23
2.2.5 Functions, packages, and modules 31
2.2.6 Built-in functions 35
2.2.7 Comment statements 37
2.3 Controlling programs with "if" and "while" 39
2.3.1 The if statement 39
2.3.2 The while statement 42
2.3.3 Break and continue 43
2.4 Lists and arrays 46
2.4.1 Lists 47
2.4.2 Arrays 53
2.4.3 Reading an array from a file 57
2.4.4 Arithmetic with arrays 58
2.4.5 Slicing 66
2.5 "For" loops 67
2.6 User-defined functions 75
2.7 Good programming style 84

3 Graphics and visualization **88**
3.1 Graphs 88

3.2 Scatter plots 99
3.3 Density plots 102
3.4 3D graphics 111
3.5 Animation 117

4 Accuracy and speed **126**
4.1 Variables and ranges 126
4.2 Numerical error 128
4.3 Program speed 134

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.[1] 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 calculations with vectors and matrices, 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. Luckily there is excellent documentation available on-line, so if there's something you want to do and it's not explained in this book, I encourage you to see what you can find. 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'll 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.

[1] Some of these also require that you have a good idea.

When you are programming in Python—developing a program, as the jargon goes—you typically work in a *development environment*, which is a window or windows on your computer screen that show the program you are working on and allow you to enter or edit lines of code. There are several different development environments available for use with Python, but the most commonly used is the one called IDLE.[2] 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.[3]) 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 chapter 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 3.2 (default, Sep 29 2012)
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 individual Python commands when you're not sure how something works, but it's not the main way that we will use Python commands. 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

[2] 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.

[3] The standard versions of Python for PC and Mac computers come with IDLE. For Linux users, IDLE does not usually come installed automatically, so you may have to install it yourself. The most widely used brands of Linux, including Ubuntu and Fedora, have freely available versions of IDLE that can be installed using their built-in software installer programs.

the screen, this one completely empty. This is an *editor window*. It behaves differently from the Python shell window. You type a complete program into this window, usually consisting of many lines. You can edit it, add things, delete things, cut, paste, 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). This is the main way in which we will use Python and IDLE in this book.

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 (useless) two-line program into the window, just as it appears here:

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

(If it's not obvious what this does, 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.[4] The names of all Python programs must end with "`.py`", so a suitable name might be "`example.py`" or something like that. (If you do not give your program a name ending in "`.py`" then the computer will not know that 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

[4]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.

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.

This same procedure is the one you'll use for running all your programs and you'll get used to it soon. It's a good idea to save your programs, as here, when they're finished and ready to run. If you forget to do it, IDLE will ask you if you want to save before it runs your program.

IDLE is by no means the only development environment for Python. If you are comfortable with computers and enjoy trying things out, there are a wide range of others available on the Internet, mostly for free, with names like PyDev, Eric, BlackAdder, Komodo, Wing, and more. Feel free to experiment and see what works for you, or you can just stick with IDLE. IDLE can do everything we'll 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.

2.2 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, so that 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 in most other programming languages) they don't have to be—they can be two, three, or more letters, or entire words if you want. 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 a `&`). Upper- and lower-case letters are distinct from one another, meaning that `x` and `X` are two different variables which can have different values.[5]

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`. Of course, there will 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's no reason why you can't 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.

[5] Also variables cannot have names that are "reserved words" in Python. Reserved words are the words used to assemble 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.)

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 the following:

- **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.
- **Float**: 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. Notice 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 $1 + 2\text{j}$ or $-3.5 - 0.4\text{j}$. 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 great advantages to having the different types. For instance, the values of the variables in 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 in writing the program.

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 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. Luckily there is a simple way to avoid it. If the quantity you're dealing with is genuinely an integer, then you should 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 computers: 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 an excellent idea 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 your variable to match it.

And 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.[6] 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.[7]

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. This doesn't quite tell us the whole story, however, because as we've said a

[6] 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 $1 + 2$i.

[7] If you have previously programmed in one of the so-called static-typed languages, such as C, C++, Fortran, or Java, then you'll 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.

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's 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.

A slightly more complicated 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 also achieves the goal of making `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 programs 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:

```
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's 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 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

(and a decimal point) in this case, but they could just as easily be letters or spaces or punctuation. If you try to do arithmetic 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 statements we have seen so far you are free to put spaces between parts of the statement. Thus "`x=1`" and "`x = 1`" do the exact same thing; the spaces have no effect. They can, however, do much to 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 the 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 the program works. Thus, 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, which takes the form "`x = 1`" or something similar. The next types of statements we will examine are statements for output and input of data in Python programs. We have already seen one example of the basic output statement, or "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 12.

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, though they look identical, produce different results in this case. Note also that each print statement starts its printing on a new line. 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 that 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 much easier to read and understand. You can also have print statements that print out only words if you like, 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 string of quantities separated by commas, or text in quotation marks, and the computer will simply 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 something like 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)
```

[8]The print statement is one of the things that 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 B for further discussion of the differences between Python versions.

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 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. It is waiting 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 in 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:

[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.

[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.

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

In fact, one doesn't 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 this program 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 if I enter the wrong thing, I get 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 rather opaque technical jargon, is complaining that we have given it an incorrect value.

It's 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 business—they are often not entirely transparent.

2.2.4 ARITHMETIC

So far our programs have done very little, certainly nothing that would be 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 $\times$ or $\div$ 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`	modulo, 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 nonintegers: `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 a few other mathematical operations available in Python as well, but they're 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 types of the variables 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 the end result will be that `x` is a floating-point variable 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` would 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 the variables `a` and `b` that went into it 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 variable, the end result will be complex.

The same rules apply to subtraction, multiplication, integer division, and the modulo operation: the end result is the same type as the starting values, or the more general type if there are two different starting types. The division operation, however—ordinary non-integer division denoted by "/"—is slightly different: it follows basically the same rules except that it never gives an integer result. Only floating-point or complex values result from division. This is necessary because you can divide one integer by another and get a noninteger result (like $3 \div 2 = 1.5$ for example), so it wouldn't make sense to have integer starting values always give an integer final result.[12] Thus if you divide any

[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

[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 a lot of confusion for exactly the reason given here: if you divided 3 by 2, for

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)`. And you can add spaces between the parts of a mathematical expression to make it easier to read; the spaces don't 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)
```

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 simply stop when it gets to that line and print a typically cryptic error message—"`SyntaxError: can't assign to operator`"—because it

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 B.

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, when it has calculated this value it assigns it 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 often very useful) effect of increasing the value of `x` by one.

Thus consider the following lines:

```
x = 0
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 0 and then the 0 gets printed out. But then what? The third line says "`x = x**2 - 2`" which in nor-

mal 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 $x^2 - 2 = 0^2 - 2 = -2$ and then set x to this new value. Then the last line of the program will print out "-2".

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 won't 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 calculates the whole of the right-hand side of the equation 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 calculated 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 for ourselves on paper, but don't worry—we will move onto more complex problems shortly.

The problem is as follows. A ball is dropped from a tower of height h. It has initial velocity zero and accelerates downwards 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 will use `input` statements to get the values of h and t from the user. Second, we will 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's 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")
```

[13]Many of the example programs in this book are also available on-line for you to download and run on your own computer if you wish. The programs, along with various other useful resources, are packaged together in a single "zip" file (of size about nine megabytes) which can be downloaded from `http://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 this file. Any mention of programs or data in the "on-line resources" also refers to the same file.

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
```

Notice that the result is negative in the second case, which means that the ball would have fallen to below ground level if that were possible, though 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's a suggestion for a possible improvement to the program above. 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 don't 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's 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 in algebra makes life a lot easier.

The same is also true of computer programs. You can make your programs substantially 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. Doing this also has the advantage of decreasing the chances that

you'll 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, just so long as you typed it correctly when you first created the variable.[14]

Using variables to represent constants in this way is one example of a programming trick that improves your programs even though it doesn't change the way they actually work. Instead it improves readability and reliability, which can be almost 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, $M = 5.97 \times 10^{24}\,\mathrm{kg}$ is the mass of the Earth, and $R = 6371\,\mathrm{km}$ 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?

---

[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.

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 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 disk 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. 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 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 miss out 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:

`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. And 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.

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 `9.86960440109` (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's 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 from statements, like this:

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

but a more succinct way to do it is to use a single line 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".[15] In most cases, however, I 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's usually better to explicitly import only those functions you actually need to use.[16]

Finally, some large packages are for convenience split into smaller subpackages, called *modules*. A module within a larger package is referred to as `packagename.modulename`. As we will see shortly, for example, 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 can 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.

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

[15] There is also another way to import the entire contents of a package in Python, with a statement of the form "`import math`". If you use this form, however, then when you subsequently use one of the imported functions you have to write, for example, `x = math.log(2.5)`, instead of just `x = log(2.5)`. Since the former is more complicated and annoying, it gets used rather rarely. Moreover the existence of the two types of import, and particularly their simultaneous use in the same program, can be quite confusing, so we will use only the "from" form in this book.

[16] A particular problem is 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 be selected in favor of 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.

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:
$$x = r\cos\theta, \qquad y = r\sin\theta. \tag{2.1}$$
3. Print out the results.

Since the formulas (2.1) involve the mathematical functions sin and cos we are going to have to import those functions from the `math` package at the start of the program. 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 if you like.)

2.2.6 BUILT-IN FUNCTIONS

There are a small number of functions in Python, called *built-in functions*, which don't 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's 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 haven't 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)
```

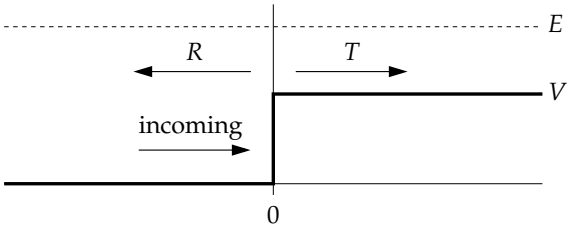
print is a function, but it is not here generating a value the way the log or input functions do. It does something with its argument `x`, namely printing it out on the screen, but it does not generate a value. This differs from the functions we are used to in mathematics, but it's allowed in Python. 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, and calculate and print the corresponding polar coordinates, with the angle θ given 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 from these two 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, $M = 1.9891 \times 10^{30}\,\mathrm{kg}$ is the mass of the Sun, and $G = 6.6738 \times 10^{-11}\,\mathrm{m^3\,kg^{-1}\,s^{-2}}$ 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 $\ell_1 = 1.4710 \times 10^{11}\,\mathrm{m}$ and $v_1 = 3.0287 \times 10^4\,\mathrm{m\,s^{-1}}$) and Halley's comet ($\ell_1 = 8.7830 \times 10^{10}\,\mathrm{m}$ and $v_1 = 5.4529 \times 10^4\,\mathrm{m\,s^{-1}}$). 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.

2.2.7 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're 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's 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 understand how it works.

Comments may seem unnecessary for short programs like this one, but when you get 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're 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'll 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're 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 don't 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 $x > \frac{1}{2}$. 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 I run this program and type in "5", I get:

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

But if I break the rules and enter 11, I 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 applying. 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 a message 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. However, it has over the years become standard practice among Python programmers to use four spaces for an indentation, and this is the number used in all the programs in this book. In fact, most Python development environments, including IDLE, automatically insert the spaces for you when they see an if statement, and they typically insert 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'll 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 statement `elif` means "else if"—if the first criterion is not met it tells the computer to try a different one. Notice that we can use both `elif` and `else`, 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.[17]) Thus, if I were to run the snippet of code above, I 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
```

[17] If you accidentally create a program with a loop that goes on for ever then you'll need to know how to stop the program: just closing the window where the program is running does the trick.

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`.

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 within statements is allowed. When we get onto 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.[18]

A variant on the idea of the break statement is the continue statement. 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, 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. (The continue statement turns out to be used rather rarely in practice. The break statement, on the other hand, gets used often and is definitely worth knowing about.)

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 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's 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's 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:
```

[18] 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. (If this doesn't make sense to you, don't worry—it will become clear later when we look at examples with more than one loop.)

```
        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 will keep 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, 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 quite 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's 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. At all times the variables `f1` and `f2` store the two most recent elements of the sequence. If `f1` is less than 1000, we print it out, then calculate the next element of the sequence 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's 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 an important role 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 quantities in Python and how to use variables to store those quantities. All the variables we have

seen so far, however, represent only a single value, a single integer, real, or complex number. But in physics it is common for a variable to represent several numbers at once. We might use a vector **r**, for instance, to represent the position of a point in three-dimensional space, meaning that the single symbol **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 as say 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 two of them, lists and arrays.[19]

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 have to be all 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'll deal with, however, the elements will be all of the same type—all integers, say, or all floats—because this is what physics calculations usually demand. Thus, for instance, in the example described above where we make a hundred measurements of a given quantity in the lab and we want to represent them on the computer, we could use a list one hundred elements long, and all the elements would presumably

[19]There are several others as well, the main ones being *tuples*, *dicts*, and *sets*. These, however, find only occasional use in physics calculations, and we will not use them in this book.

[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.

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 this particular list are all integers. Note that the elements are separated by commas and the whole list is surrounded by square brackets. Another example of a list might be `[ 1, 2.5, 3+4.6j ]`. This example has three elements of different types, one integer, one real, and one complex.

A variable can be set equal to a list:

```
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. But 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 a little 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'll 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 use a single 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

```
2.84429253067
```

which is the correct answer (to twelve 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 commonly 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
```

Some other useful built-in functions are `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.

A special, and especially useful, function for lists is the function `map`, which is a kind of meta-function—it allows you apply ordinary functions, like `log` or `sqrt`, to all the elements of a list at once. Thus `map(log,r)` takes the natural logarithm of each element of a list `r` in turn. More precisely, `map` creates a

specialized object in the computer memory, called an *iterator*, that contains the logs, one after another in order. Normally we will want to convert this iterator into a new list, which we can do with the built-in function `list`.[21] Thus, consider this snippet of code:

```
from math import log
r = [ 1.0, 1.5, 2.2 ]
logr = list(map(log,r))
print(logr)
```

This will create a list `logr` containing the logs of the three numbers 1, 1.5, and 2.2, and print it out thus:

```
[0.0, 0.4054651081, 0.7884573603]
```

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 the ones we've seen previously.[22] 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 and set that element equal to the value given, which is 6.1 in this case. The value can also be specified using a variable or a mathematical expression, thus:

[21] The difference between an iterator and a list is that the values in an iterator are not stored in the computer's memory the way the values in a list are. Instead, the computer calculates them on the fly when they are needed, which saves memory. Thus, in this case, the computer only calculates the logs of the elements of `r` when you convert the iterator to a list. In versions of Python prior to version 3, the `map` function produced a list, not an iterator, so if you are using an earlier version of the language you do not need to convert to a list using the `list` function. You can just say "`logr = map(log,r)`". For further discussion of this point, and of iterators in general, see Appendix B.

[22] 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 doesn't 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.

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

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.[23] Bear in mind that the elements are 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 won't 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

[23] 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. (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.)

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—far more than we will use lists.

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 that allow you do this in several different ways. These functions are all found in the package `numpy`.[24]

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 could similarly 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 lim-

[24] The word `numpy` is short for "numerical Python."

ited 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 `zeros` 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 also 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 aren't 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:

```
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. 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.[25,26]

The two lines above can conveniently be combined into one, like this:

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

This is a quick and neat 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 initial values. To do this we again use the `array` function, but now the argument we give it must be a *list of lists*, which gives the elements of the array row by row. For example, we can write

```
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 `a` 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:

[25] Two caveats apply here. (1) If you create an integer array from a list that has any floating-point elements, the fractional part, if any (i.e., the part after the decimal point), of the floating-point elements 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.

[26] Though it's not something we will often need to do, you can also convert an array into a list using the built-in function `list`, thus:

```
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.

```
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, the fractional part of the value will be deleted. (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]]
```

(You should check this example and make sure you understand why it does what it does.)

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 float-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 is saved in.[27])

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 the `loadtxt` function to transfer those numbers into an array so that we can perform further calculations on them.

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
```

[27] 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.

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 enormously useful in physics calculations. In general, when doing arithmetic with entire 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. Notice that when we run this program, the computer creates a new array `b` holding the results of our 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 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.[28] If we represent a vector using an array then arithmetic operations such as multiplying or dividing by a scalar quantity or adding or subtracting vectors can be written just as they would in normal mathematics. You can also add a scalar quantity to an array (or subtract one), 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 here? 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. Each element of the first array is multiplied by the corresponding element of the second. (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. This can be calculated 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))
```

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

The function `dot` takes two arrays as arguments and calculates their dot product, which would be 60 in this case.

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.

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. The `map` function also works, applying any ordinary function to all elements of a one-dimensional array and producing an iterator (see Section 2.4.1), which can then be converted into a list with the `list` function, or into an array using the `list` function then the `array` function:

```
b = array(list(map(sqrt,a)),float)
```

This will create an iterator whose elements are the square roots of the elements of `a`, then convert the iterator into an array `b` containing those values.

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)
```

That is, `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 along each axis. (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 not really any 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 on-line 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 is calculated by multiplying 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` again 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'll 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:

$$\overline{x} = \left[\prod_{i=1}^{n} x_i\right]^{1/n}. \tag{2.2}$$

Taking natural logs of both sides we get

$$\ln \overline{x} = \ln\left[\prod_{i=1}^{n} x_i\right]^{1/n} = \frac{1}{n}\sum_{i=1}^{n} \ln x_i \tag{2.3}$$

or

$$\overline{x} = \exp\left(\frac{1}{n}\sum_{i=1}^{n} \ln x_i\right). \tag{2.4}$$

In other words, the geometric mean is the exponential of the arithmetic mean of the logarithms. We can modify our previous program for the arithmetic mean to calculate the geometric mean thus:

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

Note how we combined the `map` function and the `log` function to calculate the logarithms and then calculated the arithmetic mean of the resulting values.

If we want to be clever, we can streamline this program by noting that the `numpy` package contains its own `log` function that will calculate the logs of all the elements of an array. Thus we can rewrite our program as

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

As well as being more elegant, this version of the program will probably also run a little faster, since the `log` function in `numpy` is designed specifically to work efficiently with arrays.

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 is probably not 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 gone wrong?

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 it to. 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 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 mem-

ory 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.[29]

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 work like 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's 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 separate 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.

[29] 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 trying it on the computer.)

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

2.4.5 SLICING

Here's 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's an example:

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

which produces

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

Observe what happens here. 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 can be useful in many physics calculations, particularly, as we'll 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.

Slicing also works with arrays: you can specify a subpart of an array and you get another array, of the same type as the first, as in this 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.

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]`. And `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]`.

Finally, `a[2,:]` gives you the whole of row 2 of array `a`, which means the third row since the numbering starts at zero. And `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.

2.5 "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. It turns out, however, that 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 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 out the following:

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

What's happening here is as follows. The program first creates a list called `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 round the loop. When the loop 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." Thus in this example the computer will go around the loop three times, since there are three elements in the 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. Also the statements `break` and `continue` (see Section 2.3.3) can be used with for loops the same way they're 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.

The most common use of the for loop is simply to run through a given 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)` gives 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.

(Technically, `range` produces an iterator, not a list. Iterators were discussed previously in Section 2.4.1. If you actually want to produce a list using `range` then you would write something of the form "`list(range(5))`", which creates an iterator and then converts it to a list. In practice, however, we do this very rarely, and never in this book—the main use of the `range` function is in for loops and you are allowed to use an iterator directly in a for loop without

converting it into a list first.[30])

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)
```

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

[30]In versions of Python prior to version 3, the `range` function actually did produce a list, not an iterator. If you wanted to produce an iterator you used the separate function `xrange`, which no longer exists in version 3. Both list and iterator give essentially the same results, however, so the for loops in this book will work without modification with either version of Python. For further discussion of this point, and of iterators in general, see Appendix B.

writes something like

```
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 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)
```

Notice how the upper limit of the range must be 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 doesn't work if you give it noninteger 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 them you will just get an error message because, as discussed in Section 2.2.4, the division operation always returns a floating-point quantity, 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 an integer division instead:

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

This will now work as expected. (See Section 2.2.4, page 23 for a discussion of integer division.)

Another useful function is the function arange from the numpy package, which is similar to range but generates arrays, rather than lists or iterators[31] and, more importantly, 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] as we would expect, but 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, arange 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.

[31] 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(10000000000) will produce an error message on most computers, while the equivalent expression with range will not. See Appendix B for more discussion of this point.

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 situations, however, it is often more convenient to use a for loop to calculate a sum. Suppose, for instance, 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 equal to zero. As above we'll call the variable `s`, and we want it to be a floating-point variable, so we'd do "`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:

```
5.1873775176
```

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's often the case that one can use either a for loop or a function or similar array operation to perform the same calculation. In general for loops are more flexible, but direct array operations 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: 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

$$\frac{1}{\lambda} = R\left(\frac{1}{m^2} - \frac{1}{n^2}\right). \tag{2.5}$$

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")
```

We can now understand exactly how this program works. It uses two nested for loops—a loop within another loop—with the code inside 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. And 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 will be that the program prints out a wavelength for each combination of values of m and n, which is what we want.

In fact, knowing what we now know, we can write a simpler version of this program, 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 Madelung constant

In condensed matter physics the Madelung constant gives the total electric potential felt by an atom in a solid. It depends on the charges on the other atoms nearby and their locations. Consider for instance solid sodium chloride—table salt. The sodium chloride crystal has atoms arranged on a cubic lattice, but with alternating sodium and chlorine atoms, the sodium ones having a single positive charge $+e$ and the chlorine ones a single negative charge $-e$, where e is the charge on the electron. If we label each position on the lattice by three integer coordinates (i, j, k), then the sodium atoms fall at positions where $i + j + k$ is even, and the chlorine atoms at positions where $i + j + k$ is odd.

Consider a sodium atom at the origin, $i = j = k = 0$, and let us calculate the Madelung constant. If the spacing of atoms on the lattice is a, then the distance from the origin to the atom at position (i, j, k) is

$$\sqrt{(ia)^2 + (ja)^2 + (ka)^2} = a\sqrt{i^2 + j^2 + k^2},$$

and the potential at the origin created by such an atom is

$$V(i, j, k) = \pm \frac{e}{4\pi\epsilon_0 a\sqrt{i^2 + j^2 + k^2}},$$

with ϵ_0 being the permittivity of the vacuum and the sign of the expression depending on whether $i + j + k$ is even or odd. The total potential felt by the sodium atom is then the sum of this quantity over all other atoms. Let us assume a cubic box around the sodium at the origin, with L atoms in all directions. Then

$$V_{\text{total}} = \sum_{\substack{i,j,k=-L \\ \text{not } i=j=k=0}}^{L} V(i, j, k) = \frac{e}{4\pi\epsilon_0 a} M,$$

where M is the Madelung constant, at least approximately—technically the Madelung constant is the value of M when $L \to \infty$, but one can get a good approximation just by using a large value of L.

Write a program to calculate and print the Madelung constant for sodium chloride. Use as large a value of L as you can, while still having your program run in reasonable time—say in a minute or less.

Exercise 2.10: 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 again 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. At what value of Z does the maximum binding energy per nucleon occur? (The true answer, in real life, is $Z = 28$, which is nickel.)

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 of n is defined as the product of all integers from 1 to n:

$$n! = \prod_{k=1}^{n} k. \tag{2.6}$$

We can calculate this 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.

If our calculation requires us to calculate factorials many times in various different parts of the program we could write out a loop, as above, each time, but this could get tedious quickly and would increase the chances that we would 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
```

Note how the lines of code that define the function are indented, in a manner similar to the if statements and for loops of previous sections. The indentation tells Python where the function ends.

Now, anywhere later in the program we can simply say something like

```
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.

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.[32] Note, however, that the reverse is not true—you can use a variable inside a function that is defined outside it.

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.

User-defined functions can have more than one argument. Suppose, for example, we have a point 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:

$$x = r\cos\theta, \qquad y = r\sin\theta, \qquad d = \sqrt{x^2+y^2+z^2}. \tag{2.7}$$

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's 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
```

[32] To make things more complicated, you can separately define a variable called `k` outside the function and then you are allowed to print that (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.

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

Note how the function takes three different arguments now. When we call the function we must now supply it with three different arguments and they must come in the same order—r, θ, z—that they come in 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 $z = -1.5$.

The values used as arguments for a function can be any type of quantity Python knows about, including integers and real and complex numbers, but also including, for instance, lists and arrays. 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 would 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, 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.[33]

The definition of a user-defined function—the code starting with `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. Once defined, your functions can be used in the same way as any other Python functions. You can use them in mathematical expressions. You can use them in print statements. You can even use them inside the definitions of other functions. You can also apply a user-defined function to all the elements of a list or array with the `map` function. For instance, to multiply every element of a list by 2 and subtract 1, we could do the following:

```
def f(x):
    return 2*x-1

newlist = list(map(f,oldlist))
```

This applies the function `f` to every element in turn of the list `oldlist` and makes a list of the results called `newlist`.

One more trick is worth mentioning, though it is more advanced and you should feel free to skip it if you're not interested. The functions you define do not have to be saved in the same file on your computer as your main program. You could, for example, place a function 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` to find the definition of `myfunction` and magically that function will now become available in your program. This is a very convenient feature if you write a function that you

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

want to use in many different programs: you need write it 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 by now, 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 whenever 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.[34]

EXAMPLE 2.8: 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 the number 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 "`/`" always produces a float.) 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, like all variables inside functions, it is a local variable—it exists only inside the

[34] In fact the functions in the `math` package aren't written in Python—they're 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.

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 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, however, not a very efficient way of calculating the primes—see Exercise 2.12 for a faster way of doing it.)

Exercise 2.11: 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) Using this form for the binomial coefficient, write a Python user-defined function `binomial(n,k)` that calculates the binomial coefficient for given 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 where $k = 0$.

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 (a) the total probability that a coin tossed 100 times comes up heads exactly 60 times, and (b) the probability that it comes up heads 60 or more times.

Exercise 2.12: Prime numbers

The program in Example 2.8 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 develop 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 only need to 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 finds 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 2.13: Recursion

A 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 a 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.

a) We encountered the Catalan numbers C_n previously in Exercise 2.7 on page 46. With just a little rearrangement, the definition given there can be rewritten in the form

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

Write a Python function, using recursion, that calculates C_n. Use your function to calculate and print C_{100}.

b) 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.

Comparing the calculation of the Catalan numbers in part (a) above with that of Exercise 2.7, we see that it's possible to do the calculation two ways, 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. There are some calculations, however, that are essentially impossible (or at least much more difficult) without recursion. We will see some examples later in this book.

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. **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's 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's 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.
2. **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.
3. **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.
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 similarly 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.** Use user-defined functions to represent repeated operations, especially complicated operations. Functions can greatly increase the legibility of your program. 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, because it allows you to follow the flow of the calculation when reading through your program without having to jump to a function and back. Normally you should put your function definitions at the start of your program (probably after imports and constant definitions). This ensures that each definition appears before the first use of the function it defines 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's 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 is 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 changing 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)/2 + mass*g*y \
            + 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 on the screen with backslashes at the end of each one, except the last.[35]

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's 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 their scientific goals quickly and elegantly.

[35] 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 when it's not strictly needed, so in most cases it is simpler just to use the backslash and not worry about the rules.

CHAPTER 3

GRAPHICS AND VISUALIZATION

SO FAR we have created programs that print out words and numbers, but often we will also want our programs to produce graphics, meaning pictures of some sort. In this chapter we will see how to produce the two main types of computer graphics used in physics. First, we look at that most common of scientific visualizations, the graph: a depiction of numerical data displayed on calibrated axes. Second, we look at methods for making scientific diagrams and animations: depictions of the arrangement or motion of the parts of a physical system, which can be useful in understanding the structure or behavior of the system.

3.1 GRAPHS

A number of Python packages include features for making graphs. In this book we will use the powerful, easy-to-use, and popular package `pylab`.[1] The package contains features for generating graphs of many different types. We will concentrate of three types that are especially useful in physics: ordinary line graphs, scatter plots, and density (or heat) plots. We start by looking at line graphs.[2]

To create an ordinary graph in Python we use the function `plot` from the

[1] The name `pylab` is a reference to the scientific calculation program Matlab, whose graph-drawing features `pylab` is intended to mimic. The `pylab` package is a part of a larger package called `matplotlib`, some of whose features can occasionally be useful in computational physics, although we will use only the ones in `pylab` in this book. If you're interested in the other features of `matplotlib`, take a look at the on-line documentation, which can be found at `matplotlib.org`.

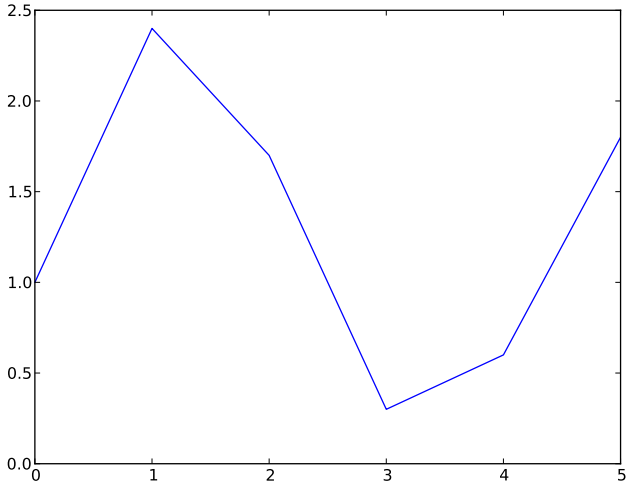
[2] The `pylab` package can also make contour plots, polar plots, pie charts, histograms, and more, and all of these find occasional use in physics. If you find yourself needing one of these more specialized graph types, you can find instructions for making them in the on-line documentation at `matplotlib.org`.

`pylab` package. In the simplest case, this function takes one argument, which is a list or array of the values we want to plot. The function creates a graph of the given values in the memory of the computer, but it doesn't actually display it on the screen of the computer—it's stored in the memory but not yet visible to the computer user. To display the graph we use a second function from `pylab`, the `show` function, which takes the graph in memory and draws it on the screen. Here is a complete program for plotting a small graph:

```
from pylab import plot,show
y = [ 1.0, 2.4, 1.7, 0.3, 0.6, 1.8 ]
plot(y)
show()
```

After importing the two functions from `pylab`, we create the list of values to be plotted, create a graph of those values with `plot(y)`, then display that graph on the screen with `show()`. Note that `show()` has parentheses after it—it is a function that has no argument, but the parentheses still need to be there.

If we run the program above, it produces a new window on the screen with a graph in it like this:

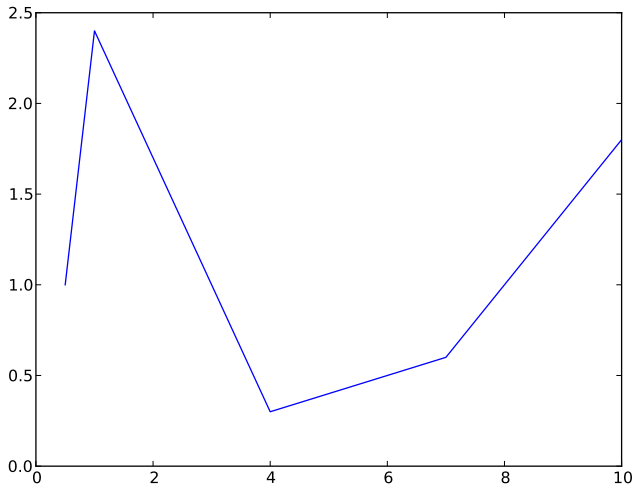


The computer has plotted the values in the list `y` at unit intervals along the x-axis (starting from zero in the standard Python style) and joined them up with straight lines.

While it's better than nothing, this is not a very useful kind of graph for physics purposes. Normally we want to specify both the x- and y-coordinates for the points in the graph. We can do this using a `plot` statement with two list arguments, thus:

```
from pylab import plot,show
x = [ 0.5, 1.0, 2.0, 4.0, 7.0, 10.0 ]
y = [ 1.0, 2.4, 1.7, 0.3, 0.6, 1.8 ]
plot(x,y)
show()
```

which produces a graph like this:



The first of the two lists now specifies the x-coordinates of each point, the second the y-coordinates. The computer plots the points at the given positions and then again joins them with straight lines. The two lists must have the same number of entries, as here. If they do not, you'll get an error message and no graph.

Why do we need two commands, `plot` and `show`, to make a graph? In the simple examples above it seems like it would be fine to combine the two into a single command that both creates a graph and shows it on the screen. However, there are more complicated situations where it is useful to have separate commands. In particular, in cases where we want to plot two or more different curves on the same graph, we can do so by using the `plot` function two or

more times, once for each curve. Then we use the `show` function once to make a single graph with all the curves. We will see examples of this shortly.

Once you have displayed a graph on the screen you can do other things with it. You will notice a number of buttons along the bottom of the window in which the graph appears (not shown in the figures here, but you will see them if you run the programs on your own computer). Among other things, these buttons allow you to zoom in on portions of the graph, move your view around the graph, or save the graph on your computer in various file formats, allowing you to view it again later, print it out on a printer, or insert it as a figure in a word processor document.

Let us apply the `plot` and `show` functions to the creation of a slightly more interesting graph, a graph of the function $\sin x$ from $x = 0$ to $x = 10$. To do this we first create an array of the x values, then we take the sines of those values to get the y-coordinates of the points:

```
from pylab import plot,show
from numpy import linspace,sin

x = linspace(0,10,100)
y = sin(x)
plot(x,y)
show()
```

Notice how we used the `linspace` function from `numpy` (see Section 2.5) to generate the array of x-values, and the `sin` function from `numpy`, which is a special version of sine that works with arrays—it just takes the sine of every element in the array. (We could alternatively have used the ordinary `sin` function from the `math` package and taken the sines of each element individually using a for loop, or using `map(sin,x)`. As is often the case, there's more than one way to do the job.)

If we run this program we get the classic sine curve graph shown in Fig. 3.1. Notice that we have not really drawn a curve at all here: our plot consists of a finite set of points—a hundred of them in this case—and the computer draws straight lines joining these points. So the end result is not actually curved; it's a set of straight-line segments. To us, however, it looks like a convincing sine wave because our eyes are not sharp enough to see the slight kinks where the segments meet. This is a useful and widely used trick for making curves in computer graphics: choose a set of points spaced close enough together that when joined with straight lines the result looks like a curve even though it really isn't.

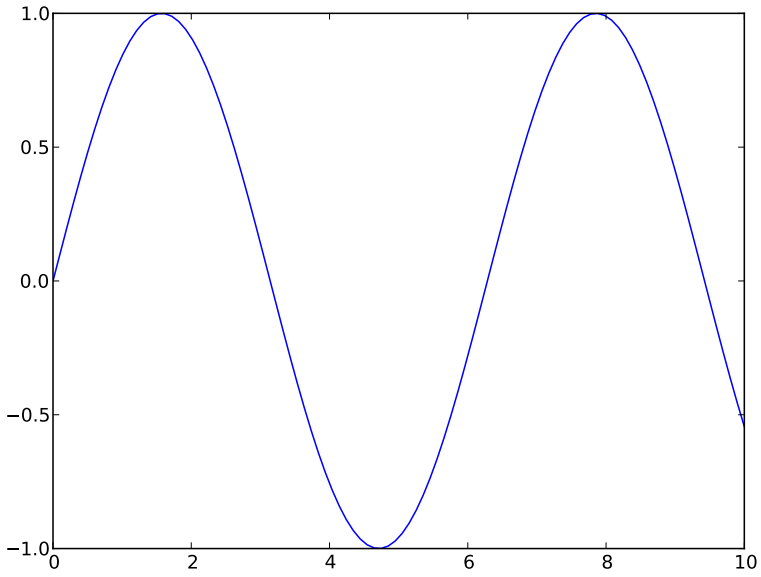


Figure 3.1: Graph of the sine function. A simple graph of the sine function produced by the program given in the text.

As another example of the use of the `plot` function, suppose we have some experimental data in a computer file `values.txt`, stored in two columns, like this:

```
0       12121.71
1       12136.44
2       12226.73
3       12221.93
4       12194.13
5       12283.85
6       12331.6
7       12309.25
...
```

We can make a graph of these data as follows:

```
from numpy import loadtxt
from pylab import plot,show

data = loadtxt("values.txt",float)
```

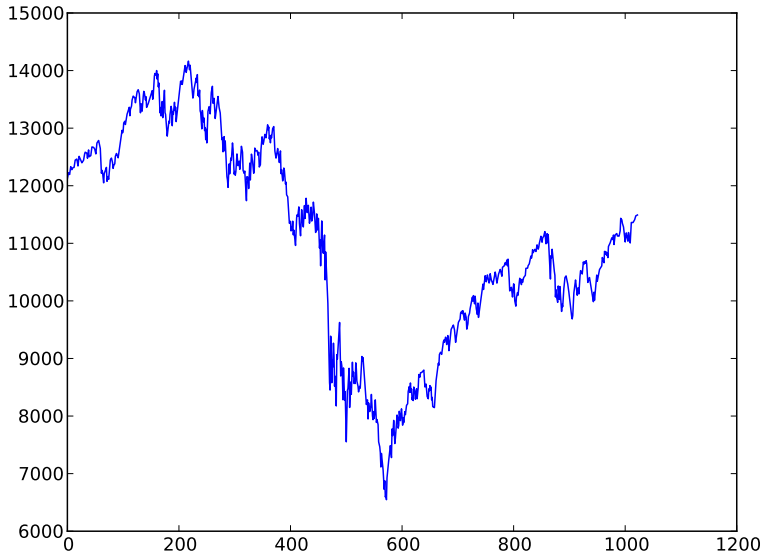


Figure 3.2: Graph of data from a file. This graph was produced by reading two columns of data from a file using the program given in the text.

```
x = data[:,0]
y = data[:,1]
plot(x,y)
show()
```

In this example we have used the `loadtxt` function from `numpy` (see Section 2.4.3) to read the values in the file and put them in an array and then we have used Python's array slicing facilities (Section 2.4.5) to extract the first and second columns of the array and put them in separate arrays `x` and `y` for plotting. The end result is a plot as shown in Fig. 3.2.

In fact, it's not necessary in this case to use the separate arrays `x` and `y`. We could shorten the program by saying instead

```
data = loadtxt("values.txt",float)
plot(data[:,0],data[:,1])
show()
```

which achieves the same result. (Arguably, however, this program is more difficult to read. As we emphasized in Section 2.7, it is a good idea to make programs easy to read where possible, so you might, in this case, want to use

the extra arrays x and y even though they are not strictly necessary.)

An important point to notice about all of these examples is that the program *stops* when it displays the graph. To be precise it stops when it gets to the show function. Once you use show to display a graph, the program will go no further until you close the window containing the graph. Only once you close the window will the computer proceed with the next line of your program. The function show is said to be a *blocking* function—it blocks the progress of the program until the function is done with its job. We have seen one other example of a blocking function previously, the function input, which collects input from the user at the keyboard. It too halts the running of the program until its job is done. (The blocking action of the show function has little impact in the programs above, since the show statement is the last line of the program in each case. But in more complex examples there might be further lines after the show statement and their execution would be delayed until the graph window was closed.)

A useful trick that we will employ frequently in this book is to build the lists of x- and y-coordinates for a graph step by step as we go through a calculation. It will happen often that we do not know all of the x or y values for a graph ahead of time, but work them out one by one as part of some calculation we are doing. In that case, a good way to create a graph of the results is to start with two empty lists for the x- and y-coordinates and add points to them one by one, as we calculate the values. Going back to the sine wave example, for instance, here is an alternative way to make a graph of $\sin x$ that calculates the individual values one by one and adds them to a growing list:

```
from pylab import plot,show
from math import sin
from numpy import linspace

xpoints = []
ypoints = []
for x in linspace(0,10,100):
    xpoints.append(x)
    ypoints.append(sin(x))

plot(xpoints,ypoints)
show()
```

If you run it, this program produces a picture of a sine wave identical to the one in Fig. 3.1 on page 92. Notice how we created the two empty lists and

then appended values to the end of each one, one by one, using a for loop. We will use this technique often. (See Section 2.4.1 for a discussion of the `append` function.)

The graphs we have seen so far are very simple, but there are many extra features we can add to them, some of which are illustrated in Fig. 3.3. For instance, in all the previous graphs the computer chose the range of x and y values for the two axes. Normally the computer makes good choices, but occasionally you might like to make different ones. In our picture of a sine wave, Fig. 3.1, for instance, you might decide that the graph would be clearer if the curve did not butt right up against the top and bottom of the frame—a little more space at top and bottom would be nice. You can override the computer's choice of x- and y-axis limits with the functions `xlim` and `ylim`. These functions take two arguments each, for the lower and upper limits of the range of the respective axes. Thus, for instance, we might modify our sine wave program as follows:

```
from pylab import plot,ylim,show
from numpy import linspace,sin

x = linspace(0,10,100)
y = sin(x)
plot(x,y)
ylim(-1.1,1.1)
show()
```

The resulting graph is shown in Fig. 3.3a and, as we can see, it now has a little extra space above and below the curve because the y-axis has been modified to run from -1.1 to $+1.1$. Note that the `ylim` statement has to come after the `plot` statement but before the `show` statement—the `plot` statement has to create the graph first before you can modify its axes.

It's good practice to label the axes of your graphs, so that you and anyone else knows what they represent. You can add labels to the x- and y-axes with the functions `xlabel` and `ylabel`, which take a string argument—a string of letters or numbers in quotation marks. Thus we could again modify our program above by importing `xlabel` and `ylabel` from `pylab` then writing

```
plot(x,y)
ylim(-1.1,1.1)
xlabel("x axis")
ylabel("y = sin x")
show()
```

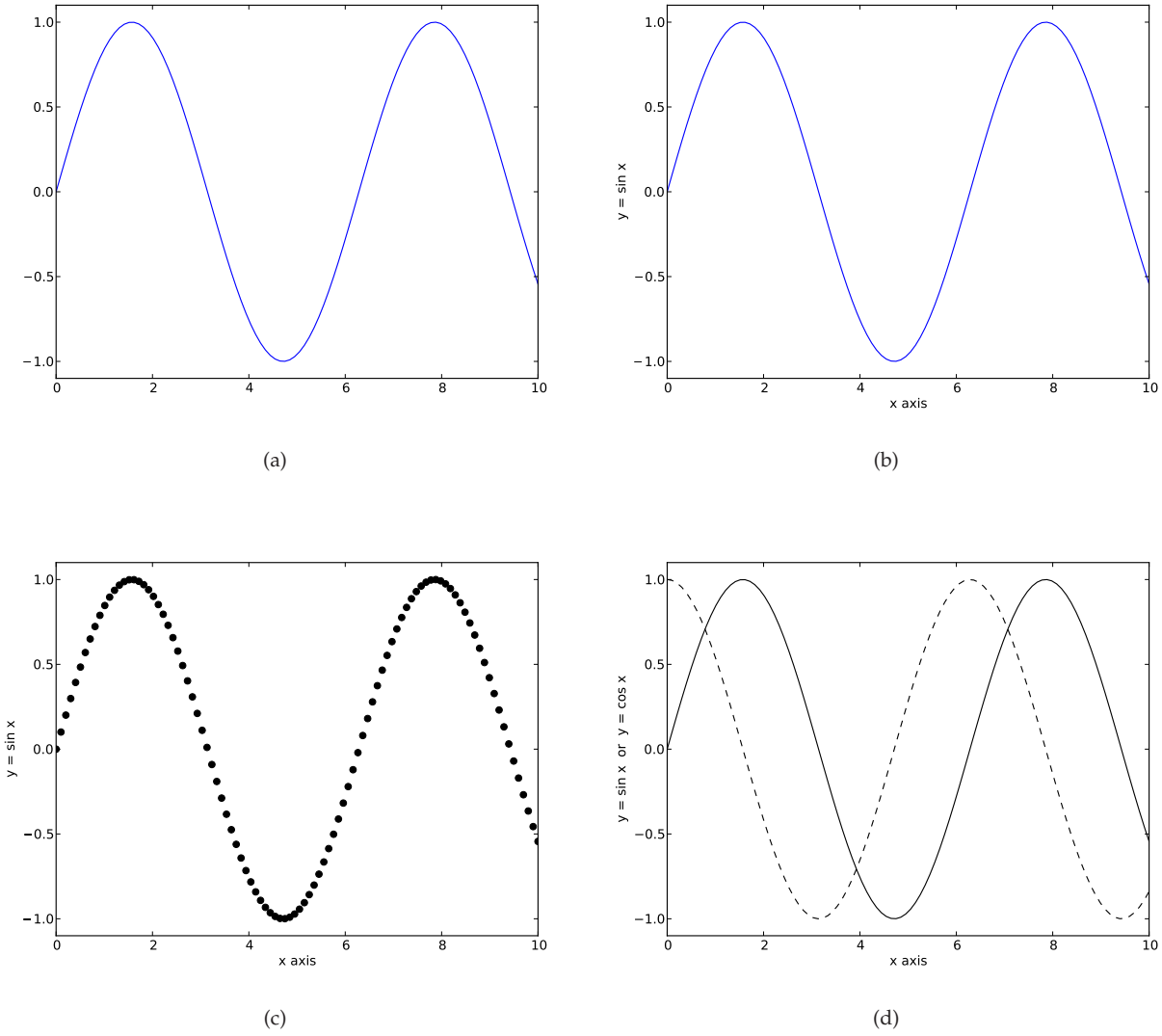


Figure 3.3: Graph styles. Several different versions of the same sine wave plot. (a) A basic graph, but with a little extra space added above and below the curve to make it clearer; (b) a graph with labeled axes; (c) a graph with the curve replaced by circular dots; (d) sine and cosine curves on the same graph.

which produces the graph shown in Fig. 3.3b.

You can also vary the style in which the computer draws the curve on the graph. To do this a third argument is added to the `plot` function, which takes the form of a (slightly cryptic) string of characters, like this:

```
plot(x,y,"g--")
```

The first letter of the string tells the computer what color to draw the curve with. Allowed letters are `r`, `g`, `b`, `c`, `m`, `y`, `k`, and `w`, for red, green, blue, cyan, magenta, yellow, black, and white, respectively. The remainder of the string says what style to use for the line. Here there are many options, but the ones we'll use most often are "`-`" for a solid line (like the ones we've seen so far), "`--`" for a dashed line, "`o`" to mark points with a circle (but not connect them with lines), and "`s`" to mark points with a square. Thus, for example, this modification:

```
plot(x,y,"ko")
ylim(-1.1,1.1)
xlabel("x axis")
ylabel("y = sin x")
show()
```

tells the computer to plot our sine wave as a set of black circular points. The result is shown in Fig. 3.3c.

Finally, we will often need to plot more than one curve or set of points on the same graph. This can be achieved by using the `plot` function repeatedly. For instance, here is a complete program that plots both the sine function and the cosine function on the same graph, one as a solid curve, the other as a dashed curve:

```
from pylab import plot,ylim,xlabel,ylabel,show
from numpy import linspace,sin,cos

x = linspace(0,10,100)
y1 = sin(x)
y2 = cos(x)
plot(x,y1,"k-")
plot(x,y2,"k--")
ylim(-1.1,1.1)
xlabel("x axis")
ylabel("y = sin x  or  y = cos x")
show()
```

The result is shown in Fig. 3.3d. You could also, for example, use a variant of the same trick to make a plot that had both dots and lines for the same data—just plot the data twice on the same graph, using two `plot` statements, one with dots and one with lines.

There are many other variations and styles available in the `pylab` package. You can add legends and annotations to your graphs. You can change the color, size, or typeface used in the labels. You can change the color or style of the axes, or add a background color to the graph. These and many other possibilities are described in the on-line documentation at `matplotlib.org`.

Exercise 3.1: Plotting experimental data

In the on-line resources[3] you will find a file called `sunspots.txt`, which contains the observed number of sunspots on the Sun for each month since January 1749. The file contains two columns of numbers, the first being the month and the second being the sunspot number.

a) Write a program that reads in the data and makes a graph of sunspots as a function of time.

b) Modify your program to display only the first 1000 data points on the graph.

c) Modify your program further to calculate and plot the running average of the data, defined by

$$Y_k = \frac{1}{2r+1} \sum_{m=-r}^{r} y_{k+m}$$

where $r = 5$ in this case (and the y_k are the sunspot numbers). Have the program plot both the original data and the running average on the same graph, again over the range covered by the first 1000 data points.

Exercise 3.2: Curve plotting

Although the `plot` function is designed primarily for plotting standard xy graphs, it can be adapted for other kinds of plotting as well.

a) Make a plot of the so-called *deltoid* curve, which is defined parametrically by the equations

$$x = 2\cos\theta + \cos 2\theta, \qquad y = 2\sin\theta - \sin 2\theta,$$

where $0 \le \theta < 2\pi$. Take a set of values of θ between zero and 2π and calculate x and y for each from the equations above, then plot y as a function of x.

---

[3] The on-line resources for this book can be downloaded in the form of a single "zip" file from `http://www.umich.edu/~mejn/cpresources.zip`.

b) Taking this approach a step further, one can make a polar plot $r = f(\theta)$ for some function f by calculating r for a range of values of θ and then converting r and θ to Cartesian coordinates using the standard equations $x = r\cos\theta$, $y = r\sin\theta$. Use this method to make a plot of the Galilean spiral $r = \theta^2$ for $0 \le \theta \le 10\pi$.

c) Using the same method, make a polar plot of "Fey's function"

$$r = e^{\cos\theta} - 2\cos 4\theta + \sin^5\frac{\theta}{12}$$

in the range $0 \le \theta \le 24\pi$.

3.2 SCATTER PLOTS

In an ordinary graph, such as those of the previous section, there is one independent variable, usually placed on the horizontal axis, and one dependent variable, on the vertical axis. The graph is a visual representation of the variation of the dependent variable as a function of the independent one—voltage as a function of time, say, or temperature as a function of position. In other cases, however, we measure or calculate two dependent variables. A classic example in physics is the temperature and brightness—also called the magnitude—of stars. Typically we might measure temperature and magnitude for each star in a given set and we would like some way to visualize how the two quantities are related. A standard approach is to use a *scatter plot*, a graph in which the two quantities are placed along the axes and we make a dot on the plot for each pair of measurements, i.e., for each star in this case.

There are two different ways to make a scatter plot using the `pylab` package. One of them we have already seen: we can make an ordinary graph, but with dots rather than lines to represent the data points, using a statement of the form:

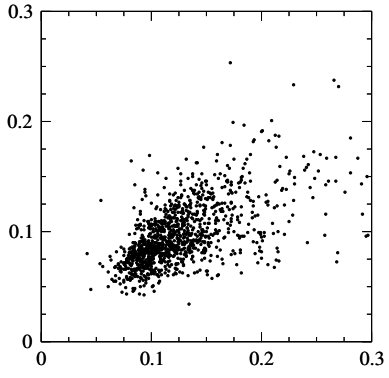
```
plot(x,y,"ko")
```

This will place a black dot at each point. A slight variant of the same idea is this:

```
plot(x,y,"k.")
```

which will produce smaller dots.

Alternatively, `pylab` provides the function `scatter`, which is designed specifically for making scatter plots. It works in a similar fashion to the `plot` function: you give it two lists or arrays,



A small scatter plot.

one containing the x-coordinates of the points and the other containing the y-coordinates, and it creates the corresponding scatter plot:

```
scatter(x,y)
```

You do not have to give a third argument telling `scatter` to plot the data as dots—all scatter plots use dots automatically. As with the `plot` function, `scatter` only creates the scatter plot in the memory of the computer but does not display it on the screen. To display it you need to use the function `show`.

Suppose, for example, that we have the temperatures and magnitudes of a set of stars in a file called `stars.txt` on our computer, like this:

```
4849.4 5.97
5337.8 5.54
4576.1 7.72
4792.4 7.18
5141.7 5.92
6202.5 4.13
...
```

The first column is the temperature and the second is the magnitude. Here's a Python program to make a scatter plot of these data:

File: `hrdiagram.py`

```
from pylab import scatter,xlabel,ylabel,xlim,ylim,show
from numpy import loadtxt

data = loadtxt("stars.txt",float)
x = data[:,0]
y = data[:,1]

scatter(x,y)
xlabel("Temperature")
ylabel("Magnitude")
xlim(0,13000)
ylim(-5,20)
show()
```

If we run this program it produces the figure shown in Fig. 3.4.

Many of the same variants illustrated in Fig. 3.3 for the `plot` function work for the `scatter` function also. In this program we used `xlabel` and `ylabel` to label the temperature and magnitude axes, and `xlim` and `ylim` to set the ranges of the axes. You can also change the size and style of the dots and many other things. In addition, as with the `plot` function, you can use `scatter`

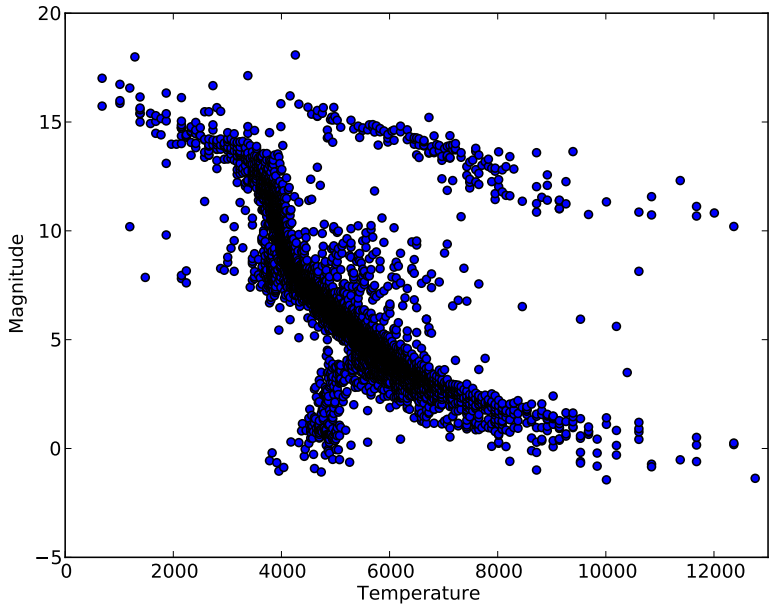


Figure 3.4: The Hertzsprung–Russell diagram. A scatter plot of the magnitude (i.e., brightness) of stars against their approximate surface temperature (which is estimated from the color of the light they emit). Each dot on the plot represents one star out of a catalog of 7860 stars that are close to our solar system.

two or more times in succession to plot two or more sets of data on the same graph, or you can use any combination of `scatter` and `plot` functions to draw scatter data and curves on the same graph. Again, see the on-line manual at `matplotlib.org` for more details.

The scatter plot of the magnitudes and temperatures in Fig. 3.4 reveals an interesting pattern in the data: a substantial majority of the points lie along a rough band running from top left to bottom right of the plot. This is the so-called *main sequence* to which most stars belong. Rarer types of stars, such as red giants and white dwarfs, stand out in the figure as dots that lie well off the main sequence. A scatter plot of stellar magnitude against temperature is called a *Hertzsprung–Russell diagram* after the astronomers who first drew it. The diagram is one of the fundamental tools of stellar astrophysics.

In fact, Fig. 3.4 is, in a sense, upside down, because the Hertzsprung–

Russell diagram is, for historical reasons,[4] normally plotted with both the magnitude and temperature axes *decreasing*, rather than increasing. One of the nice things about `pylab` is that it is easy to change this kind of thing with just a small modification of the Python program. All we need to do in this case is change the `xlim` and `ylim` statements so that the start and end points of each axis are reversed, thus:

```
xlim(13000,0)
ylim(20,-5)
```

Then the figure will be magically turned around.

3.3 DENSITY PLOTS

There are many times in physics when we need to work with two-dimensional grids of data. A condensed matter physicist might measure variations in charge or temperature or atomic deposition on a solid surface; a fluid dynamicist might measure the heights of waves in a ripple tank; a particle physicist might measure the distribution of particles incident on an imaging detector; and so on. Two-dimensional data are harder to visualize on a computer screen than the one-dimensional lists of values that go into an ordinary graph. But one tool that is helpful in many cases is the *density plot*, a two-dimensional plot where color or brightness is used to indicate data values. Figure 3.5 shows an example.

In Python density plots are produced by the function `imshow` from `pylab`. Here's the program that produced Fig. 3.5:

```
from pylab import imshow,show
from numpy import loadtxt

data = loadtxt("circular.txt",float)
imshow(data)
show()
```

The file `circular.txt` contains a simple array of values, like this:

[4] The magnitude of a star is defined in such a way that it actually increases as the star gets fainter, so reversing the vertical axis makes sense since it puts the brightest stars at the top. The temperature axis is commonly plotted not directly in terms of temperature but in terms of the so-called *color index*, which is a measure of the color of light a star emits, which is in turn a measure of temperature. Temperature decreases with increasing color index, which is why the standard Hertzsprung–Russell diagram has temperature decreasing along the horizontal axis.

```
0.0050 0.0233 0.0515 0.0795 0.1075 ...
0.0233 0.0516 0.0798 0.1078 0.1358 ...
0.0515 0.0798 0.1080 0.1360 0.1639 ...
0.0795 0.1078 0.1360 0.1640 0.1918 ...
0.1075 0.1358 0.1639 0.1918 0.2195 ...
  ...    ...    ...    ...    ...
```

The program reads the values in the file and puts them in the two-dimensional array `data` using the `loadtxt` function, then creates the density plot with the `imshow` function and displays it with `show`. The computer automatically adjusts the color-scale so that the picture uses the full range of available shades.

The computer also adds numbered axes along the sides of the figure, which measure the rows and columns of the array, though it is possible to change the calibration of the axes to use different units—we'll see how to do this in a moment. The image produced is a direct picture of the array, laid out in the usual fashion for matrices, row by row, starting at the top and working downwards. Thus the top left corner in Fig. 3.5 represents the value stored in the array element `data[0,0]`, followed to the right by `data[0,1]`, `data[0,2]`, and so on. Immediately below those, the next row is `data[1,0]`, `data[1,1]`, `data[1,2]`, and so on.

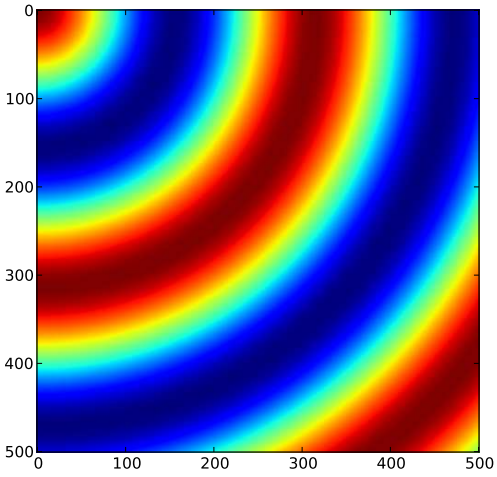


Figure 3.5: A example of a density plot

Note that the numerical labels on the axes reflect the array indices, with the origin of the figure being at the top left and the vertical axis increasing downwards. While this is natural from the point of view of matrices, it is a little odd for a graph. Most of us are accustomed to graphs whose vertical axes increase upwards. What's more, the array elements `data[i,j]` are written (as is the standard with matrices) with the *row* index first—i.e., the vertical index—and the column, or horizontal, index second. This is the opposite of the convention normally used with graphs where we list the coordinates of a point in x, y order—i.e., horizontal first, then vertical. There's nothing much we can do about the conventions for matrices: they are the ones that mathematicians settled upon centuries ago and it's too late to change them now. But the conflict between those conventions and the conventions used when plotting graphs can be confusing, so take this opportunity to make a mental note.

In fact, Python provides a way to deal with the first problem, of the origin

in a density plot being at the top. You can include an additional argument with the `imshow` function thus:

```
imshow(data,origin="lower")
```

which flips the density plot top-to-bottom, putting the array element `data[0,0]` in the lower left corner, as is conventional, and changing the labeling of the vertical axis accordingly, so that it increases in the upward direction. The resulting plot is shown in Fig. 3.6a. We will use this trick for most of the density plots in this book. Note, however, that this does not fix our other problem: indices `i` and `j` for the element `data[i,j]` still correspond to vertical and horizontal positions respectively, not the reverse. That is, the index `i` corresponds to the y-coordinate and the index `j` corresponds to the x-coordinate. You need to keep this in mind when making density plots—it's easy to get the axes swapped by mistake.

The black-and-white printing in this book doesn't really do justice to the density plot in Fig. 3.6a. The original is in bright colors, ranging through the spectrum from dark blue for the lowest values to red for the highest. If you wish, you can run the program yourself to see the density plot in its full glory—both the program, which is called `circular.py`, and the data file `circular.txt` can be found in the on-line resources. Density plots with this particular choice of colors from blue to red (or similar) are sometimes called *heat maps*, because the same color scheme is often used to denote temperature, with blue being the coldest temperature and red being the hottest.[5] The heat map color scheme is the default choice for density maps in Python, but it's not always the best. In fact, for most purposes, a simple gray-scale from black to white is easier to read. Luckily, it's simple to change the color scheme. To change to gray-scale, for instance, you use the function `gray`, which takes no arguments:

```
from pylab import imshow,gray,show
from numpy import loadtxt

data = loadtxt("circular.txt",float)
imshow(data,origin="lower")
gray()
show()
```

[5]It's not completely clear why people use these colors. As every physicist knows, red light has the longest wavelength of the visible colors and corresponds to the coolest objects, while blue has the shortest wavelengths and corresponds to the hottest—the exact opposite of the traditional choices. The hottest stars, for instance, are blue and the coolest are red.

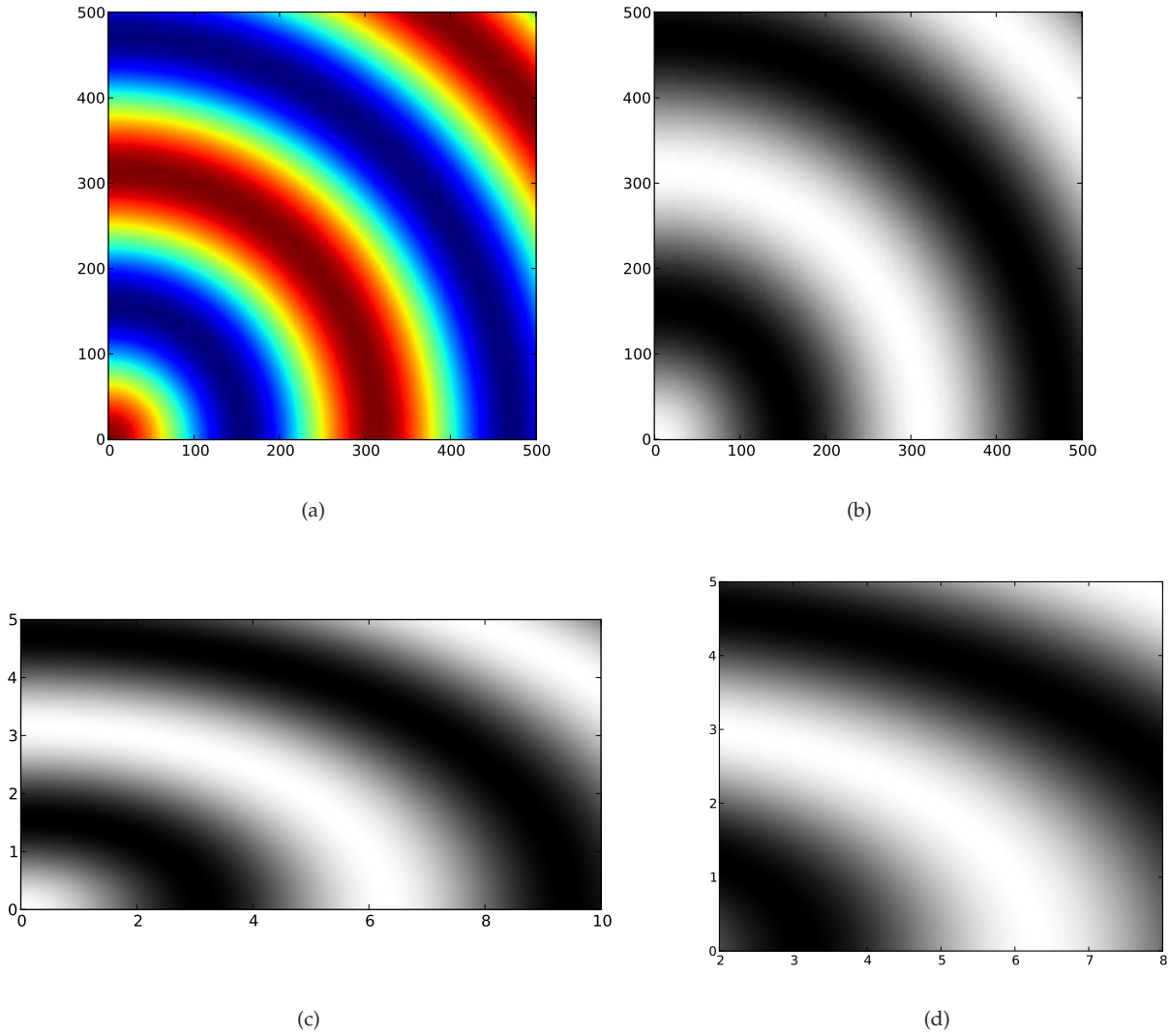


Figure 3.6: Density plots. Four different versions of the same density plot. (a) A plot using the default "heat map" color scheme, which is colorful on the computer screen but doesn't make much sense with the black-and-white printing in this book. (b) The gray color scheme, which runs from black for the lowest values to white for the highest. (c) The same plot as in panel (b) but with the calibration of the axes changed. Because the range chosen is different for the horizontal and vertical axes, the computer has altered the shape of the figure to keep distances equal in all directions. (d) The same plot as in (c) but with the horizontal range reduced so that only the middle portion of the data is shown.

Figure 3.6b shows the result. Even in black-and-white it looks somewhat different from the heat-map version in panel (a), and on the screen it looks entirely different. Try it if you like.[6]

All of the density plots in this book use the gray scale (except Figs. 3.5 and 3.6a of course). It may not be flashy, but it's informative, easy to read, and suitable for printing on monochrome printers or for publications (like many scientific books and journals) that are in black-and-white only. However, `pylab` provides many other color schemes, which you may find useful occasionally. A complete list, with illustrations, is given in the on-line documentation at `matplotlib.org`, but here are a few that might find use in physics:

`jet`	The default heat-map color scheme
`gray`	Gray-scale running from black to white
`hot`	An alternative heat map that goes black-red-yellow-white
`spectral`	A spectrum with 7 clearly defined colors, plus black and white
`bone`	An alternative gray-scale with a hint of blue
`hsv`	A rainbow scheme that starts and ends with red

Each of these has a corresponding function, `jet()`, `spectral()`, and so forth, that selects the relevant color scheme for use in future density plots. Many more color schemes are given in `pylab` and one can also define one's own schemes, although the definitions involve some slightly tricky programming. Example code is given in Appendix E and in the on-line resources to define three additional schemes that can be useful for physics:[7]

`redblue`	Runs from red to blue via black
`redwhiteblue`	Runs from red to blue via white
`inversegray`	Runs from white to black, the opposite of `gray`

There is also a function `colorbar()` in the `pylab` package that instructs Python to add a bar to the side of your figure showing the range of colors used in the plot along with a numerical scale indicating which values correspond to which colors, something that can be helpful when you want to make a more precise quantitative reading of a density plot.

---

[6] The function `gray` works slightly differently from other functions we have seen that modify plots, such as `xlabel` or `ylim`. Those functions modified only the current plot, whereas `gray` (and the other color scheme functions in `pylab`) changes the color scheme for all subsequent density plots. If you write a program that makes more than one plot, you only need to call `gray` once.

[7] To use these color schemes copy the file `colormaps.py` from the on-line resources into the folder containing your program and then in your program say, for example, "`from colormaps import redblue`". Then the statement "`redblue()`" will switch to the `redblue` color map.

As with graphs and scatter plots, you can modify the appearance of density plots in various ways. The functions `xlabel` and `ylabel` work as before, adding labels to the two axes. You can also change the scale marked on the axes. By default, the scale corresponds to the elements of the array holding the data, but you might want to calibrate your plot with a different scale. You can do this by adding an extra parameter to `imshow`, like this:

```
imshow(data,origin="lower",extent=[0,10,0,5])
```

which results in a modified plot as shown in Fig. 3.6c. The argument consists of "`extent=`" followed by a list of four values, which give, in order, the beginning and end of the horizontal scale and the beginning and end of the vertical scale. The computer will use these numbers to mark the axes, but the actual content displayed in the body of the density plot remains unchanged—the `extent` argument affects only how the plot is labeled. This trick can be very useful if you want to calibrate your plot in "real" units. If the plot is a picture of the surface of the Earth, for instance, you might want axes marked in units of latitude and longitude; if it's a picture of a surface at the atomic scale you might want axes marked in nanometers.

Note also that in Fig. 3.6c the computer has changed the shape of the plot—its *aspect ratio*—to accommodate the fact that the horizontal and vertical axes have different ranges. The `imshow` function attempts to make unit distances equal along the horizontal and vertical directions where possible. Sometimes, however, this is not what we want, in which case we can tell the computer to use a different aspect ratio. For instance, if we wanted the present figure to remain square we would say:

```
imshow(data,origin="lower",extent=[0,10,0,5],aspect=2.0)
```

This tells the computer to use unit distances twice as large along the vertical axis as along the horizontal one, which will make the plot square once more.

Note that, as here, we are free to use any or all of the `origin`, `extent`, and `aspect` arguments together in the same function. We don't have to use them all if we don't want to—any selection is allowed—and they can come in any order.

We can also limit our density plot to just a portion of the data, using the functions `xlim` and `ylim`, just as with graphs and scatter plots. These functions work with the scales specified by the `extent` argument, if there is one, or with the row and column indices otherwise. So, for instance, we could say `xlim(2,8)` to reduce the density plot of Fig. 3.6b to just the middle portion of

the horizontal scale, from 2 to 8. Figure 3.6d shows the result. Note that, unlike the `extent` argument, `xlim` and `ylim` do change which data are displayed in the body of the density plot—the `extent` argument makes purely cosmetic changes to the labeling of the axes, but `xlim` and `ylim` actually change which data appear.

Finally, you can use the functions `plot` and `scatter` to superimpose graphs or scatter plots of data on the same axes as a density plot. You can use any combination of `imshow`, `plot`, and `scatter` in sequence, followed by `show`, to create a single graph with density data, curves, or scatter data, all on the same set of axes.

EXAMPLE 3.1: WAVE INTERFERENCE

Suppose we drop a pebble in a pond and waves radiate out from the spot where it fell. We could create a simple representation of the physics with a sine wave, spreading out in a uniform circle, to represent the height of the waves at some later time. If the center of the circle is at x_1, y_1 then the distance r_1 to the center from a point x, y is

$$r_1 = \sqrt{(x - x_1)^2 + (y - y_1)^2} \tag{3.1}$$

and the sine wave for the height is

$$\xi_1(x, y) = \xi_0 \sin kr_1 \,, \tag{3.2}$$

where ξ_0 is the amplitude of the waves and k is the wavevector, related to the wavelength λ by $k = 2\pi/\lambda$.

Now suppose we drop another pebble in the pond, creating another circular set of waves with the same wavelength and amplitude but centered on a different point x_2, y_2:

$$\xi_2(x, y) = \xi_0 \sin kr_2 \qquad \text{with} \qquad r_2 = \sqrt{(x - x_2)^2 + (y - y_2)^2}. \tag{3.3}$$

Then, assuming the waves add linearly (which is a reasonable assumption for water waves, provided they are not too big), the total height of the surface at a point x, y is

$$\xi(x, y) = \xi_0 \sin kr_1 + \xi_0 \sin kr_2. \tag{3.4}$$

Suppose the wavelength of the waves is $\lambda = 5\,\text{cm}$, the amplitude is 1 cm, and the centers of the circles are 20 cm apart. Here is a program to make an image of the height over a 1 m square region of the pond. To make the image we create

an array of values representing the height ξ at a grid of points and then use that array to make a density plot. In this example we use a grid of 500×500 points to cover the 1 m square, which means the grid points have a separation of $100/500 = 0.2$ cm.

File: ripples.py

```
from math import sqrt,sin,pi
from numpy import empty
from pylab import imshow,gray,show

wavelength = 5.0
k = 2*pi/wavelength
xi0 = 1.0
separation = 20.0      # Separation of centers in cm
side = 100.0           # Side of the square in cm
points = 500           # Number of grid points along each side
spacing = side/points  # Spacing of points in cm

# Calculate the positions of the centers of the circles
x1 = side/2 + separation/2
y1 = side/2
x2 = side/2 - separation/2
y2 = side/2

# Make an array to store the heights
xi = empty([points,points],float)

# Calculate the values in the array
for i in range(points):
    y = spacing*i
    for j in range(points):
        x = spacing*j
        r1 = sqrt((x-x1)**2+(y-y1)**2)
        r2 = sqrt((x-x2)**2+(y-y2)**2)
        xi[i,j] = xi0*sin(k*r1) + xi0*sin(k*r2)

# Make the plot
imshow(xi,origin="lower",extent=[0,side,0,side])
gray()
show()
```

This is the longest and most involved program we have seen so far, so it may be worth taking a moment to make sure you understand how it works. Note in particular how the height is calculated and stored in the array `xi`. The

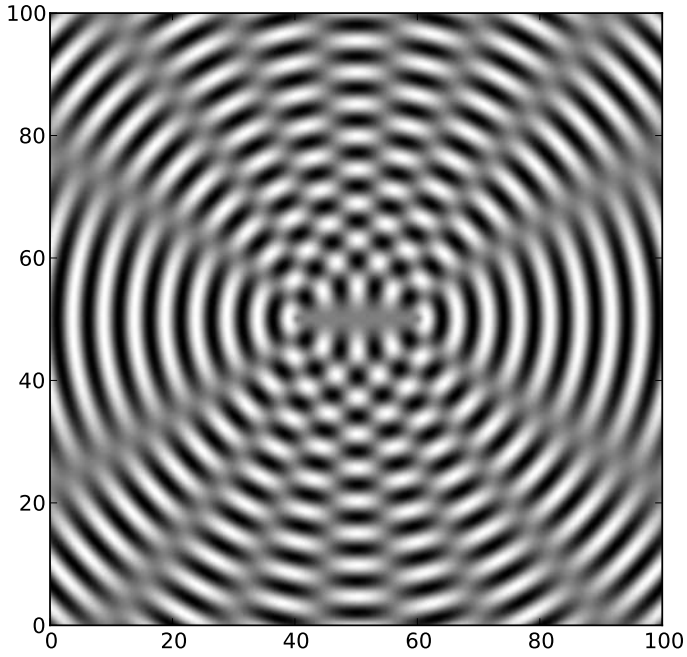


Figure 3.7: Interference pattern. This plot, produced by the program given in the text, shows the superposition of two circular sets of sine waves, creating an interference pattern with fringes that appear as the gray bars radiating out from the center of the picture.

variables `i` and `j` go through the rows and columns of the array respectively, and from these we calculate the values of the coordinates x and y. Since, as discussed earlier, the rows correspond to the vertical axis and the columns to the horizontal axis, the value of x is calculated from `j` and the value of y is calculated from `i`. Other than this subtlety, the program is a fairly straightforward translation of Eqs. (3.1–3.4).[8]

If we run the program above, it produces the picture shown in Fig. 3.7. The picture shows clearly the interference of the two sets of waves. The interference

[8]One other small detail is worth mentioning. We called the variable for the wavelength "`wavelength`". You might be tempted to call it "`lambda`" but if you did you would get an error message and the program would not run. The word "lambda" has a special meaning in the Python language and cannot be used as a variable name, just as words like "for" and "if" cannot be used as variable names. (See footnote 5 on page 13.) The names of other Greek letters—alpha, beta, gamma, and so on—are allowed as variable names.

fringes are visible as the gray bands radiating from the center.

Exercise 3.3: There is a file in the on-line resources called `stm.txt`, which contains a grid of values from scanning tunneling microscope measurements of the (111) surface of silicon. A scanning tunneling microscope (STM) is a device that measures the shape of a surface at the atomic level by tracking a sharp tip over the surface and measuring quantum tunneling current as a function of position. The end result is a grid of values that represent the height of the surface and the file `stm.txt` contains just such a grid of values. Write a program that reads the data contained in the file and makes a density plot of the values. Use the various options and variants you have learned about to make a picture that shows the structure of the silicon surface clearly.

3.4 3D GRAPHICS

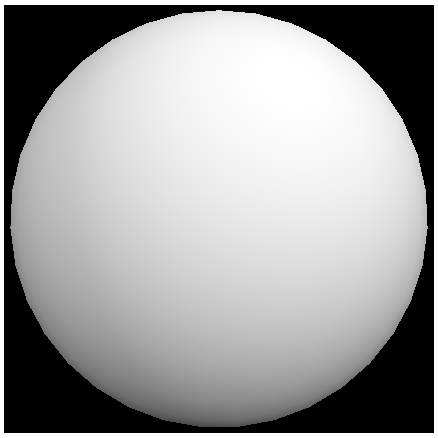
One of the flashiest applications of computers today is the creation of 3D graphics and computer animation. In any given week millions of people flock to cinemas worldwide to watch the latest computer-animated movie from the big animation studios. 3D graphics and animation find a more humble, but very useful, application in computational physics as a tool for visualizing the behavior of physical systems. Python provides some excellent tools for this purpose, which we'll use extensively in this book.

There are a number of different packages available for graphics and animation in Python, but we will focus on the package `visual`, which is specifically designed with physicists in mind. This package provides a way to create simple pictures and animations with a minimum of effort, but also has enough power to handle complex situations when needed.

The `visual` package works by creating specified objects on the screen, such as spheres, cylinders, cones, and so forth, and then, if necessary, changing their position, orientation, or shape to make them move around. Here's a short first program using the package:

```
from visual import sphere
sphere()
```

When we run this program a window appears on the screen with a large sphere in it, like this:



The window of course is two-dimensional, but the computer stores the shape and position of the sphere in three dimensions and automatically does a perspective rendering of the sphere with a 3D look to it that aids the eye in understanding the scene.

You can choose the size and position of the sphere like this

```
sphere(radius=0.5,pos=[1.0,-0.2,0.0])
```

The radius is specified as a single number. The units are arbitrary and the computer will zoom in or out as necessary to make the sphere visible. So you can set the radius to 0.5 as here, or to 10^{-15} if you're drawing a picture of a proton. Either will work fine.

The position of the sphere is a three-dimensional vector, which you give as a list or array of three real numbers x, y, z (we used a list in this case). The x- and y-axes run to the right and upwards in the window, as normal, and the z-axis runs directly out of the screen towards you. You can also specify the position as a list or array of just two numbers, x and y, in which case Python will assume the z-coordinate to be zero. This can be useful for drawing pictures of two-dimensional systems, which have no z-coordinate.

You can also change the color of the sphere thus:

```
from visual import sphere,color
sphere(color=color.green)
```

Note how we have imported the object called `color` from the visual package, then individual colors are called things like `color.green` and `color.red`. The available colors are the same as those for drawing graphs with `pylab`: red,

green, blue, cyan, magenta, yellow, black, and white.[9] The `color` argument can be used at the same time as the `radius` and `position` arguments, so one can control all features of the sphere at the same time.

We can also create several spheres, all in the same window on the screen, by using the `sphere` function repeatedly, putting different spheres in different places to build up an entire scene made of spheres. The following exercise gives an example.

EXAMPLE 3.2: PICTURING AN ATOMIC LATTICE

Suppose we have a solid composed of atoms arranged on a simple cubic lattice. We can visualize the arrangement of the atoms using the `visual` package by creating a picture with many spheres at positions (i, j, k) with $i, j, k = -L \ldots L$, thus:

File: `lattice.py`

```
from visual import sphere
L = 5
R = 0.3
for i in range(-L,L+1):
    for j in range(-L,L+1):
        for k in range(-L,L+1):
            sphere(pos=[i,j,k],radius=R)
```

Notice how this program has three nested for loops that run through all combinations of the values of `i`, `j`, and `k`. Run this program and it produces the picture shown in Fig. 3.8. Download the program and try it if you like.

After running the program, you can rotate your view of the lattice to look at it from different angles by moving the mouse while holding down either the right mouse button or the Ctrl key on the keyboard (the Command key on a Mac). You can also hold down both mouse buttons (if you have two), or the Alt key (the Option key on a Mac) and move the mouse in order to zoom in and out of the picture.

[9] All visible colors can be represented as mixtures of the primary colors red, green, and blue, and this is how they are stored inside the computer. A "color" in the `visual` package is actually just a list of three floating-point numbers giving the intensities of red, green, and blue (in that order) on a scale of 0 to 1 each. Thus red is `[ 1.0, 0.0, 0.0 ]`, yellow is `[ 1.0, 1.0, 0.0 ]`, and white is `[ 1.0, 1.0, 1.0 ]`. You can create your own colors if you want by writing things like `midgray = [ 0.5, 0.5, 0.5 ]`. Then you can use "`midgray`" just like any other color. (You would just say `midgray`, not `color.midgray`, because the color you defined is an ordinary variable, not a part of the `color` object in `visual`.)

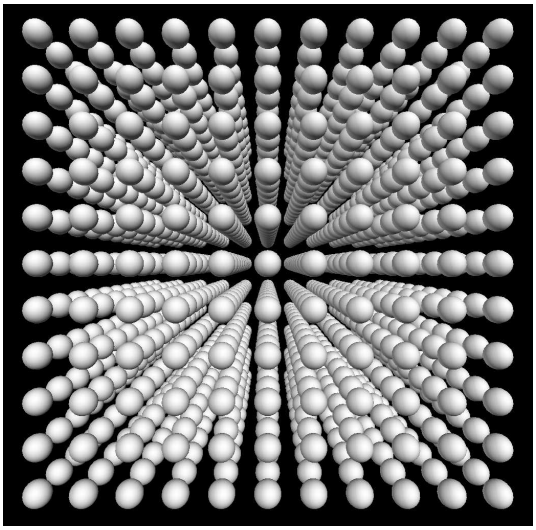
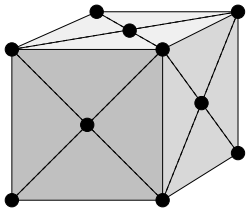


Figure 3.8: Visualization of atoms in a simple cubic lattice. A perspective rendering of atoms in a simple cubic lattice, generated using the `visual` package and the program `lattice.py` given in the text.

Exercise 3.4: Using the program from Example 3.2 above as a starting point, or starting from scratch if you prefer, do the following:



Atoms in the fcc lattice lie at the corners and center of each face of a cubic cell.

a) A sodium chloride crystal has sodium and chlorine atoms arranged on a cubic lattice but the atoms alternate between sodium and chlorine, so that each sodium is surrounded by six chlorines and each chlorine is surrounded by six sodiums. Create a visualization of the sodium chloride lattice using two different colors to represent the two types of atoms.

b) The face-centered cubic (fcc) lattice, which is the most common lattice in naturally occurring crystals, consists of a cubic lattice with atoms positioned not only at the corners of each cube but also at the center of each face. Create a visualization of an fcc lattice with a single species of atom (such as occurs in metallic nickel, for instance).

It is possible to change the properties of a sphere after it is first created, including its position, size, and color. When we do this the sphere will actually move or change on the screen. In order to refer to a particular sphere on the screen we must use a slightly different form of the `sphere` function to create it, like this:

```
s = sphere()
```

This form, in addition to drawing a sphere on the computer screen, creates a variable `s` in a manner similar to the way functions like `zeros` or `empty` create arrays (see Section 2.4.2). The new variable `s` is a variable of type "`sphere`", in the same way that other variables are of type `int` or `float`. This is a special variable type used only in the `visual` package to store the properties of spheres. Each sphere variable corresponds to a sphere on the screen and when we change the properties stored in the sphere variable the on-screen sphere changes accordingly. Thus, for example, we can say

```
s.radius = 0.5
```

and the radius of the corresponding sphere on the screen will change to 0.5, right before our eyes. Or we can say

```
s.color = color.blue
```

and the color will change. You can also change the position of a sphere in this way, in which case the sphere will move on the screen. We will use this trick in Section 3.5 to create animations of physical systems.

You can use variables of the `sphere` type in similar ways to other types of variable. A useful trick, for instance, is to create an array of spheres thus:

```
from visual import sphere
from numpy import empty
s = empty(10,sphere)
```

This creates an array, initially empty, of ten sphere-type variables that you can then fill with actual spheres thus:

```
for n in range(10):
    s[n] = sphere()
```

As each sphere is created, a corresponding sphere will appear on the screen.

This technique can be useful if you are creating a visualization or animation with many spheres and you want to be able to change the properties of any of them at will. Exercise 3.5 involves exactly such a situation, and the trick above would be a good one to use in solving that exercise.

Spheres are by no means the only shape one can draw. There is a large selection of other elements provided by the `visual` package, including boxes,

cones, cylinders, pyramids, and arrows. Here are the functions that create each of these objects:

```
from visual import box,cone,cylinder,pyramid,arrow

box(pos=[x,y,z], axis=[a,b,c], \
    length=L, height=H, width=W, up=[q,r,s])
cone(pos=[x,y,z], axis=[a,b,c], radius=R)
cylinder(pos=[x,y,z], axis=[a,b,c], radius=R)
pyramid(pos=[x,y,z], size=[z,b,c])
arrow(pos=[x,y,z], axis=[a,b,c], \
      headwidth=H, headlength=L, shaftwidth=W)
```

For a detailed explanation of the meaning of all the parameters, take a look at the on-line documentation at `www.vpython.org`. In addition to the parameters above, standard ones like `color` can also be used to give the objects a different appearance. And each element has a corresponding variable type—`box`, `cone`, `cylinder`, and so forth—that is used for storing and changing the properties of elements after they are created.

Another useful feature of the `visual` package is the ability to change various properties of the screen window in which your objects appear. You can, for example, change the window's size and position on the screen, you can change the background color, and you can change the direction that the "camera" is looking in. All of these things you do with the function `display`. Here is an example:

```
from visual import display
display(x=100,y=100,width=600,height=600, \
        center=[5,0,0],forward=[0,0,-1], \
        background=color.blue,foreground=color.yellow)
```

This will produce a window 600×600 in size, where size is measured in pixels (the small dots that make up the picture on a computer screen). The window will be 100 pixels in from the left and top of the screen. The argument "`center=[5,0,0]`" sets the point in 3D space that will be in the center of the window, and "`forward=[0,0,-1]`" chooses the direction in which we are looking. Between the two of them these two arguments determine the position and direction of our view of the scene. The background color of the window will be blue in this case and objects appearing in the window—the "foreground"—will be yellow by default, although you can specify other colors for individual objects in the manner described above for spheres.

(Notice also how we used the backslash character "\" in the code above to indicate to the computer that a single logical line of code has been spread over more than one line in the text of the program. We discussed this use of the backslash previously in Section 2.7.)

The arguments for the `display` function can be in any order and you do not have to include all of them. You need include only those you want. The ones you don't include have sensible default values. For example, the default background color is black and the default foreground color is white, so if you don't specify any colors you get white objects on a black background.

As with the `sphere` function you can assign a variable to keep track of the display window by writing, for example,

```
d = display(background=color.blue)
```

or even just

```
d = display()
```

This allows you to change display parameters later in your program. For instance, you can change the background color to black at any time by writing "`d.background = color.black`". Some parameters, however, cannot be changed later, notably the size and position of the window, which are fixed when the window is created (although you can change the size and position manually by dragging the window around the screen with your mouse).

There are many other features of the `visual` package that are not listed here. For more details take a look at `www.vpython.org`.

3.5 ANIMATION

As we have seen, the `visual` package allows you to change the properties of an on-screen object, such as its size, color, orientation, or position. If you change the position of an object repeatedly and rapidly, you can make the object appear to be moving and you have an animation. We will use such animations in this book to help us understand the behavior of physical systems.

For example, to create a sphere and then change its position you could do the following:

```
from visual import sphere
s = sphere(pos=[0,0,0])
s.pos = [1,4,3]
```

This will create a sphere at the origin, then move it to the new position $(1,4,3)$.

This is not not a very useful program, however. The computer is so fast that you probably wouldn't even see the sphere in its first position at the origin before it gets moved. To slow down movements to a point where they are visible, `visual` provides a function called `rate`. Saying `rate(x)` tells the computer to wait until $1/x$ of a second has passed since the last time you called `rate`. Thus if you call `rate(30)` immediately before each change you make on the screen, you will ensure that changes never get made more than 30 times a second, which is very useful for making smooth animations.

EXAMPLE 3.3: A MOVING SPHERE

Here is a program to move a sphere around on the screen:

File: `revolve.py`

```
from visual import sphere,rate
from math import cos,sin,pi
from numpy import arange

s = sphere(pos=[1,0,0],radius=0.1)
for theta in arange(0,10*pi,0.1):
    rate(30)
    x = cos(theta)
    y = sin(theta)
    s.pos = [x,y,0]
```

Here the value of the angle variable `theta` increases by 0.1 radians every 30th of a second, the `rate` function ensuring that we go around the for loop 30 times each second. The angle is converted into Cartesian coordinates and used to update the position of the sphere. The net result, if we run the program is that a sphere appears on the screen and moves around in a circle. Download the program and try it if you like. This simple animation could be the basis, for instance, for an animation of the simultaneous motions of the planets of the solar system. Exercise 3.5 below invites you to create exactly such an animation.

Exercise 3.5: Visualization of the solar system

The innermost six planets of our solar system revolve around the Sun in roughly circular orbits that all lie approximately in the same (ecliptic) plane. Here are some basic parameters:

Object	Radius of object (km)	Radius of orbit (millions of km)	Period of orbit (days)
Mercury	2440	57.9	88.0
Venus	6052	108.2	224.7
Earth	6371	149.6	365.3
Mars	3386	227.9	687.0
Jupiter	69173	778.5	4331.6
Saturn	57316	1433.4	10759.2
Sun	695500	–	–

Using the facilities provided by the `visual` package, create an animation of the solar system that shows the following:

a) The Sun and planets as spheres in their appropriate positions and with sizes proportional to their actual sizes. Because the radii of the planets are tiny compared to the distances between them, represent the planets by spheres with radii c_1 times larger than their correct proportionate values, so that you can see them clearly. Find a good value for c_1 that makes the planets visible. You'll also need to find a good radius for the Sun. Choose any value that gives a clear visualization. (It doesn't work to scale the radius of the Sun by the same factor you use for the planets, because it'll come out looking much too large. So just use whatever works.) For added realism, you may also want to make your spheres different colors. For instance, Earth could be blue and the Sun could be yellow.

b) The motion of the planets as they move around the Sun (by making the spheres of the planets move). In the interests of alleviating boredom, construct your program so that time in your animation runs a factor of c_2 faster than actual time. Find a good value of c_2 that makes the motion of the orbits easily visible but not unreasonably fast. Make use of the `rate` function to make your animation run smoothly.

Hint: You may find it useful to store the sphere variables representing the planets in an array of the kind described on page 115.

Here's one more trick that can prove useful. As mentioned above, you can make your objects small or large and the computer will automatically zoom in or out so that they remain visible. And if you make an animation in which your objects move around the screen the computer will zoom out when objects move out of view, or zoom in as objects recede into the distance. While this is useful in many cases, it can be annoying in others. The `display` function provides a parameter for turning the automatic zooming off if it becomes distracting, thus:

```
display(autoscale=False)
```

More commonly, one calls the `display` function at the beginning of the program and then turns off the zooming separately later, thus:

```
d = display()
d.autoscale = False
```

One can also turn it back on with

```
d.autoscale = True
```

A common approach is to place all the objects of your animation in their initial positions on the screen first, allow the computer to zoom in or out appropriately, so that they are all visible, then turn zooming off with "`d.autoscale = False`" before beginning the animation proper, so that the view remains fixed as objects move around.

FURTHER EXERCISES

3.6 Deterministic chaos and the Feigenbaum plot: One of the most famous examples of the phenomenon of chaos is the *logistic map*, defined by the equation

$$x' = rx(1 - x).$$

For a given value of the constant r you take a value of x—say $x = \frac{1}{2}$—and you feed it into the right-hand side of this equation, which gives you a value of x'. Then you take that value and feed it back in on the right-hand side again, which gives you another value, and so forth. This is an *iterative map*. You keep doing the same operation over and over on your value of x, and one of three things happens:

1. The value settles down to a fixed number and stays there. This is called a *fixed point*. For instance, $x = 0$ is always a fixed point of the logistic map. (You put $x = 0$ on the right-hand side and you get $x' = 0$ on the left.)
2. It doesn't settle down to a single value, but it settles down into a periodic pattern, rotating around a set of values, such as say four values, repeating them in sequence over and over. This is called a *limit cycle*.
3. It goes crazy. It generates a seemingly random sequence of numbers that appear to have no rhyme or reason to them at all. This is *deterministic chaos*. "Chaos" because it really does look chaotic, and "deterministic" because even though the

values look random, they're not. They're clearly entirely predictable, because they are given to you by one simple equation. The behavior is *determined*, although it may not look like it.

Write a program that calculates and displays the behavior of the logistic map. Here's what you need to do. For a given value of r, start with $x = \frac{1}{2}$, and iterate the logistic map equation a thousand times. That will give it a chance to settle down to a fixed point or limit cycle if it's going to. Then run for another thousand iterations and plot the points (r, x) on a graph where the horizontal axis is r and the vertical axis is x. You can either use the `plot` function with the options `"ko"` or `"k."` to draw a graph with dots, one for each point, or you can use the `scatter` function to draw a scatter plot (which always uses dots). Repeat the whole calculation for values of r from 1 to 4 in steps of 0.01, plotting the dots for all values of r on the same figure and then finally using the function `show` once to display the complete figure.

Your program should generate a distinctive plot that looks like a tree bent over onto its side. This famous picture is called the *Feigenbaum plot*, after its discoverer Mitchell Feigenbaum, or sometimes the *figtree plot*, a play on the fact that it looks like a tree and Feigenbaum means "figtree" in German.[10]

Give answers to the following questions:

a) For a given value of r what would a fixed point look like on the Feigenbaum plot? How about a limit cycle? And what would chaos look like?

b) Based on your plot, at what value of r does the system move from orderly behavior (fixed points or limit cycles) to chaotic behavior? This point is sometimes called the "edge of chaos."

The logistic map is a very simple mathematical system, but deterministic chaos is seen in many more complex physical systems also, including especially fluid dynamics and the weather. Because of its apparently random nature, the behavior of chaotic systems is difficult to predict and strongly affected by small perturbations in outside conditions. You've probably heard of the classic exemplar of chaos in weather systems, the *butterfly effect*, which was popularized by physicist Edward Lorenz in 1972 when he gave a lecture to the American Association for the Advancement of Science entitled, "Does the flap of a butterfly's wings in Brazil set off a tornado in Texas?"[11]

---

[10] There is another approach for computing the Feigenbaum plot, which is neater and faster, making use of Python's ability to perform arithmetic with entire arrays. You could create an array `r` with one element containing each distinct value of r you want to investigate: `[1.0, 1.01, 1.02, ...]`. Then create another array `x` of the same size to hold the corresponding values of x, which should all be initially set to 0.5. Then an iteration of the logistic map can be performed for all values of r at once with a statement of the form `x = r*x*(1-x)`. Because of the speed with which Python can perform calculations on arrays, this method should be significantly faster than the more basic method above.

[11] Although arguably the first person to suggest the butterfly effect was not a physicist at all, but the science fiction writer Ray Bradbury in his famous 1952 short story *A Sound of Thunder*, in which a time traveler's careless destruction of a butterfly during a tourist trip to the Jurassic era

3.7 The Mandelbrot set: The Mandelbrot set, named after its discoverer, the French mathematician Benoît Mandelbrot, is a *fractal*, an infinitely ramified mathematical object that contains structure within structure within structure, as deep as we care to look. The definition of the Mandelbrot set is in terms of complex numbers as follows.

Consider the equation

$$z' = z^2 + c,$$

where z is a complex number and c is a complex constant. For any given value of c this equation turns an input number z into an output number z'. The definition of the Mandelbrot set involves the repeated iteration of this equation: we take an initial starting value of z and feed it into the equation to get a new value z'. Then we take that value and feed it in again to get another value, and so forth. The Mandelbrot set is the set of points in the complex plane that satisfies the following definition:

> *For a given complex value of c, start with $z = 0$ and iterate repeatedly. If the magnitude $|z|$ of the resulting value is ever greater than 2, then the point in the complex plane at position c is* not *in the Mandelbrot set, otherwise it is in the set.*

In order to use this definition one would, in principle, have to iterate infinitely many times to prove that a point is in the Mandelbrot set, since a point is in the set only if the iteration never passes $|z| = 2$ ever. In practice, however, one usually just performs some large number of iterations, say 100, and if $|z|$ hasn't exceeded 2 by that point then we call that good enough.

Write a program to make an image of the Mandelbrot set by performing the iteration for all values of $c = x + iy$ on an $N \times N$ grid spanning the region where $-2 \le x \le 2$ and $-2 \le y \le 2$. Make a density plot in which grid points inside the Mandelbrot set are colored black and those outside are colored white. The Mandelbrot set has a very distinctive shape that looks something like a beetle with a long snout—you'll know it when you see it.

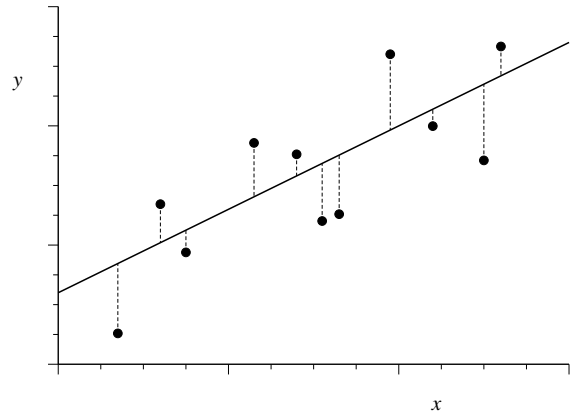
Hint: You will probably find it useful to start off with quite a coarse grid, i.e., with a small value of N—perhaps $N = 100$—so that your program runs quickly while you are testing it. Once you are sure it is working correctly, increase the value of N to produce a final high-quality image of the shape of the set.

If you are feeling enthusiastic, here is another variant of the same exercise that can produce amazing looking pictures. Instead of coloring points just black or white, color points according to the number of iterations of the equation before $|z|$ becomes greater than 2 (or the maximum number of iterations if $|z|$ never becomes greater than 2). If you use one of the more colorful color schemes Python provides for density plots, such as the "`hot`" or "`jet`" schemes, you can make some spectacular images this way. Another interesting variant is to color according to the logarithm of the number of iterations, which helps reveal some of the finer structure outside the set.

---

changes the course of history.

3.8 Least-squares fitting and the photoelectric effect: It's a common situation in physics that an experiment produces data that lies roughly on a straight line, like the dots in this figure:



The solid line here represents the underlying straight-line form, which we usually don't know, and the points representing the measured data lie roughly along the line but don't fall exactly on it, typically because of measurement error.

The straight line can be represented in the familiar form $y = mx + c$ and a frequent question is what the appropriate values of the slope m and intercept c are that correspond to the measured data. Since the data don't fall perfectly on a straight line, there is no perfect answer to such a question, but we can find the straight line that gives the best compromise fit to the data. The standard technique for doing this is the *method of least squares*.

Suppose we make some guess about the parameters m and c for the straight line. We then calculate the vertical distances between the data points and that line, as represented by the short vertical lines in the figure, then we calculate the sum of the squares of those distances, which we denote χ^2. If we have N data points with coordinates (x_i, y_i), then χ^2 is given by

$$\chi^2 = \sum_{i=1}^{N} (mx_i + c - y_i)^2.$$

The least-squares fit of the straight line to the data is the straight line that minimizes this total squared distance from data to line. We find the minimum by differentiating with respect to both m and c and setting the derivatives to zero, which gives

$$m \sum_{i=1}^{N} x_i^2 + c \sum_{i=1}^{N} x_i - \sum_{i=1}^{N} x_i y_i = 0,$$
$$m \sum_{i=1}^{N} x_i + cN - \sum_{i=1}^{N} y_i = 0.$$

For convenience, let us define the following quantities:

$$E_x = \frac{1}{N}\sum_{i=1}^{N} x_i, \qquad E_y = \frac{1}{N}\sum_{i=1}^{N} y_i, \qquad E_{xx} = \frac{1}{N}\sum_{i=1}^{N} x_i^2, \qquad E_{xy} = \frac{1}{N}\sum_{i=1}^{N} x_i y_i,$$

in terms of which our equations can be written

$$mE_{xx} + cE_x = E_{xy}\,,$$
$$mE_x + c = E_y\,.$$

Solving these equations simultaneously for m and c now gives

$$m = \frac{E_{xy} - E_x E_y}{E_{xx} - E_x^2}, \qquad c = \frac{E_{xx}E_y - E_x E_{xy}}{E_{xx} - E_x^2}.$$

These are the equations for the least-squares fit of a straight line to N data points. They tell you the values of m and c for the line that best fits the given data.

a) In the on-line resources you will find a file called `millikan.txt`. The file contains two columns of numbers, giving the x and y coordinates of a set of data points. Write a program to read these data points and make a graph with one dot or circle for each point.

b) Add code to your program, before the part that makes the graph, to calculate the quantities E_x, E_y, E_{xx}, and E_{xy} defined above, and from them calculate and print out the slope m and intercept c of the best-fit line.

c) Now write code that goes through each of the data points in turn and evaluates the quantity $mx_i + c$ using the values of m and c that you calculated. Store these values in a new array or list, and then graph this new array, as a solid line, on the same plot as the original data. You should end up with a plot of the data points plus a straight line that runs through them.

d) The data in the file `millikan.txt` are taken from a historic experiment by Robert Millikan that measured the *photoelectric effect*. When light of an appropriate wavelength is shone on the surface of a metal, the photons in the light can strike conduction electrons in the metal and, sometimes, eject them from the surface into the free space above. The energy of an ejected electron is equal to the energy of the photon that struck it minus a small amount ϕ called the *work function* of the surface, which represents the energy needed to remove an electron from the surface. The energy of a photon is $h\nu$, where h is Planck's constant and ν is the frequency of the light, and we can measure the energy of an ejected electron by measuring the voltage V that is just sufficient to stop the electron moving. Then the voltage, frequency, and work function are related by the equation

$$V = \frac{h}{e}\nu - \phi,$$

where e is the charge on the electron. This equation was first given by Albert Einstein in 1905.

The data in the file `millikan.txt` represent frequencies ν in hertz (first column) and voltages V in volts (second column) from photoelectric measurements of this kind. Using the equation above and the program you wrote, and given that the charge on the electron is 1.602×10^{-19} C, calculate from Millikan's experimental data a value for Planck's constant. Compare your value with the accepted value of the constant, which you can find in books or on-line. You should get a result within a couple of percent of the accepted value.

This calculation is essentially the same as the one that Millikan himself used to determine of the value of Planck's constant, although, lacking a computer, he fitted his straight line to the data by eye. In part for this work, Millikan was awarded the Nobel prize in physics in 1923.

CHAPTER 4

ACCURACY AND SPEED

WE HAVE now seen the basic elements of programming in Python: input and output, variables and arithmetic, loops and if statements. With these we can perform a wide variety of calculations. We have also seen how to visualize our results using various types of computer graphics. There are many additional features of the Python language that we haven't covered. In later chapters of the book, for example, we will introduce a number of specialized features, such as facilities for doing linear algebra and Fourier transforms. But for now we have the main components in place to start doing physics.

There is, however, one fundamental issue that we have not touched upon. Computers have limitations. They cannot store real numbers with an infinite number of decimal places. There is a limit to the largest and smallest numbers they can store. They can perform calculations quickly, but not infinitely quickly. In many cases these issues need not bother us—the computer is fast enough and accurate enough for many of the calculations we do in physics. However, there are also situations in which the computer's limitations will affect us significantly, so it will be crucial that we understand those limitations, as well as methods for mitigating them or working around them when necessary.

4.1 VARIABLES AND RANGES

We have seen examples of the use of variables in computer programs, including integer, floating-point, and complex variables, as well as lists and arrays. Python variables can hold numbers that span a wide range of values, including very large numbers, but they cannot hold numbers that are arbitrarily large. For instance, the largest value you can give a floating-point variable is about 10^{308}. (There is also a corresponding largest negative value of about -10^{308}.) This is enough for most physics calculations, but we will see occasional ex-

amples where we run into problems. Complex numbers are similar: both their real and imaginary parts can go up to about $\pm 10^{308}$ but not larger.[1] Large numbers can be specified in scientific notation, using an "`e`" to denote the exponent. For instance, `2e9` means 2×10^9 and `1.602e-19` means 1.602×10^{-19}. Note that numbers specified in scientific notation are always floats. Even if the number is, mathematically speaking, an integer (like `2e9`), the computer will still treat it as a float.

If the value of a variable exceeds the largest floating-point number that can be stored on the computer we say the variable has *overflowed*. For instance, if a floating-point variable `x` holds a number close to the maximum allowed value of 10^{308} and then we execute a statement like "`y = 10*x`" it is likely that the result will be larger than the maximum and the variable `y` will overflow (but not the variable `x`, whose value is unchanged).

If this happened in the course of a calculation you might imagine that the program would stop, perhaps giving an error message, but in Python this is not what happens. Instead the computer will set the variable to the special value "`inf`," which means infinity. If you print such a variable with a print statement, the computer will actually print the word "`inf`" on the screen. In effect, every number over 10^{308} is infinity as far as the computer is concerned. Unfortunately, this is usually not what you want, and when it happens your program will probably give incorrect answers, so you need to watch out for this problem. It's rare, but it'll probably happen to you at some point.

There is also a smallest number (meaning smallest magnitude) that can be represented by a floating-point variable. In Python this number is 10^{-308} roughly.[2] If you go any smaller than this—if the calculation *underflows*—the computer will just set the number to zero. Again, this usually messes things up and gives wrong answers, so you need to be on the lookout.

What about integers? Here Python does something clever. There is no largest integer value in Python: it can represent integers to *arbitrary precision*. This means that no matter how many digits an integer has, Python stores all of them—provided you have enough memory on your computer. Be aware, however, that calculations with integers, even simple arithmetic operations, take longer with more digits, and can take a very long time if there are very

[1] The actual largest number is 1.79769×10^{308}, which is the decimal representation of the binary number 2^{1024}, the largest number that can be represented in the IEEE 754 double-precision floating-point format used by the Python language.

[2] Actually 2.22507×10^{-308}, which is 2^{-1022}.

many digits. Try, for example, doing `print(2**1000000)` in Python. The calculation can be done—it yields a number with 301 030 digits—but it's so slow that you might as well forget about using your computer for anything else for the next few minutes.[3]

Exercise 4.1: Write a program to calculate and print the factorial of a number entered by the user. If you wish you can base your program on the user-defined function for factorial given in Section 2.6, but write your program so that it calculates the factorial using *integer* variables, not floating-point ones. Use your program to calculate the factorial of 200.

Now modify your program to use floating-point variables instead and again calculate the factorial of 200. What do you find? Explain.

4.2 NUMERICAL ERROR

Floating-point numbers (unlike integers) are represented on the computer to only a certain precision. In Python, at least at the time of writing of this book, the standard level of precision is 16 significant digits. This means that numbers like π or $\sqrt{2}$, which have an infinite number of digits after the decimal point, can only be represented approximately. Thus, for instance:

True value of π:	3.1415926535897932384626 . . .
Value in Python:	3.141592653589793
Difference:	0.00000000000000002384626 . . .

The difference between the true value of a number and its value on the computer is called the *rounding error* on the number. It is the amount by which the computer's representation of the number is wrong.

A number does not have to be irrational like π to suffer from rounding error—any number whose true value has more than 16 significant figures will get rounded off. What's more, when one performs arithmetic with floating-point numbers, the answers are only guaranteed accurate to about 16 figures, even if the numbers that went into the calculation were expressed exactly. If

[3] If you do actually try this, then you might want to know how to stop your program if you get bored waiting for it to finish. The simplest thing to do is just to close the window where it's running.

you add 1.1 and 2.2 in Python, then obviously the answer should be 3.3, but the computer might give 3.299999999999999 instead.

Usually this is accurate enough, but there are times when it can cause problems. One important consequence of rounding error is that you should *never use an if statement to test the equality of two floats.* For instance, you should never, in any program, have a statement like

```
if x==3.3:
    print(x)
```

because it may well not do what you want it to do. If the value of `x` is supposed to be 3.3 but it's actually 3.299999999999999, then as far as the computer is concerned it's not 3.3 and the if statement will fail. In fact, it rarely occurs in physics calculations that you need to test the equality of floats, but if you do, then you should do something like this instead:

```
epsilon = 1e-12
if abs(x-3.3)<epsilon:
    print(x)
```

As we saw in Section 2.2.6, the built-in function `abs` calculates the absolute value of its argument, so `abs(x-3.3)` is the absolute difference $|x - 3.3|$. The code above tests whether this difference is less than the small number `epsilon`. In other words, the if statement will succeed whenever `x` is very close to 3.3, but the two don't have to be exactly equal. If `x` is 3.299999999999999 things will still work as expected. The value of `epsilon` has to be chosen appropriately for the situation—there's nothing special or universal about the value of 10^{-12} used above and a different value may be appropriate in another calculation.

The rounding error on a number, which we will denote ϵ, is defined to be the amount you would have to add to the value calculated by the computer to get the true value. For instance, if we do the following:

```
from math import sqrt
x = sqrt(2)
```

then we will end up not with $x = \sqrt{2}$, but rather with $x + \epsilon = \sqrt{2}$, where ϵ is the rounding error, or equivalently $x = \sqrt{2} - \epsilon$. This is the same definition of error that one uses when discussing measurement error in experiments. When we say, for instance, that the age of the universe is 13.80 ± 0.04 billion years, we mean that the measured value is 13.80 billion years, but the true value is possibly greater or less than this by an amount of order 0.04 billion years.

The error ϵ in the example above could be either positive or negative, depending on how the variable x gets rounded off. If we are lucky ϵ could be small, but we cannot count on it. In general if x is accurate to a certain number of significant digits, say 16, then the rounding error will have a typical size of $x/10^{16}$. It's usually a good assumption to consider the error to be a (uniformly distributed) random number with standard deviation $\sigma = Cx$, where $C \simeq 10^{-16}$ in this case. We will refer to the constant C as the *error constant*. When quoting the error on a calculation we typically give the value of the standard deviation σ. (We can't give the value of the error ϵ itself, since we don't know it—if we did, then we could calculate $x + \epsilon$ and recover the exact value for the quantity of interest, so there would in effect be no error in the calculation at all.)

Rounding error is important, as described above, if we are testing the equality of two floating-point numbers, but in other respects it may appear to be only a minor annoyance. An error of one part in 10^{16} does not seem very bad. But what happens if we now add, subtract, or otherwise combine several different numbers, each with its own error? In many ways the rounding error on a number behaves similarly to measurement error in a laboratory experiment, and the rules for combining errors are the same. For instance, if we add or subtract two numbers x_1 and x_2, with standard deviations σ_1 and σ_2, then standard results about combinations of random variables tell us that the *variance* σ^2 of the sum or difference is equal to the sum of the individual variances:

$$\sigma^2 = \sigma_1^2 + \sigma_2^2. \tag{4.1}$$

Hence the standard deviation of the sum or difference is

$$\sigma = \sqrt{\sigma_1^2 + \sigma_2^2}. \tag{4.2}$$

Similarly if we multiply or divide two numbers then the variance of the result x obeys

$$\frac{\sigma^2}{x^2} = \frac{\sigma_1^2}{x_1^2} + \frac{\sigma_2^2}{x_2^2}\,. \tag{4.3}$$

But, as discussed above, the standard deviations on x_1 and x_2 are given by $\sigma_1 = Cx_1$ and $\sigma_2 = Cx_2$, so that if, for example, we are adding or subtracting our two numbers, meaning Eq. (4.2) applies, then

$$\sigma = \sqrt{C^2x_1^2 + C^2x_2^2} = C\sqrt{x_1^2 + x_2^2}. \tag{4.4}$$

I leave it as an exercise to show that the corresponding result for the error on the product of two numbers $x = x_1 x_2$ or their ratio $x = x_1/x_2$ is

$$\sigma = \sqrt{2}\,Cx. \tag{4.5}$$

We can extend these results to combinations of more than two numbers. If, for instance, we are calculating the sum of N numbers $x_1 \ldots x_N$ with errors having standard deviation $\sigma_i = Cx_i$, then the variance on the final result is the sum of the variances on the individual numbers:

$$\sigma^2 = \sum_{i=1}^{N} \sigma_i^2 = \sum_{i=1}^{N} C^2 x_i^2 = C^2 N \overline{x^2}, \tag{4.6}$$

where $\overline{x^2}$ is the mean-square value of x. Thus the standard deviation on the final result is

$$\sigma = C\sqrt{N}\,\sqrt{\overline{x^2}}. \tag{4.7}$$

As we can see, this quantity increases in size as N increases—the more numbers we combine, the larger the error on the result—although the increase is a relatively slow one, proportional to the square root of N.

We can also ask about the *fractional* error on $\sum_i x_i$, i.e., the total error divided by the value of the sum. The size of the fractional error is given by

$$\frac{\sigma}{\sum_i x_i} = \frac{C\sqrt{N\,\overline{x^2}}}{N\,\overline{x}} = \frac{C}{\sqrt{N}}\,\frac{\sqrt{\overline{x^2}}}{\overline{x}}, \tag{4.8}$$

where $\overline{x} = N^{-1}\sum_i x_i$ is the mean value of x. In other words the fractional error in the sum actually *goes down* as we add more numbers.

At first glance this appears to be pretty good. So what's the problem? Actually, there are a couple of them. One is when the sizes of the numbers you are adding vary widely. If some are much smaller than others then the smaller ones may get lost. But the most severe problems arise when you are not adding but subtracting numbers. Suppose, for instance, that we have the following two numbers:

$$x = 1000000000000000$$
$$y = 1000000000000001.2345678901234$$

and we want to calculate the difference $y - x$. Unfortunately, the computer only represents these two numbers to 16 significant figures, which means that

as far as the computer is concerned:

$$x = 100000000000000$$
$$y = 100000000000001.2$$

The first number is represented exactly in this case, but the second has been truncated. Now when we take the difference we get $y - x = 1.2$, when the true result would be 1.2345678901234. In other words, instead of 16-figure accuracy, we now only have two figures and the fractional error is several percent of the true value. This is much worse than before.

To put this in more general terms, if the difference between two numbers is very small, comparable with the error on the numbers, i.e., with the accuracy of the computer, then the fractional error can become large and you may have a problem.

EXAMPLE 4.1: THE DIFFERENCE OF TWO NUMBERS

To see an example of this in practice, consider the two numbers

$$x = 1, \qquad y = 1 + 10^{-14}\sqrt{2}. \tag{4.9}$$

Trivially we see that

$$10^{14}(y - x) = \sqrt{2}. \tag{4.10}$$

Let us perform the same calculation in Python and see what we get. Here is the program:

```
from math import sqrt
x = 1.0
y = 1.0 + (1e-14)*sqrt(2)
print((1e14)*(y-x))
print(sqrt(2))
```

The penultimate line calculates the value in Eq. (4.10) while the last line prints out the true value of $\sqrt{2}$ (at least to the accuracy of the computer). Here's what we get when we run the program:

```
1.42108547152
1.41421356237
```

As we can see, the calculation is accurate to only the first decimal place—after that the rest is garbage.

This issue, of large errors in calculations that involve the subtraction of numbers that are nearly equal, arises with some frequency in physics calculations. We will see various examples throughout the book. It is perhaps the most common cause of significant numerical error in computations and you need to be aware of it at all times when writing programs.

Exercise 4.2: Quadratic equations

Consider a quadratic equation $ax^2 + bx + c = 0$ that has real solutions.

a) Write a program that takes as input the three numbers, a, b, and c, and prints out the two solutions using the standard formula

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}.$$

Use your program to compute the solutions of $0.001x^2 + 1000x + 0.001 = 0$.

b) There is another way to write the solutions to a quadratic equation. Multiplying top and bottom of the solution above by $-b \mp \sqrt{b^2 - 4ac}$, show that the solutions can also be written as

$$x = \frac{2c}{-b \mp \sqrt{b^2 - 4ac}}.$$

Add further lines to your program to print these values in addition to the earlier ones and again use the program to solve $0.001x^2 + 1000x + 0.001 = 0$. What do you see? How do you explain it?

c) Using what you have learned, write a new program that calculates both roots of a quadratic equation accurately in all cases.

This is a good example of how computers don't always work the way you expect them to. If you simply apply the standard formula for the quadratic equation, the computer will sometimes get the wrong answer. In practice the method you have worked out here is the correct way to solve a quadratic equation on a computer, even though it's more complicated than the standard formula. If you were writing a program that involved solving many quadratic equations this method might be a good candidate for a user-defined function: you could put the details of the solution method inside a function to save yourself the trouble of going through it step by step every time you have a new equation to solve.

Exercise 4.3: Calculating derivatives

Suppose we have a function $f(x)$ and we want to calculate its derivative at a point x. We can do that with pencil and paper if we know the mathematical form of the function, or we can do it on the computer by making use of the definition of the derivative:

$$\frac{\mathrm{d}f}{\mathrm{d}x} = \lim_{\delta \to 0} \frac{f(x+\delta) - f(x)}{\delta}.$$

On the computer we can't actually take the limit as δ goes to zero, but we can get a reasonable approximation just by making δ small.

a) Write a program that defines a function `f(x)` returning the value $x(x-1)$, then calculates the derivative of the function at the point $x = 1$ using the formula above with $\delta = 10^{-2}$. Calculate the true value of the same derivative analytically and compare with the answer your program gives. The two will not agree perfectly. Why not?

b) Repeat the calculation for $\delta = 10^{-4}$, 10^{-6}, 10^{-8}, 10^{-10}, 10^{-12}, and 10^{-14}. You should see that the accuracy of the calculation initially gets better as δ gets smaller, but then gets worse again. Why is this?

We will look at numerical derivatives in more detail in Section 5.10, where we will study techniques for dealing with these issues and maximizing the accuracy of our calculations.

4.3 PROGRAM SPEED

As we have seen, computers are not infinitely accurate. And neither are they infinitely fast. Yes, they work at amazing speeds, but many physics calculations require the computer to perform millions of individual computations to get a desired overall result and collectively those computations can take a significant amount of time. Some of the example calculations described in Chapter 1 took months to complete, even though they were run on some of the most powerful computers in the world.

One thing we need to get a feel for is how fast computers really are. As a general guide, performing a million mathematical operations is no big problem for a computer—it usually takes less than a second. Adding a million numbers together, for instance, or finding a million square roots, can be done in very little time. Performing a billion operations, on the other hand, could take minutes or hours, though it's still possible provided you are patient. Performing a trillion operations, however, will basically take forever. So a fair rule of thumb is that the calculations we can perform on a computer are ones that can be done with *about a billion operations or less.*

This is only a rough guide. Not all operations are equal and it makes a difference whether we are talking about additions or multiplications of single numbers (which are easy and quick) versus, say, calculating Bessel functions or multiplying matrices (which are not). Moreover, the billion-operation rule will change over time because computers get faster. However, computers have been getting faster a lot less quickly in the last few years—progress has slowed. So we're probably stuck with a billion operations for a while.

EXAMPLE 4.2: QUANTUM HARMONIC OSCILLATOR AT FINITE TEMPERATURE

The quantum simple harmonic oscillator has energy levels $E_n = \hbar\omega(n + \frac{1}{2})$, where $n = 0, 1, 2, \ldots, \infty$. As shown by Boltzmann and Gibbs, the average energy of a simple harmonic oscillator at temperature T is

$$\langle E \rangle = \frac{1}{Z} \sum_{n=0}^{\infty} E_n e^{-\beta E_n}, \tag{4.11}$$

where $\beta = 1/(k_B T)$ with k_B being the Boltzmann constant, and $Z = \sum_{n=0}^{\infty} e^{-\beta E_n}$. Suppose we want to calculate, approximately, the value of $\langle E \rangle$ when $k_B T = 100$. Since the terms in the sums for $\langle E \rangle$ and Z dwindle in size quite quickly as n becomes large, we can get a reasonable approximation by taking just the first 1000 terms in each sum. Working in units where $\hbar = \omega = 1$, here's a program to do the calculation:

File: `qsho.py`

```
from math import exp

terms = 1000
beta = 1/100
S = 0.0
Z = 0.0
for n in range(terms):
    E = n + 0.5
    weight = exp(-beta*E)
    S += weight*E
    Z += weight

print(S/Z)
```

Note a few features of this program:

1. Constants like the number of terms and the value of β are assigned to variables at the beginning of the program. As discussed in Section 2.7, this is good programming style because it makes them easy to find and modify and makes the rest of the program more readable.
2. We used just one for loop to calculate both sums. This saves time, making the program run faster.
3. Although the exponential $e^{-\beta E_n}$ occurs separately in both sums, we calculate it only once each time around the loop and save its value in the variable `weight`. This also saves time: exponentials take significantly longer to calculate than, for example, additions or multiplications. (Of course

"longer" is relative—the times involved are probably still less than a microsecond. But if one has to go many times around the loop even those short times can add up.)

If we run the program we get this result:

```
99.9554313409
```

The calculation (on my desktop computer) takes 0.01 seconds. Now let us try increasing the number of terms in the sums (which just means increasing the value of the variable `terms` at the top of the program). This will make our approximation more accurate and give us a better estimate of our answer, at the expense of taking more time to complete the calculation. If we increase the number of terms to a million then it does change our answer somewhat:

```
100.000833332
```

The calculation now takes 1.4 seconds, which is significantly longer, but still a short time in absolute terms.

Now let's increase the number of terms to a billion. When we do this the calculation takes 22 minutes to finish, but the result does not change at all:

```
100.000833332
```

There are three morals to this story. First, a billion operations is indeed doable—if a calculation is important to us we can probably wait twenty minutes for an answer. But it's approaching the limit of what is reasonable. If we increased the number of terms in our sum by another factor of ten the calculation would take 220 minutes, or nearly four hours. A factor of ten beyond that and we'd be waiting a couple of days for an answer.

Second, there is a balance to be struck between time spent and accuracy. In this case it was probably worthwhile to calculate a million terms of the sum—it didn't take long and the result was noticeably, though not wildly, different from the result for a thousand terms. But the change to a billion terms was clearly not worth the effort—the calculation took much longer to complete but the answer was exactly the same as before. We will see plenty of further examples in this book of calculations where we need to find an appropriate balance between speed and accuracy.

Third, it's pretty easy to write a program that will take forever to finish. If we set the program above to calculate a trillion terms, it would take weeks to run. So it's worth taking a moment, before you spend a whole lot of time

writing and running a program, to do a quick estimate of how long you expect your calculation to take. If it's going to take a year then it's not worth it: you need to find a faster way to do the calculation, or settle for a quicker but less accurate answer. The simplest way to estimate running time is to make a rough count of the number of mathematical operations the calculation will involve; if the number is significantly greater than a billion, you have a problem.

EXAMPLE 4.3: MATRIX MULTIPLICATION

Suppose we have two $N \times N$ matrices represented as arrays `A` and `B` on the computer and we want to multiply them together to calculate their matrix product. Here is a fragment of code to do the multiplication and place the result in a new array called `C`:

```
from numpy import zeros
N = 1000
C = zeros([N,N],float)

for i in range(N):
    for j in range(N):
        for k in range(N):
            C[i,j] += A[i,k]*B[k,j]
```

We could use this code, for example, as the basis for a user-defined function to multiply arrays together. (As we saw in Section 2.4.4, Python already provides the function "`dot`" for calculating matrix products, but it's a useful exercise to write our own code for the calculation. Among other things, it helps us understand how many operations are involved in calculating such a product.)

How large a pair of matrices could we multiply together in this way if the calculation is to take a reasonable amount of time? The program has three nested for loops in it. The innermost loop, which runs through values of the variable `k`, goes around N times doing one multiplication operation each time and one addition, for a total of $2N$ operations. That whole loop is itself executed N times, once for each value of `j` in the middle loop, giving $2N^2$ operations. And those $2N^2$ operations are themselves performed N times as we go through the values of `i` in the outermost loop. The end result is that the matrix multiplication takes $2N^3$ operations overall. Thus if $N = 1000$, as above, the whole calculation would involve two billion operations, which is feasible in a few minutes of running time. Larger values of N, however, will rapidly become intractable. For $N = 2000$, for instance, we would have 16 billion op-

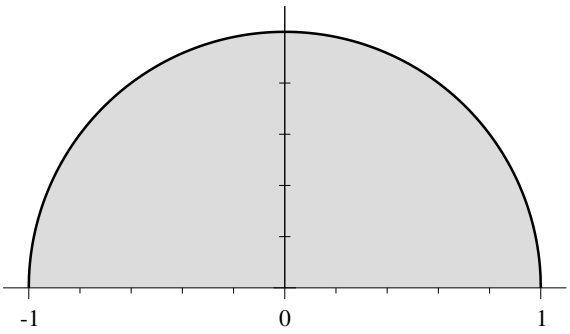
erations, which could take hours to complete. Thus the largest matrices we can multiply are about 1000×1000 in size.[4]

Exercise 4.4: Calculating integrals

Suppose we want to calculate the value of the integral

$$I = \int_{-1}^{1} \sqrt{1 - x^2}\, \mathrm{d}x.$$

The integrand looks like a semicircle of radius 1:



and hence the value of the integral—the area under the curve—must be equal to $\frac{1}{2}\pi = 1.57079632679\ldots$

Alternatively, we can evaluate the integral on the computer by dividing the domain of integration into a large number N of slices of width $h = 2/N$ each and then using the Riemann definition of the integral:

$$I = \lim_{N\to\infty} \sum_{k=1}^{N} h y_k\,,$$

---

[4]Interestingly, the direct matrix multiplication represented by the code given here is not the fastest way to multiply two matrices on a computer. *Strassen's algorithm* is an iterative method for multiplying matrices that uses some clever shortcuts to reduce the number of operations needed so that the total number is proportional to about $N^{2.8}$ rather than N^3. For very large matrices this can result in significantly faster computations. Unfortunately, Strassen's algorithm suffers from large numerical errors because of problems with subtraction of nearly equal numbers (see Section 4.2) and for this reason it is rarely used. On paper, an even faster method for matrix multiplication is the *Coppersmith–Winograd algorithm*, which requires a number of operations proportional to only about $N^{2.4}$, but in practice this method is so complex to program as to be essentially worthless—the extra complexity means that in real applications the method is always slower than direct multiplication.

where

$$y_k = \sqrt{1 - x_k^2} \qquad \text{and} \qquad x_k = -1 + hk.$$

We cannot in practice take the limit $N \to \infty$, but we can make a reasonable approximation by just making N large.

a) Write a program to evaluate the integral above with $N = 100$ and compare the result with the exact value. The two will not agree very well, because $N = 100$ is not a sufficiently large number of slices.

b) Increase the value of N to get a more accurate value for the integral. If we require that the program runs in about one second or less, how accurate a value can you get?

Evaluating integrals is a common task in computational physics calculations. We will study techniques for doing integrals in detail in the next chapter. As we will see, there are substantially quicker and more accurate methods than the simple one we have used here.