Reprint from
The *Mathematica* Conference

June, 1992   Boston, MA

857.15.10.1992

Reading Data into *Mathematica*

John M. Novak

Mathematica can be a powerful tool for the manipulation of information. To process information, however, it must first be made accessible to the kernel, or read into *Mathematica*. Luckily, *Mathematica* provides many tools to read in data, and put it into a useful form. In the following sections, we will explore some of these techniques, particularly those associated with reading files.

First, a note on how *Mathematica* treats files. A file can be viewed as a set of data to be accessed in a linear fashion, or a stream. Normally, you start at the beginning, and read in new data sequentially until you reach the end. Each new piece of data read in increments a pointer to the current position in the stream by some amount. It is also possible to adjust this pointer to an arbitrary position, so that further data can be read in from that location. To use *Mathematica* in this fashion, it is necessary that you open the file for access, using the `Open` command. The file is then read by using the `Stream` object returned by `Open`. Examples of this will be seen in the following text.

1 Organization of Data

When reading in data, it is important to consider the organization of data in the file, and the desired organization in *Mathematica*. The functions for reading in data (`Read` and `ReadList`) acquire information from the file in chunks of specified types. One then needs to determine how the chunks that can be read in correspond to the layout of data in the file.

`Byte`	single byte of data, returned as an integer, the only binary type that can be read in directly
`Real`	approximate number in Fortran-like notation
`Number`	exact or approximate number in Fortran-like notation
`Character`	single character returned as a one-character string
`Word`	sequence of characters delimited by word separators
`Record`	sequence of characters delimited by record separators
`String`	string terminated by newline
`Expression`	complete *Mathematica* expression

The chunks of data that can be read in directly.

Note that there are essentially three kinds of chunks of data that *Mathematica* can read in: those that return numbers, those that return strings, and those that return complex *Mathematica* expressions. Those that return numbers include the only binary type that *Mathematica* directly understands (a byte). All other types of data are based on text files. These include the `Real` and `Number` types; the only difference between these is that the `Number` type will treat an integer as an exact number in *Mathematica*, rather than converting it to a floating-point number.

Of the string-returning types, `Character` and `String` are fairly straightforward; the former returns the next character, while the latter returns a string containing the remainder of a line, where the end of the line is delineated by a newline character. Note that the marker for a new line will vary from machine to machine; on a Macintosh, is a carriage return (ASCII 13), while on most Unix-based machines, is a newline (ASCII 10). On MS-DOS- (or MS-Windows) based machines, it is a carriage return-newline pair of characters.

The `Word` and `Record` types are very similar in nature; they return a string that is delimited by specified substrings. Both allow different substrings to be specified for left and right delimiters. The primary difference is in intended usage, as indicated by the default delimiters; the `Word` type is delimited by whitespace, giving it the usual interpretation of a word, while the `Record` is delimited by the end-of-line marker. A `Record` corresponds to a record in a flat database stored in a text file with carriage returns between records.

The final type of data object is the `Expression`; this corresponds to *Mathematica* input, and is read in as if parsing a file as if it were being entered at a *Mathematica* command line. This means that input is processed a line at a time, until the expression appears complete at the end of a line.

The `Word` and `Record` types are particularly special; there are a number of standard options to `Read` and `ReadList` controlling how these are read in.

option name	*default value*	
`NullWords`	`False`	whether to assume an empty word exists between repeated word speparators
`WordSeparators`	`{" ", "\t"}`	separators allowed between words, can be specified as left and right separators
`TokenWords`	`{}`	words that can be used as delimiters for other words, while still being returned as words
`NullRecords`	`False`	whether to assume an empty record exists between repeated record separators
`RecordSeparators`	`{"\n"}`	separators between records
`RecordLists`	`False`	whether to break input into lists at positions of record separators, used only with `ReadList`

Options for `Read` and `ReadList`.

The `NullWords` and `NullRecords` options are useful when your data set may have multiple separators in a row, between which you do not want records to be present. An example of this would be if you had a database in which some fields were left empty. Setting `NullWords` to `True` would ensure that an empty field is read into *Mathematica* as an empty string, preserving the positions of fields in an array.

The `TokenWords` option allows additional strings to be specified as word separators, but are themselves words as well. This is useful in parsing expressions involving mathematical operators, for instance. (See The *Mathematica* Book, page 498 for an example of this usage.)

`RecordLists` is an option that can only be used with `ReadList`. It causes a new list of the objects being read in to be created whenever a record separator is encountered. This makes it easy to read in arrays of objects from text files. Before, for an array that was *n* elements across, one had to give a list of *n* objects to `Read`, or else partition the list after is was read in. Now, the syntax `ReadList[`*file*`, Number,RecordLists -> True]` can be used.

Careful consideration of the types of data that can be read in, and the options controlling these types, compared to the arrangement of data in our file, can simplify how we read in the data. For instance, suppose one had a data file with a ragged array of elements, the first element indicating how many elements follow in that row.

Here is a set of data.

```
In[1]:= !! firstdata.txt

3 1 2 4
5 1 3 3 2 3
3 2 3 3
2 2 4
```

Here is one way that we could read in our set of data.

```
In[1]:= (out = {};
        stream = OpenRead["firstdata.txt"];
        While[(n = Read[stream, Number]) =!= EndOfFile,
         AppendTo[out, ReadList[stream, Number, n]]
        ];
        Close[stream];
        out)

Out[1]= {{1, 2, 4}, {1, 3, 3, 2, 3}, {2, 3, 3}, {2, 4}}
```

We can make this operation simpler by noticing features of the structure of our data file. Here, we note that each list is in fact a record. Since we use the entire file, we do not need to maintain an intermediate stream.

```
In[2]:= out = Map[Drop[#,1]&,
         ReadList["firstdata.txt", Number, RecordLists -> True]
         ]

Out[2]= {{1, 2, 4}, {1, 3, 3, 2, 3}, {2, 3, 3}, {2, 4}}
```

2 Text Files

Files of information can be roughly placed in two classes: text files and binary files. Text files consist only of printable characters, the ASCII character set. They are sometimes referred to as ASCII files. Information in a text file is generally human-readable; a number is represented as the characters corresponding to the digits of the number. Binary files, on the other hand, store data in machine-specific formats. Both types of files can be handled in *Mathematica*, but text files are considerably easier to deal with.

Numbers can be represented in many formats. The `Read` command understands only some of the most common ways of expressing numbers. At this time, a number can be represented as a sequence of digits, possibly containing a single decimal point or with a leading sign ('−' or '+'). It can also have trailing the letter 'e' or 'E', followed by an exponent (a sign followed by a sequence of digits.) Leading whitespace is allowed. The `Read` or `ReadList` functions can read in numbers if the file position is at the start of a sequence of characters that matches the above description. If not, an `EndOfFile` marker is returned, though the stream position is left at the current location so reads of other data types can be performed. Note that this description disallows numbers with commas, or extracting strings of digits from arbitrary sequences of characters.

If a number contains a decimal point, it is always read in as a real number, whether you are using `Real` or `Number` as the data type. However, if you are using `Number`, integers (a sequence of digits, optionally followed by a positive exponent) are read in as *Mathematica* `Integer` objects. An integer with a negative exponent is turned into a `Rational`. Note that it cannot read in a rational type otherwise, however (that is, an integer, followed by a '/', followed by another integer.) The type `Real` coerces all types that it can read to reals, including integers.

Here is a file with some numbers in various formats.

```
In[3]:= !! numdata.txt

1.2343  5344
12.239e3 -34.3423E-4
34343e2    234e-2
```

First we read the data as numbers; note especially the behavior with integers.

```
In[3]:= ReadList["numdata.txt", Number]
```

Out[3]= $\{1.2343,\ 5344,\ 12239.,\ -0.00343423,\ 3434300,\ \frac{117}{50}\}$

Now we read the data in, coercing it to reals.

```
In[4]:= ReadList["numdata.txt", Real]

                                                       6
Out[4]= {1.2343, 5344., 12239., -0.00343423, 3.4343 10 , 2.34}
```

The various types of strings are generally handled in a much more straightforward fashion. The `Character` type reads in a single 8-bit character (corresponding to a single byte; remember that *Mathematica* can normally handle 16-bit characters, but at this time, `Read` cannot.) The `String` type reads in a line of text, from the current stream position to the end of the current line (marked by the machine's end-of-line character, as described above.) The stream position is set at the beginning of the next line. Note that the returned string does not contain the end-of-line character. The `Word` and `Record` types work as described above; the only thing to keep in mind here is that the separators can be any characters, and do not have to be the end-of-line or whitespace characters. This allows a string to be returned containing a carriage return, for instance.

Here is a small worksheet created in Microsoft Excel, and saved as a text file.

```
In[5]:= !! exceldata.txt

Quantity Name Cost
2 Gadget 23.04
1 Widget 15.06
4 Thingy 789.95
```

First we need to open the file for access, since we aren't going to read it in all at once this time. Note the stream object that is returned.

```
In[5]:= stream = OpenRead["exceldata.txt"]

Out[5]= InputStream[exceldata.txt, 8]
```

We can get the column headings for further use later. Because they are of a different type than the data, it is better to handle them separately.

```
In[6]:= columnheads = Read[stream, {Word, Word, Word}];
```

Now we read in the data. Note that we can mix these various types into a single read.

```
In[7]:= data = ReadList[stream, {Number, Word, Real}]

Out[7]= {{2, Gadget, 23.04}, {1, Widget, 15.06 },
  {4, Thingy, 789.95}}
```

Remember to close a stream after you are done using it; some file systems maintain a maximum number of streams that can be open by any one program at a time.

```
In[8]:= Close[stream]

Out[8]= exceldata.txt
```

The `Expression` type returns a compound *Mathematica* expression, as expressed in `InputForm` in the text file. Note that the expression is evaluated, and is handled much as input at a *Mathematica* command line is handled.

This file contains three *Mathematica* expressions. Note how the position of the ends of each line determines where the expressions end.

```
In[9]:= !! expressions.txt

(x^2 + 3 x + 2) (2 y +
3 z + Sin[Pi])
(y^2 + 2y + 1)
(z + y + Cos[x])
```

Here is what it is like in *Mathematica*. Notice that the expressions are evaluated; this can be prevented by wrapping `Hold` around the word `Expression` in the input line.

```
In[9]:= ReadList["expressions.txt", Expression]

                      2                       2
Out[9]= {(2 + 3 x + x ) (2 y + 3 z), 1 + 2 y + y ,
  y + z + Cos[x]}
```

3 Binary Files

The only binary type of data the *Mathematica* kernel can understand is a byte. Reading in an object of type `Byte` returns an integer representing the value of that byte. To handle more complex types, you would normally need to read in the bytes representing that type, then combine them in the appropriate fashion.

However, with *Mathematica* 2.1, there is a new package called `Utilities`BinaryFiles``. This adds several functions to assist you in dealing with binary files.

First, we have functions to open binary files. These are direct equivalents of their normal counterparts. They are: `OpenReadBinary`, `OpenWriteBinary`, and `OpenAppendBinary`. In fact, these are just calls to their standard equivalents, with options such as `PageWidth` set to something appropriate for binary files.

The `WriteBinary` command attempts to convert its second argument to its byte equivalent, that is, a list of integers between 0 and 255 inclusive. The conversion function can be user specified. The bytes are then written out in sequence.

Finally, we have the read functions: `ReadBinary` and `ReadListBinary`. These behave similarly to their standard counterparts, with a few differences. They accept only one option: since none of the binary types uses delimiters, there are no options for controlling the delimiter type. The option is `ByteOrder`, which can be set to `MostSignificantByteFirst` or `LeastSignificantByteFirst`. This determines whether in multibyte integers, the first byte is the most significant (high order) or least significant (low order) byte. The other major difference is that the argument that gives the type of data to be read in is not evaluated until after the data has been read into it.

This is particularly useful for certain types of structural manipulation on reading in data. `Read` evaluates the type of input; thus, if you attempt to use `Read[`*stream*`, Byte + Byte]`, it evaluates to `Read[`*stream*`, 2 Byte]`, where it then fails, because `Read` requires that all the leaves of the expression be types that can be read in. `ReadBinary`, on the other hand, given the same input, would read two bytes from *stream*, and add them together. Note that `ReadBinary` also accepts leaves that are not types that can be read in; it ignores these for the purposes of reading in data, and uses them in the evaluation of the expression once the data is read in.

There are several types of objects that can be handled by `ReadBinary` and `ReadListBinary`. Here is a list:

`Int8`	8-bit integer
`Byte`	same as `Int8`
`SignedInt8`	signed (two's complement) 8 bit integer
`Int16`	16-bit integer
`SignedInt16`	signed (two's complement) 16 bit integer
`Int32`	32-bit integer
`SignedInt32`	signed (two's complement) 32-bit integer
`Single`	single precision (approx. 8-digit) IEEE floating point
`Double`	double precision (approx. 16-digit) IEEE floating point
`CString`	null-terminated string

Types of objects that can be handled by `ReadBinary`.

Note that the signed types all understand the more common two's complement format, not the sign bit format. Also, the IEEE single-precision number takes up 4 bytes, while the double-precision fits in 8 bytes.

When read in, each of these objects is returned as its *Mathematica* equivalent. That is, the integers come back as integers, the floating points as machine-precision reals, and the `CString` as a string.

This loads the package.

```
In[10]:= << Utilities`BinaryFiles`
```

Here is a file of floating-point numbers in binary format. If we read them in as bytes, we have a good deal of difficulty in interpreting them.

```
In[11]:= ReadList["doubles.bin", Byte]

Out[11]= {61, 219, 124, 223, 217, 215, 189, 187 , 62, 17, 46,
  11, 232, 38, 214, 149, 62, 69, 121, 142, 226, 48, 140, 58,
  62, 122, 215, 242, 154, 188, 175, 72, 62, 176, 198,  247, 160,
  181, 237, 141, 62, 228, 248, 181, 136, 227, 104, 24 1, 63, 26,
  54, 226, 235, 28, 67, 45, 63, 80, 98, 77, 210, 241 , 169, 252,
  63, 132, 122, 225, 71, 174, 20, 123, 63, 185, 153, 153, 153,
  153, 153, 154, 63, 240, 0, 0, 0, 0, 0, 0, 64, 36,  0, 0, 0, 0,
  0, 0, 64, 89, 0, 0, 0, 0, 0, 0, 64, 143, 64, 0, 0, 0, 0, 0,
  64, 195, 136, 0, 0, 0, 0, 0, 64, 248, 106, 0, 0, 0, 0, 0, 65,
  46, 132, 128, 0, 0, 0, 0, 65, 99, 18, 208, 0, 0, 0, 0, 65,
  151, 215, 132, 0, 0, 0, 0, 65, 205, 205, 101, 0, 0, 0, 0, 66,
  2, 160, 95, 32, 0, 0, 0}
```

However, if we read these in using `ReadListBinary`, they now make sense.

```
In[12]:= ReadListBinary["doubles.bin", Double]

                -10        -9        -8        -7        -6
Out[12]= {1. 10   , 1. 10  , 1. 10  , 1. 10   , 1. 10  ,
  0.00001, 0.0001, 0.001, 0.01, 0.1, 1., 10., 100., 100 0.,
                        6        7        8        9        10
  10000., 100000., 1. 10 , 1. 10 , 1. 10 , 1. 10 ,  1. 10   }
```

Here is a piece of a real database for geographical information. Each coordinate is represented by a code and an ID that are bytes, followed by the latitude and longitude in minutes of arc. Each line segment is represented by a particular code.

```
In[13]:= geodata = ReadListBinary["geodata.bin",
             {Int8, Int8, SignedInt16, SignedInt16}]

Out[13]= {{3, 233, 622, 6267}, {0, 5, 514, 6 291},
  {0, 5, 600, 6349}, {0, 5, 575, 6393}, {0, 5, 7 03, 6555},
  {0, 5, 915, 6537}, {0, 5, 1054, 6395}, {0, 5, 1054, 6395},
  {0, 5, 1176, 6347}, {0, 5, 1293, 6479}, {3, 234 , 698, 6175},
  {0, 5, 625, 6267}, {3, 235, 374, 6126}, {0, 5, 430, 6025},
  {0, 5, 622, 5949}, {0, 5, 811, 6016}, {0, 5, 7 59, 6052},
  {0, 5, 698, 6175}, {3, 236, 385, 6008}, {0, 5, 170, 6077},
  {0, 5, 76, 6211}, {0, 5, 87, 6258}, {0, 5, 288 , 6207},
  {0, 5, 374, 6126}, {3, 237, 621, 5925}, {0, 5, 493, 5897},
  {0, 5, 385, 6008}}
```

Here is a preview of our next topic; we will process this to get our segments. We start by finding where the segments are.

```
In[14]:= segstarts = Flatten[Position[geodata, {3,_,_,_ }]]

Out[14]= {1, 11, 13, 19, 25}
```

We can turn this into the positions of the segments we need. (Each segment runs from the start of the current segment to one element before the next segment.)

```
In[15]:= segments = Transpose[{Drop[segstarts,-1], Drop[se
         gstarts, 1] - 1}]

Out[15]= {{1, 10}, {11, 12}, {13, 18}, {1 9, 24}}
```

Now we have the segments.

```
In[16]:= Map[ Take[geodata,#]&, segments]

Out[16]= {{{3, 233, 622, 6267}, {0, 5, 514, 6291},
   {0, 5, 600, 6349}, {0, 5, 575, 6393}, {0, 5, 703, 6555},
   {0, 5, 915, 6537}, {0, 5, 1054, 6395}, {0, 5,  1054, 6395},
   {0, 5, 1176, 6347}, {0, 5, 1293, 6479}},
  {{3, 234, 698, 6175}, {0, 5, 625, 6267}},
  {{3, 235, 374, 6126}, {0, 5, 430, 6025}, {0, 5 , 622, 5949},
   {0, 5, 811, 6016}, {0, 5, 759, 6052}, {0, 5, 698, 6175}},
  {{3, 236, 385, 6008}, {0, 5, 170, 6077}, {0, 5 , 76, 6211},
   {0, 5, 87, 6258}, {0, 5, 288, 6207}, {0, 5, 374, 6126}}}
```

4 Post-processing Input

Often, it is difficult or more time consuming to arrange the data in exactly the right form for further manipulation as it is being read in. We can take advantage of one of *Mathematica*'s most powerful features, its programming capabilities, to improve on this situation. By processing our data further after it is read in, we can easily mold our data to our purposes.

As illustrated in the example above, the list operations are some of the most useful for this sort of operation; functions such as `Position` to find where irregularly positioned marker data are located, `Take` to pull out subsections of data, and `Map` to apply functions equally to every element of our data. Another function in this vein is `Partition`; it allows us to break up data read in as a list into equally sized chunks.

We have a data file in which a bitmap is stored. The file has dimensions specified first, followed by the pixels, but with no specification of where the end of each raster row is. We first open the file for further access.

```
In[17]:= stream = OpenRead["ship.pgm"]

Out[17]= InputStream[ship.pgm, 13]
```

Here is the header information. Note that by reading in this fashion, we save the crucial piece of data (the width of the scan lines) in an easy to use form.

```
In[18]:= {id, width, height, depth} = Read[stream, {S tring,
         Number, Number, String}]

Out[18]= {P5, 164, 164, }
```

This gets the picture data, before further manipulation.

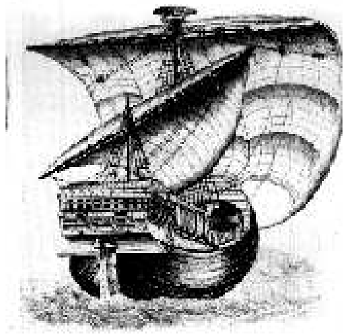
```
In[19]:= (picdata = ReadList[stream, Byte]); Close[stream]

Out[19]= ship.pgm
```

We can use partition to easily break this up into reasonable chunks.

```
In[20]:= picdata = Partition[picdata, width];
```

We can show our bitmap with a simple `ListDensityPlot`.

```
In[21]:= ListDensityPlot[-Reverse[picdata],
              Frame -> False, Mesh -> False,
              AspectRatio -> Automatic]
```



More complex post-processing is also possible. This allows *Mathematica* to be used as a powerful file translation tool; data can be imported in one form, and written in another, possibly after modifying the content in an appropriate fashion, for use by special purpose tools. (For instance, I have used *Mathematica* as a translation tool between a standard spreadsheet and a database, each of which required the data to be in a subtly different form.)

5 Conclusions

We have found many useful tools in *Mathematica* for handling data in streams. These range from being able to import simple numeric data with `Read` and `ReadList`, to being able to process data with the powerful programming capabilities present in *Mathematica*. We can easily handle text files, and with the new package `Utilities`BinaryFiles``, various binary formats as well.

These are not the limits of *Mathematica*; there are many other ways of handling data. To these, *MathLink* is a significant addition, allowing *Mathematica* to exchange structured *Mathematica* expressions with other programs. For most purposes, however, the above tools are quite sufficient, and are often simpler to implement. With these, *Mathematica* can be easily used to explore information derived from the real world.