Print determines if an expression is real, complex, or string from the first symbol in the expression. Usually this works fine, but occasionally Print guesses wrong and you may have to rearrange your expression. For example:

```
Print 1+cmplx(1,2)
```

will give an error because the first symbol, "1", is real but the expression should be complex. Changing this to

```
Print cmplx(1,2)+1
```

will work.

Printing numeric or string expressions involving structure elements must not start with the structure element. Instead an appropriate numeric or string literal must appear first so that Igor can determine what kind of expression to compile. For example rather than

```
Print astruct.astring + "hello"
```

use

```
Print "" + astruct.astring + "hello"
```

Print breaks long strings into multiple lines. If there are no natural breaks (carriage returns or semicolons) within a default length, then it breaks the string arbitrarily.

The default line length is 200 bytes. You can override this using the /LEN flag. The maximum number of bytes that can be printed on a line in the history area is 2500.

When printing waves, you can use either formatted (specified by /F) or unformatted (default) methods. Unformatted output is in an executable syntax for each printed line: `wave={}`.

Note: Executing lines printed from floating point waves will not exactly reproduce the source data due to roundoff or insufficient digits in the printed output.

Printing formatted wave data gives easily (human) readable output, and works best for small 1D and 2D waves. If the data are too large or in an unsupported format (3D or greater, or the wave is text), then the output will be unformatted. Formatting is done using spaces, so the output will look best in a fixed-width font.

Printed wave data, both formatted and unformatted, are limited to no more than 100 lines of output. When the line limit is exceeded a warning message will be printed at the end of the truncated output. For text waves, output is limited to 50 bytes of each string element, and there is no warning when a string is truncated.

See Also

The **printf** operation.

The **PrintGraphs**, **PrintTable**, **PrintLayout** and**PrintNotebook** operations.

printf

`printf` ***`formatStr`*** `[,` ***`parameter`*** `[,` ***`parameter`***`]…]`

The printf operation prints formatted output to the history area.

Parameters

formatStr is a string which specifies the formatting of the output.

The type of the parameter, string or numeric, must agree with the corresponding conversion specification in *formatStr*, or else the results will be indeterminate.

The printf parameters can be numeric or string expressions. Numeric and string structure fields are allowed except that complex structure fields and non-numeric (e.g., WAVE, FUNCREF) structure fields are not allowed.

Details

The *formatStr* contains literal text and conversion specifications.

A conversion specification starts with the % character and ends with a conversion character (for example, g, e, f, d, or s as illustrated below). In between the % and the conversion character you may include one or more flag characters, a field width specifier, and a precision specifier. The first % corresponds to the first parameter, the second % corresponds to the second parameter, etc. If *formatStr* contains no % characters, no parameters are expected.

Here are some simple examples. `numVar` is a numeric variable and `strVar` is a string variable.

```
printf "The answer is: %g\r", numVar
printf "Created wave %s\r", strVar
printf "Created wave %s, %d points\r", strVar, numVar
```

%g is a general-purpose format (floating point or scientific notation) that represents the value of numVar. %d is an integer format that represents the value of `numVar`. %s specifies that the corresponding parameter (strVar) is a string.

The "\r" in these examples appends a carriage return to the end of the printed text.

Here is a complex example using all of these elements of a conversion specification:

```
printf "%+015.4f\r", 1e6*PI
```

This prints:

```
The answer is: +003141592.6536
```

"+" is a flag character that tells printf to put a + or - sign in front of the number.

"015" is a field width specifier that tells printf to print the number in a field of at least 15 bytes, padded with leading zeros. Using "15" instead of "015" would cause printf to pad with spaces before the + sign instead of zeros after it.

".4" is a precision specifier that tells printf to print four digits after the decimal point.

"f" tells printf to use a floating point format.

The most common conversions characters are “f” for floating point, “g” for general, “d” for decimal, and “s” for string. They are interpreted as for the printf() function in the C programming language.

The escape codes \t and \r represent the tab and return characters respectively. See **Escape Sequences in Strings** on page IV-14 for more information.

Printf Flag Characters

The supported flag characters and their meanings are as follows:

-	Left align the result in the field.
+	Put a plus or minus sign before the number.
<space>	Put a space before a positive number.
#	Specifies alternate form for e, f, g, and x formats.

The meaning of the precision specifier depends on the numeric format (%g, %e, %f, %d, etc.) being used:

e, E, f	Precision specifies number of digits after decimal point.
g, G	Precision specifies maximum number of significant digits.
d, o, u, x, X	Precision specifies minimum number of digits.

You can replace both the field width and precision specifiers with an asterisk. This gets the field width or precision specifier from a parameter. For example:

```
printf "%*.*f\r" 4, 3, 1e6*PI
```

means that the field width is 4 and the precision is 3. You could use numeric expressions instead of the literal numbers to control the field width and precision algorithmically.

Printf Conversion Characters

Here is a complete list of the conversion characters supported by printf:

To include a literal percent character, use two consecutive % characters, like this:

```
Variable percentage = 37.3
Printf "%.1f%% of participants had prior medical conditions", percentage
```

Igor also supports a non-C, WaveMetrics extension to the conversion characters recognized by printf. This conversion specification starts with “%W”. It is followed by a flag digit and a format character. For example,

```
printf "%W0Ps", 12.345E-6
```

f	Converts a numeric parameter as [-]ddd.ddd, where the number of digits after the decimal point is determined by the precision specifier and defaults to 6. If the # flag is present, a decimal point will be used even if there are no digits to the right of it.
	This conversion character uses the “round-to-half-even” rule, also known as “banker’s rounding”. When the truncated digits are exactly 0.5000..., the quantity is rounded to an even number. For example:
	`Printf "%.0f\r", 15.5        // Prints 16 (rounded up to even)`
	`Printf "%.0f\r", 16.5        // Prints 16 (rounded down to even)`

e, E	Converts a numeric parameter as [-]d.ddde+/-dd, where the number of digits after the decimal point is determined by the precision specifier and defaults to 6. If you use “E” instead of “e” then printf uses a capital “E” in the number. If the # flag is present, a decimal point will be used even if there are no digits to the right of it.
g, G	Converts a numeric parameter using “f” or “e” style conversion depending on the magnitude of the number. “e” is used if the exponent is less than -4 or greater than the precision. “G” uses “f” or “E” style conversion. If the # flag is present, a decimal point will be used even if there are no digits to the right of it and trailing zeros will not be removed.

d, o, u	Converts a numeric parameter as a signed decimal integer, unsigned octal integer or unsigned decimal integer. The precision defaults to one and specifies the minimum number of digits to print.
	These conversion characters use the “round-away-from-zero” rule, like Igor’s **round** function. For example:
	`Printf "%d\r", 15.5          // Prints 16 (rounded away from zero)`
	`Printf "%d\r", 16.5          // Prints 17 (rounded away from zero)`

x, X	Converts a numeric parameter as an unsigned hexadecimal integer, rounding floating point values. Also supports integer data up to 64 bits. The “x” style uses lower case for the hexadecimal numerals “abcdef” where the “X” style uses upper case. The precision defaults to one and specifies the minimum number of digits to print. If the # flag is present, the string “0x” or “0X” is prepended to the number if it is not zero.
s	Converts a string parameter. If a precision is specified, it sets the maximum number of bytes from the string parameter to be printed. As of Igor Pro 9.00, there is no limit to the length of the string parameter. It was limited to 2400 bytes in Igor Pro 8 and to 1000 bytes before that.
b	WaveMetrics extension. Converts a numeric parameter to binary.
c	Converts a numeric parameter to a single ASCII character.
%	Prints a % sign. No parameter is used.
%W	WaveMetrics extension. See description below.

prints 12.345000µs. In this example, the “%W0” introduces the WaveMetrics conversion specification. The “0” (zero) following the “W” is the flag digit. The “P” that follows is the format specifier character, which prints the number using a prefix, in this case, “µ”.

There is only one WaveMetrics format specifier character, “P”, which prints using a prefix such as µ, m, k, or M. It recognizes two flag-digits, “0” or “1”. Option “0” prints with no space between the numeric part and the prefix character while flag “1” prints with 1 space. Numbers greater than tera or less than femto print using a power of ten notation. Here are a few examples:

```
printf "%.2W0PHz", 12.342E6      // prints 12.34MHz
printf "%.2W1PHz", 12.342E6      // prints 12.34 MHz
printf "%.0W0Ps", 12.342E-6      // prints 12µs
printf "%.0W1Ps", 12.342E-9      // prints 12 ns
```