Reprint from
The *Mathematica* Conference

June, 1992   Boston, MA

858.15.10.1992

String and Text Manipulation

John M. Novak

1 String Manipulation 2

2 Using Strings: Input and Output............ 3

3 Converting Strings 5

4 Files 6

5 Conclusions 7

Strings are a very important part of *Mathematica*. Of the various raw objects in *Mathematica* only strings can represent general textual data. While symbols can be used in some cases, it should be noted that this can lead to many problems; for instance, symbols cannot contain spaces. If you have large quantities of data, attempting to use symbols for the representation would quickly increase the size of the symbol table, degrading performance in the long run. Also, it is much more difficult to modify a symbol name to create a new symbol. Generally, symbols are not flexible enough for the sorts of manipulations possible with strings.

Strings are not only useful in handling textual data; they also can be used for output formatting. There are a number of functions in *Mathematica* dedicated to manipulating output forms. Further, input, before it is parsed, is veiwed as strings. This leads to many possibilities where string manipulation can change how *Mathematica* is seen by the user.

Mathematica as a powerful programming tool, provides many functions for the manipulation of strings. In the following sections, we will examine these, along with some possible uses.

Finally, strings and text files are often closely related, both as a type of data read from such files, and as a pattern for analyzing them. We will examine some of these connections.

1 String Manipulation

The elementary tools for string manipulation are the various `String` functions. They often parallel standard list manipulation functions. The ones we are concerned with first are those for finding out something about a string. These can be very useful in determining what modification to make to a string, or where to make it.

`StringQ[`*expr*`]`	determines whether *expr* is a string or not
`StringMatchQ[`*string*`,` *stringpattern*`]`	determines whether the first argument is matched by the pattern
`StringPosition[`*string*`,` *substring*`]`	gives a list of the positions of *substring* in the *string*
`StringLength[`*string*`]`	return number of characters in *string*
`StringByteCount[`*string*`]`	return number of bytes necessary to store *string*

Functions to return information about a string.

`StringQ` is quite straightforward; if the expression has head `String`, it returns `True`, otherwise it returns `False`. `StringMatchQ`, on the other hand, is a good deal more powerful. It allows two strings to be compared for equivalence. Further, the pattern string can also contain *string* `metacharacters`. These are wild-card characters, that is, characters that can represent other characters. The metacharacters used by *Mathematica* are `*`, which can represent a sequence of zero or more arbitrary characters, and `@`, which can represent a sequence of zero or more characters exluding upper-case letters. These metacharacters can be escaped by preceding them with a backslash (ie, `\*` for a literal asterisk.)

An example of using `StringMatchQ` with a string metacharacter.

```
In[1]:= StringMatchQ["abcdef","ab@ef"]

Out[1]= True
```

Note that the `@` character does not match upper-case letters.

```
In[2]:= StringMatchQ["abCDef", "ab@ef"]

Out[2]= False
```

`StringPosition` can also use the metacharacters in the substring. However, it returns the position of the substring, not whether the substring exists. Each position is expressed as a pair of integers, representing the start and the finish of the subexpression. This form is suitable for use with `StringTake` or `StringDrop` (described later.) Both `StringPosition` and `StringMatchQ` accept the option `IgnoreCase`; when set to `True`, the differences in case between the pattern and the string will be ignored. (The upper- and lower-case forms of letters are determined by the global symbol `$Letters`.) `StringPosition` also accepts the option `Overlaps -> False`; this causes the positions of overlapping instances of the pattern to be ignored.

This returns the location of the substring "abab" from the given string.

```
In[3]:= StringPosition["13tabababab3c&", "abab"]

Out[3]= {{4, 7}, {6, 9}}
```

Now the overlapping instance of the substring is not returned.

```
In[4]:= StringPosition["13tabababab3c&", "abab",
            Overlaps -> False]

Out[4]= {{4, 7}}
```

The functions `StringLength` and `StringByteCount` both give some indication of the size of the string; in fact, if the string only uses 8-bit characters, the two values will be identical. Remember, however, that *Mathematica* understands a 16-bit character set; if 16-bit characters are present in the string, the `StringByteCount` will be larger.

`StringTake[`*string*`,` *position*`]`	returns the substring between the characters specified by *position*
`StringDrop[`*string*`,` *position*`]`	returns the string resulting from deleting the characters specified by *position*
`StringJoin[`$string_1$`,` $string_2$`, ...]`	returns the string given by concatenating the $string_i$
`StringReverse[`*string*`]`	returns the string that is the reverse of the input string
`StringReplace[`*string*`,` *substring* `->` *replacement*`]`	returns the string in which instances of *substring* in *string* have been replaced by *replacement*

Functions that return modified strings.

The majority of the functions that modify strings are virtually identical to their list-based counterparts. `StringJoin` has a handy infix operator form that allows strings to be concatentated with the notation $string_1$ `<>` $string_2$ `<> ...`; this is often more readable than the full notation. Otherwise, `StringReplace` is the only oddball here. This function accepts a replacement rule, or list of replacement rules, and applies them to successive substrings of the string. These replacement rules do not use the metacharacters described above, but `StringReplace` does accept the `IgnoreCase` option. The behavior of `StringReplace` is closer to `ReplaceAll` than `Replace` or `ReplaceRepeated` for expressions.

StringTake behaves much like **Take** does, except only on strings.

```
In[5]:= StringTake["abcdef",{2,4}]

Out[5]= bcd
```

Lists of rules can be given to **StringReplace**.

```
In[6]:= StringReplace["abcdabde",{"ab" -> "me", "cd" -> "de"}]

Out[6]= medemede
```

Of course, as always with *Mathematica* we can combine these functions to great special purpose string manipulation functions. For instance, the following will replace a sequence of characters specified by a position with a new substring.

```
StringPositionReplace[string_String, substring_String,
                pos:{start_Integer, finish_Integer}] :=
        StringInsert[StringDrop[string, pos], substring, start]
```

2 Using Strings: Input and Output

One place that string manipulation becomes a useful tool is in dealing with *Mathematica* input and output. Output formatting in particular is a applicable to presentation of information in understandable or special forms. A full explorations of this is a new topic in its own right, but we can examine some simple examples here.

For instance, consider formatting of numbers. There are various functions in *Mathematica* for modifying how numbers are display, such as `NumberForm` and its many variants. Suppose, however, that we have a special case; we wish a set of numbers input to be written out in a very precise form, with a fixed number of digits in the mantissa and exponent, with the two separated by an "e", (a standard representation of floating point numbers printed from C or Fortan.) Some programs require that input by placed very precisely (data in columns 3-12, or some similar approach.) Here is some code to generate such a format, based on the existing number formatting code and heavy use of string manipulation.

```
padmantissa[num_, length_] :=
        Module[{str = num, zeros = length - (StringLength[num] - 1)},
                If[StringTake[num,1] === "-", zeros = zeros + 1,
                        str = "+"<>str];
                str <> Table["0",{zeros}]
        ]

padexponent[exp_, length_] :=
        Module[{str = exp, sign, zeros},
                If[StringTake[str,1] === "-",
                        sign = "-"; str = StringDrop[str,1],
                        sign = "+"];
                zeros = length - StringLength[str];
                If[zeros < 0,
                        Message[FixedNumberForm::truncexp, exp];
                        str = StringTake[str, length];
                        zeros = 0
                ];
                sign <> Table["0",{zeros}] <> str
        ]

FixedNumberForm::truncexp =
"Warning: length of exponent `1` was too great for field,
truncating...";

FixedNumberForm[expr_, digits_Integer:6, expdigits_Integer:2] :=
        ScientificForm[N[expr,digits], digits,
                NumberFormat ->
                        (SequenceForm[
                                padmantissa[#1,digits],
                                "e",padexponent[#3,expdigits]]&
                        )
        ]
```

A program to format numbers as fixed length floating point.

The option `NumberFormat` to `ScientificForm` accepts a function that determines the output form of the number, based on the mantissa, base, and exponent. These are passed in as strings, so string manipulation comes to the rescue, allowing us to fix the length of the proper fields, place signs where appropriate, and join the whole mess together. One interesting point is the use of `SequenceForm`. In this particular example, `StringJoin` could have been used with equal effect. `SequenceForm` is a function that combines the output forms of various expressions, puts them together so the objects appear in sequence. It is one of a special class of functions designed for manipulating output; these include `Subscripted`, `Superscript`, and `ColumnForm`. They are designed to accept *Mathematica* expressions, and combine their output forms in the appropriate fashion.

An example of using `FixedNumberForm`.

```
In[7]:= FixedNumberForm[{234.52^-10, 12, 234.34^23, -34.23}]

Out[7]={+1.98701e-24, +1.20000e+01, +3.20972e+54, -3.42300e+01}
```

You will note another use of strings here; the error message consists of a string. Error messages present us with another variant of manipulating strings; a related function is `StringForm`. In both of these situations, a *control* `string` is given that contains a string and markers for the placement of other objects into that string. These markers take the form `` `\#n` `` or ``` `` ```, indicating the *n*th expression passed, or the next expression not yet placed in the string, respectively. This allows us to easily create new strings from standard expressions, as is done with error messages.

A simple example of using `StringForm`.

```
In[8]:= StringForm["This expression: `1`  evaluates to this
        expression: `2`",
         HoldForm[2 + 2], 2 + 2]

Out[8]= This expression: 2 + 2  evaluates to this expression: 4
```

Mathematica input is also initially treated as a string. The global symbol `$PreRead` allows us to attach a function to pre-process an input string, before it is evaluated. This allows us to, for instance, create a new operator. Suppose, for instance, we define a new function called `NonCommutativePower`. We really don't want to type this every time we perform an evaluation with it; instead, we would rather simply type `^^`. We can take advantage of *Mathematica*ś standard form for infix notation:

Substitute the infix version of `NonCommutativePower` for all instances of `^^` in the input line.

```
In[9]:= $PreRead = StringReplace[#, "^^" ->
        "~NonCommutativePower~"]&

Out[9]= StringReplace[#1, ^^ -> ~NonCommutativePower~] &
```

It works!

```
In[10]:= (a + b)^^3

Out[10]= NonCommutativePower[a + b, 3]
```

Or does it?

```
In[11]:= "This is a string containing ^^"

Out[11]= This is a string containing ~NonCommutativePower~
```

As can be seen, this definition is not perfect; furthermore, it is overriding the standard input form for numbers in a non-decimal base. But of course, more sophisticated functions can be attached to `$PreRead`, allowing us to parse out the instances of strings, and not perform substitutions within them; doing so is left as an exercise for the student...

3 Converting Strings

There are many times in using *Mathematica* where it is convenient to change between the strings and other forms of data in *Mathematica*A common occurence is to convert strings to *Mathematica* expressions, and vice versa.

`ToExpression`[*string*]	converts a string to an expression
`ToString` [*expression*]	converts an epression to a string

Conversion between strings and expressions.

Converting a string to an expression often occurs in processing data; some programs write data out in nonstandard formats that are difficult to process. *Mathematica* can read the data in as strings, the strings can be processed to something convenient, and converted to *Mathematica* input. For instance, one common spreadsheet often writes data in the form *number*`,"`*text*`",....` This can be difficult to process using the standard capabilities of `ReadList`. However, if it is read in as strings, one can very simply map a function across the list of strings to convert to a *Mathematica* form.

The form of the function described above. The argument is a sample akin to the data in question.

```
In[12]:= ToExpression[StringJoin["{", # ,
         "}"]]&["\"abc\",123,\"def\""]

Out[12]= {abc, 123, def}
```

Conversion to expressions can also be useful if you want to perform textual manipulations on symbols; for instance, defining a set of symbols with a common extension, as:

Creating new symbols from a list of symbols.

```
In[13]:= Map[ToExpression[ToString[#]<>"x"]&,
          {a,b,c}]

Out[13]= {ax, bx, cx}
```

A more sophisticated use might perform various assignments to the new symbols. Since symbols can be looked up quickly, this can be useful for efficiency in some large programs.

Another useful conversion is to *character* `codes`. For 8-bit characters, these correspond to the standard ASCII codes for the characters. This allows us to perform numeric manipulation to modify or examine textual objects. One add result of this derives from a peculiarity of *Mathematica* strings. Unlike many systems, *Mathematica* allows strings to contain characters of any character code, including the null character (character code 0). This allows us to write binary information to files, by converting the byte representations (a sequence of integers between 0 and 255) to a string, and using the function `WriteString` to place the information in a file.

Writing binary data to a file.

```
In[14]:= WriteString["binarytst",
           FromCharacterCode[{72, 101, 108, 108, 111,
              0, 84, 104, 101, 114, 101, 33, 33}]
         ]
```

Let's take a look at what we wrote...

```
In[15]:= ReadList["binarytst", String]

Out[15]= {Hello}
```

One more conversion that gives us a great deal more flexibility in processing strings is the ability to convert a string to a `Stream` object. Once this has been done, the string can be processed using functions such as `Read` and `ReadList`. This can be useful in parsing strings, for instance.

An example of tokenizing a string by converting it to a stream.

```
In[16]:= ReadList[StringToStream["a+2 = 5"], Word,
          TokenWords -> {"+", "="}]

Out[16]= {a, +, 2, =, 5}
```

4 Files

There are some functions in *Mathematica* that are used specifically for processing files, which are distantly related to the string manipulation functions. Of course, many other functions can be used for processing files, taking the lessons of the above possibilities and combining them with the various functions for reading data.

`Find[`*stream*`, "`*text*`"]`	returns a string containing the first instance of *text* in the file
`FindList[`*stream*`, "`*text*`"]`	returns a list of strings containing *text* in the file

The `Find` and `FindList` functions.

The `Find` and `FindList` command work similarly to the Unix "fgrep" command, finding substrings in a file. They work by breaking the file up into records (as specified by the `RecordSeparators` option; the default simply breaks the file into lines) and returns those that contain the given string (or at least one of a list of strings). Another option, `AnchoredSearch`, can be set to `True` to force the match to be at the beginning of the record. As with many of the string functions, the option `IgnoreCase` can be used to match sustrings with a different case, but the use of metacharacters is not supported.

Using `FindList` to pull text out of a file.

```
In[17]:= FindList["Graphics/SurfaceOfRevolution.m",
              ":Mathematica Version:"]

Out[17]= {(* :Mathematica Version: 2.0 *)}
```

The above is a trivial example; the possibilties are limited only by your imagination.

5 Conclusions

As we have seen, strings play a vital part in *Mathematica*. They range from representing input and output, to textual data. Because of this, the importance of string manipulation functions becomes evident.

We hae examined various functions to discover the characteristics of strings, and functions to directly modify those strings. These have been applied to various useful capabilities, such as changing the behavior of input and output.

Further, functions to convert strings to other *Mathematica* types have been examined; although we only glanced at these, many of the possibilities for further manipulation are obvious.

When faced with otherwise intractable problems of handling data and text, always remember to consider the string manipulation functions!