

The wxGrid Class

Tabla de Contenidos

1. The wxGrid Class
 1. Table Interactions
 2. Changing Size/Shape of Grid/Table
 3. Data Types, Renderers and Editors
 1. Data Type Specifiers
 2. Edit/Render Objects/Controls
 4. Selections and the Cursor
 1. Cursor Manipulation
 2. Selection Set Management
 5. General Display Configuration/Queries
 1. Geometry Queries
 6. Column/Row Sizing
 1. Default Sizes
 2. Auto-sizing of Cells to Contents
 3. Overflow (Multiple-cell display/edit)
 4. Setting Individual Heights/Widths (Exceptions to Defaults)
 5. Interactive Resizing
 6. Autosizing labels
 7. Attribute Objects (Cell Display Properties)
 8. Label (Header) Display/Edit Configuration
 1. Label Queries Passed to Table
 9. Cell Display/Edit Configuration
 1. Generic Cell Display (alignment, fonts, colours)
 2. Defaults and Overall Settings
 3. Cell-Related Queries Passed to Table
 10. DragAndDrop onto Grid
 11. DragAndDrop From Grid
 12. Capturing Cell Edits when focus changes

Table Interactions

- `GetTable()` -- Return the current table object (depending on your version of wxPython, this may not be the Python object instance you passed to `SetTable`. For this reason, you will likely want to store a weak reference to your table in your Grid class and provide methods such as those in the example below.
- `SetTable(wxGridTableBase* table, bool takeOwnership = FALSE, wxGrid::wxGridSelectionMc`
-- Set the current table object. You should pass an instance of your `wxPyGridTableBase` sub-class to this method. However, see the note on `GetTable` (above), and the workaround implementation below.

This workaround code stores a Python weak reference to your `wxPyGridTableBase` instance, overriding the `GetTable` and `SetTable` methods of the `wxGrid`.

```
import weakref
class _PropertyGrid( wxGrid ):
    def SetTable( self, object, *attributes ):
        self.tableRef = weakref.ref( object )
        return wxGrid.SetTable( self, object, *attributes )
    def GetTable( self ):
        return self.tableRef()
```

Note: you can only call `SetTable` once for any given `wxGrid`. Because of this, it is generally necessary to have your `wxPyGridTableBase` class alter the grid's size and shape to reflect any changes in your table's dimensions. See **wxGrid Size/Shape Management** below for details.

Changing Size/Shape of Grid/Table

The methods described in this section control the grid's overall data structures (for instance the number of grid rows/columns). When using a data table (recommended), is often necessary to alter the grid's size to reflect changes in the data table. Here is a recipe for such changes:

```
class BasePropertyTable( wxPyGridTableBase):
    def ResetView(self):
        """Trim/extend the control's rows and update all values"""
        self.getGrid().BeginBatch()
```

```

        for current, new, delmsg, addmsg in [
            (self.currentRows, self.GetNumberRows(),
             wxGRIDTABLE_NOTIFY_ROWS_DELETED, wxGRIDTABLE_NOTIFY_ROWS_APPENDED),
            (self.currentColumns, self.GetNumberCols(),
             wxGRIDTABLE_NOTIFY_COLS_DELETED, wxGRIDTABLE_NOTIFY_COLS_APPENDED),
        ]:
            if new < current:
                msg = wxGridTableMessage(
                    self,
                    delmsg,
                    new,      # position
                    current-new,
                )
                self.getGrid().ProcessTableMessage(msg)
            elif new > current:
                msg = wxGridTableMessage(
                    self,
                    addmsg,
                    new-current
                )
                self.getGrid().ProcessTableMessage(msg)
        self.UpdateValues()
        self.getGrid().EndBatch()

        # The scroll bars aren't resized (at least on windows)
        # Jiggling the size of the window rescales the scrollbars
        h,w = grid.GetSize()
        grid.SetSize((h+1, w))
        grid.SetSize((h, w))
        grid.ForceRefresh()

    def UpdateValues( self ):
        """Update all displayed values"""
        msg = wxGridTableMessage(self, wxGRIDTABLE_REQUEST_VIEW_GET_VALUES)
        self.getGrid().ProcessTableMessage(msg)







```

Note: this code assumes that you record `self.currentRows` and `self.currentCols` on your initial table set up to allow for calculating the new size relative to the old size. It should be possible to use `wxGrid::GetNumberRows()` if you prefer.

Also note: the code uses `getGrid()` to return a pointer to the `wxGrid`, depending on your version of `wxPython`, you may be able to use `wxPyGridTableBase::GetGrid()` instead. See: Table Interactions (above) for details on this work-around.

Last note: At least on windows, the grid's scroll bars are not resized when rows and columns are deleted. Jiggling the size of the grid and forcing a refresh fixing this problem. The fix for this has been added to the demo code above.

Other size/shape management methods:

- `AppendCols(int numCols = 1, bool updateLabels = TRUE)`  Add the given number of columns to the grid
- `AppendRows(int numRows = 1, bool updateLabels = TRUE)`  Add the given number of rows to the grid
- `ClearGrid()` -- Eliminate all rows and columns from the grid (XXX I'm not sure whether this is even applicable to table-based grids)
- `DeleteCols(int pos = 0, int numCols = 1, bool updateLabels = TRUE)`  Delete numCols columns from position pos
- `DeleteRows(int pos = 0, int numRows = 1, bool updateLabels = TRUE)`  Delete numRows rows from position pos
- `GetNumberCols()` -- Retrieve the number of columns in the table
- `GetNumberRows()` -- Retrieve the number of rows in the table
- `InsertCols(int pos = 0, int numCols = 1, bool updateLabels = TRUE)`  Insert a number of columns into the grid (and associated table)
- `InsertRows(int pos = 0, int numRows = 1, bool updateLabels = TRUE)`  Insert a number of rows into the grid (and associated table)

Data Types, Renderers and Editors

Your `wxPyGridTableBase`'s `GetTypeName` returns data-type specifiers for a given row,col pair, the grid uses that to look up the appropriate renderer and editor for the cell.

- RegisterDataType(const wxString& typeName, wxGridCellRenderer* renderer, wxGridCellEdit -- Note (as of 2.3.3pre4): calling this twice for the same data type will result in your new editors not having "Create" called to create their control. You will get a message that "The wxGridCellEditor must be Created first!" if you happen to do this.

Data Type Specifiers

Usage patterns? I automate their generation, not sure if most people do that.

Edit/Render Objects/Controls



- CanEnableCellControl()
- EnableCellEditControl(bool enable = TRUE) / DisableCellEditControl()
- EnableEditing(bool edit) --
- GetDefaultEditor() --
- GetDefaultRenderer() --
- HideCellEditControl() --
- IsCellEditControlEnabled() --
- IsEditable() --
- IsReadOnly(int row, int col) --
- SaveEditControlValue() -- XXX Note: should be mentioned that you want to call this before closing the grid's parent' window!
- SetCellEditor(int row, int col, wxGridCellEditor* editor) -- XXX Hmm, is this usable with table-based grids?
- SetCellRenderer(int row, int col, wxGridCellRenderer* renderer) -- XXX Hmm, is this usable with table-based grids?
- SetDefaultEditor(wxGridCellEditor* editor) --
- SetDefaultRenderer(wxGridCellRenderer* renderer) --
- SetReadOnly(int row, int col, bool isReadOnly = TRUE) -- XXX Not really sure where this goes...
- ShowCellEditControl() --
- GetDefaultEditorForCell(int row, int col) --
- GetDefaultRendererForCell(int row, int col) --
- GetDefaultEditorForType(const wxString& typeName) --
- GetDefaultRendererForType(const wxString& typeName) --
- GetCellEditor(int row, int col) -- returning a wxPyGridCellEditor registered with the control using the grid's RegisterDataType, selected based on the data type specified by your wxPyGridTableBase's GetTypeName method
- GetCellRenderer(int row, int col) -- returning a wxPyGridCellRenderer registered with the control using the grid's RegisterDataType, selected based on the data type specified by your wxPyGridTableBase's GetTypeName method

Selections and the Cursor

The grid has both a selection set (which may include disjoint ranges) and a single-cell focus cursor. The selection set can contain arbitrary blocks of cells, while the cursor can only point to a single cell.

Cursor Manipulation

The API for interacting with the focus cursor is fairly simple.

- GetGridCursorCol() -- Retrieve the current cursor position
- GetGridCursorRow() -- Retrieve the current cursor position
- MoveCursorDown/Left/Right/Up(bool expandSelection)  move a single cell in direction
- MoveCursorDown/Left/Right/UpBlock(bool expandSelection)  move a single cell, or multiple cells if the intervening cells are empty.
- MovePageDown() -- Move cursor down such that the last visible row becomes the first
- MovePageUp() -- Move cursor up such that the first visible row becomes the last
- SetGridCursor(int row, int col) -- set the cursor to the giving coordinate, also makes sure that the coordinate is visible on screen
- MakeCellVisible(int row, int col) -- forces the particular cell to be visible, effectively works to

scroll the grid to be given cell

Selection Set Management

The grid class does not provide a simple API for retrieving the current selection set. Instead, it is necessary to catch two different events and store their values to determine the current selection set. The following sample code illustrates how this is done. The `_OnSelectedRange` method is mapped using

`EVT_GRID_RANGE_SELECT(self.grid, self._OnSelectedRange)`, while the `_OnSelectedCell` method is mapped using `EVT_GRID_SELECT_CELL(self.grid, self._OnSelectedCell)`. Because these are both command events, they can be mapped from the grid's parent window.



The following code sample shows tracking of currently selected rows in a table:

```
def _OnSelectedRange( self, event ) :
    """Internal update to the selection tracking list"""
    if event.Selecting():
        # adding to the list...
        for index in range( event.GetTopRow(), event.GetBottomRow()+1 ):
            if index not in self.currentSelection:
                self.currentSelection.append( index )
    else:
        # removal from list
        for index in range( event.GetTopRow(), event.GetBottomRow()+1 ):
            while index in self.currentSelection:
                self.currentSelection.remove( index )
    self.ConfigureForSelection()
    event.Skip()
def _OnSelectedCell( self, event ) :
    """Internal update to the selection tracking list"""
    self.currentSelection = [ event.GetRow() ]
    self.ConfigureForSelection()
    event.Skip()
```

Note: the selection of a single cell will cause a range selection event with `Selecting() == false` and the entire grid as the selection set to be sent before the single-cell selection occurs.

Note: the `Selecting() == false` ranges will be sent if the user is control-clicking to deselect particular cells within the current selection set. Because of this, the "removal from list" sub-block above becomes somewhat complex when you need to store cell-selections, not just row selections. The wxoo project has a partial implementation of a grid-selection tracking class [wx.gridselectionset.py](#), but it merely forces the whole selection set to be represented as individual cells when a deselection is called (which can be spectacularly inefficient). Improved code welcome.

Other selection-set-management methods:

- `ClearSelection()` -- Clear all currently selected cell ranges (sends a range-selection event)
- `IsInSelection(int row, int col)`  Query whether a particular cell is currently selected
- `SelectAll()` -- Strangely enough, select all cells in the table
- `SelectBlock(int topRow, int leftCol, int bottomRow, int rightCol, bool addToSelected = False)` -- Select a particular block of cells as an inclusive range. If `addToSelected` is true, add to the current selection set
- `SelectCol(int col, bool addToSelected = FALSE)` -- Select all cells in a column
- `SelectRow(int row, bool addToSelected = FALSE)`  Select all cells in a row
- `IsSelection()` -- True if there is a current selection
- `SetSelectionMode(wxGrid::wxGridSelectionMode selmode)` -- Determine whether cells, rows or columns are selected

General Display Configuration/Queries

- `EnableGridLines(bool enable = TRUE)` -- Enable drawing of the grid cell lines
- `ForceRefresh()` -- Force an Refresh of the grid's window
- `SetMargins(int extraWidth, int extraHeight)` -- Set padding space around the grid cells within the grid window

Geometry Queries

Feedback about the geometry displayed in the grid window.

- `BlockToDeviceRect(const wxGridCellCoords & topLeft, const wxGridCellCoords & bottomRight)`
- `SelectionToDeviceRect()`

- `CellToRect(int row, int col)`
- `IsVisible(int row, int col, bool wholeCellVisible = TRUE)`
- `XToCol(int x)`
- `XToEdgeOfCol(int x) --`
- `YToEdgeOfRow(int y) --`
- `YToRow(int y) --`

Column/Row Sizing

Default Sizes

The following methods allow for specifying default height and/or width values. These values are used when there is no explicitly set value for column width/height, (and where no auto-sizing is occurring (see below)).

- `SetDefaultColSize(int width, bool resizeExistingCols = FALSE) --`
- `SetDefaultRowSize(int height, bool resizeExistingRows = FALSE) --`
- `GetDefaultColSize()`, `GetDefaultRowSize()` -- return an integer representing the default pixel-width of the column headers or pixel-height of the row headers (respectively).

Auto-sizing of Cells to Contents

The following methods set the grid's cells to automatically size themselves to their content's preferred sizes. The preferred sizes are the values returned by the `wxPyGridCellRenderer`'s `GetBestSize` method. XXX are these sizes stored (potential memory exhaustion), or re-queried for each redisplay?

Note: there is a bug in the current grid implementation that makes it impossible to edit cells which are wider than the grid's displayed client area (i.e. the size of the grid window minus the labels). For this reason you should ensure that your renderer's return conservative values from `GetBestSize` or avoid using automatic sizing until the bug is fixed.

- `AutoSize()` -- All columns and all rows
- `AutoSizeColumn(int col, bool setAsMin = TRUE)` -- A single column only
- `AutoSizeColumns(bool setAsMin = TRUE)` -- All columns
- `AutoSizeRow(int row, bool setAsMin = TRUE)` -- A single row only
- `AutoSizeRows(bool setAsMin = TRUE)` -- All rows

Overflow (Multiple-cell display/edit)

`wxPython` version 2.4.x has introduced a new (default) functionality which allows renderers and editors displayed in a particular cell to overwrite neighboring cells if their data values would not normally fit in the space allotted to the cell. This will break most renderers written for earlier versions of `wxPython`, so you may want to disable the functionality by default, or disable it for particular cells/rows/columns.

The overflow state is stored in `wxGridCellAttr` objects, with functions available on the grid for setting the entire-grid-default, and for setting the mode for individual cells/ranges. See below for discussion of `wxGridCellAttr` objects and their interactions with the grid.

- `SetDefaultCellOverflow(bool allow)` -- Allows you to set the default "overflow" functionality for the grid, the default value is true, which differs from the operation of earlier `wxPython` versions.
- `GetDefaultCellOverflow()` -- Return a boolean representing the current default overflow state
- `SetCellOverflow(int row, int col, bool allow)` -- Allows you to set the "overflow" functionality for a particular cell in the grid, see notes on `wxGridCellAttr` for why you would normally not do this on table-based grid objects.
- `GetCellOverflow(int row, int col)` -- Return a boolean representing the current overflow state for the given (row,col) coordinate

Setting Individual Heights/Widths (Exceptions to Defaults)

The following methods can explicitly query and or set individual column or row width or height values. Note: the grid will set up an array for storing exceptions to default widths, so extremely large grids may consume considerable memory storing these exceptions. I am unsure whether the return values from grid cell renderers are similarly stored (I would think not).

- `GetColSize(int col) --`
- `GetRowSize(int row) --`
- `SetColSize(int col, int width) --`

- `SetRowSize(int row, int height) --`
- `SetColMinimalWidth(int col, int width) --`
- `SetRowMinimalHeight(int row, int width) --`

Interactive Resizing

The following methods allow for querying and changing the grid's ability to resize columns and/or rows by dragging on the borders between columns or rows. The grid can also (and does by default) allow for changing heights/widths by dragging grid lines in the cell area of the grid display.

- `CanDragColSize()` -- Return a boolean value indicating whether we can currently change column sizes by dragging
- `CanDragRowSize()` -- Return a boolean value indicating whether we can currently change row sizes by dragging
- `CanDragGridSize()` -- Return a boolean value indicating whether we can currently change column and row sizes by dragging on grid lines.
- `EnableDragColSize(bool enable = TRUE)` -- Enable or disable changing column sizes by dragging on dividers between column labels
- `EnableDragRowSize(bool enable = TRUE)` -- Enable or disable changing row sizes by dragging on dividers between row labels
- `EnableDragGridSize(bool enable = TRUE)` -- Enable or disable changing row and column sizes by dragging on grid lines

Autosizing labels

wxPython does not (yet) allow auto sizing of row or column labels, but a little fiddling with device contexts can achieve this. This has been tested on Windows only and your mileage may vary.

```
def AutosizeLabels(self):
    # Common setup.

    devContext = wxScreenDC()
    devContext.SetFont(self.gridCtrl.GetLabelFont())

    # First do row labels.

    maxWidth = 0
    curRow = self.gridCtrl.GetNumberRows() - 1
    while curRow >= 0:
        curWidth = devContext.GetTextExtent("M
%s"%(self.gridCtrl.GetRowLabelValue(curRow)))[0]
        if curWidth > maxWidth:
            maxWidth = curWidth
        curRow = curRow - 1
    self.gridCtrl.SetRowLabelSize(maxWidth)

    # Then column labels.

    maxHeight = 0
    curCol = self.gridCtrl.GetNumberCols() - 1
    while curCol >= 0:
        (w,h,d,l) =
devContext.GetFullTextExtent(self.gridCtrl.GetColLabelValue(curCol))
        curHeight = h + d + l + 4
        if curHeight > maxHeight:
            maxHeight = curHeight
        curCol = curCol - 1
    self.gridCtrl.SetColLabelSize(maxHeight)
```

This code operates on a control with a `wxGrid`-derived instance named `gridCtrl`. I haven't bothered to implement this very cleanly (ie. as a class inherited from `wxGrid`). I leave that as an exercise for the reader :-).

The code basically traverses each row label, calculating its width (including an extraneous "M" for spacing) and storing the widest label. Then it simply sets the row label column to that width.

It also does a similar thing for column label heights, except it calculates the desired height using character height, descender and leading gap and adding a nice 4-pixel buffer for padding.

Attribute Objects (Cell Display Properties)

`wxGridCellAttr` objects provide a mechanism for altering the display of particular rows, columns, or individual cells. In

table-based grids, you determine what attribute object will be used for a particular grid cell by overriding the `wxPyGridTableBase`'s `GetAttr` method to return a grid cell attribute object.

Note: the following methods are not really appropriate for a table-based grid, as this will force the grid to create storage for each column's attribute (which may exhaust memory on large grids). See: `wxPyGridTableBase`'s `GetAttr` method for the appropriate table-based approach. With that said, there might be some use for them where you want to customise particular rows in the grid and for some reason want to keep that code out of your table.

Note: there are methods to set the default values for the default attribute object used by the grid when no other attributes available. See section "Defaults and Overall Settings" below.

- `SetColAttr(int col, wxGridCellAttr* attr)` -- Set an attribute object as default for a particular column in the grid.
- `SetRowAttr(int row, wxGridCellAttr* attr)` -- Set an attribute object as default for a particular row in the grid.
- `CanHaveAttributes()` -- The purpose of this method is unclear, but it appears to be a query to determine whether the grid has allocated storage for attribute objects. I have no idea whether you could override the method, but I assume not (as the grid is not generally overridable).

Label (Header) Display/Edit Configuration

Each column and row in the table has an associated label (columns having their labels at the top of the grid, rows on the left-hand side).

- `GetColLabelAlignment()`, `GetRowLabelAlignment()` -- Returns a two-tuple of (horizontal, vertical) constants representing the alignment of the text labels within the column and row header spaces (respectively)
 - horizontal -- `wxALIGN_LEFT`, `wxALIGN_CENTRE` or `wxALIGN_RIGHT`
 - vertical -- `wxALIGN_TOP`, `wxALIGN_CENTRE` or `wxALIGN_BOTTOM`
- `GetColLabelSize()`, `GetRowLabelSize()` -- Return an integer representing the current pixel-height of the column headers, or the current pixel width of the row headers (respectively)
- `GetDefaultColLabelSize()`, `GetDefaultRowLabelSize()` -- return an integer representing the default pixel-height of the column headers, or default pixel width of the row headers (respectively)
- `GetLabelBackgroundColour()` -- Get the colour used for painting the label background
- `GetLabelFont()` -- Get the font used for the label text
- `GetLabelTextColour()` -- Get the colour used for the label text
- `SetColLabelAlignment(int horiz, int vert)`, `SetRowLabelAlignment(int horiz, int vert)` -- Sets the horizontal and vertical alignment of column or row (respectively) label text.
 - horizontal -- `wxALIGN_LEFT`, `wxALIGN_CENTRE` or `wxALIGN_RIGHT`
 - vertical -- `wxALIGN_TOP`, `wxALIGN_CENTRE` or `wxALIGN_BOTTOM`
- `SetColLabelSize(int height)` -- Set the column label height to a particular pixel value
- `SetRowLabelSize(int width)` -- Set the row label width to a particular pixel value
- `SetLabelBackgroundColour(const wxColour& colour)` -- Set the colour used for painting the label background
- `SetLabelFont(const wxFont& font)` -- Set the font used for the label text
- `SetLabelTextColour(const wxColour& colour)` -- Set the colour used for the label text

Label Queries Passed to Table

- `GetColLabelValue(int col)` -- The value returned by `wxPyGridTableBase`'s `GetColLabelValue`
- `GetRowLabelValue(int row)` -- The value returned by `wxPyGridTableBase`'s `GetRowLabelValue`
- `SetColLabelValue(int col, const wxString& value)` -- `wxPyGridTableBase`'s `SetColLabelValue`
- `SetRowLabelValue(int row, const wxString& value)` -- `wxPyGridTableBase`'s `SetRowLabelValue`

Cell Display/Edit Configuration

Generic Cell Display (alignment, fonts, colours)

- `GetCellAlignment(int row, int col, int* horiz, int* vert)` --
- `GetCellBackgroundColour(int row, int col)` --
- `GetCellFont(int row, int col)` --

- `GetCellTextColour(int row, int col) --`
- `SetCellAlignment(int row, int col, int horiz, int vert) --`
- `SetCellFont(int row, int col, const wxFont& font) --`
- `SetCellTextColour(int row, int col, const wxColour& colour) --`
- `SetColFormatBool(int col) --`
- `SetColFormatNumber(int col) --`
- `SetColFormatFloat(int col, int width = -1, int precision = -1) --`
- `SetColFormatCustom(int col, const wxString& typeName) --`
- `GetSelectionBackground() --`
- `GetSelectionForeground() --`
- `SetSelectionBackground(const wxColour& c) --`
- `SetSelectionForeground(const wxColour& c) --`

See also:

[Overflow \(Multiple-cell display/edit\) \(above\)](#) [Attribute Objects \(above\)](#)

Defaults and Overall Settings

- `GetGridLineColour() --`
- `GetDefaultCellAlignment() --`
- `GetDefaultCellBackgroundColour() --`
- `GetDefaultCellFont() --`
- `GetDefaultCellTextColour() --`
- `GridLinesEnabled() --`
- `SetDefaultCellAlignment(int horiz, int vert) --`
- `SetDefaultCellBackgroundColour(const wxColour& colour) --`
- `SetDefaultCellFont(const wxFont& font) --`
- `SetGridLineColour(const wxColour& colour) --`

Cell-Related Queries Passed to Table

- `GetCellEditor(int row, int col) --` returning a `wxPyGridCellEditor` registered with the control using the grid's `RegisterDataType`, selected based on the data type specified by your `wxPyGridTableBase`'s `GetTypeName` method
- `GetCellRenderer(int row, int col) --` returning a `wxPyGridCellRenderer` registered with the control using the grid's `RegisterDataType`, selected based on the data type specified by your `wxPyGridTableBase`'s `GetTypeName` method
- `GetCellValue(int row, int col) --` `wxPyGridTableBase`'s `GetValue` Note: this works with string data only!
- `IsCurrentCellReadOnly() --` XXX Not sure about what this does under the covers
- `SetCellValue(int row, int col, const wxString& s) --` `wxPyGridTableBase`'s `SetValue`==

DragAndDrop onto Grid

`wxGrid` like most windows can be the target of a `DragAndDrop` action such as `wxFileDropTarget`. The basic way of doing this is follows:

```
class GridFileDropTarget(wxFileDropTarget):
    def __init__(self, grid):
        wxFileDropTarget.__init__(self)
        self.grid = grid

    def OnDropFiles(self, x, y, filenames):
        row, col = self.grid.XYToCell(x, y)
        if row > -1 and col > -1:
            self.grid.SetValue(row, col, filenames[0])
            self.grid.Refresh()
```

Note: I'm assuming here that the grid has a user supplied method `self.SetValue`. This method can either just call `wxGrid.SetCellValue` or do the appropriate logic for virtual grids. `self.grid.Refresh()` just tells the grid that it needs to be redrawn for the new data.

Virtual grids, however, have a significant problem here. It seems that `wxGrid.XYToCell` is broken for virtual grids. (A virtual grid is a grid that uses `wxGrid.SetTable` on a `PyGridTableBase` instance.) Here is a replacement to compute `XYToCell` to compute the current cell positions. Simply add this method to your virtual grid and away you go!

```
def XYToCell(self, x, y):
    # For virtual grids, XYToCell doesn't work properly
    # For some reason, the width and heights of the labels
    # are not computed properly and thw row and column
    # returned are computed as if the window wasn't
    # scrolled
    # This function replaces XYToCell for Virtual Grids

    rowwidth = self.GetGridRowLabelWindow().GetRect().width
    colheight = self.GetGridColLabelWindow().GetRect().height
    yunit, xunit = self.GetScrollPixelsPerUnit()
    xoff = self.GetScrollPos(wxHORIZONTAL) * xunit
    yoff = self.GetScrollPos(wxVERTICAL) * yunit

    # the solution is to offset the x and y values
    # by the width and height of the label windows
    # and then adjust by the scroll position
    # Then just go through the columns and rows
    # incrementing by the current column and row sizes
    # until the offset points lie within the computed
    # bounding boxes.
    x += xoff - rowwidth
    xpos = 0
    for col in range(self.GetNumberCols()):
        nextx = xpos + self.GetColSize(col)
        if xpos <= x <= nextx:
            break
        xpos = nextx

    y += yoff - colheight
    ypos = 0
    for row in range(self.GetNumberRows()):
        nexty = ypos + self.GetRowSize(row)
        if ypos <= y <= nexty:
            break
        ypos = nexty

    return row, col
```

DragAndDrop From Grid

For complete functionality, wxGrids require access to most mouse functions. While the main grid window can be used as a source for mouse drags, this can become problematic if you still want to resize cells or allow clicks and double clicks to activate the cell. A good compromise is to allow dragging from the rows of the grids. The following Mixin enables row labels to be used as drag sources. The drag source activates a `TextDrag` that contains the string representation of the selected wells, i.e. "[0,1,2,3]". More complicated `DragAndDrop` sources can be enabled by overriding the `StartDrag` method function of the `DragGridRowMixin` below.

```
'''DragGridRowMixin

Allows users to drag rows off of grids as Drag and Drop actions.
Usage:
    mixin with a Grid class.

class MyGrid(Grid, DragGridRowMixin):
    def __init__(self, parent, id):
        # must initialize grid before DragGridRowMixin
        Grid.__init__(self, parent, id)
        DragGridRowMixin.__init__(self)

To change the type of data being sent by the Grid, override the method function
    def StartDrag(self, selectedrows):
        """start a drag operation using selectedrows"""

Currently a drag operation is performed with a PyTextDataObject containing
the data str(selectedrows)
'''
import wx
from wx.grid import Grid
```

```
class DragGridRowMixin:
    """This mixin allows the use of grid rows as drag sources, you can drag
    rows off of a grid to a text target. Only row labels are draggable, internal
    cell contents are not.
```

You must Initialize this class *after* the wxGrid initialization.
example

```
class MyGrid(Grid, DragGridRowMixin):
    def __init__(self, parent, id):
        Grid.__init__(self, parent, id)
        DragGridRowMixin.__init__(self)

    """
    def __init__(self):
        rowWindow = self.GetGridRowLabelWindow()
        # various flags to indicate whether we should have a select
        # event or a drag event
        self._potentialDrag = False
        self._startRow = None
        self._shift, self._ctrl = None, None

        wx.EVT_LEFT_DOWN(rowWindow, self.OnDragGridLeftDown)
        wx.EVT_LEFT_UP(rowWindow, self.OnDragGridLeftUp)
        wx.EVT_LEFT_UP(self, self.OnDragGridLeftUp)
        wx.EVT_MOTION(rowWindow, self.OnDragGridMotion)
        wx.EVT_KEY_DOWN(self, self.OnDragGridKeyDown)
        wx.EVT_KEY_UP(self, self.OnDragGridKeyUp)

    def OnDragGridKeyDown(self, evt):
        """Set the states of the shift and ctrl keys"""
        self._shift, self._ctrl = (evt.ShiftDown(), evt.ControlDown())

    def OnDragGridKeyUp(self, evt):
        """unset the states of the shift and ctrl keys"""
        self._shift, self._ctrl = None, None

    def OnDragGridLeftDown(self, evt):
        """The left button is down so see if we are selecting rows or not
        and do the appropriate thing. We need to do this because we
        are blocking the rowlabels so rows won't be selected."""
        x,y = evt.GetX(), evt.GetY()
        row, col = self.DragGridRowXYToCell(x,y, colheight=0)

        if not self._shift and not self._ctrl:
            self._startRow = row
            self.SelectRow(row)
        elif self._shift and not self._ctrl:
            if self._startRow > row:
                start, end = row, self._startRow
            else:
                start, end = self._startRow, row
            for row in range(start, end+1):
                self.SelectRow(row, True)
        elif self._ctrl:
            self.SelectRow(row, True)
        self._potentialDrag = True

    def OnDragGridLeftUp(self, evt):
        """We are not dragging anymore, so unset the potentialDrag flag"""
        self._potentialDrag = False
        evt.Skip()

    def OnDragGridMotion(self, evt):
        """We are moving so see whether this should be a drag event or not"""
        if not self._potentialDrag:
            evt.Skip()
            return

        x,y = evt.GetX(), evt.GetY()
        row, col = self.DragGridRowXYToCell(x,y, colheight=0)
        rows = self.GetSelectedRows()
        if not rows or row not in rows:
            evt.Skip()
            return
        self.StartDrag(rows)

    def DragGridRowXYToCell(self, x, y, colheight=None, rowwidth=None):
        # For virtual grids, XYToCell doesn't work properly
        # For some reason, the width and heights of the labels
        # are not computed properly and thw row and column
```

```

# returned are computed as if the window wasn't
# scrolled
# This function replaces XYToCell for Virtual Grids

if rowwidth is None:
    rowwidth = self.GetGridRowLabelWindow().GetRect().width
if colheight is None:
    colheight = self.GetGridColLabelWindow().GetRect().height
yunit, xunit = self.GetScrollPixelsPerUnit()
xoff = self.GetScrollPos(wx.HORIZONTAL) * xunit
yoff = self.GetScrollPos(wx.VERTICAL) * yunit

# the solution is to offset the x and y values
# by the width and height of the label windows
# and then adjust by the scroll position
# Then just go through the columns and rows
# incrementing by the current column and row sizes
# until the offset points lie within the computed
# bounding boxes.
x += xoff - rowwidth
xpos = 0
for col in range(self.GetNumberCols()):
    nextx = xpos + self.GetColSize(col)
    if xpos <= x <= nextx:
        break
    xpos = nextx

y += yoff - colheight
ypos = 0
for row in range(self.GetNumberRows()):
    nexty = ypos + self.GetRowSize(row)
    if ypos <= y <= nexty:
        break
    ypos = nexty

return row, col

def StartDrag(self, selectedrows):
    """This starts the drag event, override this to send different drag
    types."""
    tdo = wx.PyTextDataObject(str(selectedrows))
    # Create a Drop Source Object, which enables the Drag operation
    tds = wx.DropSource(self.GetGridRowLabelWindow())
    # Associate the Data to be dragged with the Drop Source Object
    tds.SetData(tdo)
    # Intiate the Drag Operation
    tds.DoDragDrop(wx.TRUE)

if __name__ == "__main__":
    # -----
    # Testing
    class GridTextDropTarget(wx.TextDropTarget):
        def __init__(self, grid):
            wx.TextDropTarget.__init__(self)
            self.grid = grid

        def OnDropText(self, x, y, text):
            # XYToCell doesn't behave quite right, so we'll just
            # grab the DragGridRowXYToCell that we fixed,
            # see the wx.Grid wiki for a better explanation
            # http://wiki.wxpython.org/index.cgi/wxGrid
            row, col = self.grid.DragGridRowXYToCell(x, y)
            if row > -1 and col > -1:
                self.grid.SetCellValue(row, col, text)
                self.grid.Refresh()

    class T(Grid, DragGridRowMixin):
        def __init__(self, parent, id, title):
            Grid.__init__(self, parent, id)
            self.CreateGrid(25,25)
            DragGridRowMixin.__init__(self)
            self.SetDropTarget(GridTextDropTarget(self))

    app = wx.PySimpleApp()
    frame = wx.Frame(None, -1, "hello")
    foo = T(frame, -1, "hello")
    frame.Show()
    app.MainLoop()

```

Capturing Cell Edits when focus changes

The default behavior of the grid control is to revert to the old value if the user changes focus before hitting enter. This can frustrate users when their edits are gone because they have clicked on another region of the GUI.

A wxGrid is actually a control made of several wxWindow components: The main grid, the table of cells, and a cell editor when a value is being modified.

To make the grid save its cell data, it's necessary to first bind to the EVT_GRID_EDITOR_CREATED event. Inside that event, then bind to the EVT_KILL_FOCUS event for the newly created control.

Note: This code is written from the perspective of the Frame containing the grid. It may of course be rewritten to embed the event handling within a class inheriting from wxGrid.

```
def __init__(self):
    self.grid = wx.grid.Grid(self, -1)
    self.Bind(wx.EVT_GRID_EDITOR_CREATED, self.OnGridEditorCreated)

def OnGridEditorCreated(self, event):
    """ Bind the kill focus event to the newly instantiated cell editor """
    editor = event.GetControl()
    editor.Bind(wx.EVT_KILL_FOCUS, self.OnKillFocus)
    event.Skip()

def OnKillFocus(self, event):
    # Cell editor's grandparent, the grid GridWindow's parent, is the grid.
    grid = event.GetEventObject().GetGrandParent()
    grid.SaveEditControlValue()
    grid.HideCellEditControl()
    event.Skip()
```

The above code snippet handles the case where a user changes focus by clicking on another control in your interface. However, clicking on the menu of a frame does not remove focus from the edit control, so the above code won't be triggered. If your users are like mine and fond of choosing Save from the menus without closing the cell editor, you can modify the above to compensate.

```
def __init__(self):
    self.grid = wx.grid.Grid(self, -1)
    self.Bind(wx.EVT_GRID_EDITOR_CREATED, self.OnGridEditorCreated)
    self.Bind(wx.EVT_MENU_OPEN, self.OnMenuOpen)

def OnGridEditorCreated(self, event):
    """ Bind the kill focus event to the newly instantiated cell editor """
    editor = event.GetControl()
    editor.Bind(wx.EVT_KILL_FOCUS, self.OnKillFocus)
    event.Skip()

def OnKillFocus(self, event):
    # Cell editor's grandparent, the grid GridWindow's parent, is the grid.
    grid = event.GetEventObject().GetGrandParent()
    grid.SaveEditControlValue()
    grid.HideCellEditControl()
    event.Skip()

def OnMenuOpen(self, event):
    self.grid.HideCellEditControl()
```

The EVT_MENU_OPEN handler above will generate an EVT_KILL_FOCUS event, saving the contents of the editor. Neither call to hide generates an EVT_GRID_EDITOR_HIDDEN event, but the call to Save does generate an EVT_GRID_CELL_CHANGE event, so you only want to call it once, in the EVT_KILL_FOCUS handler.

[Back to the wxGrid Manual](#)

When dealing with large grids the RANGE_SELECT method described in 1.4.2 can be quite slow when selecting whole columns/rows. The following code treats that special case separately and is much faster:

```
def _OnRangeSelect:
    top,bottom = event.GetTopRow(),event.GetBottomRow()+1
    left,right = event.GetLeftCol(),event.GetRightCol()+1

    if event.Selecting():
        # adding to the list...
```

```

        if bottom-top == self.GetNumberRows():
            self.currentSelection[0] = range(top, bottom)
            for index in range(left, right):
                if index not in self.currentSelection[1]:
                    self.currentSelection[1].append(index)
        elif right-left == self.GetNumberCols():
            self.currentSelection[1] = range(left, right)
            for index in range(top, bottom):
                if index not in self.currentSelection[0]:
                    self.currentSelection[0].append(index)
        else:
            for index in range(top, bottom):
                if index not in self.currentSelection[0]:
                    self.currentSelection[0].append(index)
            for index in range(left, right):
                if index not in self.currentSelection[1]:
                    self.currentSelection[1].append(index)
    else:
        # removal from list
        if bottom-top == self.GetNumberRows():
            for index in range(left, right):
                while index in self.currentSelection[1]:
                    self.currentSelection[1].remove(index)
        elif right-left == self.GetNumberCols():
            for index in range(top, bottom):
                while index in self.currentSelection[0]:
                    self.currentSelection[0].remove(index)
        else:
            for index in range(top, bottom):
                while index in self.currentSelection[0]:
                    self.currentSelection[0].remove(index)
            for index in range(left, right):
                while index in self.currentSelection[1]:
                    self.currentSelection[1].remove(index)

```

Additionally one has to catch the mouse click on the row/col labels to reset the current selection:

```

def __init__(self, ...):
    ...
    self.Bind(wx.grid.EVT_GRID_LABEL_LEFT_CLICK, self.OnLabelLeftClick)
    ...

def OnLabelLeftClick(self, evt):
    self.currentSelection = [[], []]
    evt.Skip()

```

--Christian

This document was created with Win2PDF available at <http://www.win2pdf.com>.
The unregistered version of Win2PDF is for evaluation or non-commercial use only.
This page will not be added after purchasing Win2PDF.