CHAPTER 1

Importing data from files

Most experimentally biased scientists and engineers carefully store the data resulting from past work. These archived data may be stored in many different formats. Indeed, data that you are collecting now may be in several different formats because it comes from different experiments that use different data acquisition techniques, software, and computer hardware. If you analyze these data with a system other than that with which they were acquired, chances are that you will need to perform some sort of conversion. You will certainly need to "import" those data into whichever system you are using.

Your *Mathematica* system is no exception to this general rule, but the task of reading in, or importing, data from non-*Mathematica* files is really rather easy. Once the data are in *Mathematica*, you will be able to use all of *Mathematica*'s functionality to graph and to analyze them – perhaps exporting the data at a later date. *Mathematica* gives you considerable flexibility in how you handle data. For example, if your data have a particular structure, you will be able to retain that structure, even if it contains a mix of numeric data types and text. If you are used to dealing with mixed-type structures in C (or records in Pascal) then you will be able to build upon your experience with such data groupings when you work with *Mathematica*.

In this chapter we show you how to import data from ASCII and binary files; we also show you how, in general terms, to extract data that may be embedded amidst other, nonpertinent, contents of a file, and to maintain any structure that might be present. First, we begin by showing you how to make *Mathematica* navigate around the filesystem of your computer.

1.1 File operations in general

Before you can import data from a file, you need to tell *Mathematica* where that file is to be found. All computer systems operate some kind of filing system. At the top-most level, the filing system will use as the first element of a file's name the name of the disk on which the data are stored. For example, the computer on which this is being written has three disks attached, named `Macintosh HD`, `Q127 Alpha`, and `Q127 Beta`; the latter two disks are actually one physical hard disk that is partitioned into two logical disks by the operating system – so the computer sees two independent disks. Such physical partitioning does not affect what we are about to show: it is the name (or symbol) of the disk on which your data are resident that is significant.

Within each disk there will be some kind of lower-level partitioning. A disk on a multi-user system will have directories, one for each user. A disk on a single-user machine (or a user's directory on a multi-user machine) also will have directories or folders. These, too, may contain many sets of nested subdirectories. Typically, each level of directory structure is delineated by some special character like \, /, or :, depending on which operating system is in use – DOS, UNIX, or Macintosh respectively.

Before you begin to read and write files using any applications software, you should feel confident about how your computer system defines the disk and directories that contain a file, and also what filenames are allowed. For example, you should know how many characters you can use in a filename, whether you are disallowed certain characters (like spaces or colons) within a filename, and which character is used to indicate a new level within the filing system's directory structure. We suggest also that you try out new techniques only on files for which you have backup copies – which preferably are stored away on media not connected to your computer.

1.1.1 Locating files

At any time during a *Mathematica* session, *Mathematica* has a default directory for ongoing work. By default, it is this directory that *Mathematica* will access for all file operations. You can see the name of the default directory by using the **`Directory`** function, here shown for a Macintosh system in which filenames can have spaces, and in which levels within the file system are delineated by colons:

In:

```
    Directory[]
```

Out:

```
    Q127Beta:Mathematica 2.2
```

As returned by **`Directory`**, the current default directory is called `Mathematica 2.2` and is located on a disk called `Q127Beta`. If the default directory happens to contain the file you wish *Mathematica* to read, then you need go no further. If the file is elsewhere, then you have three options. Either you can set the default directory to be the directory that contains the file, you can explicitly specify the full filename whenever you want to access the file, or you can add the name of the directory in which the file is located to *Mathematica*'s **`$Path`** variable. **`$Path`** is a list of directories through which *Mathematica* will search in order to find a file. For example, if you know the directory name, then you can use the **`SetDirectory`** function to make *Mathematica* use that directory as the default. The **`FileNames`** function will then list all the files within that directory.

In:

```
SetDirectory["Q127Alpha:Sam's observations"]
```

Out:

```
Q127Alpha:Sam's observations
```

In:

```
FileNames[]
```

Out:

```
{First experiment, noGood.dat}
```

Note that the filenames within the chosen directory have been returned as a list, enclosed in curly brackets, like all *Mathematica* lists. If you get the directory name wrong, *Mathematica* will warn you with a message like

Out:

```
 SetDirectory::cdir: Cannot set current directory to
Q127Alpha:Sam's experiment.
```

If you are going to work often on a file or on a group of files that will always be resident in one particular directory, adding that directory to *Mathematica*'s default search path can save you time. You can see which device and directory combinations are already on the default search path by just typing **`$Path`**.

In:

```
$Path
```

Out:

```
{Q127Beta:Mathematica 2.2:Packages,
Q127Beta:Mathematica 2.2:Packages:StartUp, :}
```

You can append your chosen directory's name to the search path using the **`AppendTo`** function.

In:

```
AppendTo[$Path,"Q127Alpha:Sam's observations"]
```

Out:

```
{Macintosh HD:Mathematica 2.2.2:Packages,
Macintosh HD:Mathematica 2.2.2:Packages:StartUp, :,
Q127Alpha:Sam's observations}
```

Now, *Mathematica* will automatically search the directory `Sam's observations` on the disk `Q127Alpha`. (You can also use the **`PrependTo`** function to make the new directory name the first element in **`$Path`**.) Once you have established that *Mathematica* can access the directory containing the required file, you can proceed to make *Mathematica* read it.

In addition to knowing where a file is located in your computer's filesystem, you also need to know the type of file and how the data are structured within the file.

1.2 File types

Data are stored typically in one of two main types of file: ASCII or binary. In ASCII files, data are stored in a printable format using the ASCII code; data in binary files are store in base-2 form. The advantage of using ASCII data files is that usually you can type or print the files to see what is in them. Accessing data in binary files is not so simple because you need to know how the data were stored in the files – owing to the different ways that computers store integers and floating point numbers, just to take two examples. Trying to type or print a binary file can produce strange results – including "random" flashing of the terminal screen and multiple formfeeds from printers – because the device to which the contents of the file are being sent will interpret some of the binary values as device control codes, which are interpreted as instructions for the device to behave in a particular way. However, binary files normally occupy less space, and so they are especially useful for large sets of data. *Mathematica* can read and write both binary and ASCII files with ease.

If your data are contained in multiple files, we show you later in the chapter how you can use *Mathematica* to access all the files automatically, or even how to use the data themselves to specify which files have to be accessed next or how the data are to be stored.

1.3 Data structures

Before reading in any data, you will find it useful to know both how the data are arranged in the file to be read and how you want to use the data after it is read into *Mathematica*. The former is essential; the latter is merely desirable. If the data are in some kind of structure, or are related in some way, then you might want to keep that structure or relationship. In C or Pascal, such structures are given special names like `struct` and `record`. *Mathematica* handles data and structures in a more general way: all groups of data are lists, and a list can contain identical or differently typed members.

For example, a chronological date might consist of three numbers – the year, the month, and the day. In *Mathematica*, you can keep these three quantities within a list. Here is a list (containing a date) that has been assigned to a variable name **`myDate`**:

In:

```
    myDate={1900,1,1}
```

Out:

```
    {1900, 1, 1}
```

Of course, you might want to mix data of different types within this date structure. For example, a different format of date might contain two integers and a string.

In:

```
    myDate2={1900,"January",1}
```

Out:

```
    {1900, January, 1}
```

Note that although *Mathematica* has not displayed `January` in quotes, it is treated as a string; you can verify this by using the **`FullForm`** function to explicitly display the attributes of **`myDate2`**.

In:

```
    FullForm[myDate2]
```

Out:

```
    List[1900, "January", 1]
```

In each case, you can extract parts of the date without worrying about the type of data held in that part. Here, we use the double bracket ([[ ]]) form of the function **`Part`** which returns list parts.

In:

```
    month=2;
    myDate[[month]]
```

Out:

```
    1
```

In:

```
myDate2[[month]]
```

Out:

```
January
```

Maintaining any structure inherent in your data has many advantages; related data can be kept in structures, and you can write object-oriented functions to operate on those structures (for example, see Maeder (1994) and Riddle & Dick (1994)).

1.4 Simple ASCII files

The simplest method of interchanging information between computer programs is to use ASCII files to store that information. The information components are easily read by eye, by word processor, or by spreadsheet and most applications programs support the import and export of data in this format. If you are authoring your own programs, writing information to a file in ASCII will be supported by the compiler or interpreter that you are using. For import into *Mathematica*, you might find it easier to always include a dividing character or space between, say, numbers. Tightly packed, formatted FORTRAN output, for example, will require a little more effort to read into *Mathematica* compared with the same output where, say, a space has been placed between every item output. Of course, number-separating spaces also make that information easier to read by eye.

In this section, we look at importing numbers and strings from files, but we also cover other nonexclusively related issues – string – number conversion, accessing variably named files, and content cataloging – that are of more general interest. So, even if you are disinterested in reading data from ASCII files for now, you still might want to browse through these other issues.

1.4.1 Numbers and strings from free-format ASCII files

Free-format files, in this context, are files in which each item in the file is bounded by a delimiting character, normally a space. The most straightforward function that you can use to read in free-format data is **ReadList**, which takes two arguments: the name of the file from which the data are to be read, and the type of data that are to be read. Here, we read in data from a file that contains a short list of metals and their melting points.

In:

```
myMetals=ReadList["Q127Alpha:Sam's observations:metals",
                   {Word,Number}]
```

Out:

```
{{Bismuth, 271}, {Lead, 327}, {Lithium, 179}, {Iron,
1537}, {Copper, 1085}}
```

There are several points worth noting. The filename is enclosed in double quotes and we have specified that we expect to read in one or more structures and that each structure consists of a word (the metal's name) and a number (the numeric melting-point of the metal). We have specified that each structure has a word and a number by enclosing those type names in curly brackets, *Mathematica*'s notation for a list. The types that *Mathematica* recognizes are: **Byte**, **Character**, **Real** (an approximate number in a FORTRAN-like format – for example, 1 or 1.342 or 1.3e4), **Number** (an exact number, such as 5 or 1025, or an approximate number in FORTRAN-like format), **Word** (delimited by word-separating characters that you can define), **Record** (delimited by record-separating characters that you can define), **String** (delimited by a new line), or **Expression** (a complete *Mathematica* expression).

Once these data have been read, we can access them individually or grouped in their structure by using the **Part** function. For example, the second part of the list **myMetals** can be obtained by

In:

```
Part[myMetals,2]
```

Out:

```
{Lead, 327}
```

or by the more usual form of the **Part** function, with double square brackets:

In:

```
myMetals[[2]]
```

Out:

```
{Lead, 327}
```

We reach the next level of the structure by specifying a further level with the **Part** function:

In:

```
myMetals[[2,2]]
```

Out:

```
327
```

By specifying **Number**, we have made *Mathematica* assume that an exact number is to be read. The file `metalsAgain` contains a mix of integer and real values. Because we have used **Number**, numbers read are left in their purest form: integers as integers, reals as reals.

In:

```
ReadList["Q127Alpha:Sam's observations:metalsAgain",
         {Word,Number}]
```

Out:

```
{{Bismuth, 271.1}, {Lead, 327}, {Lithium, 179.3},
{Iron, 1537}, {Copper, 1085}}
```

Had we specified **Real** as the type, *Mathematica* would have converted all the numbers to their approximate form (identifiable by the omnipresence of the decimal point), regardless of the original form of the number:

In:

```
ReadList["Q127Alpha:Sam's observations:metalsAgain",
         {Word,Real}]
```

Out:

```
{{Bismuth, 271.1}, {Lead, 327.}, {Lithium, 179.3},
{Iron, 1537.}, {Copper, 1085.}}
```

We also could have read in each line as a string by specifying the imported type to be **String**.

In:

```
myValues=ReadList["Q127Alpha:Sam's observations:
                  metals",{String}]
```

Out:

```
{{Bismuth 271}, {}, {Lead 327}, {}, {Lithium 179}, {},
{Iron 1537}, {}, {Copper 1085}}
```

Note that there are several empty lists, **{}**, in **myValues**. These may be caused, for example, by blank lines in the file. In this instance, using **String** makes separating out the names and the melting points more difficult, should we want to do so later. In general, therefore, it is best to make all parts of your data as accessible as possible. For example, although we can still access each member of the list, it is now more difficult to access the numbers as separate entities – and an error message is generated if we use **Part** inappropriately.

In:

```
myValues[[3]]
```

Out:

```
{Lead 327}
```

In:

```
myValues[[3,2]]
```

Out:

```
Part::partw: Part 2 of {Lead 327} does not exist.
{{Bismuth 271}, {}, {Lead 327}, {}, {Lithium 179}, {},
{Iron 1537}, {}, {Copper 1085}}[[3,2]]
```

To overcome this problem, we can read each item within the file as a separate entity by using the type **Word**.

In:

```
myNewValues=ReadList["Q127Alpha:Sam's
                      observations:metals",
                      {Word}]
```

Out:

```
{{Bismuth}, {271}, {Lead}, {327}, {Lithium}, {179},
 {Iron}, {1537}, {Copper}, {1085}}
```

Note that when you read in data as type **Word**, each entity is read as a string. You will not be able to perform mathematical operations on the number-like strings directly because *Mathematica* treats strings as literals – and you need to display the full format of the string to see that it is indeed a string.

In:

```
myNewValues[[4]]+1
```

Out:

```
{1 + 327}
```

In:

```
FullForm[ myNewValues[[4]] ]
```

Out:

```
List["327"]
```

Of course, we can still access and mathematically manipulate the numbers, but to do so we need to know how to convert between strings and numbers.

1.4.2 String – number conversion

The ability to convert between numbers expressed as strings and numbers proper (expressed as numerals) enables you to extract usable numbers from strings (and vice versa) and to create variably valued strings that you can then use, for example, to access multiple files. We have seen that the elements of the list **myNewValues** are strings: when we display them fully using **FullForm**, the elements are enclosed in double quotes. If you want to manipulate a value encoded as a string, then you need to make the string a *Mathematica* expression. The function **ToExpression** converts its argument into a *Mathematica* expression; it will therefore convert a string to a manipulatable number. Here, we create the one-item-long list **myNumber**, whose (single) element we can then manipulate.

In:

```
myNumber=ToExpression[ myNewValues[[4]] ]
```

Out:

```
{327}
```

In:

```
myNumber+3
```

Out:

```
{330}
```

Here, we have converted a string to a number. You can also use **ToExpression** to convert any string (say, a symbolic formula) into a usable *Mathematica* expression.

The inverse of **ToExpression** is **ToString**. You can use the **ToString** function to make any *Mathematica* expression a string. You may find it useful to apply the function **FullForm** to see exactly how *Mathematica* is formatting your data; without using **FullForm**, the following three expressions – a list of numbers, a string version of that list, and a list of strings – appear identical:

In:

```
myNumbers=Table[i,{i,10}]//FullForm
```

Out:

```
List[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

In:

```
myString=ToString[myNumbers]//FullForm
```

Out:

```
"{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}"
```

In:

```
myString2=Table[ToString[i],{i,10}]//FullForm
```

Out:

```
List["1", "2", "3", "4", "5", "6", "7", "8", "9", "10"]
```

1.4.3 Files with embedded comments

If your data files have comments, you may want to extract those comments for, say, a log of what each file contains or else to use the comment information to decide how to process the file. Most ASCII data files should contain