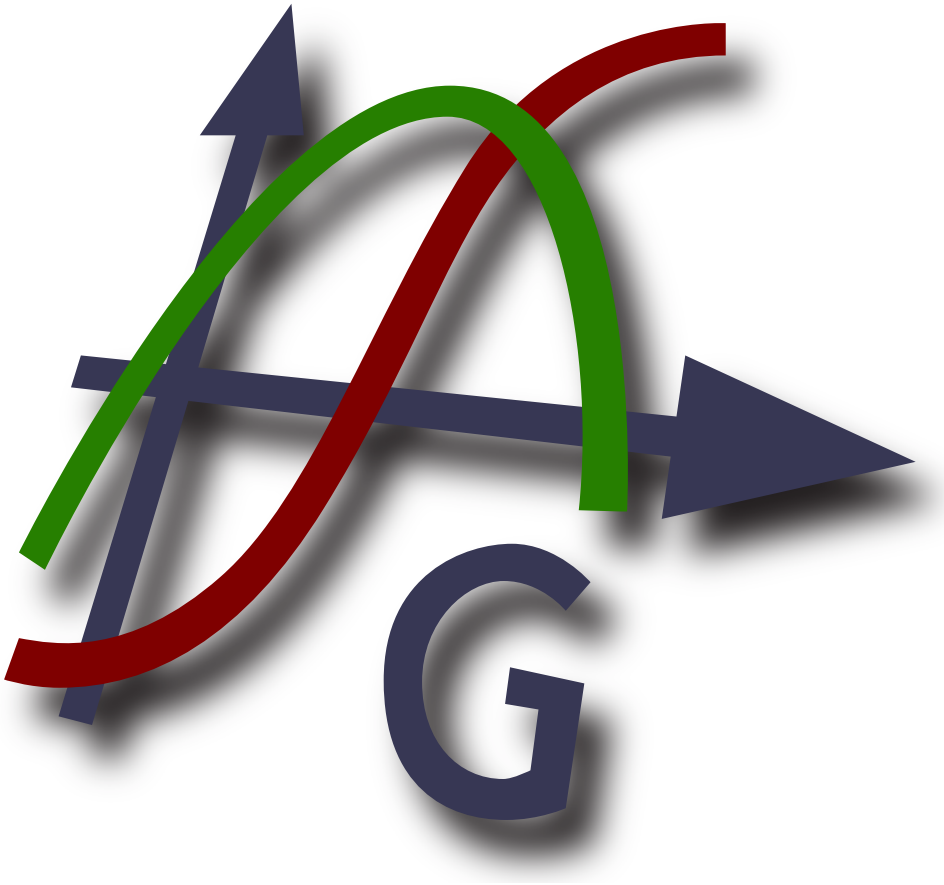
Scripting Engine Documentation for Graph



Version 4.4

Table of Contents

Overview

The scripting engine can be used to create plugins or to enter commands and access advanced features directly in Graph. In both cases you need to install the 32 bit version of Python 3.2 from http://www.python.org. Documentation of the Python language may be found installed with Python or online [http://docs.python.org/3.2/].

Plugins

Plugins are Python scripts and are usually distributed in source form as .py files. The plugin files are placed in the `Plugins` directory where Graph is installed, and will automatically be found and loaded by Graph. A plugin will usually register some a callback function to be called when some event occur when the plugin is loaded. This can for example be a function that is called when a menu item is selected. This function will when do the actual work of the plugin. The work done when the plugin is loaded should be kept to a minimum to keep loading time low.

Python interpreter

You can also use the scripting engine through the Python interpreter shown when you press **F11** inside Graph. In this interpreter you can write Python expressions and that way do very advanced things. It is also an easy way to test code before it is used in a plugin.

Graph module

Graph.**Axes**
Structure of class `Graph.TAxes` with axes settings. These settings are stored in the .grf file.

Graph.**Property**
Structure of class `Graph.TProperty` with global settings. These settings are stored in the users profile.

Graph.**GuiFormatSettings**
Structure of class `Graph.TGuiFormatSettings` with global formatting settings. Changes to these settings are not stored.

Graph.**PlotSettings**
Structure of class `Graph.TPlotSettings` with global plot settings. Changes to these settings are not stored.

Graph.**GuiSettings**
Structure of class `Graph.TGuiSettings` with global GUI settings. Changes to these settings are not stored.

Graph.**VersionInfo**
A tuple containing the five components of the version number of Graph: `Major`, `Minor`, `Release`, `Build`, and `ReleaseLevel`. All values except `ReleaseLevel` are integers; the release level is `'beta'` or `'final'`. The `VersionInfo` value corresponding to the Graph version 4.3 is `(4, 3, 0, 384, 'final')`. The components can also be accessed by name, so `Graph.VersionInfo[0]` is equivalent to `Graph.VersionInfo.Major` and so on.

Graph.**Redraw**()
Redraws the the graphing area.

Graph.**Update**()
Forces all graph elements, i.e. functions etc., to recalculate, redraw the plotting area and update the function list.

Graph.**CreateAction**(*Caption, OnExecute, Hint="", ShortCut="", IconFile=None, OnUpdate=None, AddToToolBar=True*)
Creates a new action with *Caption* as the shown text. Actions are used in the user interface, for example in the toolbar and menus. *OnExecute* is a function with the action as argument that is called when the action is triggered. *Hint* is an optional tooltip for the action. *ShortCut* is an optional shortcut as a text string, e.g. "Ctrl+Shift+C". *IconFile* is a file name for an image file that will be used as icon for the action. The file name can be a fully qualified path or a path relative to the Plugin directory. *OnUpdate* is an optional function with the action as argument that will be called when Graph is idle. This can be used to update the action, for example change its visibility state or enable/disable the action. *AddToToolBar* indicates if the action will be available to add to the toolbar by the user.

Graph.**AddActionToMainMenu**(*Action*)
Adds an action to the main menu unde the Plugin top menu.

Graph.**AddActionToContextMenu**(*Action*)
Adds an action to the context menu for the function list.

Graph.**LoadDfmFile**(*FileName*)
Load a DFM text file given by *FileName* usually created by Embarcadero Delphi or C++ Builder. A TForm object created from the file is returned.

Graph.**LoadDefault**()
Replaces the current coordinate system with the default settings. This is basically the same as selecting File → New in the menu.

`Graph.`**BeginMultiUndo**()
Used to group several related changes in the undo stack, so they can be undone as one thing. Call **EndMultiUndo()** to end the grouping.

`Graph.`**EndMultiUndo**()
Used to end undo grouping started with **BeginMultiUndo()**.

`Graph.`**OnNew** , `Graph.`**OnLoad**, `Graph.`**OnSelect**, `Graph.`**OnClose**, `Graph.`**OnEdit**, `Graph.`**OnAnimate**, `Graph.`**OnDelete**, `Graph.`**OnAxesChanged**, `Graph.`**OnZoom**, `Graph.`**OnOptionsChanged**, `Graph.`**OnCustomFunctionsChanged**, `Graph.`**OnNewElem**, `Graph.`**OnChanged**, `Graph.`**OnMoved**

List	Signature	Description
OnNew	Function()	Called after a new coordinate system is created.
OnLoad	Function()	Called after a coordinate system is loaded from a file.
OnSelect	Function(*`Elem`*)	Called when an element in the function list has been selected with the new element in *`Elem`*.
OnClose	Function()	Called when Graph is shutting down.
OnEdit	Function(*`Elem`*)	Called when the user wants to edit an element. The element to edit is given in *`Elem`*. If the function handles the edit it should return True to prevent Graph from handling it.
OnAnimate	Function(*`Data, Var, Value`*)	When an animation is created this is called every time a frame is created. *`Data`* is the temporary data used for creating the animation. *`Var`* is the constant changed in every frame and *`Value`* is the new value of the constant.
OnDelete	Function(*`Elem`*)	Called when an element is about to be deleted.
OnAxesChanged	Function()	Called when the axes settings have been changed.
OnZoom	Function()	Called when the user has zoomed in out out.
OnOptionsChanged	Function()	Called when the user has made changes in the *Options* dialog.
OnCustomFunctions	Function()	Called when the user has made changes to custom functions or constants.
OnNewElem	Function(*`Elem`*)	Called when a new element has been created.
OnChanged	Function(*`Elem`*)	Called when an element has been changed by the user.
OnMoved	Function(*`Elem`*)	Called when the user has moved an element in the function list.

`Graph.`**Eval**(*`Expression`* [, *`Trigonometry`*])
Evaluates *`Expression`*, which is a string with an expression like "sin(0.3)+3^2.5". The expression is evaluated using real numbers only and the result is returned as a floating point number. *`Trigonometry`* can be `Graph.Radian` or `Graph.Degree`. If *`Trigonometry`* is not specified, the value in `Graph.Axes.Trigonometry` will be used. The function will raise the exception `Graph.`**EFuncError** if an error occurs.

`Graph.`**EvalComplex**(*`Expression`* [, *`Trigonometry`*])
The same as **Eval**() except that this function evaluates using complex numbers, and a complex number is returned.

`Graph.`**SaveAsImage**(*`FileName`* [, *`FileType, Width, Height`*])
Saves the current coordinate system as an image file, where *`FileName`* specifies the file name. If given *`FileType`* specifies the file type, which can be Enhanced Metafile (1), Scalable Vector Graphics (2), Bitmap (3), PNG (4), JPEG (5) or PDF (6). If *`FileType`* is not specified, the format will be guessed

from the extension in the file name. *`Width`* and *`Height`* specifies the resolution of the image file. If they are left out, the same resolution as shown on the screen will be used.

`Graph`.**Selected**
This is the currently selected item in the function list. Don't try to change this. Instead you can change `Graph`.**FunctionList**.**Selected**.

`Graph`.**Constants**
This is the interface to the *Custom functions/constants* dialog. The index is the name of the constant or function. The value is a tuple where the first element is either a numeric value or a text defining the function or constant. The rest of the elements in the tuple are the parameters to the function.
A constant does not have any parameters. For example a constant R=8.314472 can be created as `Graph.Constants["R"] = (8.314472,)` A custom function as sinc(x)=sin(x)/x can be created as `Graph.Constants["sinc"] = ("sin(x)/x", "x")`

`Graph`.**CustomFunctions**
This can be used to create custom functions implemented in Python. **CustomFunctions** is a dictionary where the key is the function name and the value is the Python function. For example the function sinc(x)=sin(x)/x can be implemented in Python like this: `Graph.CustomFunctions["sinc"] = lambda x: math.sin(x)/x`

`Graph`.**FunctionList**
This is a list of `Graph.TGraphElem` elements which are plotted by Graph. It is the same list that is shown in the GUI. To plot an element, you just add it to the list. To make sure the undo functionality works, you should not change an element already in the list. Instead you should replace the old element in the list with a new element. `Graph`.**FunctionList**.**Selected** can be used to read and set the selected element in the GUI.

`Graph`.**PluginData**
PluginData is a dictionary like object where a plugin can store data. The data is stored in the .grf file. Graph itself does not use this, it is only for use by plugins Every plugin should use a unique value as key in the **PluginData** dictionary. The value assigned should be a tuple, which may contain anything that can be passed to xmlrpc, e.g. tuples, lists, strings, numbers.

`Graph`.**LoadFromFile**(*`FileName, AddToRecent=True, ShowErrorMessages=True`*)
Loads a grf file from the file specified by FileName. If *`AddToRecent`* is True, the file name is added to the list of recent files in the File menu. If *`ShowErrorMessages`* is True, a dialog with error information is shown if a problem occur, else errors are ignored. The function returns True if the file was loaded without errors, else False is returned.

`Graph`.**SaveToFile**(*`FileName, Remember=True`*)
Saves the current data to a grf file specified by *`FileName`*. If *`Remember`* is True, Graph will remember the file name and use it when saving with File → Save.

`Graph`.**Import**(*`FileName`*)
Imports the content of a grf file specified by FileName into the current coordinate system, excluding axes settings. An exception is thrown on errors.

`Graph`.**ImportPointSeries**(*`FileName, Separator=0`*)
Imports the content of a text file as one or more point series. *`Separator`* indicates the separator used. It is usually ',', ' ', ';' or '\t'. If *`Separator`* is 0, the actual separator is auto detected from the content of the file. An exception is thrown on errors.

class `Graph.TProperty`

`TProperty`.**RoundTo**
Indicates the number of decimals used when showing numbers.

`TProperty`.**SavePos**
If `True` the window size and position is saved at program termination.

`TProperty`.**ComplexFormat**
This indicates the format used when showing complex numbers. It can be `Graph.cfReal`, `Graph.cfRectangular` or `Graph.cfPolar`.

`TProperty`.**CheckForUpdate**
When this is `True` Graph will check for updates when started.

`TProperty`.**DefaultFunction**, `TProperty`.**DefaultPoint**, `TProperty`.**DefaultPointLine**, `TProperty`.**DefaultShade**, `TProperty`.**DefaultTrendline**, `TProperty`.**DefaultRelation**, `TProperty`.**DefaultTangent**, `TProperty`.**DefaultDif**
A tuple with default settings for functions, point markers, point lines, shadings, trendlines, relations, tangents and derivatives with style, color and size.

`TProperty`.**DefaultPointLabelFont**, `TProperty`.**DefaultLabelFont**
VCL object of type `TFont` with default font settings for point series labels and text labels.

`TProperty`.**ShowTipsAtStartup**
Indicates if Tip of the Day should be shown at startup.

`TProperty`.**Language**
This indicates the currently selected GUI language.

`TProperty`.**FontScale**
This specifies the scaling in percent of the user interface, including forms and fonts. The default is 100.

`TProperty`.**CustomDecimalSeparator**
This specifies if another decimal separator than the one from the locale settings should be used when data is imported and exported.

`TProperty`.**DecimalSeparator**
This is the decimal separator used for importing and exporting of data when **CustomDecimalSeparator** is True.

class `Graph.TAxis`

The `TAxis` class represents settings for one of the axes. You cannot create new instances of this class but should access it through `Graph.Axes.xAxis` or `Graph.Axes.yAxis`.

`TAxis`.**Min**
The minimum value of the axis.

`TAxis`.**Max**
The maximum value of the axis.

`TAxis`.**LogScl**
Specifies if the axes is scaled logarithmic.

`TAxis`.**MultipleOfPi**
Specifies if numbers, tick marks and grid lines should be based on numbers that are a multiple of # instead of integers.

`TAxis`.**ShowLabel**
If True the text specified in **Label** will be shown next to the axis.

`TAxis`.**ShowNumbers**
If True numbers will be shown along the axis at the position of the tick marks.

`TAxis`.**ShowTicks**
Id True, tick marks are shown along the axis. The distance between the tick marks are given in **TickUnit**. If **LogScl** is True, the grid lines are shown at $\mathbf{TickUnit}^N$ where N is an integer.

`TAxis`.**ShowGrid**
If True, grid lines are shown perpendicular to the axis. The distance between the grid lines are given in **GridUnit**. If **LogScl** is True, the grid lines are shown at **GridUnit**^{N} where N is an integer, with minor grid lines between the major grid lines.

`TAxis`.**AutoTick**
If True the value in **TickUnit** will be automatically calculated every time there is a change.

`TAxis`.**AutoGrid**
If True the value in **GridUnit** will be automatically calculated every time there is a change.

`TAxis`.**Label**
This specifies a text string that will be shown next to the axis when **ShowLabel** is True.

`TAxis`.**AxisCross**
A floating point value indicating the axis crosses the other axis.

`TAxis`.**TickUnit**
The distance between the tick marks on the axis. If **AutoTick** is True, this value will be automatically calculated every time the image is updated.

`TAxis`.**GridUnit**
The distance between grid lines on the axis. If **AutoGrid** is True, this value will be automatically calculated every time the image is updated.

`TAxis`.**Visible**
When True the axis is shown in the image.

`TAxis`.**ShowPositiveArrow**
When True an arrow is shown in the positive end of the axis.

`TAxis`.**ShowNegativeArrow**
When True an arrow is shown in the negative end of the axis.

`TAxis`.**NumberPlacement**
This specifies where the numbers along the axis is shown relative to the tick marks. For the x-axis, the numbers can be centered below (`Graph.npCenter`) or shown below a little to the left of the tick marks (`Graph.npBefore`). For the y-axis, the numbers can be centered to the left of the tick marks (`Graph.npCenter`) or shown below to the left (`Graph.npBefore`).

class `Graph.TAxes`

`TAxes`.**xAxis**
Structure of class `Graph.TAxis` with settings for the x-axis.

`TAxes`.**yAxis**
Structure of class `Graph.TAxis` with settings for the y-axis.

`TAxes`.**AxesColor**
This specifies the color of the axes. See the VCL documentation [http://docwiki.embarcadero.com/VCL/en/Graphics.TColor].

`TAxes`.**GridColor**
This specifies the color of the grid lines. See the VCL documentation [http://docwiki.embarcadero.com/VCL/en/Graphics.TColor].

`TAxes`.**BackgroundColor**
This specifies the background color of the image. See the VCL documentation [http://docwiki.embarcadero.com/VCL/en/Graphics.TColor].

`TAxes`.**NumberFont**
This specifies the font used to write the numbers along the axes. See the VCL documentation [http://docwiki.embarcadero.com/VCL/en/Graphics.TFont].

`TAxes`.**LabelFont**
This specifies the font used to write the labels shown at the end of the axes. See the VCL documentation [http://docwiki.embarcadero.com/VCL/en/Graphics.TFont].

`TAxes`.**LegendFont**
This specifies the font used to write the text in the legend. See the VCL documentation [http://docwiki.embarcadero.com/VCL/en/Graphics.TFont].

`TAxes`.**TitleFont**
This specifies the font used to write the title above the coordinate system. See the VCL documentation [http://docwiki.embarcadero.com/VCL/en/Graphics.TFont].

`TAxes`.**Title**
This specifies the title shown above the coordinate system. Set this to an empty string to not show a title.

`TAxes`.**ShowLegend**
When True the legend will be shown in the coordinate system.

`TAxes`.**Trigonometry**
This specifies if trigonometric functions calculate in radians or degrees. Valid values are `Graph.Radian` and `Graph.Degree`

`TAxes`.**AxesStyle**
Indicates how the axes are show. Valid values are `Graph.asNone`, `Graph.asCrossed` and `Graph.asBoxed`.

`TAxes`.**LegendPlacement**
This specifies where the legend is placed in the image. Valid values are `Graph.lpCustom`, `Graph.lpTopRight`, `Graph.lpBottomRight`, `Graph.lpTopLeft` and `Graph.lpBottomLeft`.

`TAxes`.**LegendPos**
This is a tuple with the (x,y) coordinates of the top left corner of the legend. It is only used when **LegendPlacement** is `Graph.lpCustom`.

`TAxes`.**CalcComplex**
When True Graph will use complex numbers when plotting functions, which will slow down the plotting. It does not affect other evaluations than the plotting of functions.

`TAxes`.**GridStyle**
Indicates how the grid is shown. Valid values are `Graph.gsLines`, the default which is shown as lines, and `Graph.gsDots`, which shows a dot where the grid cross.

class `Graph.TGuiFormatSettings`

`TGuiFormatSettings`.**CartesianPointFormat**
This string specifies the format used to show cartesian coordinates for point series. %1% in the string indicates the x-coordinate and %2% indicates the y-coordinate.

`TGuiFormatSettings`.**DegreePointFormat**
This string specifies the format used to show polar coordinates in degrees for point series. %1% in the string indicates the angular coordinate and %2% indicates the radial coordinate.

`TGuiFormatSettings`.**RadianPointFormat**
This string specifies the format used to show polar coordinates in radians for point series. %1% in the string indicates the angular coordinate and %2% indicates the radial coordinate.

class Graph.TPlotSettings

TPlotSettings.**AxisWidth**
Width of the axes on the screen in pixel.

TPlotSettings.**GridWidth**
Width of grid lines on the screen in pixels.

TPlotSettings.**xNumberDist**, TPlotSettings.**yNumberDist**
Specifies the distance in pixels of the numbers on the screen from the x- and y-axis.

TPlotSettings.**TickWidth**, TPlotSettings.**TickLength**
Specifies the width and length of the tick marks in pixels on the screen.

class Graph.TGuiSettings

TGuiSettings.**MajorZoomIn**, TGuiSettings.**MinorZoomIn**, TGuiSettings.**MajorZoomOut**, TGuiSettings.**MinorZoomOut**
When zooming the zoom rate specifies how large a unit will be after on both axes compared to before the zoom. This means that a zoom rate of 1 will make no change. A zoom rate of 2 will double the size on both axes while a zoom rate of 0.5 will half the size of both axes. **MajorZoomIn** and **MajorZoomOut** are used when zooming in and out normally, while **MinorZoomIn** and **MinorZoomOut** are used when zooming in and out when Shift is held down.

TGuiSettings.**MajorStepSize**, TGuiSettings.**MinorStepSize**
When stepping up, down or sideways, the step size indicates how much is stepped as fraction of the image size, i.e. 0.1 means that each step is 10% of the image size. **MajorStepSize** is used when stepping normally, while **MinorStepSize** is used when stepping with Shift held down.

TGuiSettings.**MouseZoomIn**, TGuiSettings.**MouseZoomOut**
When zooming in or out with the mouse scrolling wheel or similar, the zoom rate specifies how large a unit will be after on both axes compared to before the zoom. This means that a zoom rate of 1 will make no change. A zoom rate of 2 will double the size on both axes while a zoom rate of 0.5 will half the size of both axes.

Graph elements

class Graph.TGraphElem

TGraphElem.**Visible**
This specifies if the element is shown in the graphing area.

TGraphElem.**ShowInLegend**
Indicates if the element is shown in the legend.

TGraphElem.**LegendText**
This is the text string shown in the legend.

TGraphElem.**Parent**
This is a read only attribute indicating the parent element when the object is in the function list.

TGraphElem.**PluginData**
PluginData is a dictionary like object where a plugin can store data local to an element. The data is stored in the .grf file. Graph itself does not use this, it is only for use by plugins Every plugin should use a unique value as key in the **PluginData** dictionary. The value assigned should be a tuple, which may contain anything that can be passed to xmlrpc, e.g. tuples, lists, strings, numbers.

TGraphElem.**ChildList**
This is a list of child elements as shown in the function list.

`TGraphElem`.**Clone**()
This creates a new copy of the object.

class `Graph.TBaseFuncType`

`TBaseFuncType`.**sList**
This is a list of data points used to plot the function. Each entry is a tuple with 3 elements, the independent variable, x-coordinate and y-coordinate.

`TBaseFuncType`.**Points**
This is a list of pixel coordinates used to plot the function.

`TBaseFuncType`.**PointNum**
This is a list of values indicating the number of continues points in each segment.

`TBaseFuncType`.**Color**
This specifies the color of the function.

`TBaseFuncType`.**Size**
This specifies the width of the function in pixels on the screen.

`TBaseFuncType`.**Style**
This specifies the line style of the function.

`TBaseFuncType`.**From**, `TBaseFuncType`.**To**
From and **To** specifies the the range of the function. Standard functions may use `float("-inf")` and `float("inf")` for an infinite range.

`TBaseFuncType`.**StartPointStyle**, `TBaseFuncType`.**EndPointStyle**
These indicates the style of the end points. Use 0 if you don't want an endpoint.

`TBaseFuncType`.**DrawType**
This indicates how the functions is plotted. Valid values are `Graph.dtAuto`, `Graph.dtDots` and `Graph.dtLines`.

`TBaseFuncType`.**MakeDifFunc**()
This method will create and return the first derivative of the function.

`TBaseFuncType`.**Eval**(*`t`*)
This evaluates the function at the specified independent variable *`t`*. The result is a tuple if the (x,y) coordinate pair.

`TBaseFuncType`.**CalcArea**(*`From, To`*)
Calculates the signed area over the range between *`From`* and *`To`* by numeric integration. The calculated area is between the function and the x-axis for standard and parametric functions, while it is the area between the function and the center for polar functions.

class `Graph.TStdFunc`

class `Graph`.**TStdFunc**(*`Str`*)
Creates a standard function from the expression in *`Str`* with "x" as the independent variable.

`TStdFunc`.**Text**
This is the same string as was passed to the constructor.

class `Graph.TParFunc`

class `Graph`.**TParFunc**(*`xStr, yStr`*)
Creates a parametric function from the expressions in *`xStr`* and *`yStr`* where the independent variable is "t".

TParFunc.**xText** TParFunc.**yText**
These are the same strings as was passed to the constructor.

class Graph.TPolFunc

class Graph.**TPolFunc**(*Str*)
Creates a polar function from the expression in *Str* where the independent variable is "t".

TParFunc.**Text**
This is the same string as was passed to the constructor.

class Graph.TTan

class Graph.**TTan**()
Creates a new tangent or normal. It must be attached to a function to be plotted.

TTan.**Valid**
This property is True if the tangent is valid, i.e. the function has a first derivative at **t**.

TTan.**t**
This is the value where the tangent/normal interacts with the its parent function.

TTan.**TangentType**
This indicates if the object is a tangent or normal. Valid values are Graph.ttTangent and Graph.ttNormal.

class Graph.TPointSeries

class Graph.**TPointSeries**()
Creates a new point series.

TPointSeries.**FillColor**, TPointSeries.**FrameColor**, TPointSeries.**Size**, TPointSeries.**Style**
These properties sets the color filling the markers, the border color of the markers and the style of the markers.

TPointSeries.**LineColor**, TPointSeries.**LineSize**, TPointSeries.**LineStyle**
These properties sets the color, size and style of the line between the markers.

TPointSeries.**xErrorBarType**, TPointSeries.**yErrorBarType**
These specifies the type of vertical and horizontal error bars. Valid values are Graph.ebtNone for no error bars, Graph.ebtFixed for fixed size error bars, Graph.ebtRelative for error bars being a percentage of the coordinate value, and Graph.ebtCustom for a custom specified error bar for every point.

TPointSeries.**xErrorValues**, TPointSeries.**yErrorValues**
These contains the value used for the error bars when **xErrorBarType** or **yErrorBarType** is Graph.ebtFixed or Graph.ebtRelative.

TPointSeries.**Interpolation**
This specifies the interpolation algorithm used for drawing lines between the markers. Valid values are Graph.iaLinear, Graph.iaCubicSpline, Graph.iaHalfCosine and Graph.iaCubicSpline2.

TPointSeries.**ShowLabels**
Set this to True to show coordinate labels next to the markers.

TPointSeries.**Font**
This sets the font used when drawing the coordinate labels.

`TPointSeries`.**LabelPosition**
This specifies where the coordinate labels are placed relative to the markers. Valid values are `Graph.lpAbove`, `Graph.lpBelow`, `Graph.lpLeft`, `Graph.lpRight`, `Graph.lpAboveLeft`, `Graph.lpAboveRight`, `Graph.lpBelowLeft` and `Graph.lpBelowRight`.

`TPointSeries`.**PointType**
This specifies if the coordinates in **Points** are in polar or cartesian coordinates. Valid values are `Graph.ptCartesian` and `Graph.ptPolar`.

`TPointSeries`.**Points**
This is a list of tuples with (x,y) coordinates for the points.

`TPointSeries`.**PointData**
This is a list with a tuple with 4 elements for each point. The tuple contains the first coordinate, second coordinate, x-error and y-error, all in text form. The first and second coordinates are (x,y) coordinates if **PointType** is `Graph.ptCartesian`, and (θ,r) coordinates if **PointType** is `Graph.ptpolar`.

class `Graph.TTextLabel`

class `Graph`.**TTextLabel**()
Creates a new text label object.

`TTextLabel`.**Text**
This is the text in the label in Rich Text format.

`TTextLabel`.**BackgroundColor**
This is the background color of the label. Use 0x1fffffff for transparent.

`TTextLabel`.**Placement**
This specifies the placement of the label. Valid values are `Graph.lpUserTopLeft`, `Graph.lpAboveX`, `Graph.lpBelowX`, `Graph.lpLeftOfY`, `Graph.lpRightOfY`, `Graph.lpUserTopRight`, `Graph.lpUserBottomLeft` and `Graph.lpUserBottomRight`.

`TTextLabel`.**Rotation**
This specifies the rotation of the label in degrees.

`TTextLabel`.**xPos**, `TTextLabel`.**yPos**
`xPos` and `yPos` indicates the (x,y) coordinate of the label when **Placement** is `Graph.lpUserTopLeft`, `Graph.lpUserTopRight`, `Graph.lpUserBottomLeft` or `Graph.lpUserBottomRight`.

class `Graph.TShading`

class `Graph`.**TShading**()
Creates a new shading object. It must be attached to a function to be plotted.

`TShading`.**ShadeStyle**
This specifies the type of shading. Valid values are `Graph.ssAbove`, `Graph.ssBelow`, `Graph.ssXAxis`, `Graph.ssYAxis`, `Graph.ssBetween` and `Graph.ssInside`.

`TShading`.**BrushStyle**
This is a `vcl.TBrushStyle` that specifies the brush style used to plot the shading.

`TShading`.**Color**
This specifies the color of the shading.

`TShading`.**Func2**
This must specify the second function when **BrushStyle** is `Graph.ssBetween`.

`TShading`.**sMin**, `TShading`.**sMax**
This is the start and end values on the function for the shading.

`TShading`.**sMin2**, `TShading`.**sMax2**
When **ShadeStyle** is `Graph.ssBetween` this is the start and end value on **Func2** for the shading.

`TShading`.**ExtendMinToIntercept**, `TShading`.**ExtendMaxToIntercept**,
`TShading`.**ExtendMin2ToIntercept**, `TShading`.**ExtendMax2ToIntercept**
When True, **sMin** and **sMin2** are decreased and **sMax** and **sMax2** are increased until the function is crossing the axis, the edge of the graphing area, itself or another graph depending of the value in **ShadeStyle**.

`TShading`.**MarkBorder**
When True a line will be drawn around the shading.

class `Graph.TRelation`

class `Graph`.**TRelation**(*`Str, [ConstraintStr]`*)
Creates a new relation object with the relation specified in *`Str`* and an optional constraint specified in *`ConstraintStr`*.

`TRelation`.**BrushStyle**
This is a `vcl.TBrushStyle` that specifies the brush style used to plot inequalities.

`TRelation`.**Color**
This specifies the color of the relation.

`TRelation`.**RelationType**
This read only attribute indicates if the relation is an equation (`Graph.rtEquation`) or inequation (`Graph.rtInequality`).

`TRelation`.**Size**
This is the width of the plot of the inequation or the width of the borderline around the inequality. **Size** may be 0 if no borderline is wanted.

`TRelation`.**Text**
This is the text of the equation or inequality.

`TRelation`.**Constraints**
This is the text of the constraints.

`TRelation`.**Eval**(*`x, y`*)
Evaluate the relation at the given x- and y-coordinates and return the result.

`vcl` module

The `vcl` module is an interface to the Embarcadero Visual Component Library (VCL) used by Delphi and C++ Builder. Documentation for the VCL can be found online at http://docwiki.embarcadero.com/VCL/en/Main_Page.

Python is case sensitive, but as the VCL is written in Delphi, which is not case sensitive, much of the `vcl` module is not case sensitive. This means it doesn't care if you write `TForm` or `tform`. However it is recommended to use the same case as shown in the documentation as the case sensitivity may change in the future.

Classes

VCL classes as `TForm` and `TButton` are found in the `vcl` module. You create a new VCL object by instantiating a VCL class in the same way you always create Python objects. All positional arguments are passed on to the constructor for the VCL class. VCL classes can have several constructors. The first constructor that matches the passed arguments will be used, e.g. `vcl.TForm(None)` will use the constructor that takes an owner component as argument. Keyword arguments will be assigned to the properties of the object after the object has been constructed, e.g. `Form = vcl.TForm(None, Caption="Test dialog")` is the same as `Form = vcl.TForm(None); Form.Caption = "Test dialog"`

Functions

Global functions as `TextToShortCut` can be found in the `vcl` module and are called like `vcl.TextToShortCut("Ctrl+A")`.

Objects

The `vcl` module contains some global objects as `Application`, `Mouse`, `Clipboard` and `Screen`. Other objects can be directly constructed or returned from a function. `None` in Python can be used to pass a NULL pointer to a VCL function instead of an object.

The Python object is a proxy object to the actual VCL object. Per default VCL objects created directly in Python are owned by the proxy object. The VCL object will therefore be destroyed when the proxy object in Python is destroyed. Objects returned from a function or accessed through properties are not owned and will continue to exist after the proxy object has been destroyed. The proxy object has an **_owned** property that specify if the proxy object owns the underlying VCL object.

Object methods and properties are accessed as you normally would in Python so you can use `Form.Show()` and `Form.Caption = "Test"`. If a method is overloaded, the first one that matches the parameters will be called.

Events

Events can either be global functions or methods in an object, which can be assigned like properties. The event handler must be able to take the expected arguments which will be passed.

Sometimes an event takes a reference as argument. In that case the actual argument is an object with a property called **Value** that can be used to access the actual referenced value.

Types

Most types can be used directly, e.g. strings, numbers and booleans. VCL sets are converted to Python strings, e.g. `Form.Font.Style = "fsBold,fsItalic"` will make the font bold and italic. Enumerations are always returned as strings but can be assigned as strings or integers, e.g. `Form.WindowState = "wsMaximized"` and `Form.WindowState = 2` will both maximize the window. Records are converted to tuples with one element for every item in the record. Similarly a function that expects a record must be passed a tuple, e.g. `Form1.ClientToScreen((100,50))`.

VCL example

```
# This script will show a dialog where you can enter a value.
# The event will check that only digits are entered.
# If the OK button is pressed, the entered value will be printed to the console.
import vcl
import string

def HandleKeyPress(Sender, Key):
  if not Key.Value in string.digits:
    Key.Value = '\0'

Form = vcl.TForm(None, Caption="Value dialog", Width=190, Height=110)
Label = vcl.TLabel(None, Parent=Form, Caption="Value:", Top=12, Left=8)
Edit = vcl.TEdit(None, Parent=Form, OnKeyPress=HandleKeyPress, Text="0", Top=8, Left=50)
OkButton = vcl.TButton(None, Parent=Form, Caption="OK", Default=True, ModalResult=1,
 Top=50, Left=8)
CancelButton = vcl.TButton(None, Parent=Form, Caption="Cancel", Cancel=True, ModalResult=2,
 Top=50, Left=100)
if Form.ShowModal() == 1:
  print("Result:", Edit.Text)
```