1

Preliminaries

1.1 Introduction

The aim of this book is to give a practical introduction to performing simulations of molecular systems. This is accomplished by summarizing the theory underlying the various types of simulation method and providing a programming library, called pDynamo, which can be used to perform the calculations that are described. The style of the book is pragmatic. Each chapter, in general, contains some theory about related simulation topics together with descriptions of example programs that illustrate their use. Suggestions for further work (or exercises) are listed at the end.

By the end of the book, readers should have a good idea of how to simulate molecular systems as well as some of the difficulties that are involved. The pDynamo library should also be a reasonably convenient starting point for those wanting to write programs to study the systems they are interested in. The fact that users have to write their own programs to do their simulations has advantages and disadvantages. The major advantage is flexibility. Many molecular modeling programs come with interfaces that supply only a limited range of options. In contrast, the simulation algorithms in pDynamo can be combined arbitrarily and much of the data generated by the program is available for analysis. The drawback is that the programs have to be written – a task that many readers may not be familiar with or have little inclination to do themselves. However, those who fall into the latter category are urged to read on. pDynamo has been designed to be easy to use and should be accessible to everyone even if they have only a minimum amount of computing experience.

This chapter explains some essential background information about the programming style in which pDynamo and the example programs are written. Details of how to obtain the library for implementation on specific machines are left to the appendices.

1.2 Python

All the example programs in this book and much of the programming library are written in the programming language Python. The rest of the library, which most readers will never need to look at, consists of code for which computational efficiency is paramount and is written in C. The reasons for the choice of Python were threefold. First, it is a powerful and modern programming language that is fun to use! Unlike languages such as C and FORTRAN, it is an interpreted language, which means that programs can be run immediately without going through separate compilation and linking steps. Second, Python is open-source software that is free and runs under a wide variety of operating systems and, third, there is a very active development community that is continually enhancing the language and adding to its capabilities.

Most computer languages are easiest to learn by example and Python is no exception. The following, simple program illustrates several basic features of the language:

```
1  """Example 0."""
2
3  import math
4
5  # . Define a squaring function.
6  def Square ( x ):
7      return x**2
8
9  # . Create a list of integers.
10 values = range ( 10 )
11
12 # . Loop over the integers.
13 for i in values:
14     x = float ( i )
15     print "%5d%10.5f%10.5f%10.5f" \
               % ( i, x, math.sqrt ( x ), Square ( x ) )
```

Line 1 is the program's *documentation string* which, in principle, should give a concise description of what the program is supposed to do. All the examples in this book, however, have documentation strings of the type `"""Example n."""` to save space and to avoid duplicating the explanations that occur in the text.

Lines 2, 4, 8 and 11 are blank and are ignored.

Line 3 makes the standard Python *module* `math` accessible to the program. Python itself and programs written using Python – including pDynamo – consist of modules which must be explicitly *imported* if their contents are to be used.

The `import` statement has a number of different forms and the one shown is the simplest. With this form, module items are accessed by prefixing the item's name with the module name, followed by a dot (`.`) character. Thus, the function `sqrt` from the module `math`, which is used on *line 15* of the program, is accessed as `math.sqrt`. An alternative form, which is sometimes preferable, is `from math import sqrt`. This makes it possible to refer to the function `sqrt` by its name only without the `math.` prefix.

Lines 5, 9 and 12 are *comments* which are included to make the program easier to understand. Python ignores all characters from the hash character (`#`) until the end of the line.

Lines 6–7 define a very simple Python *function*. Functions are named collections of instructions that can be *called* or *invoked* at different points in a program. They behave similarly in Python to functions in other languages, such as C and FORTRAN.

Line 6 is the function definition line. It starts with the word `def` which tells Python that a function definition is coming and terminates with a colon (`:`). The second word on the line, `Square`, is the name that we are giving to the function and this is followed by the function's *arguments* which appear in parentheses. Arguments are variables that the function needs in order to work. Here there is only one, `x`, but there can be many more.

The function definition line is followed by the *body* of the function. This would normally consist of several lines but here there is only one, *line 7*. Python is unusual among programming languages in that the lines in the function body are determined by line indentation. In other languages, such blocks of code are delimited by specific characters, such as the matching braces {...} of C or the `FUNCTION` ... `END FUNCTION` keywords of FORTRAN 90. The number of spaces to indent by is arbitrary – in this book it is always four – but all lines must be indented by the same amount and instructions after the end of the function must return to the original indentation level.

Line 7 is very simple. The second part of the line contains the expression `x**2` which computes the square of the function's argument `x`. The `**` symbol denotes the power operator and so the expression tells Python to raise `x` to the power of 2. The first part of the line is the keyword

`return` which says that the result of the squaring calculation is to be *returned* to the place from which the function was called.

Line 10 is the first executable line of the example and illustrates several more features of the Python language – *built-in functions*, *sequence types* and *variable assignment*. `range` is one of Python's built-in functions and is always available whenever the Python interpreter is invoked. It produces a sequence of integers, in this case ten of them, starting with the value `0` and finishing with the value `9`. Python, like C, but unlike FORTRAN, starts counting from zero and not from one. The integers are returned as a *list* which is one of Python's built-in sequence data types and is one of the things that makes Python so attractive to use. Finally the list of integers is assigned to a variable with the name `values`. Python differs from many languages in that variables do not need to be declared as being of a particular type. In C, for example, an integer variable would have to be declared with a statement such as "`int i ;`" before it could be used. These declarative statements do not exist in Python and so `values` can be assigned arbitrarily to refer to any data type.

Line 13 shows one of the forms of *iteration* in Python. The statement takes the list referred to by `values` and assigns each of its elements to the variable `i` in turn. The iteration stops when the end of the list is reached. The lines over which iteration is to occur are determined by line identation in exactly the same way as those in the body of a function.

Line 14 is the first line of the loop specified by the `for` construct in *line 13*. It takes the integer referred to by the variable `i`, converts it to a *floating-point number* using the built-in function `float` and then assigns it to the variable `x`.

Line 15 is printed as two lines in the text, due to the restricted page width, but it is logically a single statement. The presence of the backslash character (\) at the end of the line indicates to Python that the subsequent line is to be treated as a continuation of the current one.

The statement prints the values of `i` and of `x`, the square root of `x`, which is calculated by invoking the function `sqrt` from the module `math`, and the square of `x`, calculated using the previously defined function `Square`. These items are grouped together at the end of the line in a *tuple*. Tuples, like lists, are one of Python's built-in sequence data types and are constructed by enclosing the items that are to be in the tuple in parentheses. Tuples differ from lists, though, in that they are *immutable*, which means that their contents cannot be changed once they have been created.

The style in which the quantities are printed is determined by the *formatting string*, which is enclosed in double quotes `""`. This is placed after the `print` keyword and is separated from the tuple of items to be printed by the `%` character. Python employs a syntax for formatting operations that is very similar to that of the C language. Output fields start with a `%` character and so, in this example, there are four output fields in the string, one for each of the items to be printed. The first output field is `%5d`, which says that an integer, coded for by the letter `d`, is to be printed in a field `5` characters wide. The remaining fields are identical and have the form `%10.5f`. They are for the output of floating-point numbers (`f`) in fields `10` characters wide but with `5` of these characters occurring after the decimal point.

It is not, of course, possible to master a language from a single, short program but readers should gain in expertise and come to appreciate more fully the capabilities of the language as they work through the examples and exercises in the book. One of the great advantages of Python for learning is that it can be used interactively and so it is quick and easy to write simple programs to test whether one really understands what the language is doing.

1.3 Object-oriented programming

Python admits various programming styles but all the modules in pDynamo are written using an *object-oriented* approach in which the basic unit of programming is the *class*. A class encapsulates the notion of an *object*, such as a file or a molecule, and groups together the data or *attributes* needed to describe the object and the functions or *methods* that are required to manipulate it. Classes are used by *instantiating* them so that, for example, a program for modeling the molecules methanol and water would create two instances of the class molecule, one to represent methanol and one water.

There are other important aspects of object-oriented programming that pDynamo employs but which will only be alluded to briefly, if at all, later on. Two of these are *inheritance* and *polymorphism*. Inheritance is the mechanism by which a new class is defined in terms of an existing one. For example, a class for manipulating organic molecules could be derived from a more general class for molecules. The new class would inherit all the attributes and methods defined by its parent class but would also have attributes and methods specific for organic molecules. Polymorphism is related to inheritance and is the ability to redefine methods for derived classes. Thus, the general molecule class could have a method for chemical reactions but this would

be *overridden* in the organic molecule class because the rules for implementing reactions for organic molecules are different from those of the general case.

It is time to consider a simple, hypothetical example. Suppose there were a *class hierachy* designed for writing to text files. The *base class* would be a general class, `TextFileWriter`, and there would be a *subclass*, `DataFileWriter`, designed for writing either a specific type of data or data in a specific format. The base class has the following (partial) specification:

```
1  class TextFileWriter ( object ):
2      """The base class for objects that write to text files."""
3
4      def __init__ ( self, filename ):
5          """Instance initializer from |filename|."""
6          self.name = filename
7          ... other initialization here ...
8
9      def Close ( self ):
10         """Close the file."""
11         ... contents here ...
```

Line 1 says that a class of name `TextFileWriter` is being defined and that it is subclassed from the class `object` which is the base class for all Python objects.

Line 2 is the documentation string for the class.

Lines 4 to 7 define a special method, `__init__`, that is called when an instance of a class is created.

The method has two arguments. The first, `self`, denotes the instance of the class that is calling the method (hence the name `self`). Python requires that this argument appears in the specification of all instance methods in a class but that it should not be present when a method is actually invoked. We shall see examples of this later in the section. The second argument, `filename`, gives the name of the file to which data are to be written.

The body of the method, *line 6* onwards, would be used to perform various 'start-up' operations on the newly created instance. This could include the initialization of various attributes that the instance may need and the setting up of the necessary data structures for writing to a file with the given name. The only operation that we show explicitly is on *line 6* because it illustrates how an attribute of an instance can be defined – in this case, the attribute, `name`, of the instance `self` that points to the

name of the file. The dot-notation identifies the attribute as belonging to the instance and is employed when accessing an attribute as well as for its definition.

Lines 9 to 11 define a method, `Close`, that would be called when writing to the file has terminated.

The subclass is specified as follows:

```
1 class DataFileWriter ( TextFileWriter ):
2     """A class for writing data to a text file."""
3
4     def WriteData ( self, data ):
5         """Write |data| to the file."""
6         ... contents here ...
```

Line 1 defines `DataFileWriter` as a subclass of `TextFileWriter`.

Lines 4 to 6 define a method, `WriteData`, that would be called to write the argument `data` to the file.

The methods `__init__` and `Close` are absent from the specification of the subclass, which means that they are inherited from the parent class, `TextFileWriter`, and behave in an identical fashion.

Once the classes and its methods have been defined, data could be written to a file of the class `DataFileWriter` with the following series of commands:

```
1 datafile = DataFileWriter ( "myfile.dat" )
2 datafile.WriteData ( data )
3 datafile.Close ( )
4 print "Data written to the file", datafile.name
```

Line 1 creates an instance of the class `DataFileWriter` that is called `datafile`. The instance is produced using the name of the class followed by parentheses that contain the arguments, excluding `self`, that are to be passed to the class's `__init__` method. In this case there is a single argument, `"myfile.dat"`, which is a character string that contains the name of the file to be written. In the rest of this book, we shall refer to statements in which instances of a class are generated using the class name as *constructors*.

Line 2 calls the `WriteData` method of the instance with the data to be written. Methods of an instance are most usually invoked using the dot-notation. This is similar to the way in which the attributes of an instance are

accessed but with the difference that the method's arguments appear in parentheses after the method name. As discussed above, the `self` argument that occurs in the specification of the method is absent.

Line 3 closes the file as writing to the file has terminated. The `Close` method takes no arguments but parentheses are still required.

Line 4 prints a short informational message. This version of the print statement is simpler than that used previously in that no formatting information is present. Instead, Python chooses suitable defaults for the way in which the string `"Data written to the file"` and the `name` attribute of `datafile` are written.

Although the class notation is very elegant, it can be a little cumbersome to use. This is why, for many classes, pDynamo supplies 'helper' functions that provide a shorthand way of using the class without having to explicitly instantiate it. An appropriate helper function for the class `DataFileWriter` would simply be one that 'wraps' the four-line program given previously. It would have the form:

```
def DataFile_Write ( filename, data ):
    """Write |data| to the data file with name |filename|."""
    datafile = DataFileWriter ( filename )
    datafile.WriteData ( data )
    datafile.Close ( )
    print "Data written to the file", datafile.name
```

and would be used as follows

```
DataFile_Write ( "myfile.dat", data )
```

1.4 The pDynamo library

Like many large Python libraries, pDynamo is hierarchically organized into *packages* and modules. A package is a named collection of modules that are put together because they perform logically related tasks, whereas a module is a collection of Python classes, functions and other instructions that are grouped in a single Python file. As we saw in the example program of Section 1.2, modules from a library can be used in Python programs by importing them with the `import` keyword. The same syntax is possible with packages, so the

statement `import mypackage` would make the contents of the package called `mypackage` accessible.

pDynamo consists of three principal packages. The first and most fundamental package is `pCore`. It contains modules implementing various basic data structures and algorithms that are independent of molecular applications. The second package is `pDynamo`, which has modules for representing and manipulating molecular systems and for performing molecular simulation. The third package is `pBabel`, which has modules that read and write information for chemical systems in various formats. The packages are arranged hierarchically because `pBabel` depends upon both `pDynamo` and `pCore`, `pDynamo` upon `pCore`, but not `pBabel`, and `pCore` upon neither.

The purpose of this book is not to provide a detailed description of each of the pDynamo packages. Instead, only a subset of pDynamo's classes and functions will be introduced as needed. Some, whose behaviour and construction are deemed important for the arguments being pursued in the text, will be described in detail, whereas others will be mentioned only in passing. A summary of all the items from the pDynamo library appearing in the book is given in Appendix 1, whereas full documentation will be found online with the library's source code and the book's example programs.

1.5 Notation and units

To finish, a few general points about the notation and units used in this book and the program library will be made. In the text, all program listings and the definitions of classes, methods, functions and variables have been represented by using characters in typewriter style, e.g. `molecule`. For other symbols, normal typed letters are used for scalar quantities whereas bold face italic letters are employed for vectors and bold face roman for matrices. Lower case letters have generally been taken to represent the properties of individual atoms whereas upper case letters represent the properties of a group of atoms or, more usually, the entire system. Lower case roman subscripts normally refer to atoms, upper case roman subscripts to entire structures and Greek subscripts to other quantities, such as the Cartesian components of a vector or quantum chemical basis functions. The more common symbols are listed in Tables 1.1, 1.2 and 1.3.

The units of most of the quantities either employed or calculated by pDynamo are specified in Table 1.4. All the quantum chemical algorithms use *atomic units* internally although little input or output is done in them. Nevertheless, for completeness, Table 1.5 lists some quantities in atomic units and their pDynamo equivalents.

Table 1.1 *Symbols that denote quantities for atoms or for the entire system*

Symbol	Description
Atomic quantities	
α_i	Isotropic dipole polarizability for atom i
$\boldsymbol{\zeta}_i$	Mass-weighted first derivatives of potential energy
$\boldsymbol{a}_i$ $(\equiv \ddot{\boldsymbol{r}}_i)$	Acceleration of atom i
$\boldsymbol{f}_i$	Force on atom i
$\boldsymbol{g}_i$	First derivatives of potential energy with respect to coordinates of atom i
$\mathbf{h}_{ij}$	Second derivatives of potential energy with respect to coordinates of atoms i and j
m_i	Mass of atom i
$\boldsymbol{p}_i$	Momentum vector for atom i
$\boldsymbol{q}_i$	Vector of mass-weighted Cartesian coordinates for atom i
q_i	Partial charge for atom i
$\boldsymbol{r}_i$	Vector of Cartesian coordinates, (x_i, y_i, z_i), for atom i
r_{ij}	Distance between two atoms i and j
$\boldsymbol{s}_i$	Vector of Cartesian fractional coordinates for atom i
$\boldsymbol{v}_i$ $(\equiv \dot{\boldsymbol{r}}_i)$	Velocity vector for atom i
w_i	Weighting factor for atom i
x_i	x Cartesian coordinate of atom i
y_i	y Cartesian coordinate of atom i
z_i	z Cartesian coordinate of atom i
System quantities	
$\boldsymbol{\mu}$	Dipole-moment vector
σ_{IJ}	Root mean square coordinate deviation between structures I and J
$\boldsymbol{A}$	$3N$-dimensional vector of atom accelerations
$\boldsymbol{D}$	$3N$-dimensional coordinate displacement vector
$\boldsymbol{F}$	$3N$-dimensional vector of atom forces
$\boldsymbol{G}$	$3N$-dimensional vector of first derivatives
G_{RMS}	Root mean square (RMS) gradient
$\mathbf{H}$	$(3N \times 3N)$-dimensional matrix of second derivatives of system
$\boldsymbol{\mathcal{I}}$	Inertia matrix
$\mathbf{M}$	$3N \times 3N$ diagonal atomic mass matrix
O	System observable or property
$\boldsymbol{P}$	$3N$-dimensional vector of atom momenta
Q	Charge
$\boldsymbol{R}$	$3N$-dimensional vector of atom coordinates
$\boldsymbol{R}_{\mathrm{c}}$	Centre of charge, geometry or mass
$\boldsymbol{S}$	$3N$-dimensional vector of atom fractional coordinates
$\boldsymbol{V}$	$3N$-dimensional vector of atom velocities