



interskill
learning

Numeric, Justification, and Conversion Functions

By proceeding with this courseware you agree with [these terms and conditions](#). Interskill Learning Pty. Ltd. © 2019



Objectives

Numeric, Justification, and Conversion Functions

In this module, you will explore the numeric, justification, and conversion built-in functions in the REXX language. These functions help analyze and format numeric values and data strings.

After completing this module, you will be able to:

- Identify Numeric Functions
- Identify Justification Functions
- Identify Conversion Functions





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.

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

You will now focus on the justification functions and numeric functions.

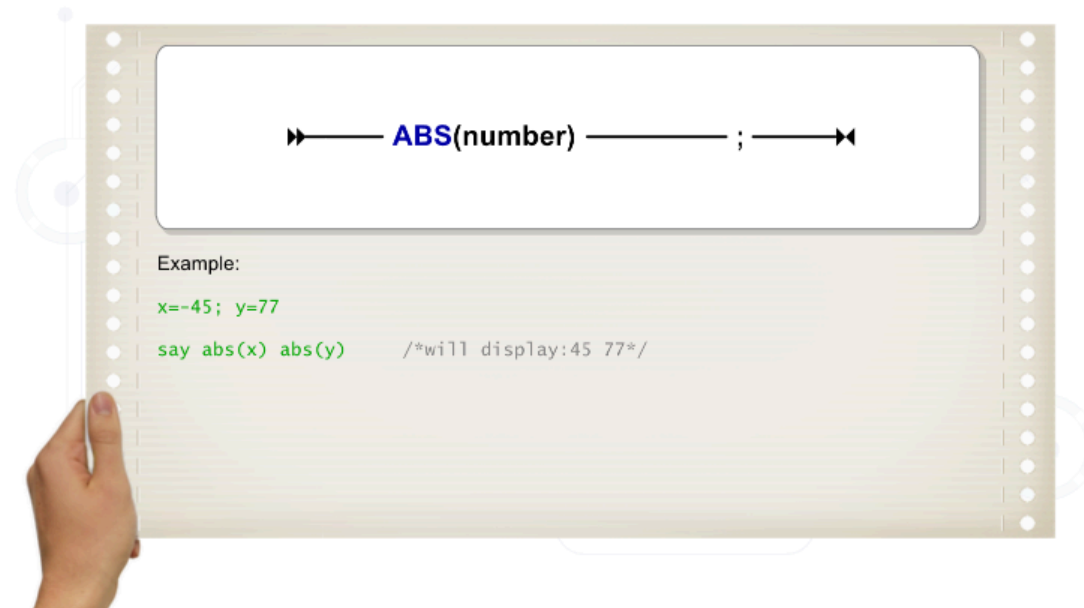




ABS	Returns the absolute value of an entered number.
FORMAT	Rounds and formats a given number according to specified lengths of integer part and decimal part.
MAX	Returns the highest value number in a specified list.
MIN	Returns the smallest value number in a specified list.
RANDOM	Returns a random number within a specified range. When a "seed" is specified, the random numbers generated will be the same each time the routine is executed. After specifying a seed, it will be "remembered" for all further executions of the RANDOM function, and should not be coded again.
SIGN	Returns an indicator of the sign of a given number; "1" for positive, "-1" for negative, and "0" for 0.
TRUNC	Truncates a numeric value to a specified number of decimal places.

Numeric functions are used to interrogate, manipulate, and present numeric values.

Listed above are the numeric functions that you will examine.



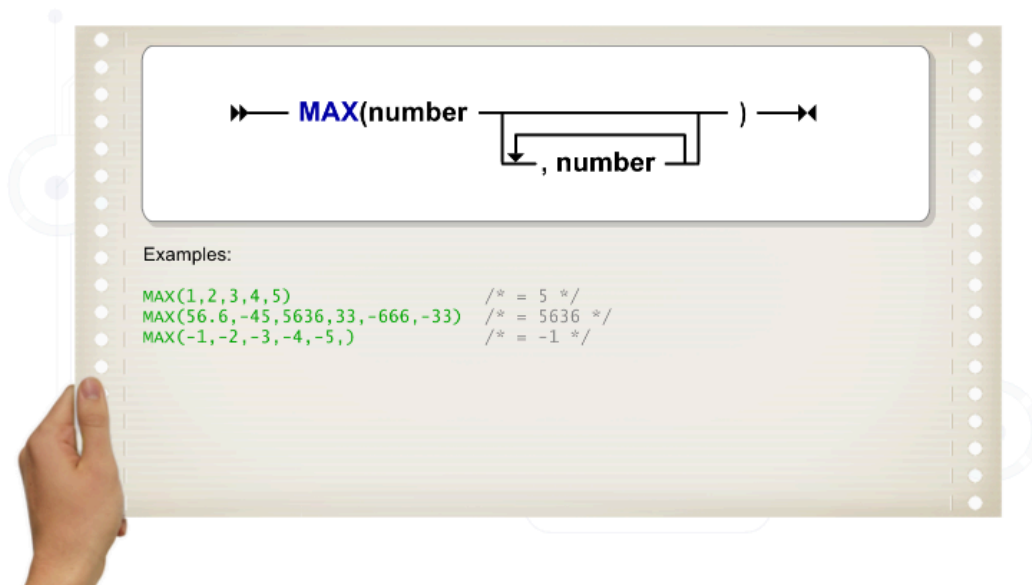
The ABS function returns the absolute value of a number, that is, it makes negative numbers positive. For example, this piece of code:

```
if y<0 then x=-y  
else x=y
```

is equivalent to coding the following:

```
x=abs(y)
```





The MAX function returns the largest number from the list of numbers specified. A maximum of 20 numbers can be passed to the MAX function, but using a MAX function within the MAX function will effectively increase that number.

For example:

```
MAX(1,23,56,24,14,8,9,56,73,43,76,78,45,15,48,26,59,6,28,max(6,34,2,8,39,75,93,72.31))
```

would return the value 93 from the inner MAX function and then use that value in the list of values for the outer function.



Diagram illustrating the MIN function syntax: $\text{MIN}(\text{number } \overbrace{\text{, number}} \text{)}$

Examples:

```
MIN(1,2,3,4,5)           /* = 1 */  
MIN(56.6,-45,5636,33,-666,-33) /* = -666 */  
MIN(-1,-2,-3,-4,-5,)    /* = -5 */
```

The MIN function returns the smallest number from the list of numbers specified.

As with the MAX function, the MIN function can have a maximum of 20 parameters, but can contain nested MIN functions.



The diagram shows the function signature `RANDOM()` with a large bracket indicating the range of parameters. Below the signature, a smaller bracket groups `min, max` as the range parameters, and another bracket groups `max, seed` as the optional parameters. The examples section lists four calls to the function with their respective comments.

```
Examples:  
  
RANDOM()      /* number ranging from 0 to 999 */  
RANDOM(5)     /* number ranging from 0 to 5 */  
RANDOM(5,10)  /* number ranging from 5 to 10 */  
RANDOM(5,10,6) /* always 8 */
```

The RANDOM function generates a pseudo-random number in the range specified by `max` or `min,max`.

If `min` or `max` are not specified, they default to 0 and 999 respectively.

The `seed` value can be defined to enable successive calls to the RANDOM function to be reproducible for testing purposes. If `seed` is not used the first time that RANDOM is called, the numbers will change each time the program runs.



SIGN(number)

Examples:

```
SIGN(5) /* = 1 */  
SIGN(-5) /* = -1 */  
SIGN(0) /* = 0 */
```

This code:

```
result=sign(x)
```

is equivalent to:

```
if x=0 then result=0  
else if x>0 then result=1  
else result=-1
```

The SIGN function returns the value of 1, 0, or -1 depending on the value of the number that is passed to it.

If the number is greater than zero, SIGN returns 1. If the number is less than zero, SIGN returns -1.

If the number is zero, SIGN returns 0.



→ **TRUNC**(number , n) →

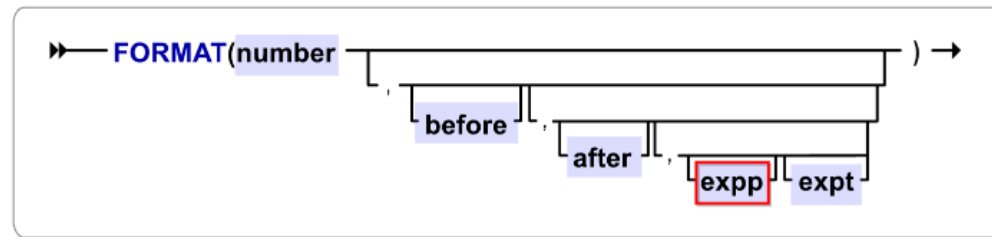
Examples:

```
TRUNC(345.678) /* = 345 */  
TRUNC(345.678,2) /* = 345.67 */  
TRUNC(345.678,6) /* = 345.678000 */
```

The TRUNC function truncates a **number** at **n** decimal places. The default value for **n** is 0.

If **n** is omitted, the **number** will be truncated with zero decimal places and an integer will be returned.

With TRUNC, excess numeric values will be dropped, not rounded.



If the number is to be displayed in scientific notation, `expp` specifies the number of places to set aside for the exponent part of the output. If `expp` is not specified, REXX will use whatever amount of space is necessary.

FORMAT rounds numbers and formats them. The `number` must be passed to FORMAT. All other parameters are optional.

The `before` and `after` operands specify how many places before and after the decimal point. If they are not specified, the `number` will contain the same number of places that it did before the FORMAT operation.

The `expp` and `expt` operands define how and when exponent values will be displayed.

Mouse-over the syntax for a description of each parameter.

```

EDIT          USER1.ISPF.ISPCLIB(NUMERICS)          Columns 00001 00072
Command ==>                                         Scroll ==> CSR
***** ***** Top of Data *****
000010 /* Numeric Functions REXX Example */
000030 n.1 = random(1,300,1234)
000040
000050 do c1 = 2 to 12
000060 n.c1 = random(1,300)
000070 end
000080
000090 hn = max(n.1,n.2,n.3,n.4,n.5,n.6,n.7,n.8,n.9,n.10,n.11,n.12)
000100 ln=min(n.1,n.2,n.3,n.4,n.5,n.6,n.7, min(n.8,n.9,n.10,n.11,n.12))
000110
000120 fn = format(hn,4,2)
000130
000140 say n.1 n.2 n.3 n.4 n.5 n.6 n.7 n.8 n.9 n.10 n.11 n.12
000150 say "The Maximum no. is = "hn ": The Minimum no. is = " ln
000160 say "The formatted maximum no. is = "fn
***** ***** Bottom of Data *****

```

```

n.1 - 80
n.2 - 213
n.3 - 216
n.4 - 12
n.5 - 163
n.6 - 291
n.7 - 169
n.8 - 112
n.9 - 20
n.10 - 96
n.11 - 178
n.12 - 187
hn - 291
ln - 12
fn - 291.00

```

```

%NUMERICS
80 213 216 12 163 291 169 112 20 96 178 187
The Maximum no. is 291: The Minimum no. is 12
The Formatted maximum no. is 291.00

```

Click Play to see some of the numeric functions and how they can be used.



CENTRE or CENTER	Returns a string of a specific length with the specified string centered within it. If they are necessary, the pad characters can also be specified.
JUSTIFY	Formats a given string by adding pad characters between words in the string to justify to both left and right margins of a specified length.
LEFT	Returns a string of a specified length containing the left-most characters of the string.
RIGHT	Returns a string of a specified length containing the right-most characters of the string.
STRIP	Removes a specified number of leading or trailing characters from a given string.

Justification functions enable data to be formatted, particularly for reporting display purposes.

Listed above are the functions that help format data.



Diagram illustrating the `JUSTIFY(string,length[,pad])` function signature. The `string` parameter is shown with arrows indicating it is formatted and justified to both margins. The `length` parameter is shown with a bracket indicating it is the target length. The `pad` parameter is shown with a bracket indicating it is the character used for padding.

Examples:

```
JUSTIFY('Now is the time',21)      /* 'Now is the time' */
JUSTIFY('Now is the time',21,'*') /* 'Now**is**the**time' */
JUSTIFY('Now is the time',12)     /* 'Now is the t' */
```

The JUSTIFY function returns `string` formatted and justified to both margins. The `pad` character is used between blank delimited words to make up characters if `string` is shorter than `length`.

If `string` is longer than `length`, it will be truncated to the next space, if available, and then justified by using the remaining words.

The default `pad` character is a blank space.

Note: JUSTIFY is a non-SAA built-in function and is currently only available under TSO/E and VM.



CENTRE(string, length ,pad)
CENTER(string, length ,pad)

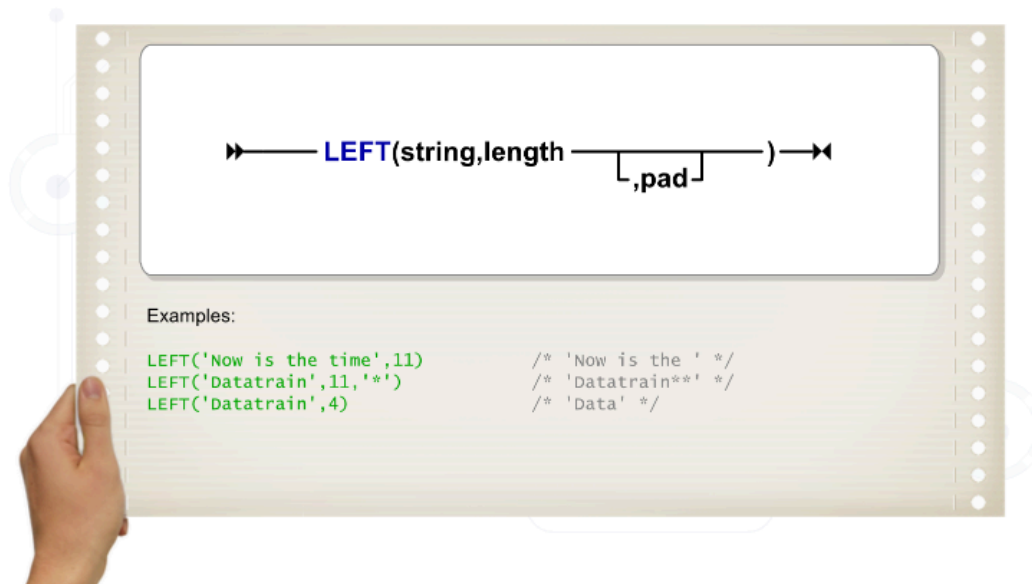
Examples:

```
CENTER('Datatrain',13)      /* ' Datatrain ' */  
CENTER('Datatrain',13,'*') /* '**Datatrain**' */  
CENTER('Datatrain',5)     /* 'tatra' */
```

The CENTER function returns a string centered within a character sequence of **length** characters long.

If the **string** is longer than **length**, truncation occurs. If the **string** is shorter than **length**, the **pad** character is used to make up the **length**.

The **pad** character is a blank space by default.



The LEFT function starts at the left of `string` and returns the number of characters specified by `length`.

If `string` is shorter than `length`, the string is padded to the right with the `pad` character. The default `pad` character is a blank space. If `string` is longer than `length`, truncation occurs on the right.

Specifying `LEFT(string,6)` is the same as specifying the following:

`SUBSTR(string,1,6)`



Diagram illustrating the RIGHT function syntax: `RIGHT(string, length [, pad])`. The `pad` parameter is shown in a box with a bracket, indicating it is optional.

Examples:

```
RIGHT('Now is the time',11)      /* 'is the time' */  
RIGHT('Datatrain',11,'*')      /* '**Datatrain' */  
RIGHT('Datatrain',5)           /* 'train' */
```

The RIGHT function starts at the right of `string` and returns the number of characters specified by `length`.

If `string` is shorter than `length`, the `string` is padded to the left with the `pad` character. The default `pad` character is a blank space.

If `string` is longer than `length`, truncation occurs on the left.





The whiteboard contains a diagram of the `SPACE` function signature: `SPACE(string, n, pad)`. The `string` parameter is enclosed in a box with arrows pointing left and right. The `n` and `pad` parameters are also enclosed in boxes with arrows pointing left and right. Below the diagram, the text "Examples:" is followed by four lines of code, each with its output in comments:

```
SPACE(' Data train ')\nSPACE('Data train ',2, '*')\nSPACE('Data train ',, '+')\nSPACE('Data train ',0, '+')
```

/* 'Data train' */\n/* 'Data**train' */\n/* 'Data+train' */\n/* 'Datatrain' */

The SPACE function replaces all spaces between each word in `string` with the number of `pad` characters specified by `n`. Leading and trailing blanks are always removed.

The default for `n` is 1, and the default for `pad` is a blank space.





option can be:

Option	Meaning	Description
B	Both	Removes both leading and trailing chars from string. This is the default.
L	Leading	Removes leading char from string.
T	Trailing	Removes trailing char from string.

The STRIP function returns **string** with the leading, trailing, or both chars removed, depending on the value of **option**. The default for **char** is a blank space.





Examples:

Option	Meaning	Description
B	Both	<code>STRIP(' A B ')</code> /* 'A B' */ <code>STRIP('**A**B**','B','**')</code> /* 'A**B' */
L	Leading	<code>STRIP('**A**B**','L','**')</code> /* 'A**B**' */
T	Trailing	<code>STRIP('**A**B**','T','**')</code> /* '**A**B' */ <code>STRIP('**A**B**','trailing','**')</code> /* '**A**B' */

Any characters that follow the first character of the `option` parameter are ignored, but specifying the entire word shown in the "Meaning" column above helps to document the program.

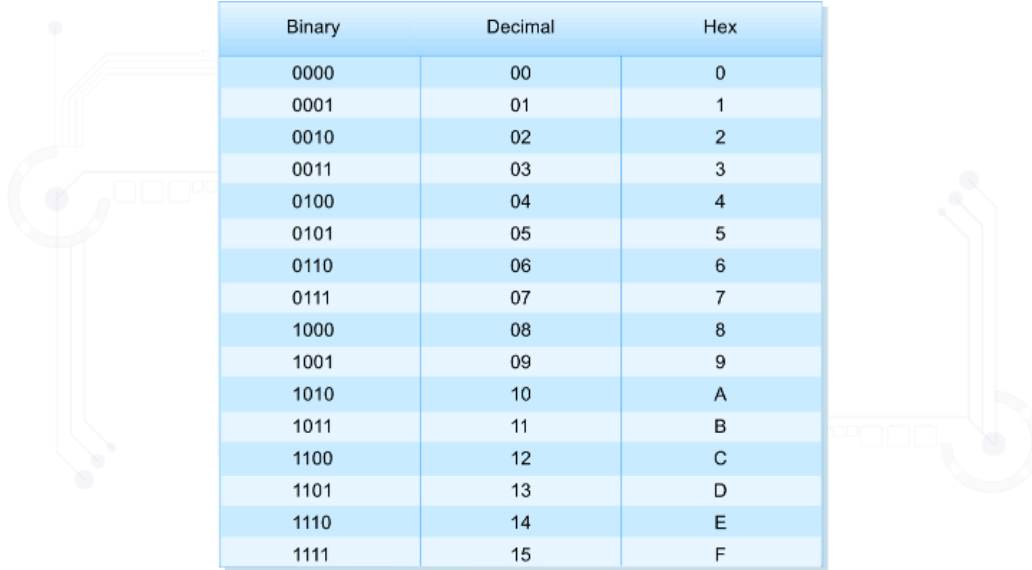
Note that `option` is not case-sensitive and the first character of `option`, if used, is the only character that is required.



```
EDIT          USER1.ISPF.ISPCLIB(JUSTIFY)          Columns 00001 00072
Command ==>>                                     Scroll ==> CSR
***** Top of Data *****
000010 /* REXX */
000020 heading1 = 'Name item cost'
000031 rec1a = 'Fred Nurk'
000041 rec1b = 'tool'
000051 rec1c = '$65.80'
000061 rec2a = 'Harry Houdini'
000071 rec2b = 'handcuffs'
000081 rec2c = '$192.60'
000091 rec3a = ' Peter Piper'
000100 rec3b = 'peppers'
000110 rec3c = '$6.00'
000120
000130 heading = justify(heading1,56)
000140
000150 say heading
000160
000170 say left(rec1a,20) center(rec1b,15,'-') right(rec1c,19)
000180 say substr(rec2a,1,20) centre(rec2b,15,'-') right(rec2c,19)
000190 say left(strip(rec3a),20) centre(rec3b,15,'-') right(rec3c,19)
***** Bottom of Data *****
```

```
%justify
Name          item          cost
Fred Nurk     ----tool-----    $65.80
Harry Houdini --handcuffs---    $192.60
Peter Piper   ---peppers---      $6.00
```

Click Play to see some of the justification functions and how they are used.

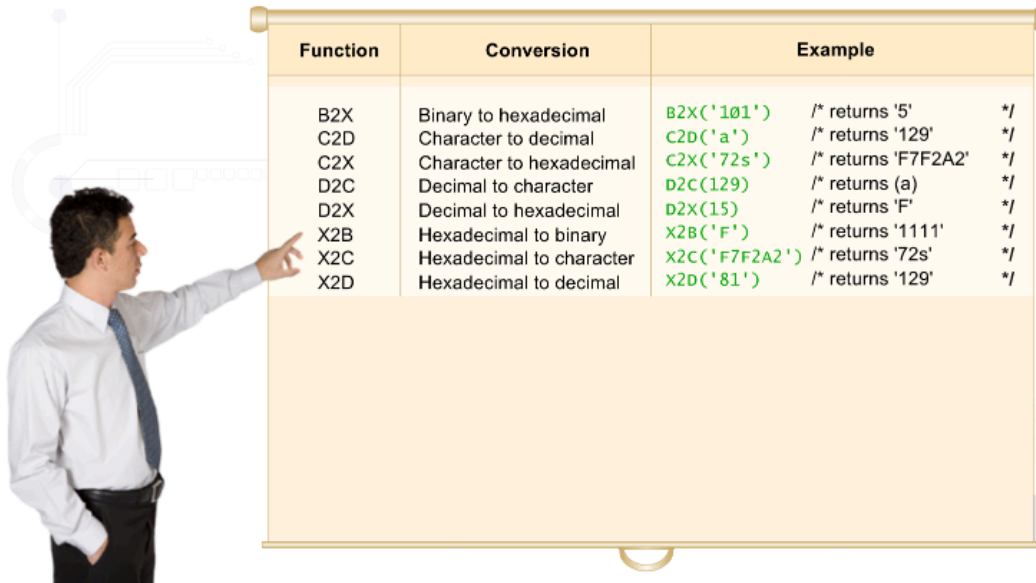


Binary	Decimal	Hex
0000	00	0
0001	01	1
0010	02	2
0011	03	3
0100	04	4
0101	05	5
0110	06	6
0111	07	7
1000	08	8
1001	09	9
1010	10	A
1011	11	B
1100	12	C
1101	13	D
1110	14	E
1111	15	F

Electronic components have two basic states: on or off, and open or closed. Computers must therefore work at the very base level, binary, which is also called base 2. This means that all states are referred to as 0 or 1, which are the two numbers allowed in base 2.

It is difficult to read large amounts of zeros and ones so binary digits are grouped into groups of 4, giving 16 combinations from 0000 to 1111. Each of these 16 combinations is represented by the characters 0-9 and the letters A-F. These characters are referred to as hex characters or a base 16 and they can also be represented as a decimal number or base 10. For example, Binary 1111 is the same as hex "F" and decimal 15.

Click Play to see the binary, decimal, and hex representations of the numbers 0-15.

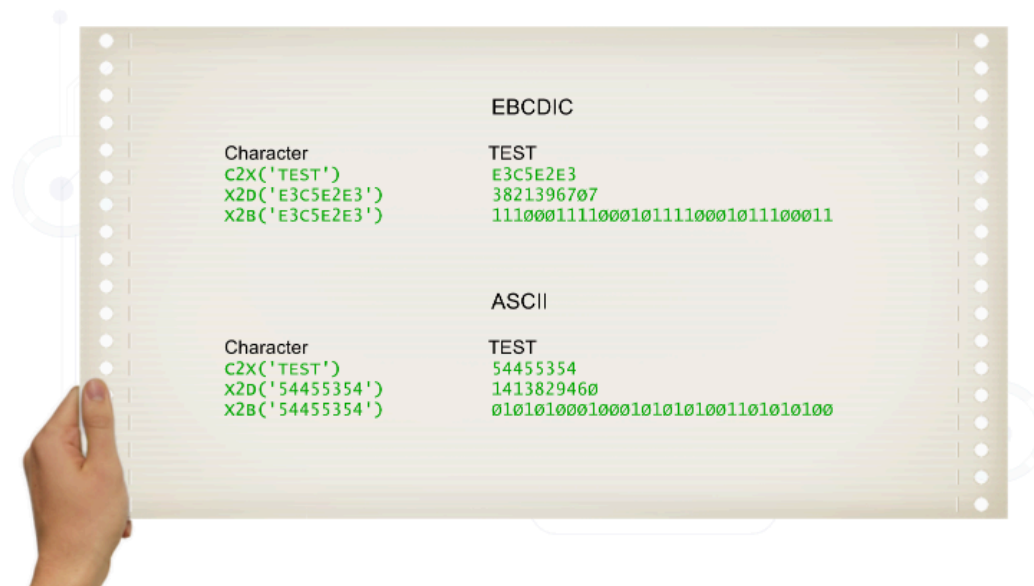


Function	Conversion	Example
B2X	Binary to hexadecimal	<code>B2X('101')</code> /* returns '5' */
C2D	Character to decimal	<code>C2D('a')</code> /* returns '129' */
C2X	Character to hexadecimal	<code>C2X('72s')</code> /* returns 'F7F2A2' */
D2C	Decimal to character	<code>D2C(129)</code> /* returns (a) */
D2X	Decimal to hexadecimal	<code>D2X(15)</code> /* returns 'F' */
X2B	Hexadecimal to binary	<code>X2B('F')</code> /* returns '1111' */
X2C	Hexadecimal to character	<code>X2C('F7F2A2')</code> /* returns '72s' */
X2D	Hexadecimal to decimal	<code>X2D('81')</code> /* returns '129' */

To represent displayed and stored characters in the computer, two hex characters are joined to enable a total of 256 characters, which is 16 x 16, to be defined. Approximately 100 of these are "displayable" and the rest are used to instruct the computer on what to do. Displayable hex values are referred to as character format.

Although hex is the usual format to display non-displayable characters, it is possible to display and manipulate data in binary, hex, decimal, and character formats. For example, the displayable character **A** on an IBM mainframe can also be referred to as hex `C1`, decimal `193`, or binary `11000001`.

Listed above are conversion functions that are used to convert data from one format to another.



The two different hex representation standards that are used in the most common computers are EBCDIC and ASCII.

EBCDIC is used by most IBM mainframe systems, such as z/OS and z/VM. ASCII is used by most mid-range and small systems, such as PCs. Because of this difference, applications using these functions are rarely ported from one platform to another.

Click Play to see an example of the difference between ASCII and EBCDIC conversions.



`BITAND('0101'b,'0001'b)` - Binary "and" logic. If both values are 1, the result is 1.

```
0101 "and"
0001
-----
0001
```

`BITOR('0101'b,'0001'b)` - Binary "or" logic, if either value is 1, the result is 1.

```
0101 "or"
0001
-----
0101
```

`BITXOR('0101'b,'0001'b)` - Binary "exclusive or" logic. If either value is 1, but not both,

```
0101 "or"
0001
-----
0100
```

`BITOR('T','E')`

```
"T" -> "54"x (ASCII) -> 01010100
"E" -> "45"x (ASCII) -> 01000101
-----
"U" <- "55"x (ASCII) <- 01010101
```

Three related functions are used for the Boolean logic operations of "and", "or", and "exclusive or".

These functions enable two binary strings to be compared and Boolean logic to be performed at a bit level. Using bit flags can save space and improve the efficiency of programs.

Click Play to see how the three binary logic functions calculate their results.

