



String Manipulation in Dynamic Text

Zephyr Associates, Inc.
P.O. Box 12368
Zephyr Cove, NV 89448

775-588-0654
800-789-5323
Fax 775-588-8423

www.styleadvisor.com

String Concatenation

You may have encountered the term string concatenation in your exploration of Dynamic Text. If you haven't, you will. You'll find that it happens all the time.

So, what is string concatenation? It's the process of combining two or more strings into one. In English, you do this by putting them next to each other and it just happens naturally. In DT a string is a separate object and two strings next to each other are still two separate objects unless you concatenate them.

The string concatenation operator in DT is two periods in a row (`..`). If I have a string variable named "String1" and another named "String2" and I want to combine them into one variable called "Combo" it'd look like this:

```
Combo = String1 .. String2
```

Now the variable Combo contains the contents of both String1 and String2. In case that's not clear, let's look at an entire block of code and its output:

```
<%  
String1 = "abc"  
String2 = "def"  
  
Combo = String1 .. String2  
  
Display(Combo)  
%>
```

This will output "abcdef". That's exactly the same thing that two Display() functions in a row (Display(String1) Display(String2)) would do. If there's already a way to do it, why bother? Good question. I think I have a good answer.

There are DT functions that require a single string as an argument. If I want them to regard two strings as one, I can't just stick them next to each other. I have to combine them into a single object and use that as an argument.

For example, `Display()` takes a single string as an argument. Suppose I want to display a number variable and a description of that variable. To get a single `Display()` function to show both, I have to concatenate them into one argument like this:

```
<%  
ret = Value(1, "Return", "Scan #1|Page 2")  
Display("My manager's return was " .. ret)  
%>
```

This will output something like, “My manager’s return was 10.34”. I just noticed that this isn’t a complete sentence. It lacks a period. Well, I can just concatenate one onto the end like this:

```
Display("My manager's return was " .. ret .. ".")
```

This is how string concatenation leads to code that’s difficult to read. You almost have to count quote marks to make sense of that string. If you want to display a paragraph that mentions several variable values, you can end up with really complicated looking strings, especially since the periods at the ends of sentences look like half of a concatenation operator. If it’s not in quotes it’s either a variable or a concatenation operator.

Please note as well that this example block doesn’t check to see if `ret` has a value. Do as I say, not as I do. I’m trying to write uncluttered examples, but *you* should *always* check to see if the `Value()` function returned a value before trying to do anything with it.

We could show the same result by using three `Display()` functions without inserting any line feeds like this:

```
Display("My manager's return was ")  
Display(ret)  
Display(".")
```

Personally, I find this just as confusing as trying to sort out one concatenated argument.

Note that it is often possible to use + to concatenate strings, but since this is our addition operator and strings that look like numbers will be added rather than concatenated by + it’s best to use + only for addition and .. only for concatenation.

Find and Replace

Suppose you snag a string out of a qualitative field in a Scan and you don't like the way it's worded. It mentions "S&P 500" and you really need it to say "S&P 500 Index". DT has a function which will replace all instances of a string with another version. We call it `Replace()`.

`Replace()` takes three arguments. First is the string you want to operate on (I'm going to call that the source string as we go along). Second is the substring you want to change (let's call it the search string for future reference). Third, as you must have seen coming, is the replacement string. `Replace()` will return a string in which it has replaced all instances of the second argument with the third one. If there are no instances of the search string then `Replace()` will return the original string unchanged, so it doesn't hurt to run `Replace()` on a string when there's no need.

For example:

```
<%  
strat = Value(1, "Strategy", "Scan #1|Page 2")  
Replace(strat, "S&P 500", "S&P 500 Index")  
%>
```

This block of code gets the Strategy field for the first manager in a Scan Table and replaces every instance of "S&P 500" with "S&P 500 Index" and displays it. If you were going to do further manipulation to the new and improved strategy, you could assign it back into the *strat* variable like this:

```
strat = Replace(strat, "S&P 500", "S&P 500 Index")
```

This would allow you to do another `Replace()` on another substring if you needed to...say to replace "ain't" with "is not".

What if you don't need an entire string...you just want a part of it? That's where the `Substring()` function comes in. `Substring` takes three arguments:

```
Substring(source string, start character, end character)
```

The first is the string you want to look at. The second and third are numbers describing the position in the main string of the first and last character that you're trying to get. `Substring()` returns a string consisting of the start through the end characters in the source string. This is easier to see than to describe, so let's go:

```
<%  
MyString = "This string has the word baloney in it. "  
ShorterString = Substring(MyString, 25, 32)  
Display(ShorterString)  
%>
```

This block of code will display the word "baloney" by looking at the string in the variable *MyString* and returning the 25th through the 32nd characters. Just as a reminder, the numeric arguments to `Substring()` are called indexes, and the process of pulling things out of a longer thing by specifying their position is called indexing. If you leave off the second numeric argument (the ending index), `Substring` will start at the starting index and go to the end of the string. If the above block said `Substring(MyString, 25)` you'd get back "baloney in it."

How did I know that the word "baloney" was the 25th through the 32nd characters in that sentence? Well, I counted them. If only there were a way to find a specific substring and get starting and ending indexes that `Substring()` could use. Well, there is. It's a function called `Find()`.

`Find()` takes two arguments.

```
Find(source string, search string)
```

First is the source string you want to search. Second is the substring you want to search for. `Find()` returns two values. The starting index and the ending index of the search string if it finds it in the source string. Because `Find()` returns two different numbers, we're going to have to learn a new syntax for assigning values to variables:

```
StartIndex, EndIndex = Find(MyString, "baloney")
```

What this will do is assign the first value returned by `Find()` to the variable *StartIndex*, and the second value returned by `Find()` to the variable *EndIndex* all in one line. Pretty slick, huh? You can do this any time you're assigning variables, but it may make your code less legible if you do it too much:

```
one, two = 1, 2
```

This does exactly the same thing as

```
one = 1  
two = 2
```

I'll leave it up to your own conscience to decide if this improves your code or not. Anyway, this is the only way to get the second index from the `Find()` function. Let's improve our example code from above:

```
<%  
MyString = "This string has the word baloney in it. "  
FirstChar, LastChar = Find(MyString, "baloney")  
  
ShorterString = Substring(MyString, FirstChar, LastChar)  
  
Display(ShorterString)  
%>
```

This returns exactly the same thing as before, but this time you didn't have to count characters yourself. The other advantage of this is that the `Substring()` line will work even if you rearrange the sentence in *MyString*. You don't need to know where "baloney" starts and ends. You just tell DT to find it and then use the starting and ending indexes to fish out the baloney you're interested in. So if the sentence was "My baloney has a first name" the variable *ShorterString* would still get the word "baloney".

Now if you've been wondering how this is useful, because the example would work as well if it was only one line long and just said "Display("baloney")", well there are a couple of things it's good for.

First, you can use Find() to determine if the search string exists in the source string. If the first value returned by Find() is Nil, then the Find() function didn't find the search string in the source string. You can use this fact to test to see if a substring exists in a source string like this:

```
<%  
IsIndex = Find(ManagerName(1), "Index")  
if IsIndex then  
  Display("This manager thinks it's an Index")  
end  
%>
```

This block of code checks to see if the word "Index" appears anywhere in the name of the first manager in the workbook. Since we're not going to do anything with the end index we needn't even bother to assign the ending index to a variable. The variable *IsIndex* will either be Nil because "Index" doesn't appear in the manager's name, in which case the test will fail, or it will have a number representing the position of the first character in the word Index in the manager's name. If "Index" is found anywhere in the manager's name the test passes and we display the message "This manager thinks it's an Index".

Another thing you can do with Find() and Substring() is discard extraneous parts of strings. Suppose you need the number of managers from the header of a Manager versus Universe Table for a particular time period. Well, the header contains the time period and a blank line and the string " mng" around the number. You can use Find() to find out where all the extra junk appears and use Substring() to grab just the number.

```
<%  
Header = StatisticAtPosition(3, "Manager vs Universe Table #1|Page 2")  
Blankstart, Blankend = Find(Header, "|")  
Mngstart = Find(Header, " mng")  
  
NumberOfManagers = Substring(Header, (Blankend + 1), (Mngstart - 1))  
  
Display(NumberOfManagers)  
%>
```

Let's step through this example line by line to hammer home exactly what's being done. First we assign the contents of the third statistic cell (skipping the Median Rank and Volatility columns) in the first MVU table on page 2 to a variable called *Header*. We know by looking at the table that this cell looks like this:

YTD
408 mng

Next we find the starting and ending indexes of the string "|" (the blank line) in the header and assign them to variables called *Blankstart* and *Blankend*. Remember that the vertical bars represent line feeds and it takes two in a row to make a blank line. We only care about the ending index of this substring, but the only way to assign the ending index to a variable is to assign the beginning index as well. We just won't use the variable *Blankstart*. I've been known to name throwaway variables like this "NotUsed" just to be explicit about my intentions. Next we do the same thing for the starting index for the string " mng" and assign it to *Mngstart*. Note that the search string includes the blank space between the number and "mng". Since we can get the starting index by itself and that's all we're interested in, we don't even bother assigning the ending index to a variable. Now that we know where the stuff we want to discard starts and ends, we can figure out where the number we care about starts and ends. The next line uses *Substring()* to grab the part of the header that starts one character after the blank line and ends one character before the ending string and assigns it to a variable called *NumberOfManagers*. The last line displays *NumberOfManagers* just to prove that we did it right.

What if my source string contains more than one instance of the search string? Which one does Find() find? The short answer is the first one from the left. The longer answer is that there is an optional third argument for Find() that allows you to specify the position in the source string at which you want to start looking for your search string. It's like saying, "Look for the substring "index" but ignore the first n characters".

Here's a completely artificial string to illustrate this option.

```
<%  
MyString = "aa slkfsf bb skldfjsj aa skfjskf aa"  
Find(MyString, "aa", 5)  
%>
```

This returns the number 23 since that's the first instance of the search string "aa" in *MyString* after the fifth character. So what good is this? For one thing, it'll let you count how many times the search string appears in the source string. Do the first Find() with the third argument set to 1, then do another Find() with the third argument set to the ending index of the search string plus 1, and repeat until you don't find one. Keep track of all the successful finds as we did earlier in this doc and you've got a count of how many times the search string appears in the source string. An example of this advanced string manipulation can be found in Exhibit 1 at the end of this document.

Looking for punctuation in a string may not return the results you expect because many punctuation characters have specialized meaning within Dynamic Text code. Here's a list of the so-called "magic characters" that mean things in Dynamic Text search strings:

```
^$().[]*+~?
```

If you want to be sure that Find() will find any of these characters stick a percent sign (%) in front of it in your search string. To find a period, for example, try this:

```
Find("This contains a period. Or two. ", "%.")
```

Other String Functions

We've provided a few other string manipulation functions that may come in handy.

Length()

The first one I'm going to cover isn't technically a string function because it works on other types of objects as well, but it's pretty useful for strings. `Length()` takes one argument. If that argument is a string, `Length()` returns the number of characters in the string. For example:

```
<%  
  
MyString = "One"  
Length(MyString)  
  
%>
```

This example assigns the string "One" to a variable called "MyString" then uses `Length()` to count characters in "One". It will return 3. `Length` will count space characters and punctuation, so `Length("One, Two.")` will return 9, not 6.

ToUpper() and ToLower()

These two functions are surprisingly useful when you're trying to find a string and you don't know if it will be capitalized or not. `ToUpper()` takes a string as an argument and returns the same string with all letters shifted to upper case. `ToLower()` does the opposite. Neither has any affect on spaces or punctuation or numbers. For example:

```
<%  
  
MyString = "My baloney has a first name, it's O S C A R"  
  
ToUpper(MyString)  
Display("|")  
ToLower(MyString)  
  
%>
```

This example assigns a long string to the variable "MyString" then converts all the letters to upper case, then goes to the next line (that's what "`Display("|`") does), then converts all the letters in the same string to lower case, resulting in something like this:

```
MY BALONEY HAS A FIRST NAME, IT'S O S C A R  
my baloney has a first name, it's o s c a r
```

This is handy when doing something like getting a database ID from a Scan and not knowing if it will be "MSE" or "mse" or even "Mse". If you force the ID to either upper or lower case before checking to see if it's the right letters, you can safely ignore the case that you got from the scan. This is what's called making your code case-insensitive.

WrapText()

This function takes two arguments, a string and a number of lines. What it returns is the same string with line breaks inserted so that it appears on more than one line...up to the number of lines specified. The number is the maximum number of lines you want your text to appear on. If it can make a nicely justified presentation with fewer lines it might. It will only make breaks between words. For example:

```
<%  
MyString = "This is four words"  
WrapText(MyString, 4)  
%>
```

Will not put one word on each line as you might expect. You'll probably get:

```
This  
is four  
words
```

If the words were all of similar length it might be on four lines, but the number is only a recommendation. We told DT, "break this string into several lines, but don't go over four". You can use this to try to fit long manager names into a cell in a Notes table by deciding the longest name that will fit on one line, then measuring the manager's name with the Length() function and using WrapText() if it's too long to fit. This will, of course, fail if the manager's name doesn't have many blanks in it but that's probably rare.

```
<%  
Man = ManagerName(1)  
if Length(Man) > 25 then  
  WrapText(Man, 2)  
else  
  Display(Man)  
end  
%>
```

This block assigns the first manager's name to the variable "Man" then checks the length of the name. If it's longer than 25 characters it tries to use WrapText() to break it into two lines, otherwise it just displays it.

Combine()

This function is designed to format lists. It takes a variable number of arguments. First you specify the separator you want to use between the items in your list (commas, hyphens, some word, whatever) then you provide a list of items to display. If you're trying to display the results of a bunch of analysis parameter functions in a comma delimited list, this will save you from having to do a complicated bunch of concatenations. For example:

```
<%  
Combine(", ", ManagerName(1), ManagerName(2), BenchmarkName())  
%>
```

Will display a list containing the first two managers in the workbook and the Market Benchmark separated by a comma followed by a space.

ManagerA, ManagerB, S&P 500 Index

Since the arguments to Combine() are separated by commas when calling the function, the syntax may not be completely clear, so I'll change the separator argument to something different:

```
<%  
Combine(" and ", ManagerName(1), ManagerName(2), BenchmarkName())  
%>
```

This will display a list that looks like this:

ManagerA and ManagerB and S&P 500 Index

To display the same list using Display() you'd have to do some serious concatenation:

```
<%  
Display(ManagerName(1) .. " and " .. ManagerName(2) .. " and " ..  
        BenchmarkName())  
%>
```

The Escape Character (\)

By now you're probably ready to escape from this tutorial, but we should go over one more concept...escaping characters to prevent DT from evaluating them. Suppose you have a string surrounded by single-quotes and somewhere in the middle you decide to use an apostrophe. This will be a problem because we use the same character for apostrophe and for single-quote so the apostrophe will look to DT as if you'd decided to end the string at that point. For example:

```
<%  
Display('This ain't right')  
%>
```

will result in an error that says “{Error: Unexpected "<%" found.} ” because the apostrophe is being evaluated as a single-quote and will close the string. Then the one at the end of the string (well, what a normal person would recognize as the end) looks like the beginning of another string. Since there isn't a matching closing single-quote for that one, the end marker looks like part of the string. In this case it looks like you started a block of code and never finished it, so DT complains that it doesn't know what to do with that begin marker at the top.

There are two ways to deal with this type of problem. Actually three, since you could avoid using contractions the way you should in formal documentation, but you've probably noticed that writing like a grown-up isn't a choice I'll make willingly.

First, you can change the single-quotes at the beginning and end of the string to double-quotes like this:

```
<%  
Display("This ain't right")  
%>
```

This works well for me. I use so many contractions that I've gotten in the habit of using double-quotes around my strings so that I can use apostrophes without having to do anything special. If you habitually use single-quotes around strings, however, you can display apostrophes when you need them by escaping them. That is, by preceding the apostrophe with a backslash like this:

```
<%  
Display('This ain\'t right')  
%>
```

The backslash tells DT to display the very next character instead of evaluating it. I end up needing the escape character when I want a string to contain a quotation because I use double-quotes for strings and grammar dictates double-quotes for quotations as well. For example:

```
<%Display("Mark Twain said, \"Eschew surplussage\".")%>
```

Exhibit 1

Script to count the number of times a search string appears in a longer string:

```
<%  
MyString = "aa slkfsf bb skldfsjsj aa skfjskf aa asdfsfsf"  
Offset = 1 --Initialize Offset to 1 (used by Find() to skip past  
           --the previously found match)  
Found = 0 --Initialize a variable to count how many matches were found  
repeat    --Loop through MyString looking for matches and counting them  
  StartChar, EndChar = Find(MyString, "aa", Offset)  
  if StartChar then    --Only true if search string is found  
    Found = Found + 1  --Add 1 to variable to count matches  
    Offset = EndChar + 1 --Move Offset to past the first match  
  end  
until (not StartChar) --Until you don't find a match  
Display(Found)  
%>
```

In this example, the output will be 3.