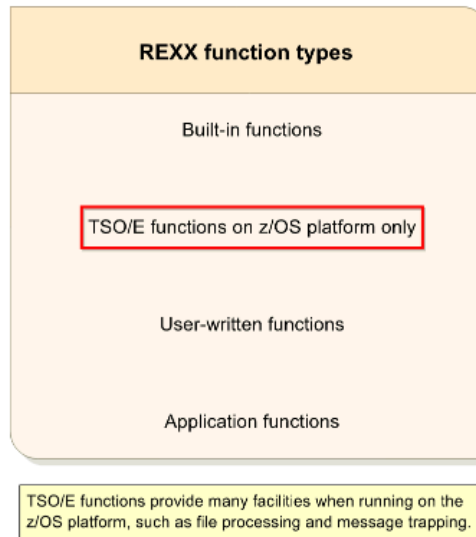




String Functions

By proceeding with this courseware you agree with [these terms and conditions](#). Interskill Learning Pty. Ltd. © 2021





While REXX programs using just keyword instructions can be written to do most things, many complicated and commonly used facilities have to be coded in special "functions" that can be included in any standard REXX clause or expression.

Functions perform specific actions or calculations, and return a result.

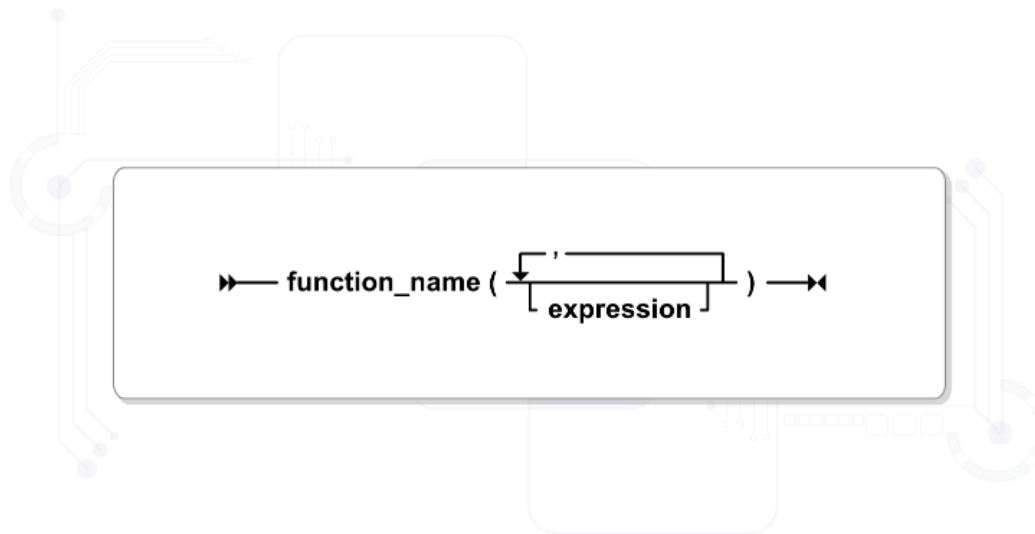
Listed above are the four types of functions. **Mouse-over** each function for a brief description. You will now focus on the built-in functions that are available with the REXX interpreter.



String functions	Perform various comparison, interrogation, and manipulation actions on data strings
Text and word functions	Interrogate and manipulate words and specific data within a string
Justification functions	Justify and format text and data strings
Numeric functions	Interrogate and format numeric values
Character conversion functions	Convert and manipulate binary, hex, and character values
Environment functions	Interrogate the environment that the REXX program is running under, and return settings and definitions
Stream I/O functions	Used for file processing on many platforms

More than 50 built-in functions are available in REXX, depending on the platform and the version of REXX that is running.

For the purposes of this course, we have divided the built-in functions into the groups listed above.



The REXX function call is the same basic format for all functions, regardless of the action they perform or the result they return.

The number and type of expressions depends on the function that is being called. For example, a numeric function may expect a number and will cause an error if something other than a valid number is passed as an expression. Too many or too few expressions can also cause errors.

REXX requires parentheses to be coded, even if there are no expressions. In fact, REXX will not recognize a function as such unless a left bracket immediately follows its name. Do not code a space between the function name and the left bracket.

Function expression examples

```
lastname = SUBSTR('FRED FROG',6,4)
```

```
Name = "FRED FROG"  
lastname = SUBSTR(name,6,4)
```

```
Name = "FRED FROG"  
Len = LENGTH(name)  
lastname = SUBSTR(name, len-3,4)
```

```
Name = "FRED FROG"  
lastname = SUBSTR(name, LENGTH(name)-3,4)
```

```
Name = "FRED FROG"  
lastname = SUBSTR(name, POS(' ',name)+1, LENGTH(name)-POS(' ',name))
```

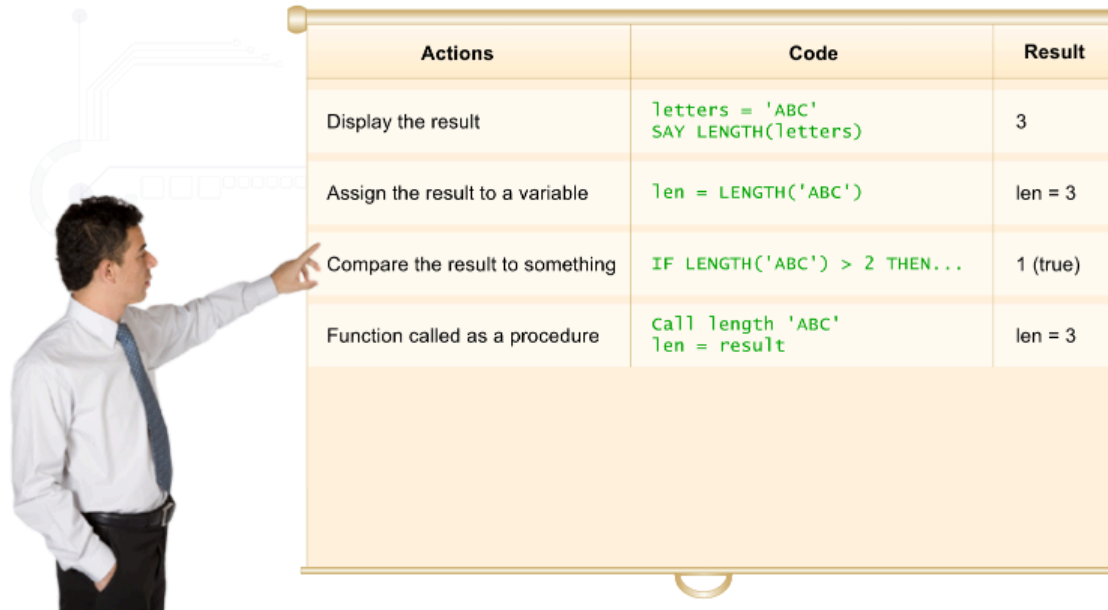
Note: Be careful when using functions within functions. Although the code is more efficient, it is usually more difficult to understand and debug.

The expression or expressions coded between the parentheses of a function are passed as arguments to the function after being interpreted and evaluated.

The expression passed to a function can be any valid REXX expression consisting of variables, literals, arithmetic evaluations, or even other functions. Obviously, the resulting string must be a value that is expected or required by the function, or an error could occur.

A powerful feature of REXX is that multiple functions can be combined or nested to produce a result.

The above examples of expressions used in a function would yield the same result.



Actions	Code	Result
Display the result	<pre>letters = 'ABC' SAY LENGTH(letters)</pre>	3
Assign the result to a variable	<pre>len = LENGTH('ABC')</pre>	len = 3
Compare the result to something	<pre>IF LENGTH('ABC') > 2 THEN...</pre>	1 (true)
Function called as a procedure	<pre>Call length 'ABC' len = result</pre>	len = 3

REXX functions that execute successfully will always return a result that the interpreter will replace the function with; it is up to the programmer to decide what to do with the result.

Functions can also be called like any procedure, but they rarely are because this requires more lines of code.

Shown here are some examples of functions in use.



- ARG
- COMPARE
- COPIES
- DATATYPE
- DELSTR
- INDEX
- INSERT
- LASTPOS
- LENGTH
- OVERLAY
- POS**
- REVERSE
- SUBSTR
- SYMBOL
- VERIFY
- XRANGE

Returns the character position of a substring in a given string.

String functions interrogate, compare, and manipulate character strings of data. Listed above are the standard built-in functions that fall into this loosely defined category.

Mouse-over each function for a brief description.





» — **COPIES**(string,n) — «

Examples:

```
COPIES('abc',2)      /* 'abcabc' */  
COPIES('*',20)      /* '********************' */
```

The COPIES function returns **n** copies of **string** concatenated together.





Diagram of the DELSTR function signature: `DELSTR (string, n, length)`. The parameter `length` is enclosed in a box with arrows pointing to the right, indicating it is optional.

Examples:

```
DELSTR('Datatrain',5)           /* 'Data' */  
DELSTR('Datatrain',5,1)        /* 'Datarain' */  
DELSTR('They are white sheep',4,5) /* 'The white sheep' */
```

The DELSTR function returns `length` characters deleted from `string`, starting at `n`.

If `length` is not specified, all characters from `n` are deleted.

INSERT(new,target , n , length , pad)

Examples:

```
INSERT('new','old') /* = 'newold' */  
INSERT('new','old',2) /* = 'oldnewd' */  
INSERT('new','old',2,5,'*') /* = 'oldnew**d' */  
INSERT('new','old',7,5,'*') /* = 'old****new**' */
```

The INSERT function will insert the characters **new** into **target**. All other parameters are optional; **n** specifies the location in **target** after which to insert **new**. The default for **n** is 0, which means to insert before the beginning of **target**.

If **n** is greater than the **length** of **target**, padding occurs using the character specified by **pad**. The default **pad** character is a blank space.

The default for **length** is the length of the new string, but if it is specified, a length less than the length of **new** will cause **new** to be truncated. A **length** greater than **new** will cause **new** to be padded with the **pad** character.



► **OVERLAY**(new,target , [n] , [length] , [pad]) ◄

Examples:

```
OVERLAY('when','whatever') /* = 'whenever' */
OVERLAY('when','whatever',5) /* = 'whatwhen' */
OVERLAY('when','whatever',,6) /* = 'when er' */
OVERLAY('when','whatever',10) /* = 'whatever when' */
OVERLAY('when','whatever',10,6'*) /* = 'whatever*when*' */
```

The OVERLAY function will replace characters in **target** with characters from **new**; that is, it will overlay **target** with **new**, starting from position **n** in target.

If **n** is omitted, overlaying begins at position 1 in **target**. If **length** is specified, it is used to pad or truncate **new**.

If padding is required, the **pad** character specified by pad will be used. Otherwise, the default pad character of blank will be used.





► **REVERSE** (string) ◄

Examples:

```
REVERSE('abcde') /* 'edcba' */  
REVERSE('ward') /* 'draw' */  
REVERSE('toot') /* 'toot' */
```

The REVERSE function returns a character string swapped end-over-end; that is, the first character of `string` becomes the last and vice-versa.



The diagram shows the function signature `SUBSTR (string, n, length, pad)` with arrows indicating the start and end of the function. Brackets below the parameters identify `length` and `pad`.

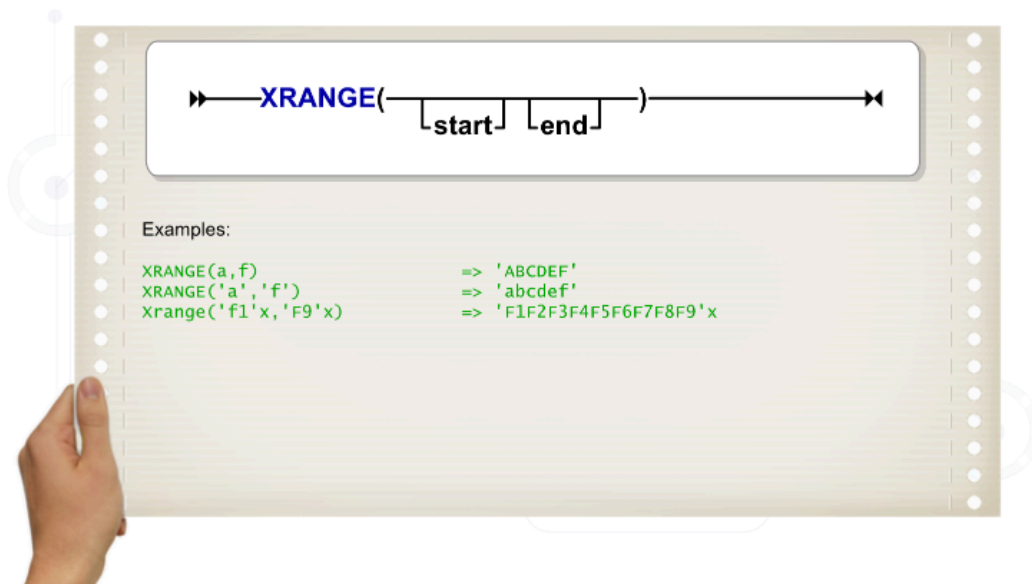
Examples:

```
SUBSTR('abcdefghij',7)          /* = 'ghij' */  
SUBSTR('abcdefghij',7,2)       /* = 'gh' */  
SUBSTR('abcdefghij',7,7,'%')   /* = 'ghij%%%' */
```

The SUBSTR function returns a substring of `string`, starting at character `n`, of `length` characters.

If `length` is not specified, the remainder of `string` is returned. If `length` is specified, it is used to truncate or pad the remainder of `string`.

The default `pad` character is a blank space.



The `XRANGE` function returns a string of all one-byte codes between and including a specified start and end character in a given string. This can be useful when, for example, the entire alphabet must be assigned to a variable.

Care should be taken as `XRANGE` looks at the hex value of the start and end characters and would include every hex value within the range. These characters can vary depending on whether the platform is using ASCII or EBCDIC.

Diagram illustrating the ARG function syntax: `ARG(n [,option])`. The parameter `n` is labeled as the 1st ARG, and the optional parameter `[,option]` is labeled as the 2nd ARG and 3rd ARG.

Examples:

```

/* following "Call name 'x1','y1'" */
ARG()      -> 3
ARG(1)     -> 'x1'
ARG(2)     -> ''
ARG(3)     -> 'y1'
ARG(n)     -> '' /* for n>=4 */
ARG(1,'e') -> 1 /* true - Arg 1 Exists */
ARG(2,'E') -> 0 /* false - Arg 2 is null*/
ARG(2,'o') -> 1 /* true - Arg 2 Omitted */
ARG(3,'o') -> 0 /* false - Arg 3 exists */
ARG(4,'o') -> 1 /* true - Arg 4 Omitted */

```

When a REXX routine is executed, parameters or arguments can be passed to the program or procedure and normally accessed by using the ARG keyword instruction. However, a comma in the parameter list is interpreted differently, depending on the command or instruction used. When using the CALL keyword instruction to execute a procedure, a comma is considered to be an argument delimiter.

The ARG function can be used to determine how many arguments have been passed (`ARG()`), the value of an argument (`ARG(n)`), and whether an argument exists (`ARG(5, "E")`) or is omitted (`ARG(3, 'o')`).

When a REXX is executed by an explicit or implicit EXEC command for TSO/E, only one argument is passed across and commas are considered to be literal characters in the parameter string.



▶ **COMPARE**(string1,string2 ,pad) ▶

Examples:

```
COMPARE('abc','abc')      /* returns 0 - 'abc' = 'abc' */
COMPARE('ab ','ab')       /* returns 0 - 'ab ' = 'ab ' */
COMPARE('ab ','ab','-')   /* returns 3 - 'ab ' /= 'ab-' */
COMPARE('ab','abc')       /* returns 3 - 'ab ' /= 'abc' */
```

The COMPARE function compares two strings and returns 0 if they match. If the strings are not equal, COMPARE returns the position of the first character that does not match.

COMPARE pads the shorter string with the specified character, which is a blank space by default.



» **DATATYPE**(string ,type) «

The **DATATYPE** function is very useful for verifying program input data and can be used to prevent unexpected errors.

Examples:

```
PULL input
IF DATATYPE(input) = 'NUM' THEN answer = 10*input
ELSE SAY 'Not a valid number'
```

```
DATATYPE(99)      /* returns NUM */
DATATYPE('99A')  /* returns CHAR */
```

The **DATATYPE** function is useful for verifying program input data and preventing unexpected errors. **DATATYPE** compares a string to the REXX definition of a string type. If only **string** is specified, **DATATYPE** returns **NUM** if the string is a valid number; otherwise, it returns **CHAR**.



Diagram illustrating the DATATYPE function syntax: `DATATYPE(string, type)`. The `string` parameter is enclosed in a box, and the `type` parameter is also enclosed in a box. A hand is shown pointing to the table below.

Type	Meaning	Description
A	Alphanumeric	Returns 1 if string contains only characters from the ranges a-z, A-Z, and 0-9.
B	Binary	Returns 1 if string contains only the characters 0 or 1, or both.
L	Lowercase	Returns 1 if string contains only characters from the range a-z.
M	Mixed	Returns 1 if string contains only characters from the ranges a-z and A-Z.
N	Number	Returns 1 if string is a valid REXX number.
S	Symbol	Returns 1 if string contains characters that are valid REXX symbols.
U	Uppercase	Returns 1 if string contains only characters from the range A-Z.
W	Whole	Returns 1 if string is a REXX whole number.
X	HeXadecimal	Returns 1 if string contains only characters from the ranges a-f, A-F, 0-9, and blank when blanks appear only between pairs of hexadecimal characters..

If `string` and `type` are both specified, DATATYPE compares the contents of `string` to the `type` and returns 1 (true) if they match; otherwise, 0 (false) is returned.

Listed above are the types supported by the DATATYPE function.



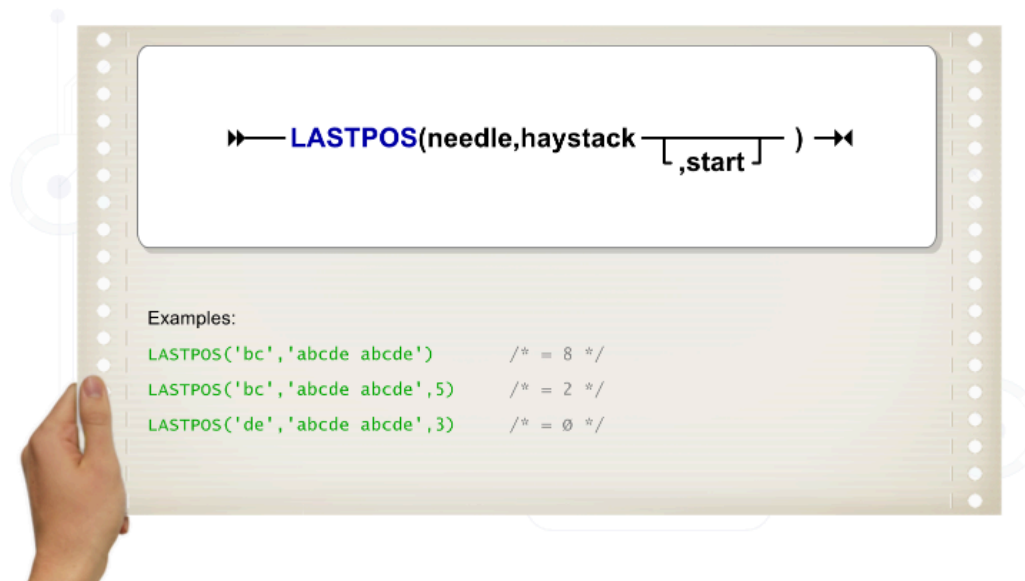
DATATYPE(string ,type)

Type	Meaning		Description
A	Alphanumeric	DATATYPE('99B','ALPHA')	/* returns 1 */
B	Binary	DATATYPE('01010111','Bin.')	/* returns 1 */
L	Lowercase	DATATYPE('abc','lc')	/* returns 1 */
M	Mixed	DATATYPE('Datatrain','M')	/* returns 1 */
N	Number	DATATYPE('99.999','N')	/* returns 1 */
S	Symbol	DATATYPE('99B','sym')	/* returns 1 */
U	Uppercase	DATATYPE('abc','UC')	/* returns 0 */
W	Whole	DATATYPE('99','W')	/* returns 1 */
X	HeXadecimal	DATATYPE('A9 B3','X')	/* returns 1 */

The type, if used, is the only character that is required. Any characters that follow it are ignored, but specifying the meaning helps to document your program.

Hexadecimal must start with X and type is not case-sensitive.

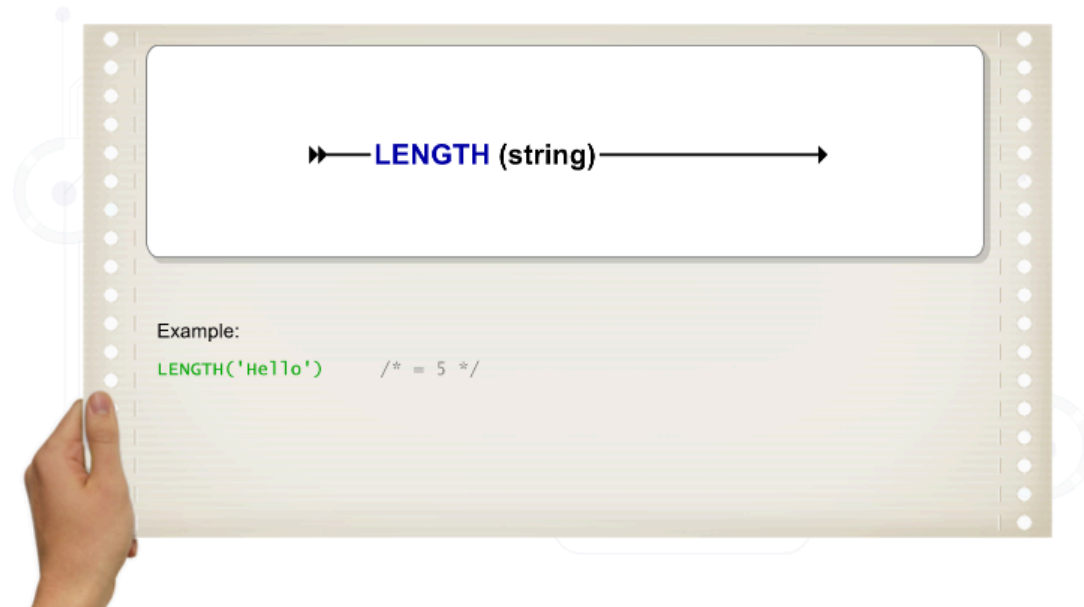
Listed above are some examples of the DATATYPE function and its returned values.



The LASTPOS function returns the starting position of the last occurrence of **needle** in **haystack**. If **needle** is not found in **haystack**, 0 is returned.

If **start** is not specified, the search begins at the end of **haystack** and continues left toward the beginning of **haystack**. If **start** is specified, searching commences at **start** and continues left toward the beginning of **haystack**.

The value returned will always be 0 when **needle** is not found or the number of the characters counting from the first character in the string to the first character of the **needle**.



The `LENGTH` function returns the length of the string passed to it.

► `POS(needle, haystack [start])` ►

Examples:

```
POS('bc', 'abcdefg')      /* = 2 */
POS('bc', 'abcdefg', 4)   /* = 0 */
POS('z', 'abcdefg')      /* = 0 */
```

The `POS(needle, haystack)` function is the same as `INDEX(haystack, needle)` function.

The `POS` function returns the starting position of `needle` in `haystack`. If `needle` cannot be located in `haystack`, `POS` returns 0.

If `start` is not specified, the search commences at the start of `haystack` and continues right towards the end of `haystack`. If `start` is specified, searching commences at `start` and continues right towards the end of `haystack`.

The `POS(needle, haystack)` function is the same as the `INDEX(haystack, needle)` function.



SYMBOL(value)

Where value is a variable or literal character string.

Examples:

```
Drop X.2
Y=2
SYMBOL('y')      -> 'VAR'
SYMBOL(y)        -> 'LIT' /* has tested "2" */
SYMBOL('x.y')    -> 'LIT' /* has tested x.2 */
SYMBOL(2)        -> 'LIT' /* a constant symbol */
SYMBOL('*')      -> 'BAD' /* not a valid symbol */
```

The SYMBOL function interrogates the REXX variable pool to determine whether a variable has been set to a value.

It returns 'BAD' if the specified string is not a valid REXX symbol, 'VAR' if the string is the name of a used variable, or 'LIT' otherwise.

The diagram shows the syntax for the VERIFY function: `VERIFY(target, reference [, option [, start]])`. The parameters are: `target` (the string to be verified), `reference` (the set of allowed characters), `option` (optional, can be 'MATCH'), and `start` (optional, the starting position for the match).

Examples:

```
VERIFY('this is Datatrain','dathns')=> 3
VERIFY(16925.71,0123456789) => 6
VERIFY('XYZ3TU5','1234567890','M',5) -> 7
VERIFY('XYZ3TU5','1234567890','N',4) -> 5
```

The VERIFY function verifies that a specified string only contains characters from a specified reference string by returning the position of the first character that is not in the reference, or 0 if the string is composed only of characters in the reference.

Alternatively, the function can determine the first character of the string that is in the reference by using the MATCH option. A start position can also be defined.

Mouse-over the syntax for a brief description of its parameters.