Skin Expressions

Was this page helpful?

- We have highlighted your search term **skin expressions** for you. If you'd like to remove the search term, click here.

Overview

Skin expressions (functions and properties) provide great flexibility to the skin designer, but sometimes simple property replacement is not enough. One such case is when you need to combine a property with a localized string. Skin expressions provide a way to calculate more complex operations based on skin properties, localizable strings and other.

A skin expression is a construct that allows calculations to be performed on property values and the result substituted in the releavant skin control tag/attribute. It can be used (almost) anywhere a property can, e.g. as a label text, a texture filename, etc. It can even be used within <define> tags to avoid repeating the same expression over and over. It cannot however be used in visibility conditions.

An expression is any of the following:

- a string literal in single quotes - e.g. 'This is a string literal'
- an integer literal - e.g. 123
- a floating point literal - e.g. 12.34 or even 12. (note the dot at the end)
- a property - e.g #itemcount
- a function call - e.g. string.format(123, #itemcount)

Function calls have the following general syntax:

```
functionname([expression[,expression[...]]])
```

Because each argument to a function call is also an expression, it is possible to use nested function calls as in:

```
string.format('page {0}/{1}', div(#itemindex, 10), div(#itemcount, 10))
```

Notice that there are no operators (yet).

To distingush expressions from plain text, an expression has to be enclosed in a #(...) construct. E.g.

```
<control>
      <description>Number of Files Label</description>
      <type>label</type>
```

```
        <id>1</id>
        <posX>462</posX>
        <posY>682</posY>
        <label>#selectedindex/#(string.formatcount(#itemcount, 'no items|{0} item|{0} items'))</label>
        <align>left</align>
        <textcolor>White</textcolor>
        <font>font10</font>
        <visible>string.equals(#selectedindex)+string.equals(#itemcount)</visible>
        <animation effect="fade" time="250">WindowOpen</animation>
        <animation effect="fade" time="250">WindowClose</animation>
    </control>
```

In the above xml code fragment, only the #(string.formatcount(...)) function call and its enclosed arguments are expressions, #selectedindex is not. Notice however that the #(...) construct has to be used only once, and that is where the expression is to be evaluated. That means that expressions in <define> tags should not be enclosed in #(...).

White-space is ignored but in order to make parsing and caching of expressions faster, expressions are compared including white-space. If you use some expression multiple times or in multiple screens make sure you use the same white-space. The following two will return the same result but if both are used in a skin, two expressions will be cached:

```
string.format(123,#itemcount)
string.format ( 123, #itemcount )
```

To ensure the best performance of your skin, decide on a white-space usage convention and stick to it.

Functions

MediaPortal comes with a set of core skin functions that can be used in skin xml. Plugin Developers can provide additional custom functions through their plugins. All functions - core and custom - are treated equally. The only difference is that core functions are guaranteed to be present in every MediaPortal installation.

Currently the following core functions, grouped in categories, are supported:

- String
- Conversion
- Math
- Date/Time
- Conditionals / Flow Control
- Boolean

String

String functions let you manipulate strings in several ways. They either take strings as arguments or return a string as a result (often both). Since skin properties are always strings and the purpose of skin expressions is usually to construct some text to display or some filename to use as texture, these functions are the nost commonly used ones.

string.contains(*string a, string b***)**

> [Since 1.1]
>
> Returns true if string 'a' contains string 'b'.

string.starts(*string a, string b***)**

> [Since 1.1]
>
> Returns true if string 'a' starts with string 'b'.

string.equals(*string a, string b***)**

> [Since 1.1]
>
> Returns true if string "a" and "b" are identical. Some notes:

- The compare is case sensitive
- Quotes (") are evaluated as well, so string.equals(a,"b") is completly different from string.equals(a,b)
- You can use operators like + | & but be carefull with spaces:
    - string.equals(a,b) | string.equals(c,d) <<< not working
    - string.equals(a,b)|string.equals(c,d) <<< OK

string.format(*format string id*, *arg1*, *arg2*, ...**)**

> [Since 1.2]
>
> Format parameters according to *format string* or localized string referenced by *format string id.*
>
> The format string contains format items in the syntax:
>
> {*index*[,*length*][:*formatString*]}
>
> Elements in square brackets are optional. The following describes each element.
>
> *index*
>
> > The zero-based position in the parameter list of the object to be formatted. If there is no parameter in the index position, an error is returned
>
> *,length*
>
> > The minimum number of characters in the string representation of the

parameter. If positive, the parameter is right-aligned; if negative, it is left-aligned.

:*formatSpecifier*

A standard or custom format string that is supported by the object to be formatted. Possible values for formatSpecifier are the same as the values supported by the object's ToString(format) method. If formatSpecifier is not specified a default format is used.

For more details on the syntax of format string, see the .Net Framework function String.Format(). Especially useful are Standard and Custom number/date formats.

Examples:

```
#(string.format('{0}/{1} items', #selectedindex, #itemcount))
#(string.format(100, #selectedindex, #itemcount))
```

Both will return a string like this: "3/85 items"

(assuming string with id=100 contains the localizable string "{0}/{1} items')

Some number format examples (assuming $starrating = 4.56 and #votes = 1234567):

```
#(string.format('{0:f1} ({1:n})', cflt(#starrating), cint(#votes)))
```

Returns **4.6 (1,234,567)** on en-US regional settings but **4,6 (1.234.567)** on el-GR. The standard number formats use the regional settings to determine how to format the numbers.

```
#(string.format('{0:0.0} ({1:#,0})', cflt(#starrating), cint(#votes)))
```

Returns **4.6 (1,234,567)** on any regional settings. Regional settings still determine group and decimal separators but not whether they will be used or not. Also note that the 0.0 format will output 4.0 if #starrating = 4.

Some date format examples (assuming #date = 17/3/2010):

```
#(string.format('{0:D})', cdate(#date)))
```

Returns **Wednesday, March 17, 2010** on en-US regional settings but **Τετάρτη, 17 Μαρτίου 2010** on el-GR. The standard date formats use the regional settings to determine how to format the date.

```
#(string.format('{0:dd/mm/yy})', cdate(#date)))
```

Returns **17/03/10** on any regional settings.

```
#(string.format('{0:MMMM d, yyyy})', cdate(#date)))
```

Returns **March 17, 2010** on en-US regional settings and **Μάρτιος 17, 2010** in el-GR. Note that the format does not change based on regional settings but the month names get translated.

In general use standard number/date formats to show the values formatted according to the user's regional settings. Use custom number/date formats to show the values formatted the way you want regardless of the user's regional settings.

Also never forget to convert properties to the proper type before formatting. Trying to format an unconverted (string) property as date, will simply output the property string unmodified.

string.formatcount(*value*, *multi format***)**

string.formatcount(*value*, *multi format id***)**

[Since 1.2]

Format *value* according to one of three formats depending on whether *value* is 0, 1 or > 1. The argument *multi format* (or the localizable string specified by *multi format id*) should contain the the formats separated by the | character.

Example:

```
#(string.formatcount(#itemcount, 'no items|{0} item|{0} items))
```

If #itemcount is 0 this function will return "no items"

If #itemcount is 1 this function will return "1 item"

If #itemcount is greater than 1 this function will return something like "53 items"

L(*Id***)**

[Since 1.2]

Return the localized string for ID *Id*. When used alone, it is equivalent to simply using a string ID. But this function allows you to combine a localizable string with literal text and properties as in the following example:

```
<label>#selectedindex/#itemcount #(L(101))</label>
```

This will display something like "8/53 items" *(provided that string with id = 100 contains the localizable string "items")*

Note though, that you can achieve the same results using **string.format**().

string.ltrim(*string*[, *charsToTrim*])

[Since 1.2]

Trim whitespace or *charsToTrim* (if supplied) from the start (left) of the string.

Example:

```
<label>string.ltrim(#date)</label>
<label>string.ltrim(#date,'A,B,C')</label>
```

string.rtrim(*string*[, *charsToTrim*])

[Since 1.2]

Trim whitespace or *charsToTrim* (if supplied) from the end (right) of the string.

Example:

```
<label>string.rtrim(#time)</label>
<label>string.rtrim(#time,'A,M,P')</label>
```

string.trim(*string*[, *charsToTrim*])

[Since 1.2]

Trim whitespace or *charsToTrim* (if supplied) from both ends of the string.

Example:

```
<label>string.trim(#time)</label>
<label>string.trim(#time,'1,2,3,4,5,6,7,8,9,0')</label>
```

Conversion

Back Up

Skin Properties in MediaPortal are always strings. But some functions either require some other type (integer, date etc.) as parameter, or work differently based on the type of the parameters passed. In most cases strings are implicitly converted to the required type. But if the conversion is ambiguous, or result in loss of precision, it cannot be applied implicitly and you have to explicitly convert to the desired type. At other times the implicit conversion will choose a type to convert to that is not appropriate for your needs. In these cases you may choose to explicitly convert to the type you desire. To explicitly convert any value to some other type, use one of the following functions.

cint(*value*)

[Since 1.2]

Convert *value* to integer. If *value* is not a number, returns an error.

cflt(*value*)

[Since 1.2]

Convert *value* to float. If *value* is not a number, returns an error.

cdate(*value*)

[Since 1.2]

Convert *value* to date. If *value* is not a valid date, reutrns an error.

Math

Back Up

Some times the values returned in properties are not exactly how you want them. You many need to do some calculations on these values. Although operators are not supported (yet) you can still do basic math using function syntax.

neg(*arg***)**

[Since 1.2]

Return the negative of *arg*.

add(*arg1*, *arg2*, ...**)**

[Since 1.2]

Return the sum of all arguments (i.e. *arg1*+*arg2*+....*argN*).

sub(*arg1*, *arg2***)**

[Since 1.2]

Return the *difference arg1 - arg2*.

mul(*arg1*, *arg2*, ...**)**

[Since 1.2]

Return the product of all arguments (i.e. *arg1 * arg2 * ... * argN*).

div(*arg1*, *arg2***)**

[Since 1.2]

Return the quotient *arg1 / arg2*. If arg2 is zero, a "division by zero" error will be returned.

math.round(*number*[, *digits*])

[Since 1.2]

Round *number* to the closest number having *digits* decimal digits. If digits is not supplied, 0 is assumed (i.e. round to closest integer).
Note that you can also use negative values for digits to round to multiples of 10, 100 etc.

Examples:

math.round(16.32, 1) returns **16.3**

math.round(16.36, 1) returns **16.4**

math.round(16.32) returns **16**

math.round(16.32, -1) returns **20**

math.ceil(*number*[, *digits*])

[Since 1.2]

Return the smallest number having *digits* decimal digits that is greater than or equal to *number*. Similar to **math.round()** but instead of rounding, truncates upwards.

Examples:

math.ceil(16.32, 1) returns **16.4**

math.ceil(16.36, 1) returns **16.4**

math.floor(*number*[, *digits*])

[Since 1.2]

Return the largest number having *digits* decimal digits that is lesst than or equal to *number*. Similar to **math.round()** but instead of rounding, truncates downwards.

Examples:

math.floor(16.32, 1) returns **16.3**

math.floor(16.36, 1) returns **16.3**

Date/Time

Back Up

Working with dates is always tricky. You can't treat them as strings, you can't treat them as numbers. You have to use specialized functions that work on dates. Core functions have been included to allow adding and subtracting dates and times as well as extracting date/time parts. There are two basic types used:

- **date** which is actually date/time. It can hold any date and/or time
- **timespan** which can hold the difference between two dates/times. It can later be added/subtracted from any date/time, or specific parts of it extracted.

date.add(*interval*, *number*, *date*)

[Since 1.2]

Add the *number* of *intervals* to *date* and return the resulting date. number can be positive or negative and in some cases even decimal. The valid values for *interval* are:

- **d** or **dd** or **y** or **dy** or **w** or **dw**: Days
- **wk** or **ww**: Weeks
- **m** or **mm**: Months
- **q** or **qq**: Quarters
- **yy** or **yyyy**: Years
- **h** or **hh**: Hours
- **n** or **nn**: Minutes
- **s** or **ss**: Seconds
- **ms**: Milliseconds

date.add(*date*, *timespan*)

[Since 1.2]

Add *timespan* to *date* and return the resulting date. Timespans represent the difference between two dates.

date.sub(*date1*, *date2*)

[Since 1.2]

Subtract two dates, returning the timespan from date2 to date1.

date.add(*date*, *timespan*)

[Since 1.2]

Subtract *timespan* from *date* and return the resulting date. Timespans represent the difference between two dates.

date.extract(*interval*, *date*)

date.extract(*interval*, *timespan*)

[Since 1.2]

Extract a date part from a date or timespan. Return the number of *interval*s in *date* or *timespan*. The valid values for *interval* are:

- **d** or **dd**: Day of month (dates) or Days (timespans)
- **y** or **dy**: Day of year (only use with dates)
- **w** or **dw**: Day of week (only use with dates)
- **wk** or **ww**: Week of year (dates) or weeks (timespans)
- **m** or **mm**: Month (only use with dates)
- **q** or **qq**: Quarter (only use with dates)
- **yy** or **yyyy**: Year (only use with dates)
- **h** or **hh**: Hours
- **n** or **nn**: Minutes
- **s** or **ss**: Seconds
- **ms**: Milliseconds

Example:

```
<label>Recorded #(string.formatcount(date.extract('d', date.sub(cdate(#date), cdate(#recordeddate))), 'today|yesterday|{0} days ago'))</label>
```

depending on #date and #recordeddate, would show:

Recorded today

or

Recorded yesterday

or

Recorded 3 days ago

Conditionals / Flow control

Back Up

You often need to check for certain conditions in your skin and provide a different UI based on that. Maybe you want to change background based on the time of day, or use an appropriate greeting text. Conditionals do exactly that, and there are several flavours to choose from depending on the complexity of the situation.

iif(*condition*, *true part*, *false part***)**

[Since 1.2]

If *condition* is true return *true part* else return *false part*. *condition* can be any expression returning a boolean result.

Example:

```
<label>#(iif(not(#ispaused), 'Pause', 'Play'))</label>
<label>#(L(iif(not(#ispaused), 101, 102)))</label>
```

Will show either Play or Pause depending on the current state *(assuming strings with ids 101 and 102 are 'Pause' and 'Play' respectively)*

choose(*index*, *arg1*, *arg2*, ...**)**

[Since 1.2]

Return the *index*th argument. *index* is 0-based.

Example:

```
#(choose(date.extract('dw', #date), 'Mon', 'Tue', 'Wed', 'Thu', 'Fri',
'Sat', 'Sun'))
```

Will return the abbreviated day name.

switch(*condition1*, *value1*, *condition2*, *value2*, ...)

[Since 1.2]

Return the first *value* for which the corresponding *condition* is true.

Example:

```
<define>#hour: date.extract('h', #time)</define>
...
<texture>#(switch(and(gte(#hour, 6), lt(hour, 12)), 'back-morning.png', and(gte(#hour, 12), lt(#hour, 15)), 'back-noon.png', and(gte(#hour, 15), lt(#hour, 19)), 'back-evening.png', or(gte(#hour, 19), lt(#hour, 6)), 'back-night.png'))</texture>
```

Assuming the texture referred to is that of the background, this will rotate the background between morning, noon, evening and night based on the time of day.

Boolean

Back Up

Having conditional functions would be of no use, without a way to build as complex conditions as necessary to suit your needs. Boolean funtions allow you to build such conditions. There are two types of boolean functions:

- Comparison functions allow you to check for specific values / ranges of values

- Boolean logic functions can be used to combine conditions to build more complex ones

eq(*value1*, *value2***)**

[Since 1.2]

Return true if *value1* = *value2*.

neq(*value1*, *value2***)**

[Since 1.2]

Return true if *value1* <> *value2*.

gt(*value1*, *value2***)**

[Since 1.2]

Return true if *value1* > *value2*.

gte(*value1*, *value2***)**

[Since 1.2]

Return true if *value1* >= *value2*.

lt(*value1*, *value2***)**

[Since 1.2]

Return true if *value1* < *value2*.

lte(*value1*, *value2***)**

[Since 1.2]

Return true if *value1* <= *value2*.

not(*condition***)**

[Since 1.2]

Return true if *condition* is false.

and(*condition1*, *condition2*, ...**)**

[Since 1.2]

Return true if all *condition*s are true.

or(*condition1*, *condition2*, ...**)**

[Since 1.2]

Return true if one *condition* is true.