APPENDIX B

DIFFERENCES BETWEEN PYTHON VERSIONS

THE Python programming language is continually being updated and improved by its creators. The most recent version is version 3, which is the version used in this book, but version 2 is still available and finds wide use. (The much earlier version 1, which dates back to the 1980s, is now seen very rarely.) For our purposes the differences between versions 2 and 3 are not very great, and you can use the book with version 2 quite easily, but there are a small number of significant differences between the versions that you will need to know about, which are described in this appendix.

If you don't want to read through the whole appendix in detail, then the main thing you need to know is that to use version 2 with this book you should include at the beginning of all your programs the following statement:

```
from __future__ import division,print_function
```

(Note the two underscore characters "`_`" on either side of the word "`future`".) This statement tells version 2 of Python to behave in the way version 3 does with respect to the two most important differences, the differences in the division of integers and the structure of the print command. Also, everywhere that the programs in this book use the function "`input`", you should use instead the function "`raw_input`". For details, see below.

Division returns a floating-point value: In version 2 of Python the division of two integers, one by another, returns another integer, any fractional part of the result being discarded. Thus `3/2` gives 1, not 1.5. In version 3 the same operation gives 1.5. Furthermore, even if the result of a division is in fact an integer, the operation will still give a floating-point value in version 3. Thus `4/2` gives a floating-point 2.0, not an integer 2.

If you are using version 2 of Python, you can duplicate the behavior of version 3 with respect to division by including the statement

```
from __future__ import division
```

at the start of your program. If you are using version 2 I recommend you do this with all your programs if you are using this book—it will prevent a lot of misunderstandings and annoyances.

Print is a function: In version 3 of Python the print command is a *function*, where in version 2 it is a *statement*. The main practical difference between the two is that in version 3 you must enclose the argument(s) of a print command within parentheses, while in version 2 you would not. Thus in version 2 you might say

```
print "The energy is",E
```

while in version 3 you would say

```
print("The energy is",E)
```

In most other respects the two commands behave in the same way.

If you are using version 2 of Python, you can duplicate the behavior of the version 3 print command by including the statement

```
from __future__ import print_function
```

at the start of your program. Again if you are using version 2 I recommend that you do this in all your programs when you are using this book. It will avoid a lot of annoyances.

Input returns a string: In version 3 of Python the input function always returns a *string*, no matter what you type in, even if you type in a number. In version 2, by contrast, the input statement takes what you type and evaluates it as an algebraic expression, then returns the resulting value. Thus if you write a program that includes the statement

```
x = input()
```

and you enter "2.5", the result will be different in versions 2 and 3. In version 2, `x` will be a floating-point variable with numerical value 2.5, while in version 3 it will be a string with string value "2.5". In version 2 if you entered an actual string like "Hello" you would get an error message, while in version 3 this works just fine.

Version 2 of Python includes another function called `raw_input`, which behaves the same way that `input` does in version 3. Thus if you are using version 2, you can still use the programs in this book if you just write "`raw_input`" everywhere that the (version 3) programs in the book currently say "`input`". (In version 3 `raw_input` no longer exists.)

There is only one integer type: In version 2 of Python there are two types of integer variables called int and long. Variables of type int are restricted to numbers in the range $\pm 2^{31}$, but arithmetic using them is very fast; variables of type long can store numbers of arbitrary size but arithmetic using them is slower. In version 3 of Python there is only one type of integer variable, called int, which subsumes both the earlier types. For smaller integer values version 3 will automatically use old-style ints with their fast arithmetic while for larger values it will automatically use old-style longs but slower arithmetic. You do not need to worry about the distinction between the two—Python takes care of it all for you.

In fact, this change appeared earlier than version 3 of Python, starting in version 2.4. If you are using version 2 of the language, it's most likely that you are using either version 2.6 or 2.7, in which case you don't need to worry about this point—version 2.4 and later already behave the same way as version 3 with respect to integer variables and you do not need to do anything special to request this behavior.

Iterators: An *iterator* is an object in Python that behaves something like a list. It is a collection of values, one after another, but it differs from a list in that the values are not stored in the memory of the computer waiting for you to look them up; instead they are calculated on the fly, which saves memory. Thus, for example, in version 2 of Python the function `range` returns an actual list of numbers, which occupies space in the computer memory. This can cause problems if the list is very large. For instance, in version 2 on most computers the statement

```
for n in range(10000000000):
```

will give an error message because there is not enough memory to store the huge list generated by the `range` function. To get around this problem version 2 provides another function called `xrange`, which acts like `range` but produces an iterator. Thus "`xrange(100)`" behaves in many respects like a list of 100 elements, but no actual list is created. Instead, when you ask for, say, the tenth element, the computer just works out what the tenth element ought to be and hands it to you. The value is never stored anywhere. Thus you could say

```
for n in xrange(1000000000000):
```

and the program would run just fine, without crashing (although it would take forever to finish because the loop is so long).

In version 3 of Python `range` behaves the way `xrange` does in version 2, producing an iterator, not a true list. Since the most common use of `range` by

far is in for-loops, this is usually an improvement: it saves memory and often makes the program run faster. Sometimes, however, you may actually want to generate a real list from a range. In that case you can use a statement of the form

```
x = list(range(100))
```

which will create an iterator then convert it into a list. In version 2 of Python you do not need to do this (although it will work fine if you do).

(It's worth noting that the function `arange` in the package `numpy`, which is similar to `range` but works with arrays rather than lists, really does create an array, not an iterator. It calculates all the values of the array and stores them in memory, rather than calculating them on the fly. This means that using `arange` with large arguments can slow your program or cause it to run out of memory, even in Python version 3.)

Another situation in which iterators appear in version 3 is the `map` function, which we studied in Section 2.4.1. Recall that `map` applies a given function to each element of a list or array. Thus

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

applies the natural logarithm function separately to each element of the list `[1.0,1.5,2.2]`. In version 2 of Python this operation produces another list with the three logarithms in it, but in version 3 it produces an iterator. That is, it doesn't actually compute the logarithms and store them away. Instead it waits until you ask for the value of a particular element of `logr` and then calculates what that value is. Again, this is usually an improvement—it typically saves time and memory space. Sometimes, however, you may need a real list, in which case you can generate one with a statement of the form

```
logrlist = list(logr)
```

which converts the iterator into a true list.

There are a number of other differences between versions 2 and 3 of Python, but none of them will affect the programs in this book. For our purposes, the ones above are the ones you need to know about. A full description of the differences can be found on-line at the main Python web site `www.python.org`.