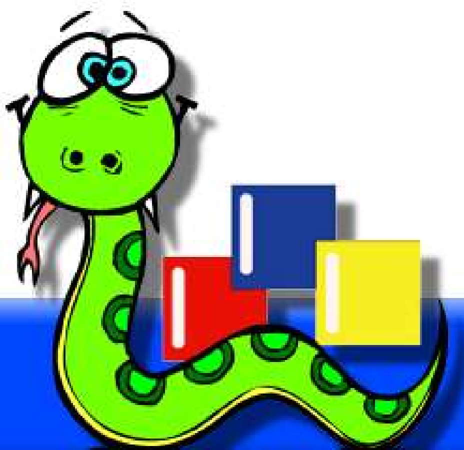
Advanced wxPython Nuts and Bolts

Robin Dunn

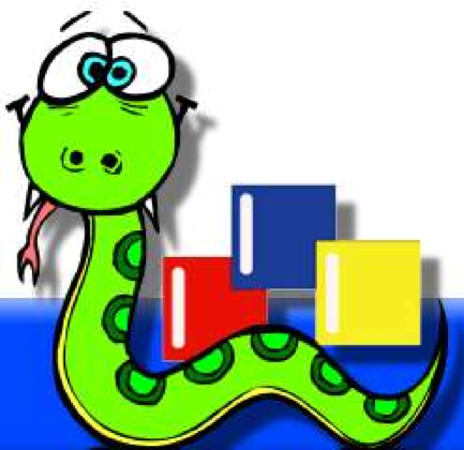
O'Reilly Open Source Convention

July 26–30, 2004



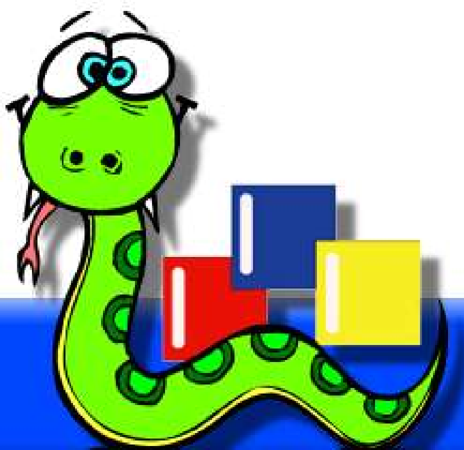
Presentation Overview

- Introduction
- 2.5 Migration
- wx.ListCtrl
- Virtual wx.ListCtrl
- wx.TreeCtrl
- wx.gizmos.TreeListCtrl
- wx.grid.Grid
- ScrolledPanel
- wx.HtmlWindow
- Keeping the UI Updated
- Data transfer
  - data objects
  - clipboard
  - drag and drop
- Sizers and more sizers
- Creating custom widgets
- Double buffered drawing

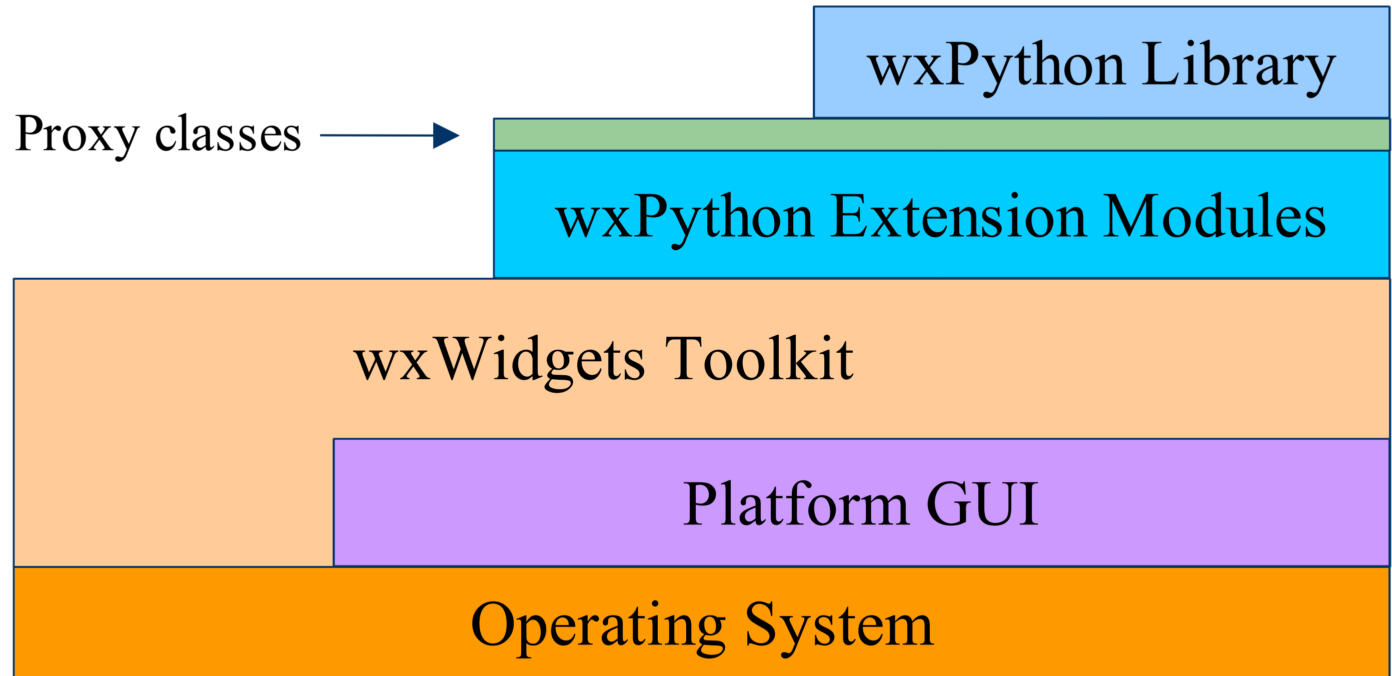


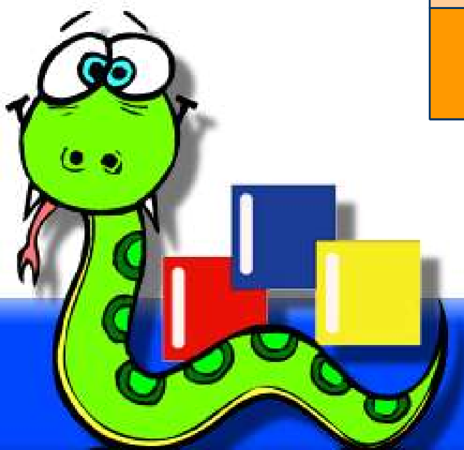
Introduction to wxPython

- wxPython is a GUI toolkit for Python, built upon the wxWidgets C++ toolkit. (See http://wxWidgets.org/)
  - Cross platform: Windows, Linux, Unix, OS X.
  - Uses native widgets/controls, plus many platform independent widgets.
- Mature, well established projects.
  - wxWidgets: 1992
  - wxPython: 1996



Introduction: architecture





Introduction: partial class hierarchy

wx.Object

wx.EvtHandler

wx.Window

wx.TopLevelWindow

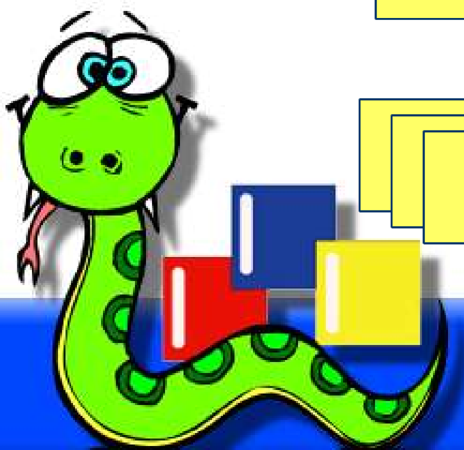
wx.Panel

wx.Control

wx.Frame

wx.Dialog

wx.ScrolledWindow



wxPython 2.5 Migration Highlights

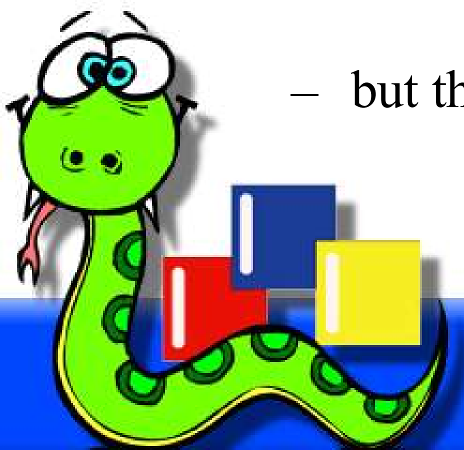
- wx "Namespace"
  - Instead of

```
from wxPython.wx import *
class MyFrame(wxFrame):
    ...
```

  - we now use this by default

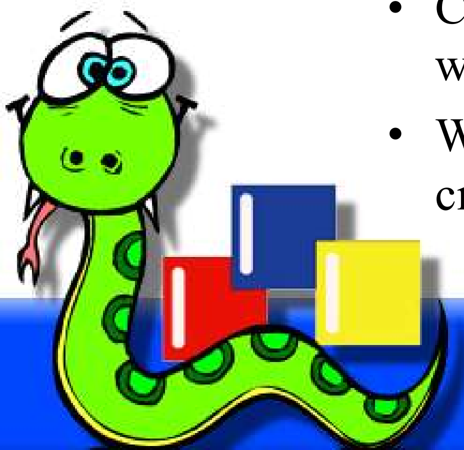
```
import wx
class MyFrame(wx.Frame):
    ...
```

  - but the old "namespace" is still available via compatibility wrappers



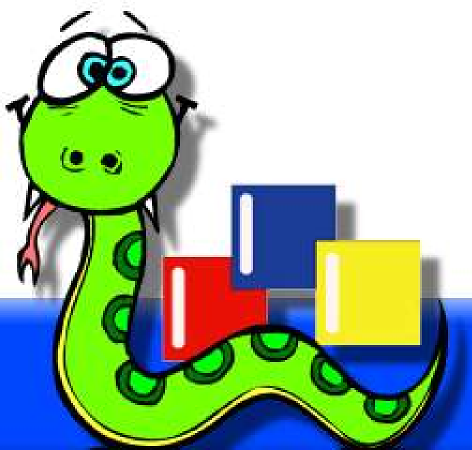
wxPython 2.5 Migration Highlights

- Module initialization
  - Due to changes in the C++ library, wxWidgets is not initialized until the wx.App object is created.
  - Some beneficial side-effects:
    - Can import wx to do things like check version number, get docstrings, etc. without the overhead of fully initializing the library (connecting to the display server, etc.)
    - The thread that creates the wx.App object will be considered the "main" thread, instead of the one that imports the wx module
  - Some not so good side-effects:
    - Can not create any GDI objects or do any UI related operation until the wx.App object is created.
    - Will get a wx.PyNoAppError if we can detect the situation ☺, or a crash otherwise ☹



wxPython 2.5 Migration Highlights

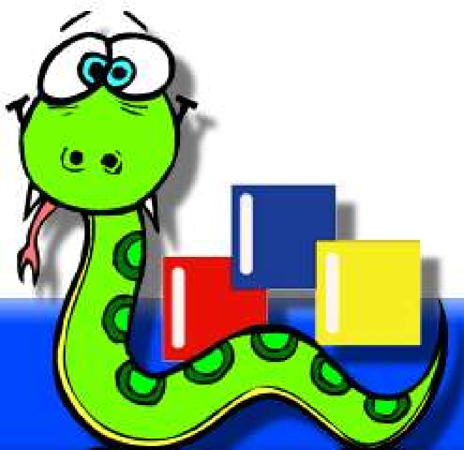
- Have switched away from a heavily customized legacy version of SWIG to a nearly standard SWIG 1.3.x.
  - All classes are now derived from `object` and so are new- style classes supporting properties, static functions, meta-classes, etc.
  - More modern python proxy code is generated
  - Wrapper code somewhat more efficient
  - Will be able to take advantage of more advanced features in the future



wxPython 2.5 Migration Highlights

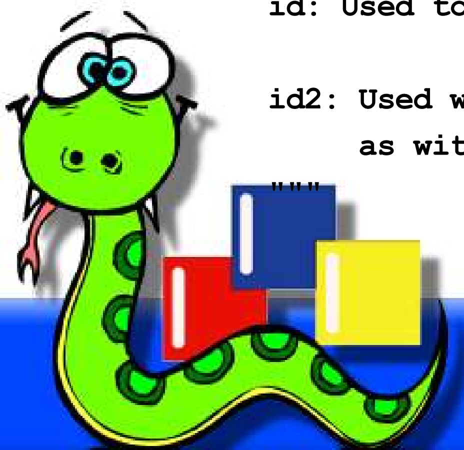
- Event binding updates
  - All **EVT_*** event binder functions have been converted to instances of the **wx.PyEventBinder** class.
  - New **Bind** method that helps reduce the need for keeping track of window and menu Ids

```
self.Bind(wx.EVT_SIZE,   self.OnSize)
self.Bind(wx.EVT_BUTTON, self.OnButtonClick, theButton)
self.Bind(wx.EVT_MENU,   self.OnExit, id=wx.ID_EXIT)
```



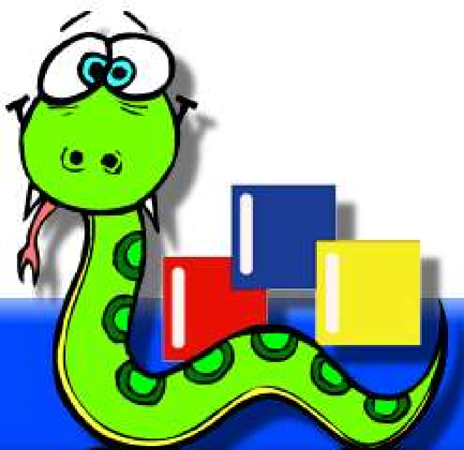
wxPython 2.5 Migration Highlights

```
def Bind(self, event, handler, source=None, id=wx.ID_ANY, id2=wx.ID_ANY):
    """
    Bind an event to an event handler.

    event: One of the EVT_* objects that specifies the type of event to bind

    handler: A callable object to be invoked when the event is delivered to self.
             Pass None to disconnect an event handler.

    source: Sometimes the event originates from a different window than self, but
            you still want to catch it in self.  (For example, a button event
            delivered to a frame.)  By passing the source of the event, the
            event handling system is able to differentiate between the same
            event type from different controls.

    id: Used to specify the event source by ID instead of instance.

    id2: Used when it is desirable to bind a handler to a range of IDs, such
         as with EVT_MENU_RANGE.
    """
```

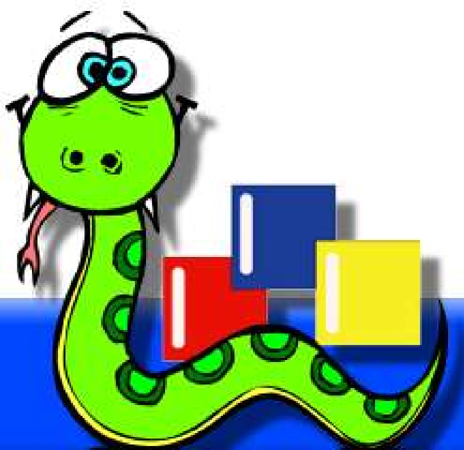


wxPython 2.5 Migration Highlights

- See http://wxPython.org/MigrationGuide.html for more.



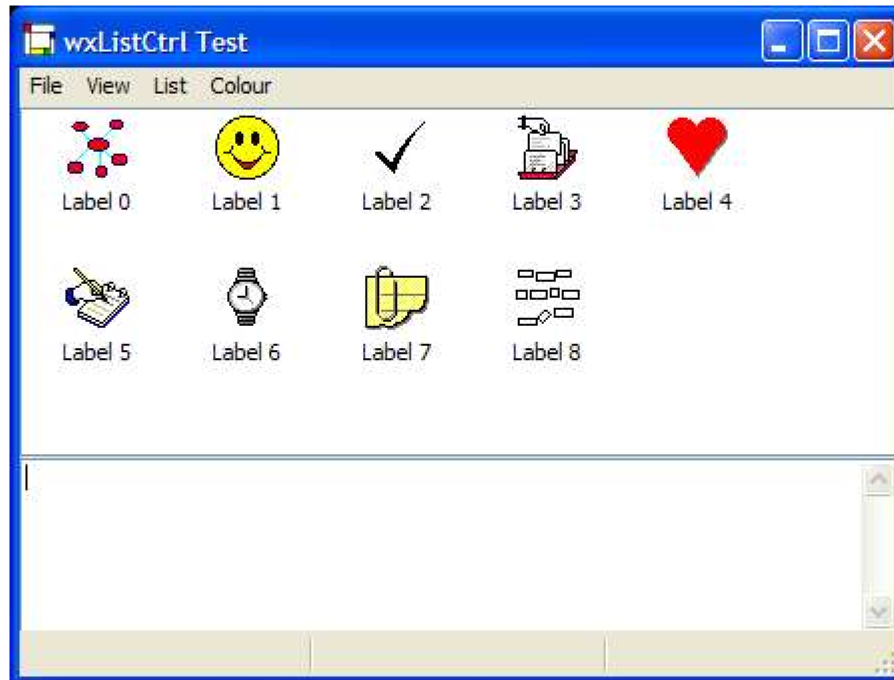
Questions?

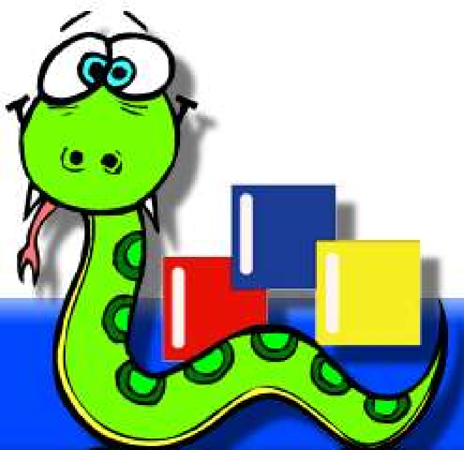


wx.ListCtrl

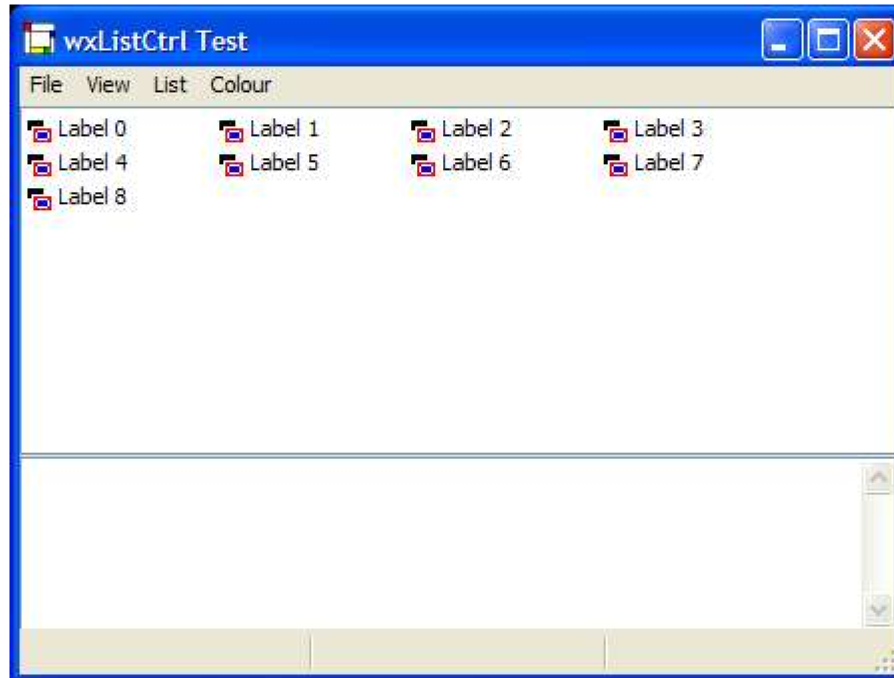
- Presents a list of items with one of several possible views
  - List
  - Report
  - Icon
- Supports various attributes and operations on the list data
  - Icons, and colors
  - Sorting
  - multiple selection
- The same native control that is used by Windows Explorer
- On GTK and Mac it is implemented generically, but there may be native versions eventually.

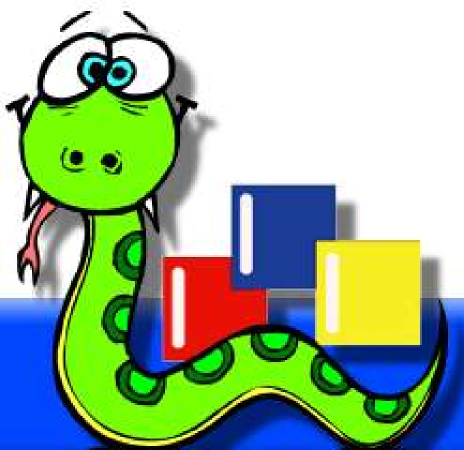
wx.ListCtrl: Icon view



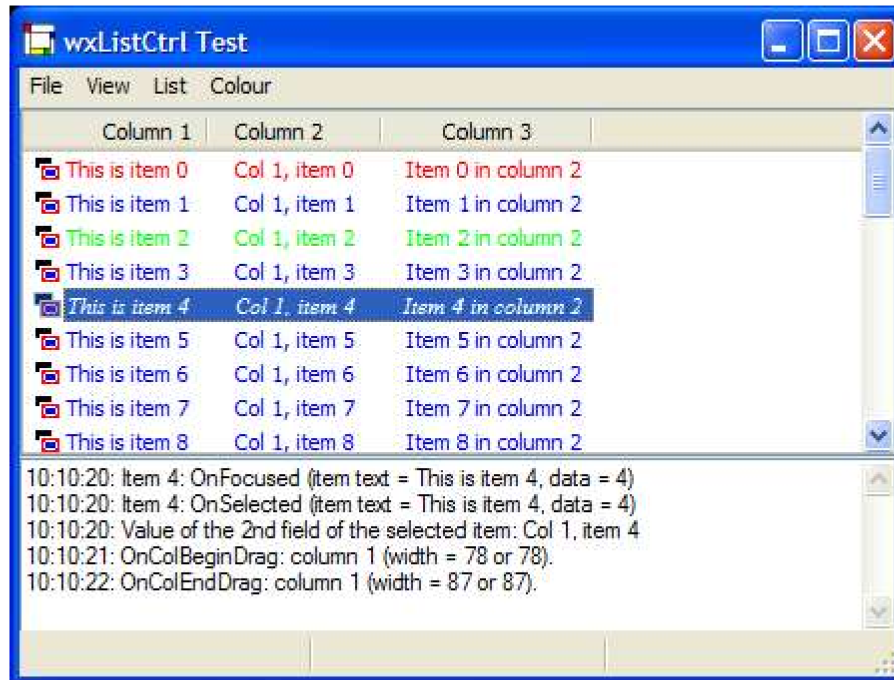


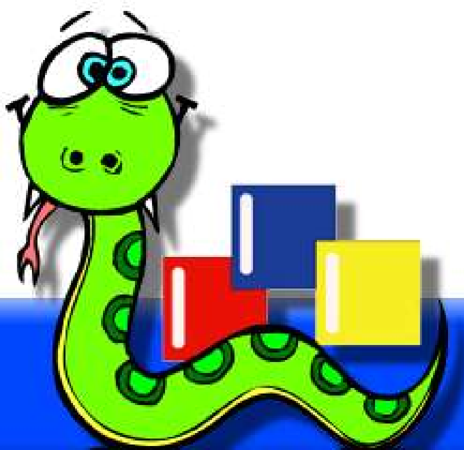
wx.ListCtrl: list (small icon) view





wx.ListCtrl: report view



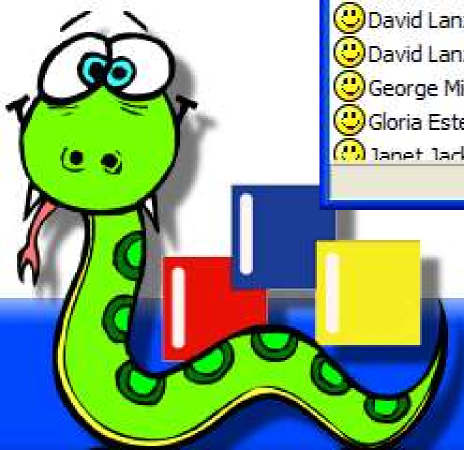


wx.ListCtrl: report view

RunDemo: ListCtrl

File

Artist ▲	Title	Genre
Bad English	The Price Of Love	Rock
Billy Joel	Blonde Over Blue	Rock
Billy Joel	Famous Last Words	Rock
Billy Joel	Lullabye (Goodnight, My Angel)	Rock
Billy Joel	The River Of Dreams	Rock
Billy Joel	Two Thousand Years	Rock
Blue Man Group	Drumbone	New Age
Blue Man Group	Endless Column	New Age
Blue Man Group	Klein Mandelbrot	New Age
Cusco	Dream Catcher	New Age
Cusco	Geronimos Laughter	New Age
Cusco	Ghost Dance	New Age
DNA featuring Suzanne Vega	Tom's Diner	Rock
David Arkenstone	Papillon (On The Wings Of The Butterfly)	New Age
David Arkenstone	Stepping Stars	New Age
David Arkenstone	Carnation Lily Lily Rose	New Age
David Lanz	Behind The Waterfall	New Age
David Lanz	Cristofori's Dream	New Age
David Lanz	Heartsounds	New Age
David Lanz	Leaves on the Seine	New Age
George Michael	Praying For Time	Rock
Gloria Estefan	Here We Are	Rock
Janet Jackson	Alright	Rock

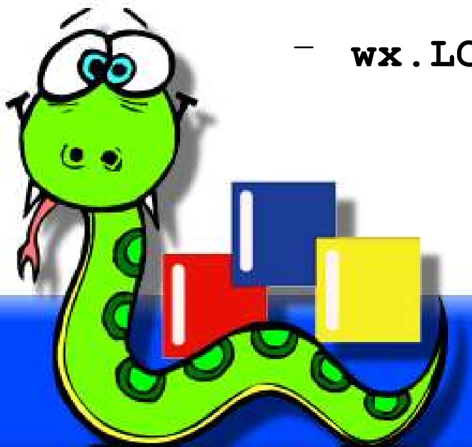


wx.ListCtrl

- Same basic constructor as other windows:

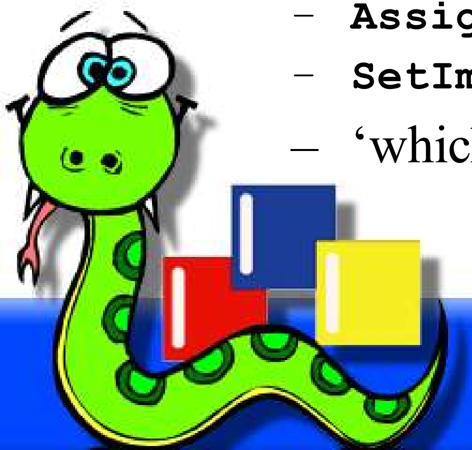
```
__init__(self, parent, id=-1, pos=wx.DefaultPosition, size=wx.DefaultSize,
        style=wx.LC_ICON, validator=wx.DefaultValidator,
        name=ListCtrlNameStr)
```

- Set view style as a style flag in the constructor
- Other useful styles:
  - **wx.LC_SINGLE_SEL**: set single selection mode
  - **wx.LC_HRULES, wx.LC_VRULES**: draw lines between rows and cols in report mode.
  - **wx.LC_NO_HEADER**: don't use a column header in report mode.
  - **wx.LC_EDIT_LABELS**: allow the labels to be edited



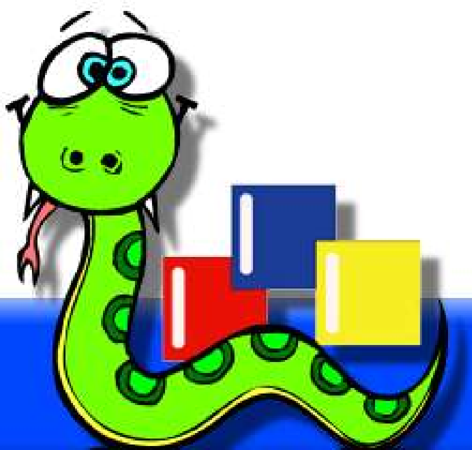
wx.ListCtrl

- Common events to bind:
  - **EVT_LIST_ITEM_SELECTED**
  - **EVT_LIST_ITEM_DESELECTED**
  - **EVT_LIST_ITEM_ACTIVATED**
  - **EVT_LIST_COL_CLICK**
- In report mode you must define the columns, even if there is only one.
  - **InsertColumn(col, heading, format=wx.LIST_FORMAT_LEFT, width=-1)**
- Icons are stored in a **wx.ImageList** to facilitate reuse:
  - **AssignImageList(imageList, which)**
  - **SetImageList(imageList, which)**
  - 'which' is **wx.IMAGE_LIST_NORMAL**, or **wx.IMAGE_LIST_SMALL**



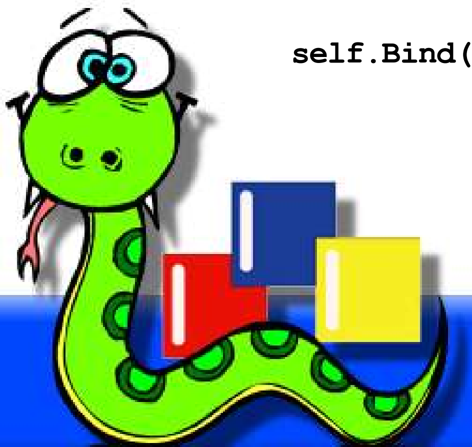
wx.ListCtrl

- Add items to the wx.ListCtrl with one of the InsertItem methods:
  - `InsertStringItem(index, label)`
  - `InsertImageStringItem(index, label, imageIndex)`
  - `InsertItem(item)`
- Set values for additional columns in report mode:
  - `SetStringItem(index, col, label)`



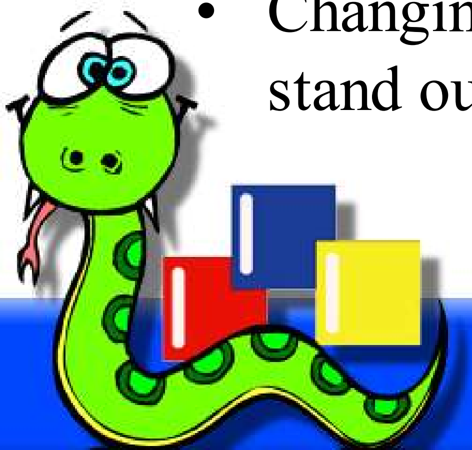
wx.ListCtrl: example

```
il = wx.ImageList(16,16)
imgidx1 = il.Add(bmp1)
imgidx2 = il.Add(bmp2)

lc = wx.ListCtrl(self, style=wx.LC_REPORT|wx.LC_SINGLE_SEL)
lc.AssignImageList(il)
lc.InsertColumn(0, “Column 1”)
lc.InsertColumn(1, “Column 2”, wx.LIST_FORMAT_RIGHT)
lc.InsertColumn(2, “Column 3”)

for data in GetDataItems():
    index = lc.InsertImageStringItem(sys.maxint, data[0], imgidx1)
    lc.SetStringItem(index, 1, data[1])
    lc.SetStringItem(index, 2, data[2])

self.Bind(wx.LIST_ITEM_SELECTED, self.OnSelect, lc)
```



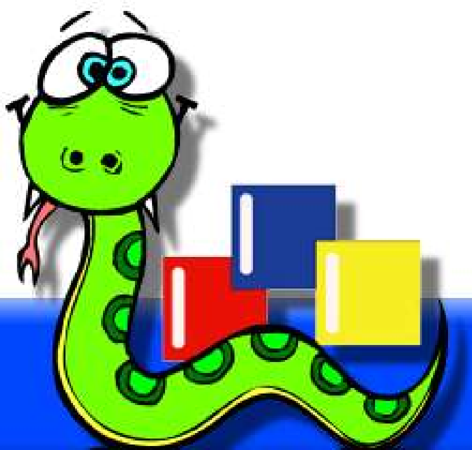
wxListCtrl: tips and tricks

- `wx.lib.mixins.listctrl` has some useful helper classes:
  - `ColumnSorterMixin`
  - `ListCtrlAutoWidthMixin`
  - `TextEditMixin`
  - etc.
- Each item can have an integer "data value" associated with it that will not change when the item is resorted or etc. This value can be used to map to the original data objects, or whatever.
- Changing the color of items is an easy way to make them stand out.

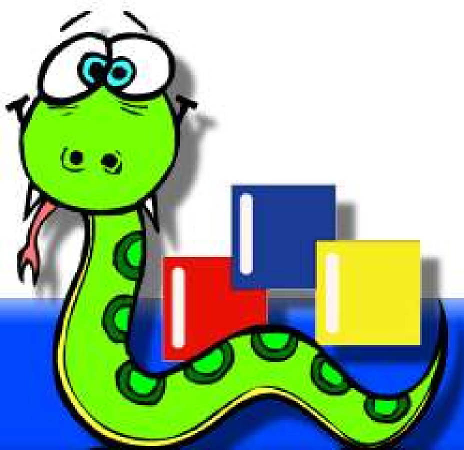


wx.ListCtrl: gotcha's

- Currently icons can only be used for the first column in report mode
- Data duplication issues…
- In serious need of a redesign…

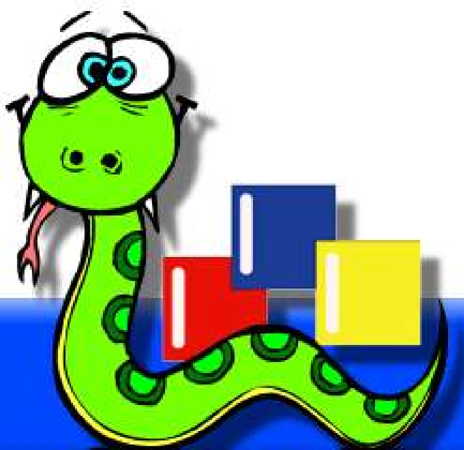


Questions?



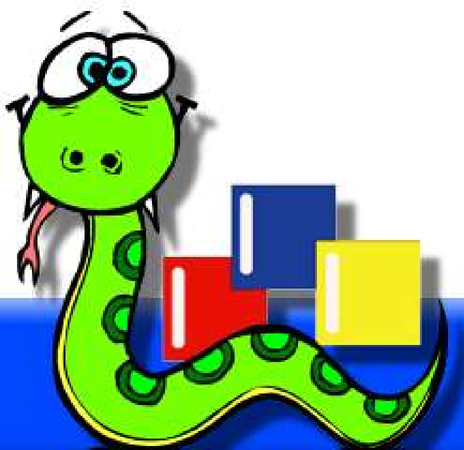
Virtual wx.ListCtrl

- wx.ListCtrl also has **`wx.LC_VIRTUAL`** style
- Can only be used with report mode
- Data items are never added to the control, instead the control asks you for the data as it needs them for display
- Not just the strings, but attributes and image list id's too.
- A virtual wx.ListCtrl can easily display millions of items with very little overhead.



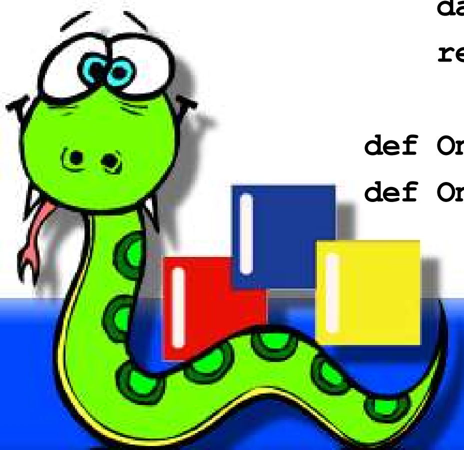
Virtual wx.ListCtrl

- Simply need to call **`SetItemCount`** and to override a few methods:
  - **`OnGetItemText(item, column)`**
  - **`OnGetItemImage(item)`**
  - **`OnGetItemAttr(item)`**
- **`EVT_LIST_CACHE_HINT`** can be bound and used to prefetch data items the wx.ListCtrl thinks it will need soon.



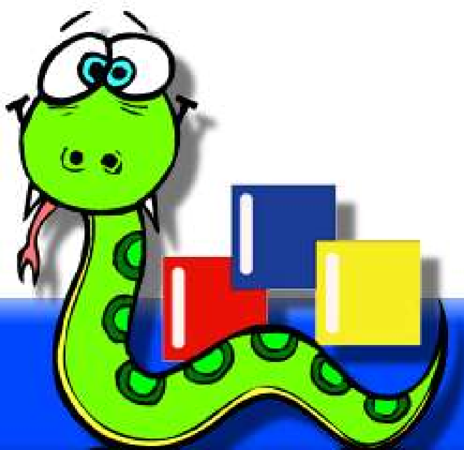
Virtual wx.ListCtrl: example

```
class MyVirtualListCtrl(wx.ListCtrl):
    def __init__(self, parent, dataSource)
        wx.ListCtrl.__init__(self, parent,
                style=wx.LC_REPORT|wx.LC_SINGLE_SEL|wx.LC_VIRTUAL)
        self.dataSource = dataSource
        self.InsertColumn(0, "Column 1")
        self.InsertColumn(1, "Column 2")
        self.InsertColumn(2, "Column 3")
        self.SetItemCount(dataSource.GetCount())
        self.Bind(wx.EVT_LIST_CACHE_HINT, self.DoCacheItems)

    def DoCacheItems(self, evt):
        self.dataSource.UpdateCache(evt.GetCacheFrom(), evt.GetCacheTo())

    def OnGetItemText(self, item, col):
        data = self.dataSource.GetItem(item)
        return data[col]

    def OnGetItemAttr(self, item):  return None
    def OnGetItemImage(self, item): return -1
```

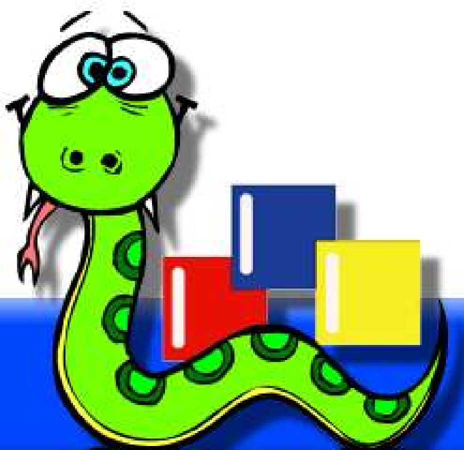


Virtual wx.ListCtrl: gotcha's

- Any operation that would need to visit every item in the list will not be done in a virtual wx.ListCtrl.
  - auto-sizing columns
  - sorting
  - etc.

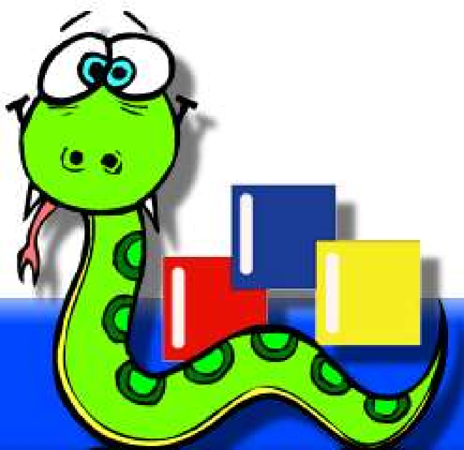


Questions?



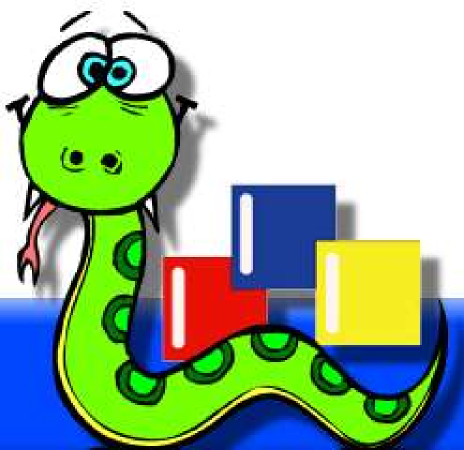
wx.TreeCtrl

- Presents a view of hierarchical data with expandable and collapsible nodes.
- The same native control that is used by Windows Explorer
- On GTK and Mac it is implemented generically, but there may be native versions eventually.
- Not all items have to be preloaded into the control, but can be done on demand for a virtual view.

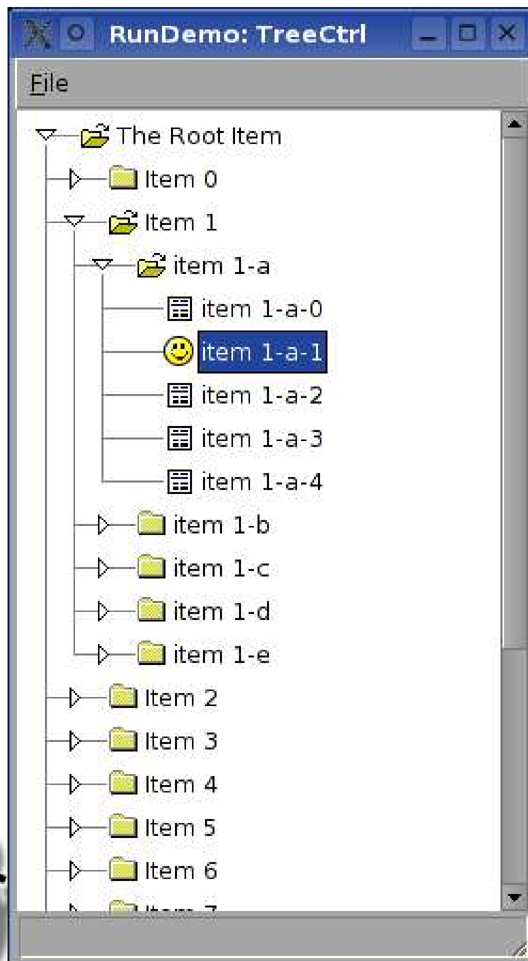


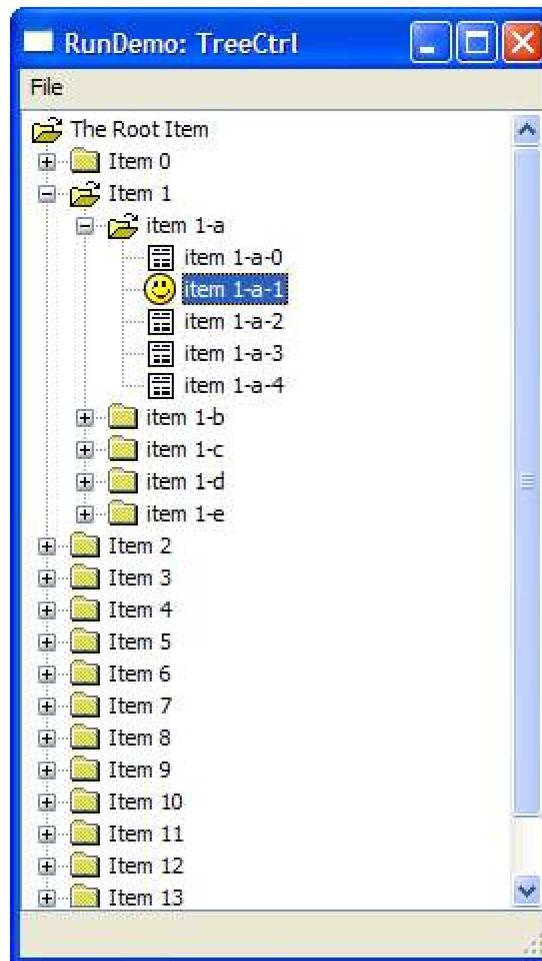
wx.TreeCtrl

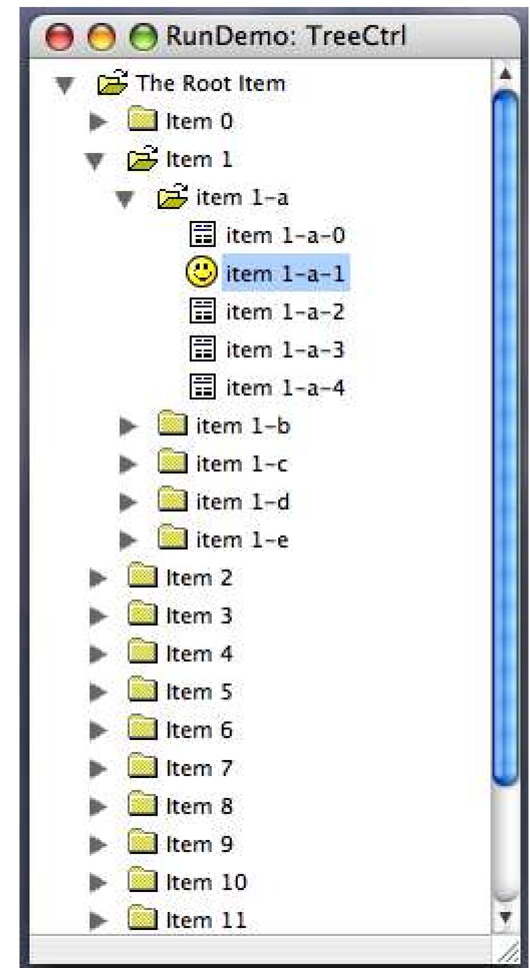
- The root node can be hidden to give the appearance of multiple roots.
- Nodes can have images associated with them, that are changed based on state of the node.

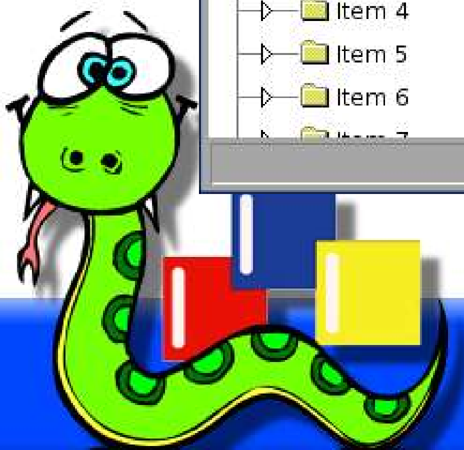


wx.TreeCtrl



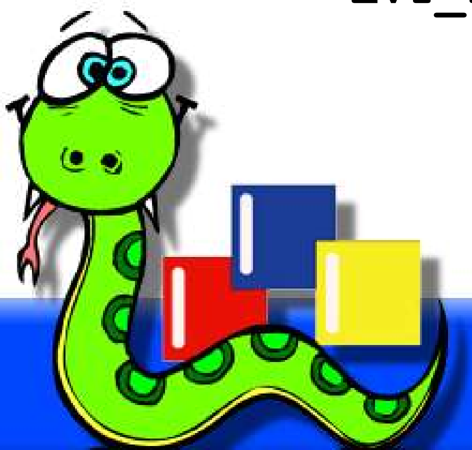






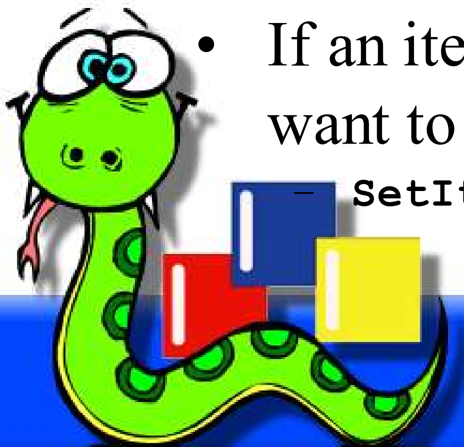
wx.TreeCtrl

- Interesting Styles:
  - `wx.TR_HIDE_ROOT`
  - `wx.TR_MULTIPLE`
  - `wx.TR_FULL_ROW_HIGHLIGHT`
  - `wx.TR_EDIT_LABELS`
- Common Events:
  - `EVT_TREE_ITEM_ACTIVATED`
  - `EVT_TREE_SEL_CHANG(ED,ING)`
  - `EVT_TREE_ITEM_COLLAPS(ED,ING)`
  - `EVT_TREE_ITEM_EXPAND(ED,ING)`



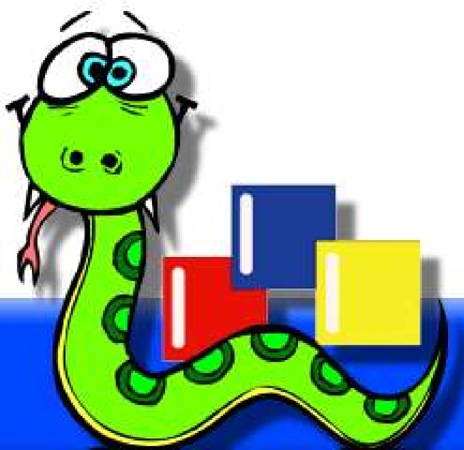
wx.TreeCtrl

- Nodes in a wx.TreeCtrl are referred to using a class named **`wx.TreeItemId`** which is used when accessing items, adding child items, traversing the tree structure, finding the currently selected item, etc.
- The root, or top-level parent node is always added first
  - **`AppendRoot(text, image=-1, selImage=-1)`**
- After that, nodes can be added in any order, as long as their parent node already exists
  - **`AppendItem(parent, text, image=-1, selImage=-1)`**
- If an item should appear to have children, but you don't want to add them yet:
  - **`SetItemHasChildren(item, hasChildren=True)`**



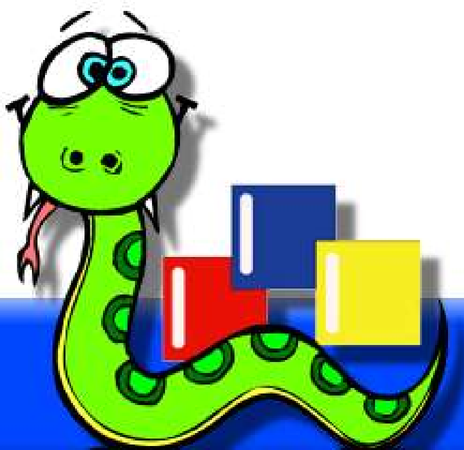
wx.TreeCtrl

- Icons are stored in a wx.ImageList to facilitate reuse:
  - **AssignImageList(imageList)**
  - **SetImageList(imageList)**
- Nodes have various states, each of which can have an associated icon
  - **SetItemImage(item, image, which)**
    - **wx.TreeItemIcon_Normal**
    - **wx.TreeItemIcon_Expanded**
    - **wx.TreeItemIcon_Selected**
    - **wx.TreeItemIcon_SelectedExpanded**



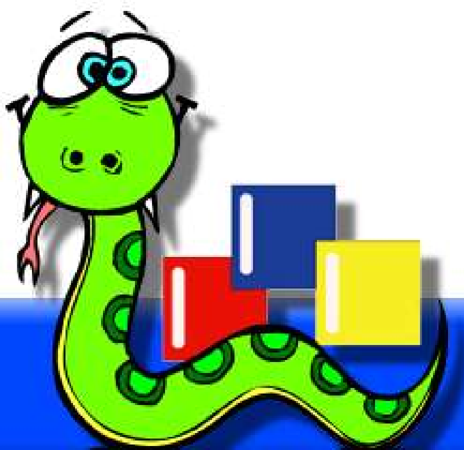
wx.TreeCtrl

- Nodes can have arbitrary Python objects associated with them, which can be used to link tree items with your actual data objects, etc.
  - `SetItemPyData(item, object)`
  - `GetItemPyData(item)`
- Nodes can have non-standard attributes:
  - `SetItemTextColour(item, color)`
  - `SetItemBackgroundColour(item, color)`
  - `SetItemFont(item, color)`
  - `SetItemBold(item, bold=True)`



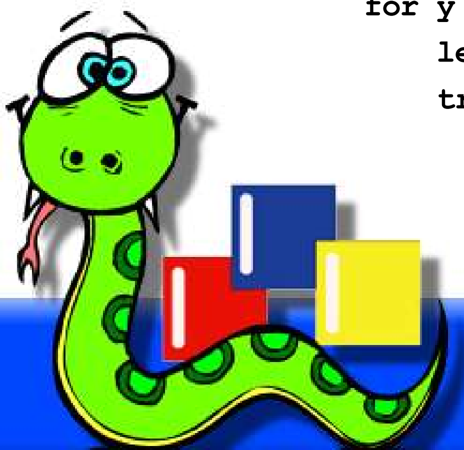
wx.TreeCtrl

- Many node traversal methods to choose from:
  - `GetRootItem()`
  - `GetItemParent(item)`
  - `GetFirstChild(item)`
  - `GetNextChild(item, cookie)`
  - `GetLastChild(item)`
  - `GetNextSibling(item)`
  - `GetPrevSibling(item)`
  - `GetFirstVisibleItem()`
  - `GetNextVisible(item)`



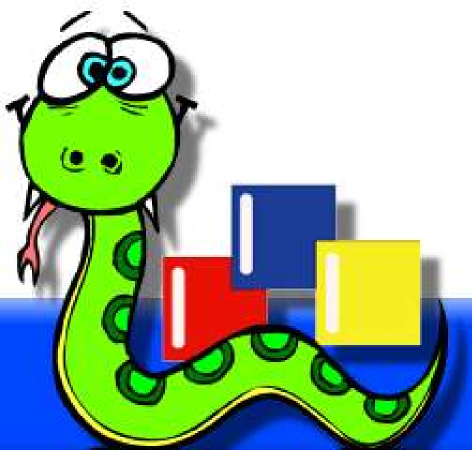
wx.TreeCtrl: example

```
il = wx.ImageList(16,16)
fldridx     = il.Add(folderBmp)
fldropenidx = il.Add(folderOpenBmp)
leafidx     = il.Add(leafBmp)

tree = wx.TreeCtrl(self)
root = tree.AddRoot("root item")
tree.SetItemImage(root, fldridx, wx.TreeItemIcon_Normal)
tree.SetItemImage(root, fldropenidx, wx.TreeItemIcon_Expanded)

for x in range(5):
    item = tree.AppendItem(root, "Item %d" % x)
    tree.SetItemImage(item, fldridx, wx.TreeItemIcon_Normal)
    tree.SetItemImage(item, fldropenidx, wx.TreeItemIcon_Expanded)
    for y in range(5):
        leaf = tree.AppendItem(item, "leaf %d-%d" % (x,y))
        tree.SetItemImage(leaf, leafidx, wx.TreeItemIcon_Normal)
```

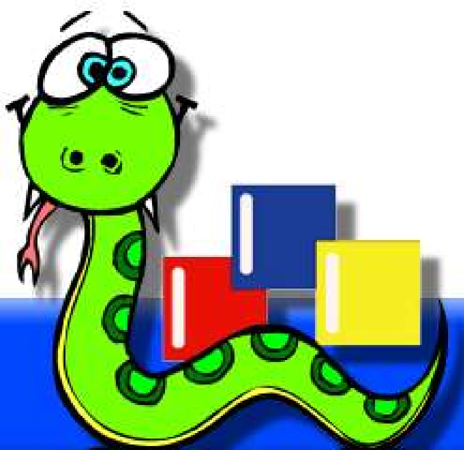


wx.TreeCtrl: gotcha's

- On MS Windows the native control won't sort items unless they have a data value, even if it is not used by the sort. Calling SetPyData(item, None) for every item solves the problem.
- If images are used at all then every node should have an image assigned, otherwise there will be alignment issues.

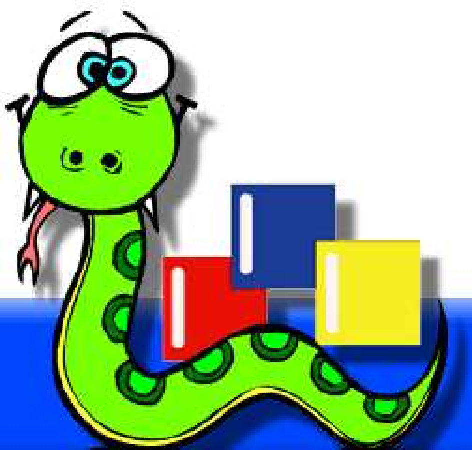


Questions?

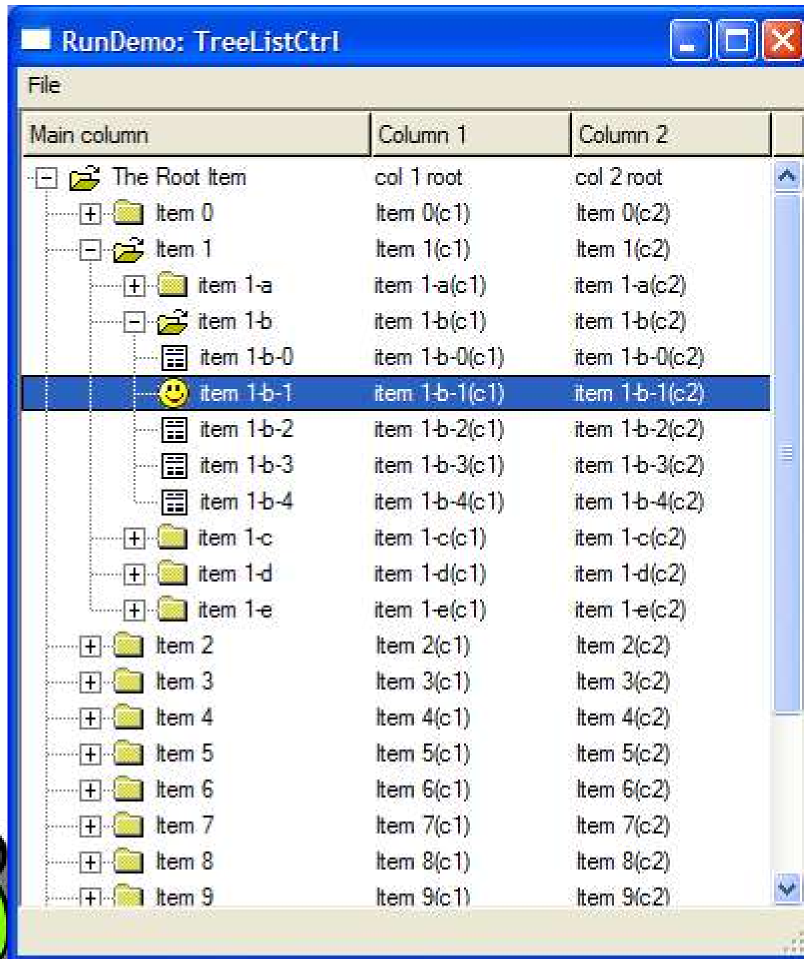


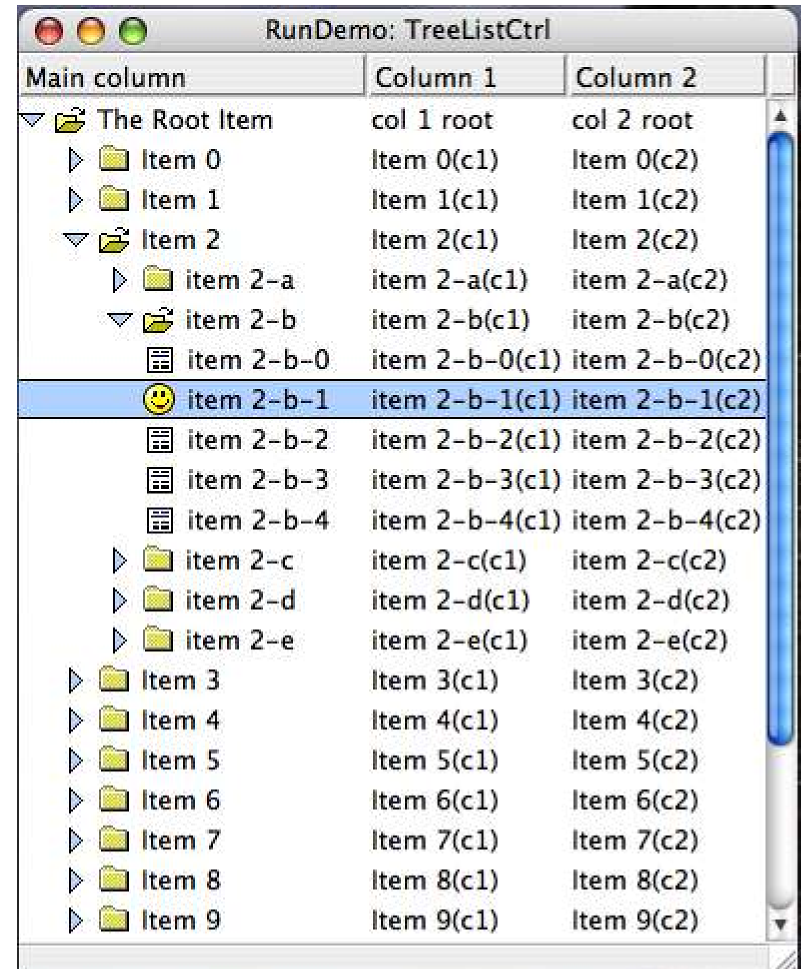
wx.gizmos.TreeListCtrl

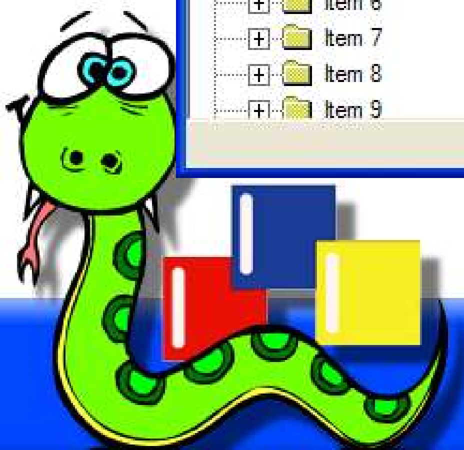
- A generic control that combines `wx.ListCtrl` and `wx.TreeCtrl`.
- Looks like a `wx.ListCtrl` with an embedded `wx.TreeCtrl`, but from a programming perspective it is a `wx.TreeCtrl` with columns.



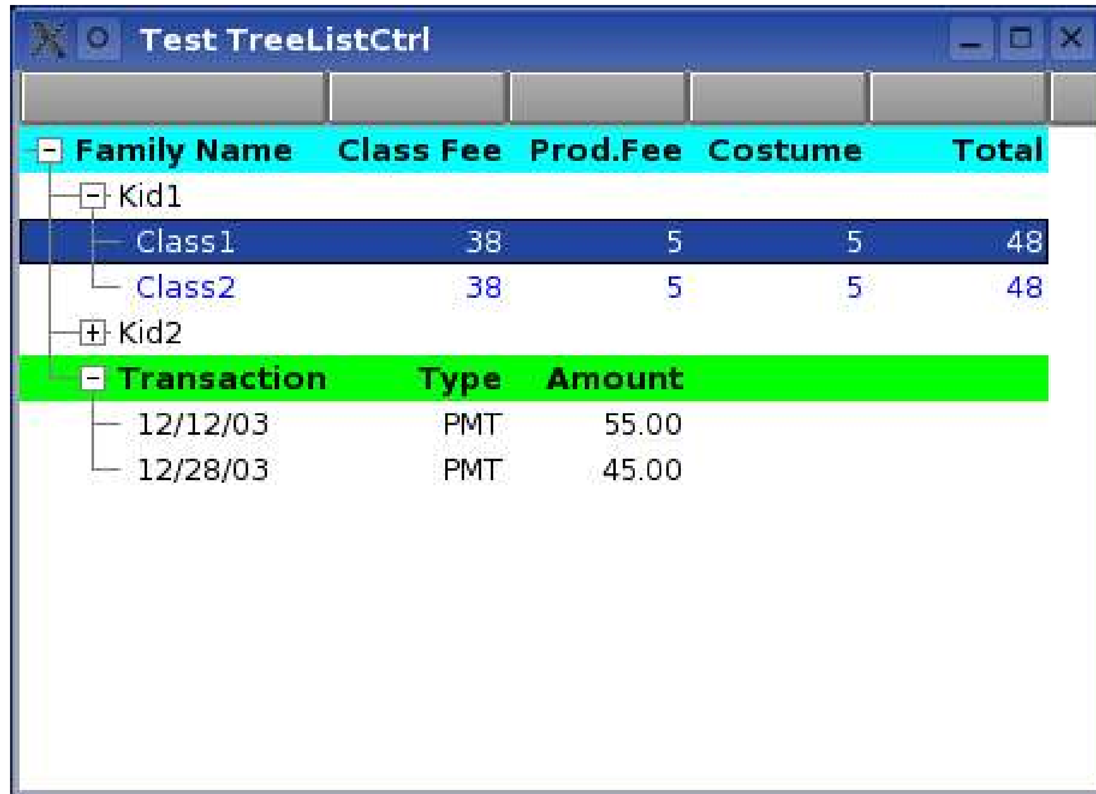
wx.gizmos.TreeListCtrl

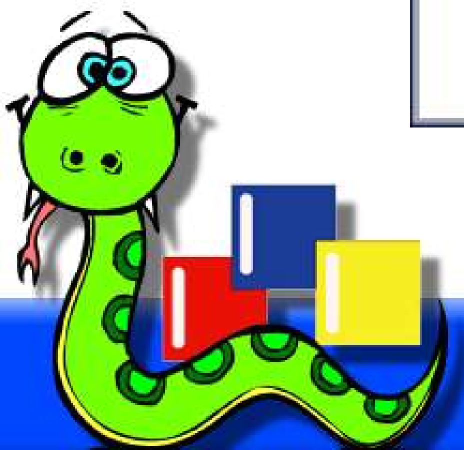




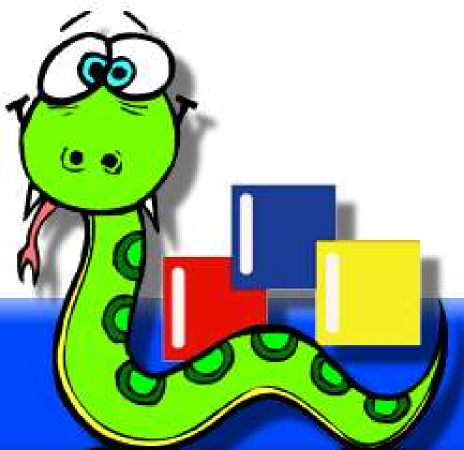


wx.gizmos.TreeListCtrl



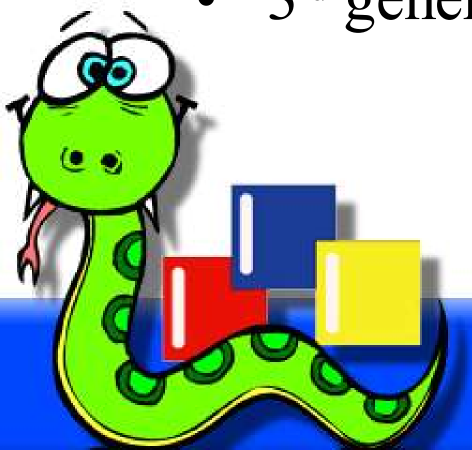


Questions?



wx.grid.Grid

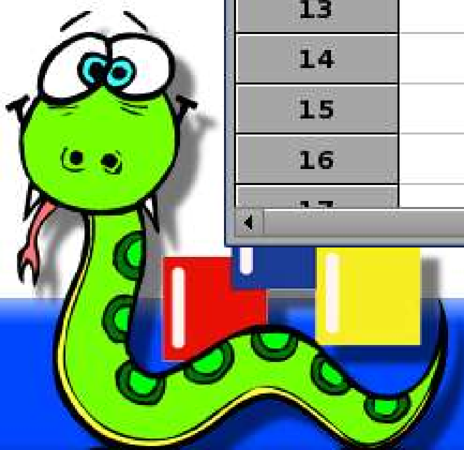
- A window implementing spreadsheet-like capabilities
- Uses a plug-in architecture where various pieces of functionality can be replaced by other classes
  - data table
  - cell editor
  - cell renderer
  - attribute provider
- Very powerful, and very complex
- 3rd generation rewrite being considered



wx.grid.Grid

Simple Grid Demo

	Custom	column	labels	D	E	F
1	*First cell*				Limited text	
2		Another cell		Hello OSCON!		
3			Yet another cell			
4				This cell is read-only		
5						
6	123					
7	123.34			You can veto editing this cell		
8						
9						
10		This default cell will overflow into neighboring cells, but not if you turn overflow off.				
11						
12		This cell is set to span 3 rows and 3 columns				
13						
14						
15						
16						



wx.grid.Grid

wx.ScrolledWindow

wx.grid.Grid

GridTableBase

GridStringTable

PyGridTableBase

GridCellAttrProvider

PyGridCellAttrProvider

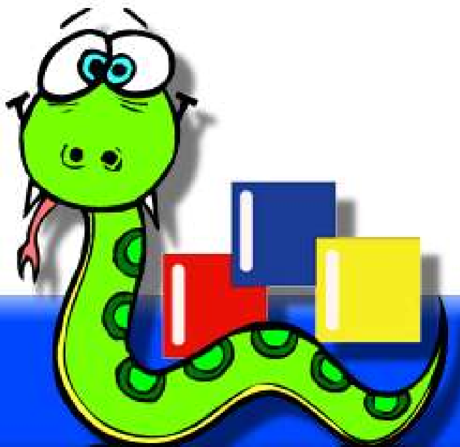
GridCellRenderer

PyGridCellRenderer

GridCellAttr

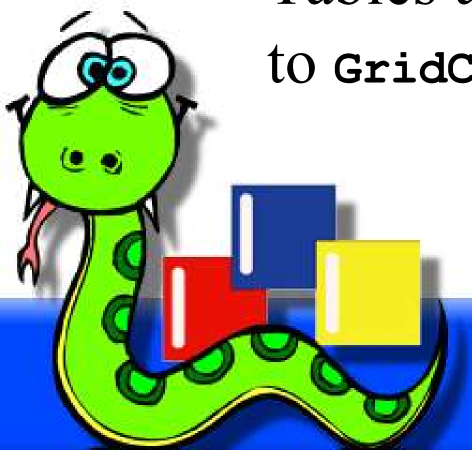
GridCellEditor

PyGridCellEditor



wx.grid.Grid

- By default uses the **`GridStringTable`** class
  - Data setters and getters in the **`Grid`** class pass through to the table
  - So default usage is conceptually similar to a non-virtual **`ListCtrl`**
- "Virtualizing" **`Grid`** is as simple as plugging in a custom table class.
  - Data for all cells is requested from the table as needed for display
  - Editors send new values to the table for update
  - Data items can be non-string types
- Tables are also the attribute provider, which passes through to **`GridCellAttrProvider`** by default.

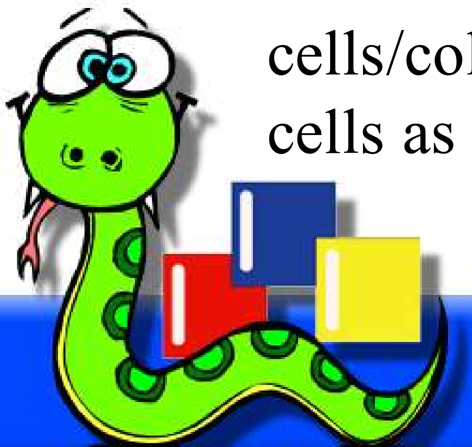


wx.grid.Grid

- `GridCellAttr` defines the look and feel for each cell.
  - Font and Colors
  - Alignment
  - Read-only/read-write
  - Overflow settings
  - row/col spanning
  - Editor
  - Renderer
- Columns and rows can have a `GridCellAttr` too, and overlaps are merged.

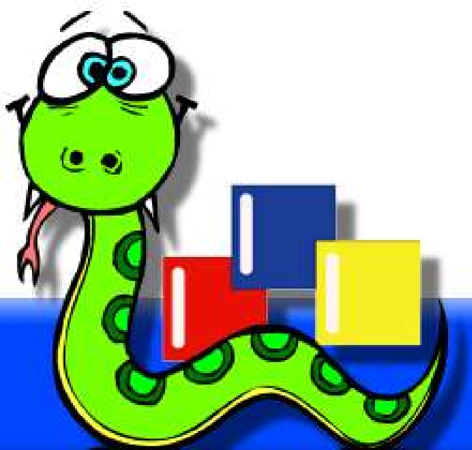
wx.grid.Grid

- Data-type specific *renderers* are used for drawing the contents of the cells.
  - Each has a Draw method that is called when the grid paints itself
- Data-type specific *editors* are used for editing the contents of the cells, each manages a single a `wx.Control`
  - Passes data between the table and the control
  - shows/hides the control as needed
  - Only one editor can be active at once.
- Editors and Renderers can be part of the attributes set for cells/cols/rows, or it an be driven from the data type of the cells as reported by the table.

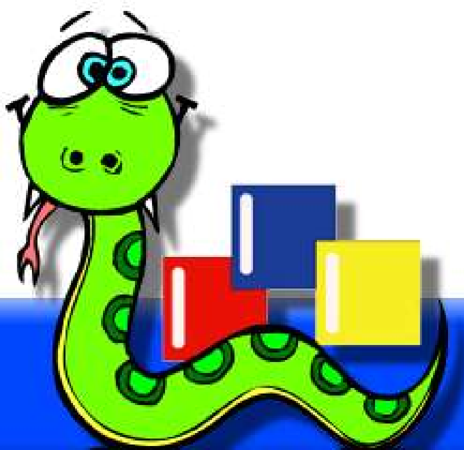


wx.grid.Grid: gotcha's

- The C++ Editor, Renderer and Attr classes use a reference counting scheme that doesn't blend well with Python's.
  - Have to manually call IncRef and DecRef methods
  - Can be very tricky to get it right.
- There are multiple types of selections, that can all be active at once.
- More help at: http://wiki.wxpython.org/index.cgi/wxGrid_20Manual

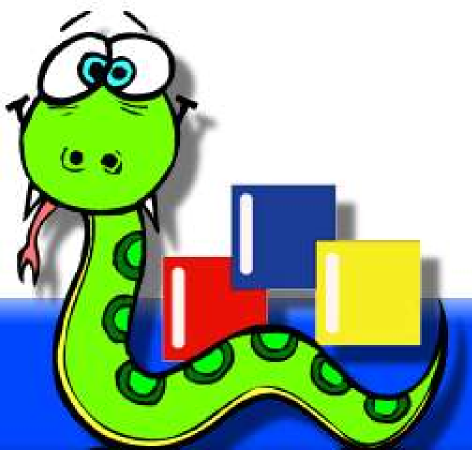


Questions?



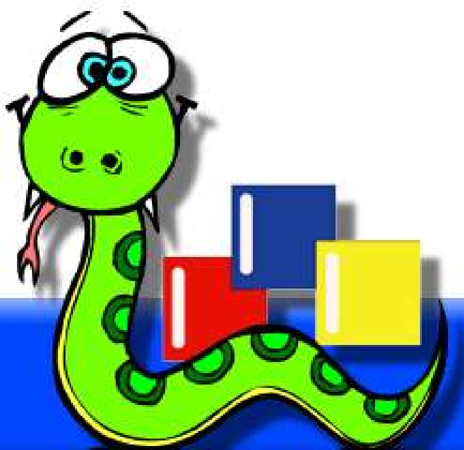
wx.lib.scrolledpanel.ScrolledPanel

- **Problem**: Sometimes you can't fit all the desired controls on the space available to a `wx.Panel`, and `wx.ScrolledWindow` doesn't automatically scroll when child windows change focus.
- A simple Python class to the rescue!
- `ScrolledPanel` uses its sizer to determine what the virtual size of the window should be, based on what the sizer calculates as its minimum size



wx.lib.scrolledpanel.ScrolledPanel

- As each child control receives the focus the scrolled panel will scroll itself enough to make the child visible.
- Just need to call `SetupScrolling` after all children and the sizer have been created.



wx.lib.scrolledpanel.ScrolledPanel

RunDemo: ScrolledPanel

File

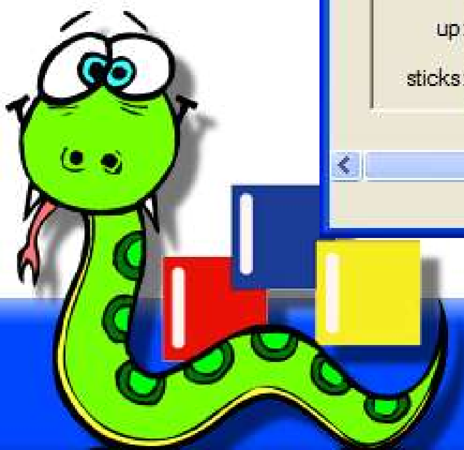
ScrolledPanel extends wx.ScrolledWindow, adding all the necessary bits to set up scroll handling for you.

Here are three fixed size examples of its use. The demo panel for this sample is also using it -- the wxStaticLine below is intentionally made too long so a scrollbar will be activated.

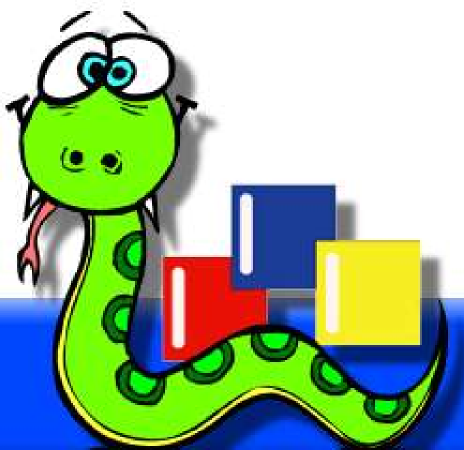
my: my
shoe: shoe
three: three
four: four
shut: shut
the: the
door: door
five: five
six: six
pick: pick
up: up
sticks: sticks

one two buckle my shoe three four shut the door five six

one two buckle my shoe
three four shut the door
five six pick up sticks

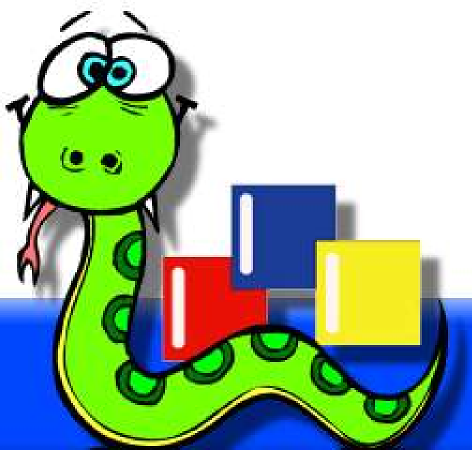


Questions?



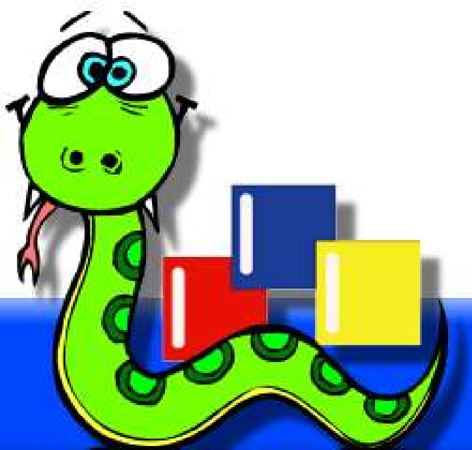
wx.HtmlWindow

- A simple HTML viewer that supports a subset of the HTML standard.
- Not meant to be a full HTML browser, but is useful for many other things:
  - fancy “About Box” and other dialogs
  - generic “rich” text viewer
  - display widget for the results of database queries
  - enhanced widget layout
  - etc.

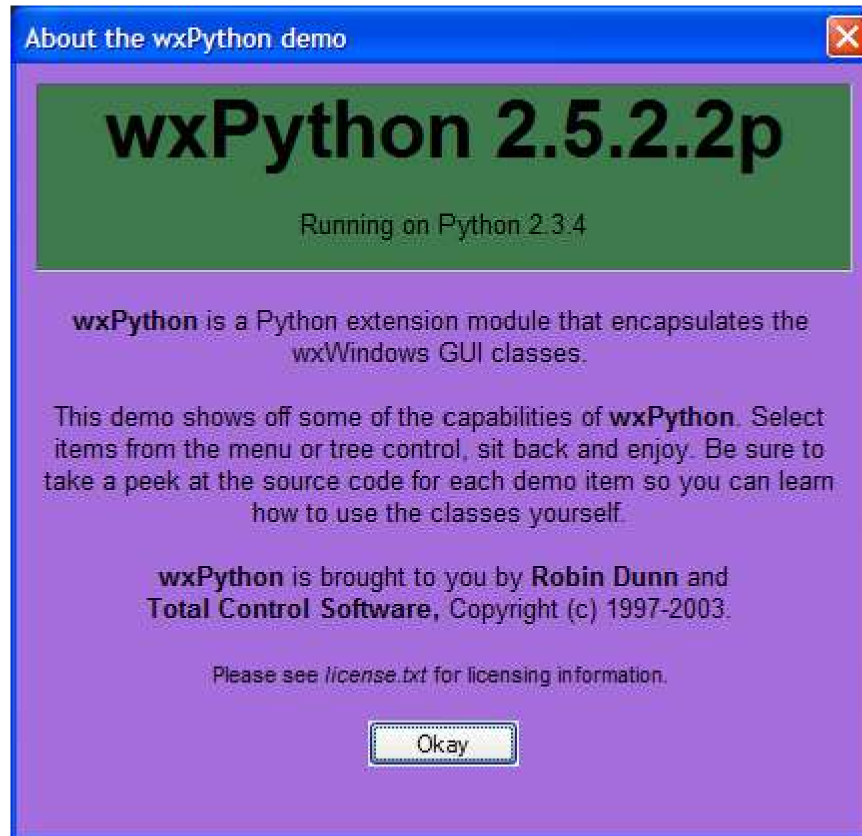


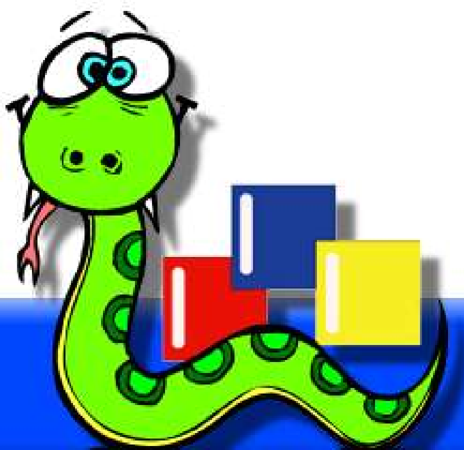
wx.HtmlWindow

- Plug-in tag handlers allow extending the abilities in various ways. (See wx/lib/wxpTag.py for an example)
- Plug-in file system handlers allow various standard and non-standard protocols specified in the URLs.
  - http, ftp
  - zip
  - mem



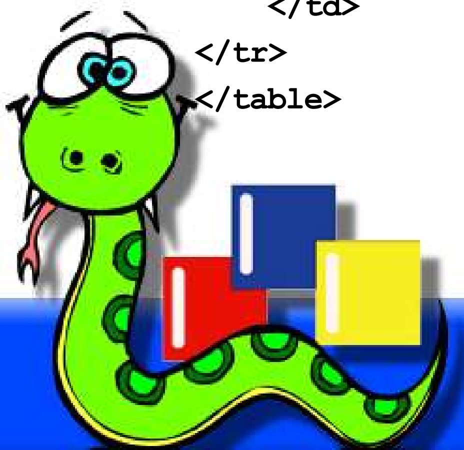
wx.HtmlWindow





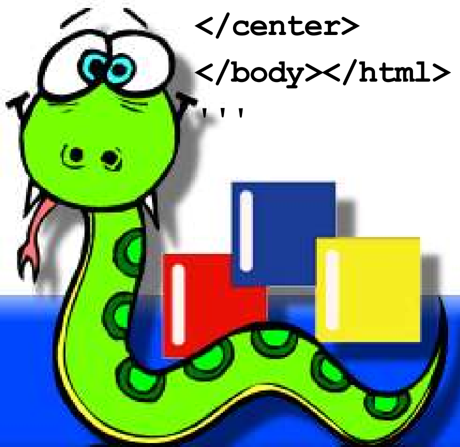
wx.HtmlWindow

```
import sys
import wx
import wx.html
import wx.lib.wxpTag

class MyAboutBox(wx.Dialog):
    text = '''
<html>
<body bgcolor="#AC76DE">
<center><table bgcolor="#458154" width="100%%" cellspacing="0"
cellpadding="0" border="1">
<tr>
    <td align="center">
    <h1>wxPython %s</h1>
    Running on Python %s<br>
    </td>
</tr>
</table>
```



wx.HtmlWindow

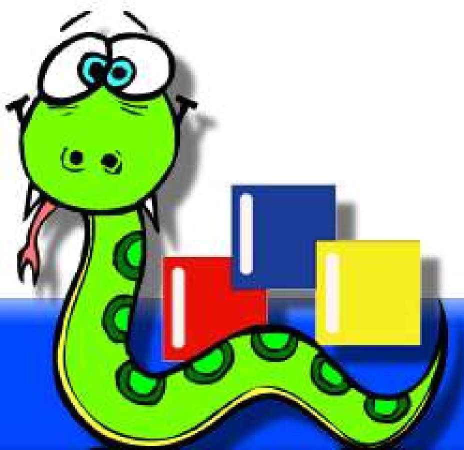
```
<p><b>wxPython</b> is a Python extension module that encapsulates the wxWindows GUI
    classes.</p>

<p>This demo shows off some of the capabilities of <b>wxPython</b>.  Select items from
    the menu or tree control, sit back and enjoy.  Be sure to take a peek at the source
    code for each demo item so you can learn how to use the classes yourself.</p>

<p><b>wxPython</b> is brought to you by <b>Robin Dunn</b> and<br>
<b>Total Control Software,</b> Copyright (c) 1997-2003.</p>

<p><font size="-1">Please see <i>license.txt</i> for licensing information.</font></p>

<p><wxp module="wx" class="Button">
    <param name="label" value="Okay">
    <param name="id"    value="ID_OK">
</wxp></p>
</center>
</body></html>
'''
```



wx.HtmlWindow

```
 def __init__(self, parent):
        wx.Dialog.__init__(self, parent, -1, 'About the wxPython demo',)
        html = wx.html.HtmlWindow(self, -1, size=(420, -1))
        py_version = sys.version.split()[0]
        html.SetPage(self.text % (wx.VERSION_STRING, py_version))
        btn = html.FindWindowById(wx.ID_OK)
        ir = html.GetInternalRepresentation()
        html.SetSize( (ir.GetWidth()+25, ir.GetHeight()+25) )
        self.SetClientSize(html.GetSize())
        self.CentreOnParent(wx.BOTH)

if __name__ == '__main__':
    app = wx.PySimpleApp()
    dlg = MyAboutBox(None)
    dlg.ShowModal()
    dlg.Destroy()
    app.MainLoop()
```

Questions?

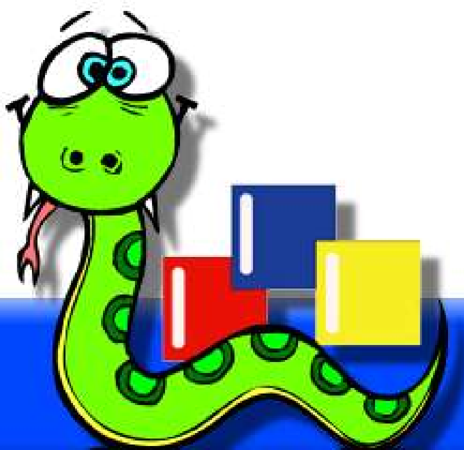


Keeping the UI updated

- When an app has many menu or toolbar items, controls, labels, etc. that can be enabled/disabled, checked/unchecked, toggled, etc. then it can be very difficult to keep them all updated as program status changes.
- `wx.UpdateUIEvent` is a mechanism to make it much simpler
- For common cases, the event handler merely checks the program state and calls a method of the event object. wx.Widgets takes care of the rest.
- Custom actions can also be done if needed.

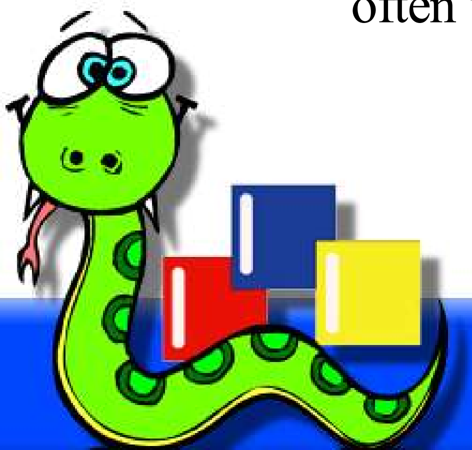
Keeping the UI updated

```
    self.Bind(wx.EVT_MENU, self.OnDoSomething, id=ID_SOMETHING)
    self.Bind(wx.EVT_UPDATE_UI, self.OnCheckSomething, id=ID_SOMETHING)

...

def OnDoSomething(self, evt):
    self.someFlag = not self.someFlag
    self.DoSomething()

def OnCheckSomething(self, evt):
    evt.Check(self.someFlag)
```

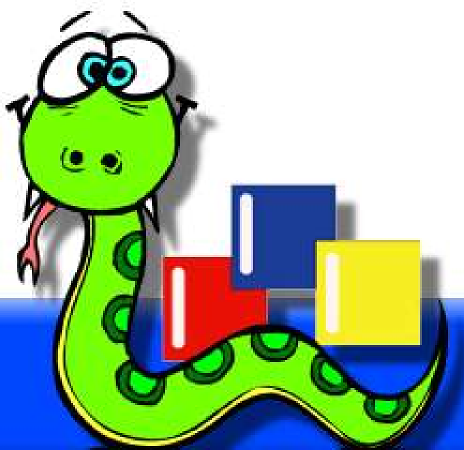


Keeping the UI updated

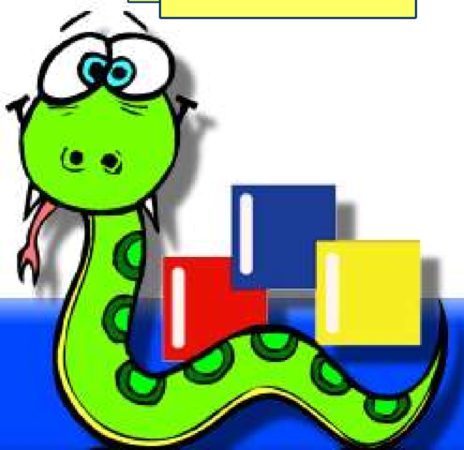
- `wx.UpdateUIEvent` is sent in idle time, before menus are shown, and at other key points.
- If the overhead of sending the events to every window becomes excessive then:
  - Call `wx.UpdateUIEvent.SetMode(wx.UPDATE_UI_PROCESS_SPECIFIED)`
  - Set the `wx.WS_EX_PROCESS_UPDATE_EVENTS` extra style for only the windows that you wish to receive the event.
  - Or call `wx.UpdateUIEvent.SetUpdateInterval` to change how often they are sent.



Questions?

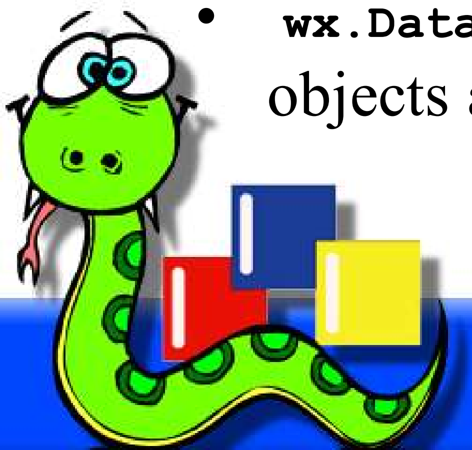


Data Transfer



Data Transfer: data objects

- Represent the data that is being transferred via the clipboard or drag and drop
- "Smart" data
  - knows what formats it supports
  - knows how to render itself in any of them
  - can defer the creation or copying of the data until needed
- Several standard data object formats are supported
- Custom formats are easily created
- `wx.DataObjectComposite` can hold one or more simple data objects and makes all of them available at once



Data Transfer: data objects

- Creating a custom data object:

```
self.format = wx.CustomDataFormat("MyFormat")
dataobject = wx.CustomDataObject(self.format)
```

- Putting a Python object in it:

```
dataobject.SetData(cPickle.dumps(myDataObject, 1))
```

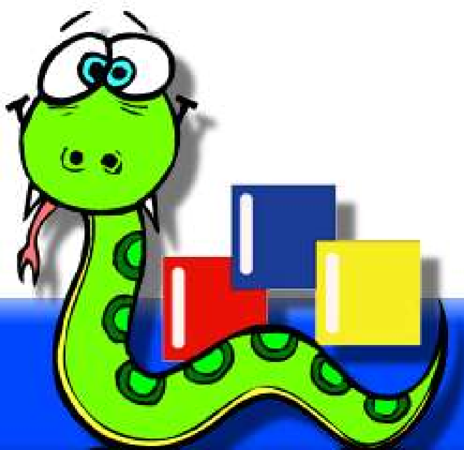
- Fetching a Python object:

```
obj = cPickle.loads(dataobject.GetData())
```

- Any app that understands "MyFormat" can transfer these data objects via the clipboard and DnD.

Data Transfer: Clipboard

- Transfers data objects via typical Cut, Copy and Paste mechanisms
- Should normally use the global `wx.TheClipboard` instance, but singleton is not enforced.
- `wx.Clipboard` uses a simple open, check/read/write, close metaphor
- Should only keep the clipboard open momentarily.



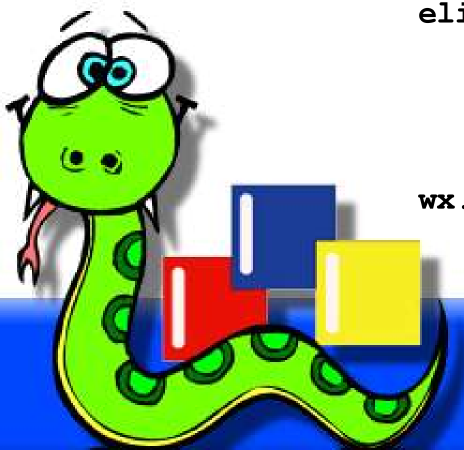
Data Transfer: Clipboard

- Write some text to the clipboard:

```
if wx.TheClipboard.Open():
    wx.TheClipboard.SetData(wx.TextDataObject("Some text"))
    wx.TheClipboard.Close()
```

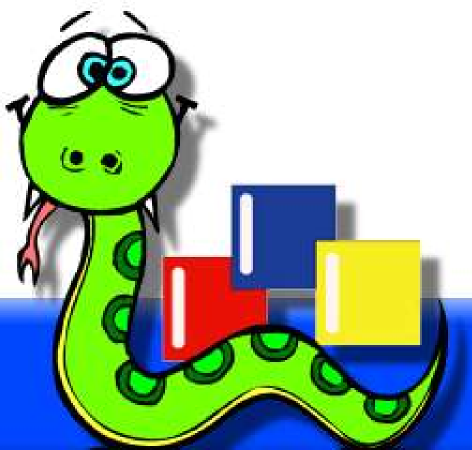
- Read either text or a custom formatted data object from the clipboard:

```
if wx.TheClipboard.Open():
    if wx.TheClipboard.IsSupported(wx.DataFormat(wx.DF_TEXT)):
        data = wx.TextDataObject()
        success = wx.TheClipboard.GetData(data)
        text = data.GetText()
    elif wx.TheClipboard.IsSupported(self.format):
        data = wx.CustomDataObject(self.format)
        success = wx.TheClipboard.GetData(data)
        obj = cPickle.loads(dataobject.GetData())
    wx.TheClipboard.Close()
```



Data Transfer: Drag and Drop

- Uses same data format and object classes as `wx.Clipboard`
- DnD fucntionality is divided into two main classes, the `wx.DropSource` and the `wx.DropTarget`
- The source is “in control” of the operation
  - provides the data object
  - specifies if it is able to be moved or just copied
  - initiates the drag and is modal until the drop is completed or canceled

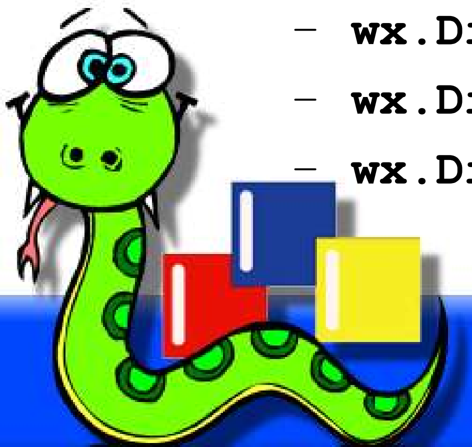


Data Transfer: Drag and Drop

- Initiating a DnD (usually in response to a mouse event):

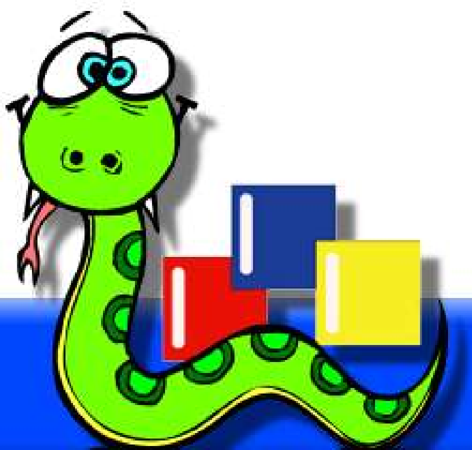
```
data = wx.TextDataObject("This is some data")
source = wx.DragSource(self)
source.SetData(data)
result = source.DoDragDrop(flags)
```

- **`DoDragDrop`** does not return until the mouse button is released
- If **`GiveFeedback`** is overridden in a class derived from **`wx.DropSource`** then it is called on every mouse move
- The flags parameter specifies what drag ops are allowed:
  - **`wx.Drag_CopyOnly`**
  - **`wx.Drag_AllowMove`**
  - **`wx.Drag_DefaultMove`**



Data Transfer: Drag and Drop

- The return value of `DoDragDrop` lets you know what happened so you can respond to it
  - `wx.DragError`
  - `wx.DragNone`
  - `wx.DragCopy`
  - `wx.DragMove`
  - `wx.DragCancel`

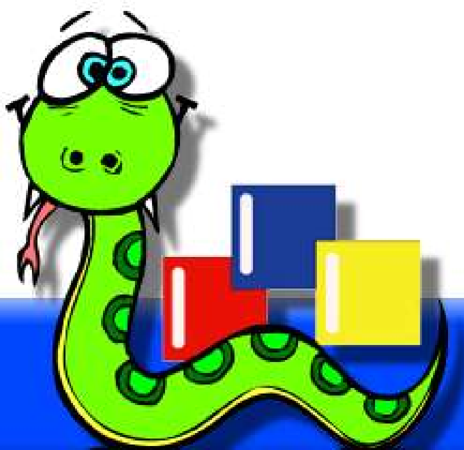


Data Transfer: Drag and Drop

- For any window to be the target of a DnD operation it must have a **`wx.DropTarget`** instance assigned to it.

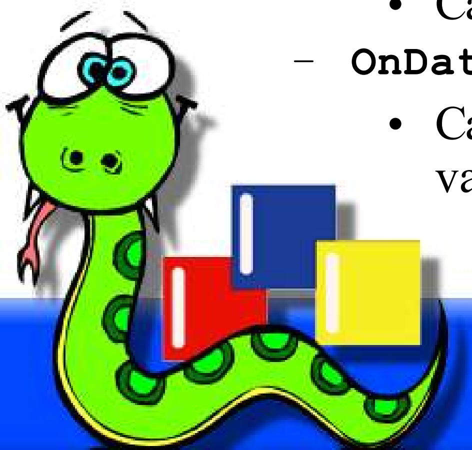
```
self.SetDropTarget(MyDropTarget)
```

- The drop target has a data object selected into it which serves to let the DnD system know what data format(s) are accepted by the target, as well as to serve as the place to fetch the data from.



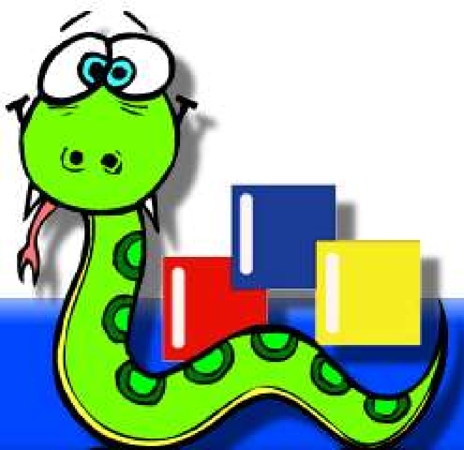
Data Transfer: Drag and Drop

- The **`wx.DropTarget`** class has several overridable methods that are used to facilitate the data transfer, and provide visual indicators. All are optional except **`OnData`**.
  - **`OnEnter(x, y, defResult)`**
    - Called when the mouse enters the target window
  - **`OnLeave()`**
    - Called when the mouse leaves the target window
  - **`OnDragOver(x, y, defResult)`**
    - Called as the mouse moves over the window, return value indicates default visual feedback to give
  - **`OnDrop(x, y)`**
    - Called when the drop happens, return **`False`** to veto the drop
  - **`OnData(x, y, defResult)`**
    - Called when **`OnDrop`** returns **`True`**. Should call **`GetData`**, return value indicates the result of the DnD.



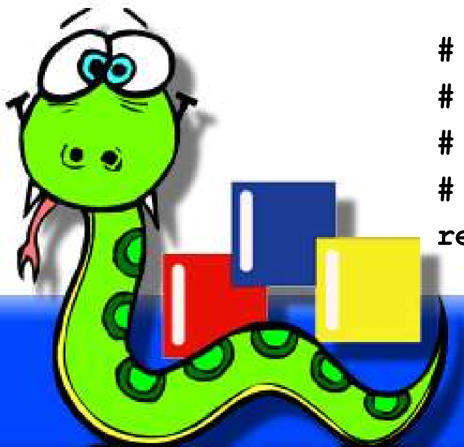
Data Transfer: Drag and Drop

- Predefined drop target classes for text and files
  - **`wx.TextDropTarget`**
    - override **`OnDropText(x, y, text)`**
  - **`wx.FileDropTarget`**
    - override **`OnDropFiles(x, y, files)`**

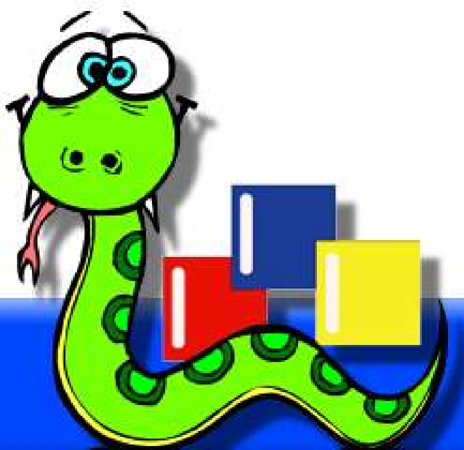


Data Transfer: Drag and Drop

```
class MyDropTarget(wx.PyDropTarget):
    def __init__(self, window):
        wx.PyDropTarget.__init__(self)
        self.window = window
        format = wx.CustomDataFormat("MyFormat")
        self.data = wx.CustomDataObject(format)
        self.SetDataObject(self.data)

    def OnData(self, x, y, defResult):
        # copy the data from the drag source to my data object
        if self.GetData():
            # convert the data object and do something with it
            datastr = self.data.GetData()
            obj = cPickle.loads(datastr)
            self.window.DoSomethingWithDroppedObject(obj)

        # What is returned signals the source what to do with the
        # original data object (if it was a move then the original
        # should be removed, etc.)  In most cases just return the
        # default passed to us
        return defResult
```

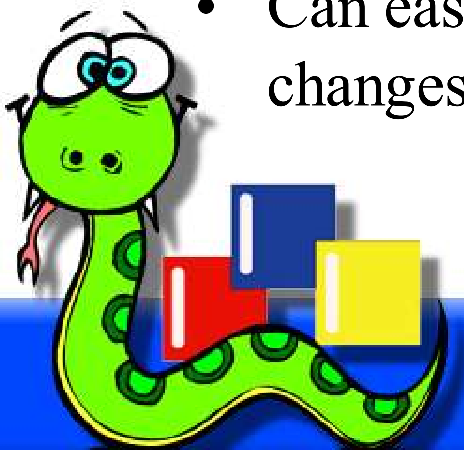


Questions?



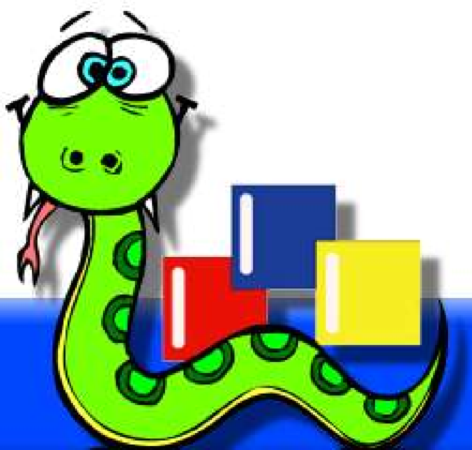
Sizers

- A widget layout mechanism that calculates the size and position of widgets under its control based on the minimal size required by each widget, the available space, and a sizer-specific algorithm.
- Adapts layouts to the needs of different platforms with no changes in the programmer's code.
- Automatically recalculates the layout when the container window changes size.
- Can easily manage recalculating layout when a widget changes state (perhaps a new font or label)



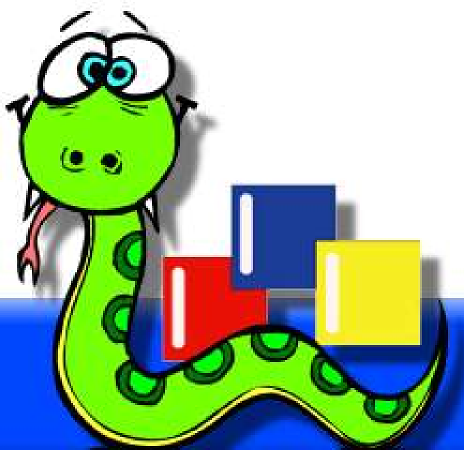
Sizers

- Some folks need a bit of a paradigm shift to understand sizers, but once they do it becomes much easier for them and they are able to do any kind of layout they need.
- Sizers can be nested.
- Windows with sizers of their own can also be managed by their parent window's sizer (IOW, a panel on a panel) and the sizers do the RightThing.



Sizers

- Each item managed by the sizer is allotted a certain amount of space, and items can be variously aligned within that space, or expanded to fill it.
- Items can have empty border space on any or all sides



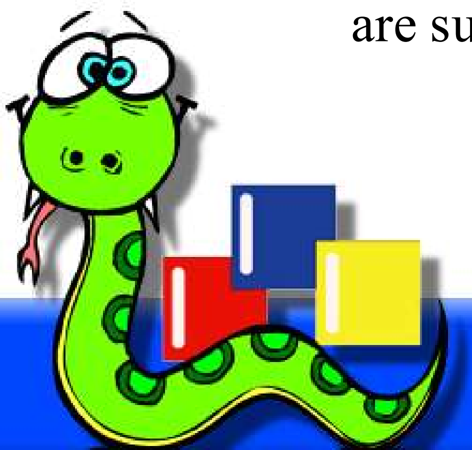
Sizers: how do they work?

- Most controls know their minimum "best size", and the sizers query that value to determine the defaults of the layout.
- The size that a control is created with can override the "best size" and and it can also be explicitly set with **`window.SetMinSize`** or **`window.SetSizeHints`**. [*]
- Most controls will adjust their "best size" if attributes of the control change, such as the label or the font.

- [*] This is a change from earlier versions where the size of the window when it was added to the sizer took precedence.

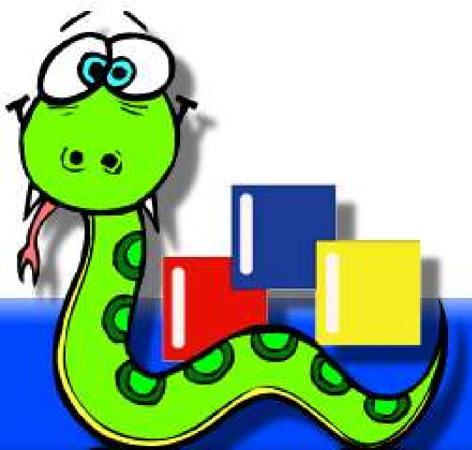
Sizers: how do they work?

- The "best size" of non-control windows is determined from the window's sizer if it has one, otherwise it is large enough to show all the window's children at their current size. If there are no children then it is the window's minsize if it has one, or the current size.
- When a sizer's `Layout` method is called it will:
  - determine the minsize of all items under its control
  - adjust the size and position of all items according to its layout algorithm, which will also recursivly do the same for any items that are sub-sizers



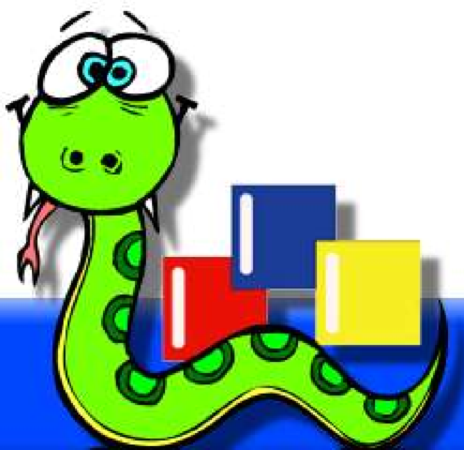
Sizers: how do they work?

- Windows that have a sizer will call **`Layout`** in the default **`EVT_SIZE`** handler, so whenever the size is changed the layout of child widgets is rechecked.
  - This also means that if a child window managed by a sizer has its own sizer then adjusting the size of the child will cause **`Layout`** of the child's sizer to be called and recursively do the layout of the grandchild widgets, etc.
- The order items are added to the sizer is (usually) significant.



Sizers: wx.GridSizer

- In a `wx.GridSizer` all *cells* are the same size, which will default to the size of the largest item.
- All the items are positioned within their cells as defined by the alignment and border flags, if any.
- A gap between cells for rows and columns can be specified
- Always expands the cells to take all space given to the sizer



Sizers: wx.GridSizer

GridSizer Test

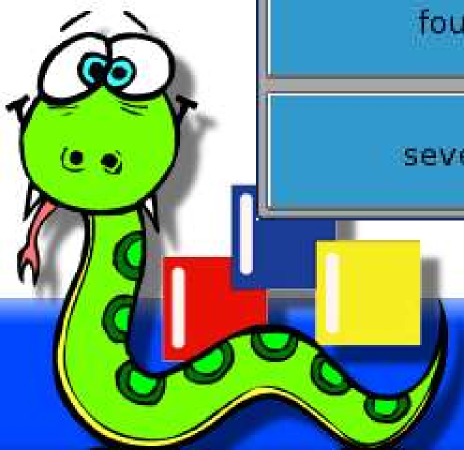
one	two	three
four	five	six
seven	eight	nine

GridSizer Test <2>

one	two	three
four	five	six
seven	eight	nine

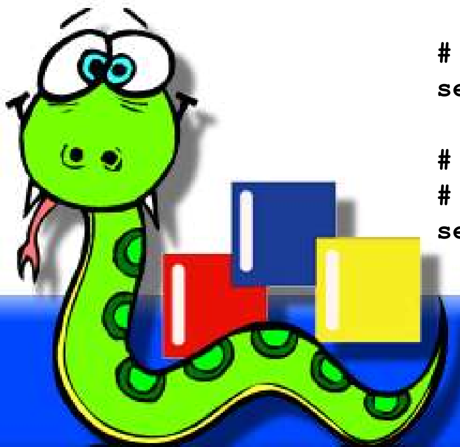
GridSizer Test <3>

one	two	three
four	five	six
seven	eight	nine



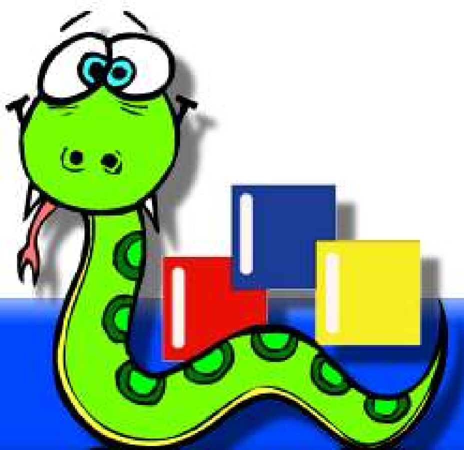
Sizers: wx.GridSizer

```
labels = "one two three four five six seven eight nine".split()
class TestFrame(wx.Frame):
    def __init__(self, makeLarger=False, useExpand=False):
        wx.Frame.__init__(self, None, -1, "GridSizer Test")
        if useExpand:
            flag = wx.EXPAND
        else:
            flag = 0

        # Create the sizer
        sizer = wx.GridSizer(rows=3, cols=3, hgap=5, vgap=5)

        # Create some block windows
        for label in labels:
            bw = BlockWindow(self, label=label)
            sizer.Add(bw, 0, flag)

        if makeLarger:
            center = self.FindWindowByName("five")
            center.SetMinSize((150,50))

        # Tell this window to use the sizer for layout
        self.SetSizer(sizer)

        # Change the size of the window to be the minimum
        # needed by the sizer
        self.Fit()
```

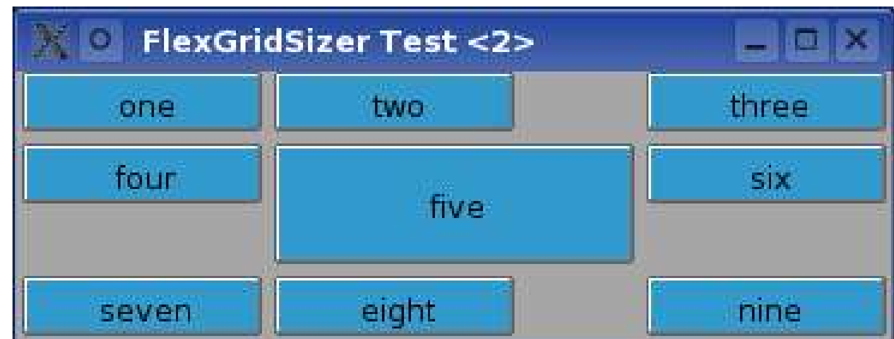


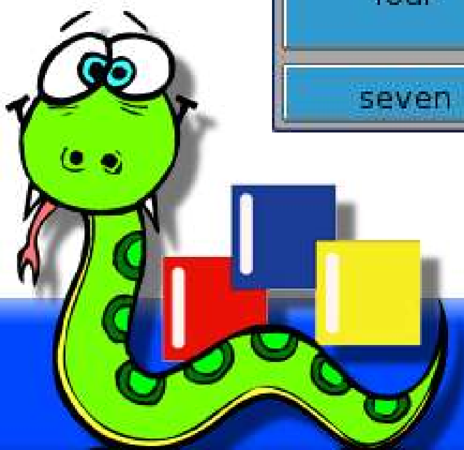
Sizers: wx.FlexGridSizer

- Derives from **`wx.GridSizer`**, but not all cells are required to be the same size.
- All cells in each row are as tall as the tallest item in that row
- All cells in each column are as wide as the widest item in that column
- Rows and columns are not stretchable by default, but you can specify which rows and columns should stretch when there is additional space available.



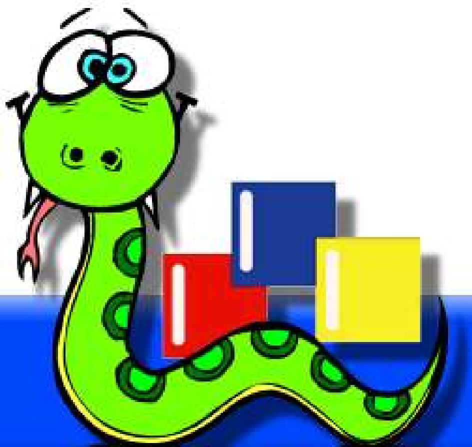
Sizers: wx.FlexGridSizer





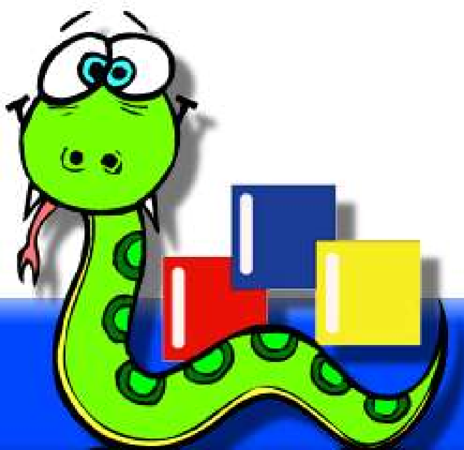
Sizers: wx.FlexGridSizer

```
class TestFrame(wx.Frame):
    def __init__(self, makeLarger=False, useExpand=False):
        wx.Frame.__init__(self, None, -1, "FlexGridSizer Test")
        if useExpand:
            flag = wx.EXPAND
        else:
            flag = 0

        # Create the sizer
        sizer = wx.FlexGridSizer(rows=3, cols=3, hgap=5, vgap=5)

        # Create some block windows
        for label in labels:
            bw = BlockWindow(self, label=label)
            sizer.Add(bw, 0, flag)

        if makeLarger:
            center = self.FindWindowByName("five")
            center.SetMinSize((150,50))
            sizer.AddGrowableCol(1)
            sizer.AddGrowableRow(1)

        # Tell this window to use the sizer for layout
        self.SetSizer(sizer)

        # Change the size of the window to be the minimum
        # needed by the sizer
        self.Fit()
```

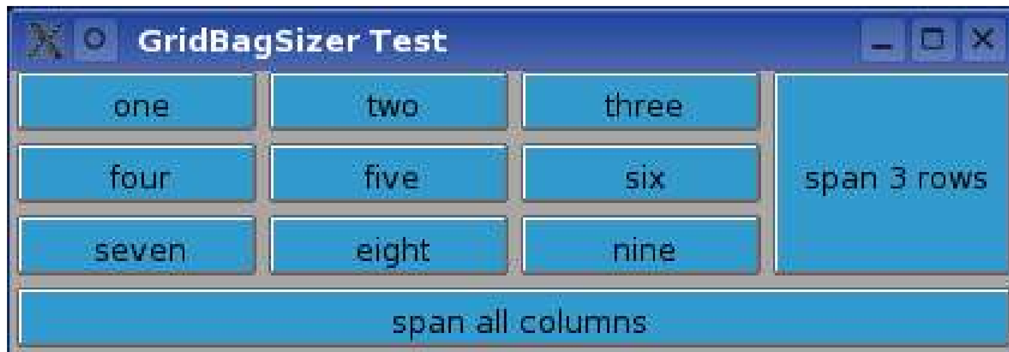


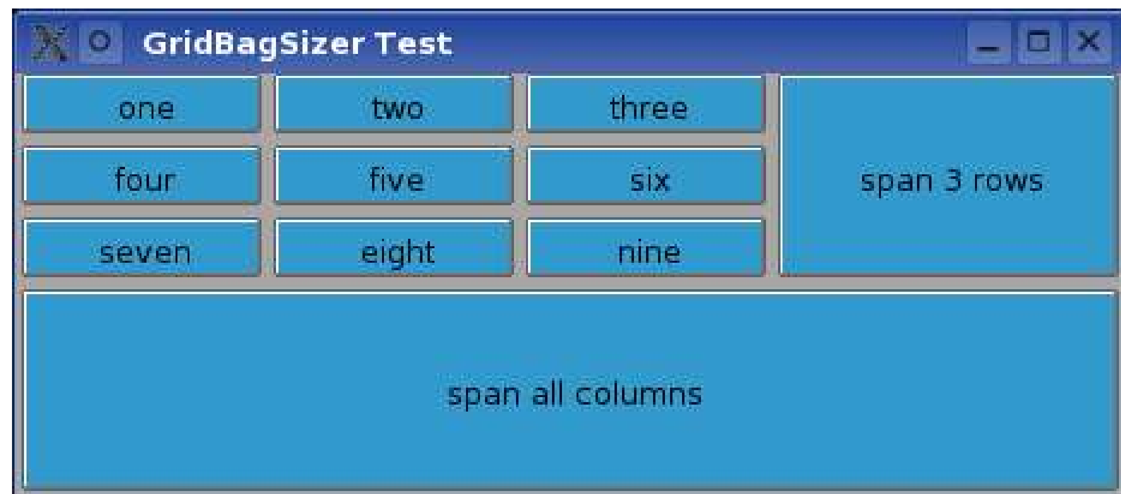
Sizers: wx.GridBagSizer

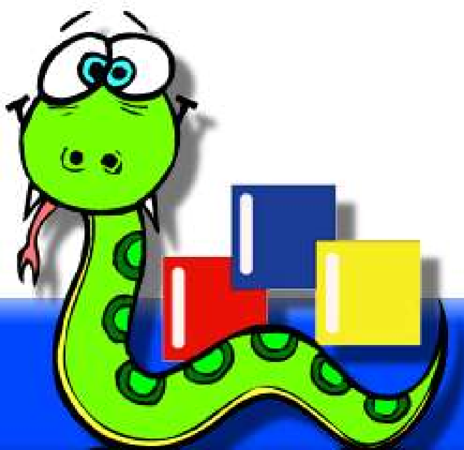
- Derives from **`wx.FlexGridSizer`**
- Also lays out items in a virtual grid, but in this case items are positioned at the specific *cell* specified in the **`Add`** method.
- This means that the order that items are added is not significant as with the other sizers.
- Items can span rows or columns.



Sizers: wx.GridBagSizer

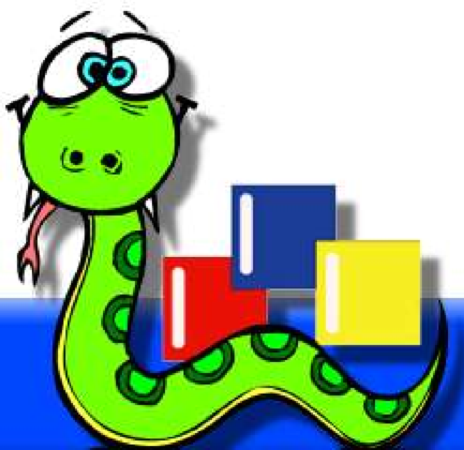






Sizers: wx.GridBagSizer

```
class TestFrame(wx.Frame):
    def __init__(self):
        wx.Frame.__init__(self, None, -1, "GridBagSizer Test")

        # Create the sizer
        sizer = wx.GridBagSizer(hgap=5, vgap=5)

        # Create some block windows in a basic grid
        for col in range(3):
            for row in range(3):
                bw = BlockWindow(self, label=labels[row*3 + col])
                sizer.Add(bw, pos=(row,col))
```

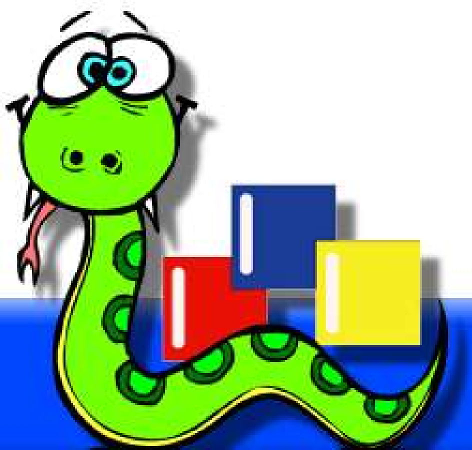


Sizers: wx.GridBagSizer

```
# add a window that spans several rows
bw = BlockWindow(self, label="span 3 rows")
sizer.Add(bw, pos=(0,3), span=(3,1), flag=wx.EXPAND)

# add a window that spans all columns
bw = BlockWindow(self, label="span all columns")
sizer.Add(bw, pos=(3,0), span=(1,4), flag=wx.EXPAND)

# make the last row and col be stretchable
sizer.AddGrowableCol(3)
sizer.AddGrowableRow(3)

# Tell this window to use the sizer for layout
self.SetSizer(sizer)

# Change the size of the window to be the minimum
# needed by the sizer
self.Fit()
```

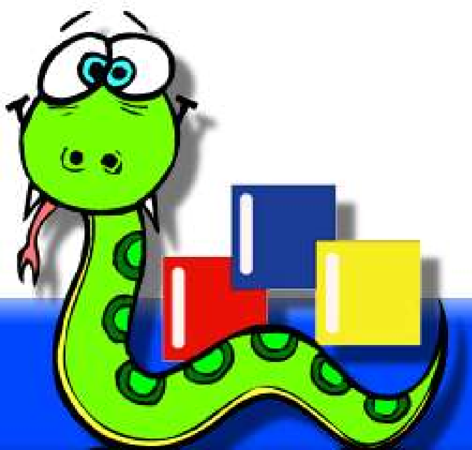
Sizers: wx.BoxSizer

- The `wx.BoxSizer` simply lays out its items in either a horizontal row, or a vertical stack. This is the sizer's primary dimension.
- Items can be added such that they get only their minimal size needed, or a proportion of the available free space.
- Items can expand to fill all available space in the alternate dimension.



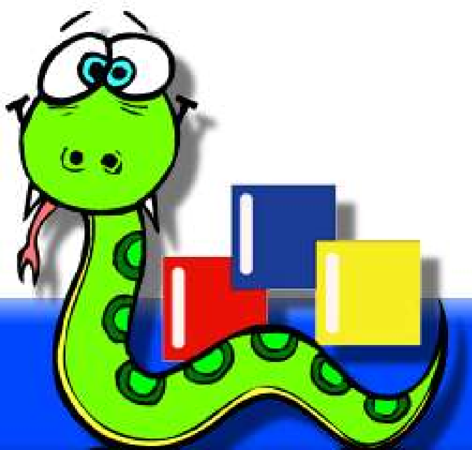
Sizers: wx.BoxSizer

```
# Create the sizer
sizer = wx.BoxSizer(wx.VERTICAL)

# Create some block windows
for label in labels:
    bw = BlockWindow(self, label=label, size=(200,30))
    sizer.Add(bw, flag=wx.EXPAND)
```



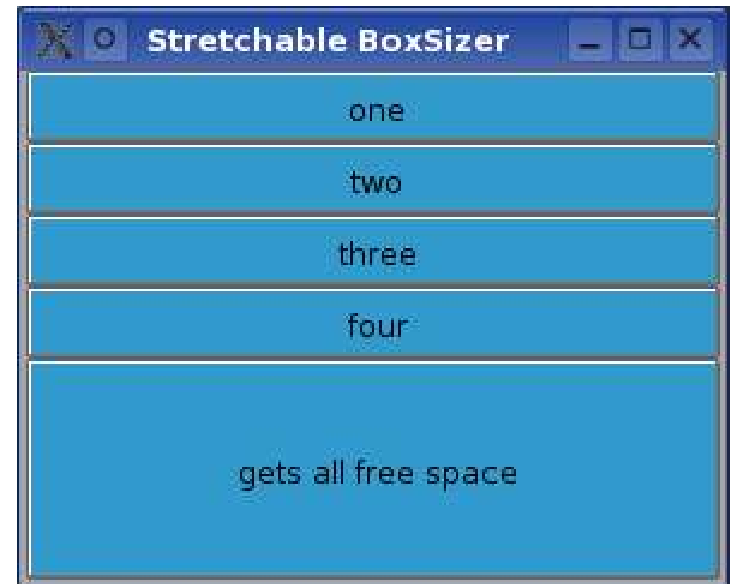
Sizers: wx.BoxSizer

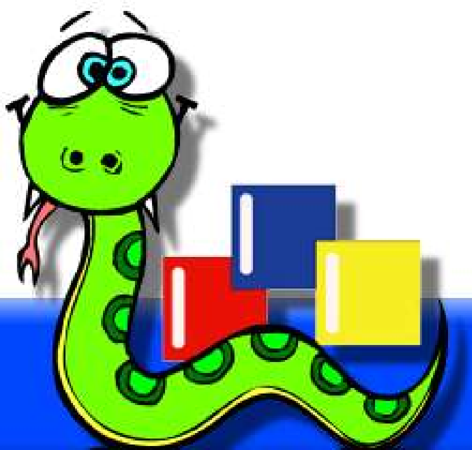
```
# Create the sizer
sizer = wx.BoxSizer(wx.HORIZONTAL)

# Create some block windows
for label in labels:
    bw = BlockWindow(self, label=label, size=(75,30))
    sizer.Add(bw, flag=wx.EXPAND)
```



Sizers: wx.BoxSizer

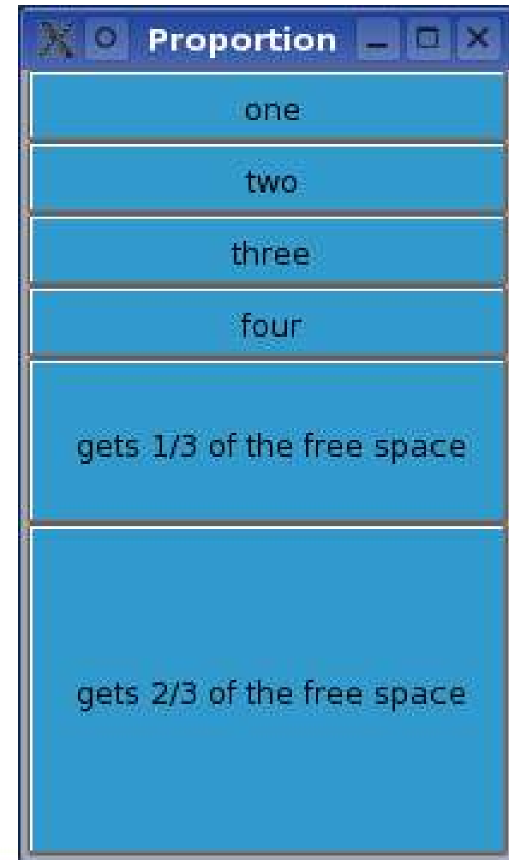
```
# Create the sizer
sizer = wx.BoxSizer(wx.VERTICAL)

# Create some block windows
for label in labels:
    bw = BlockWindow(self, label=label, size=(200,30))
    sizer.Add(bw, flag=wx.EXPAND)

# Add an item that takes all the free space
bw = BlockWindow(self,
        label="gets all free space",
        size=(200,30))
sizer.Add(bw, 1, flag=wx.EXPAND)
```

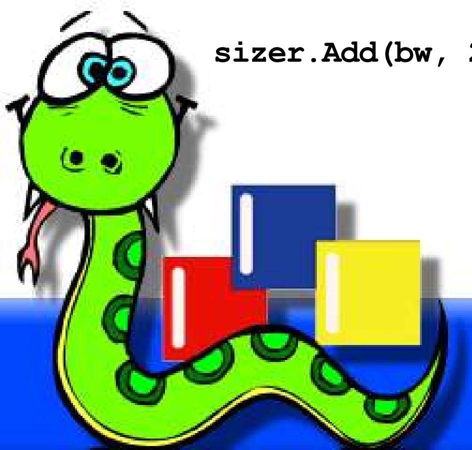




Sizers: wx.BoxSizer

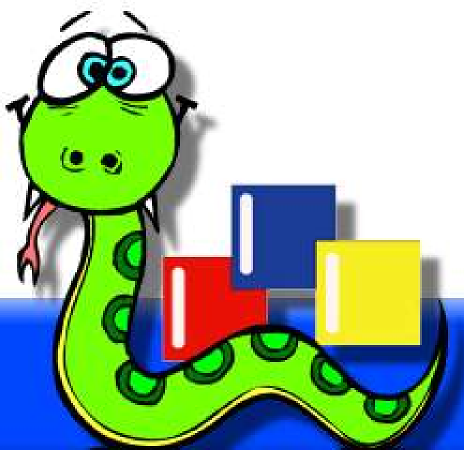
```
# Create the sizer
sizer = wx.BoxSizer(wx.VERTICAL)

# Create some block windows
for label in labels:
    bw = BlockWindow(self, label=label, size=(200,30))
    sizer.Add(bw, flag=wx.EXPAND)

# Add an item that takes one share of the free space
bw = BlockWindow(self, label="gets 1/3 of the free space",
                 size=(200,30))
sizer.Add(bw, 1, flag=wx.EXPAND)

# Add an item that takes 2 shares of the free space
bw = BlockWindow(self, label="gets 2/3 of the free space",
                 size=(200,30))
sizer.Add(bw, 2, flag=wx.EXPAND)
```



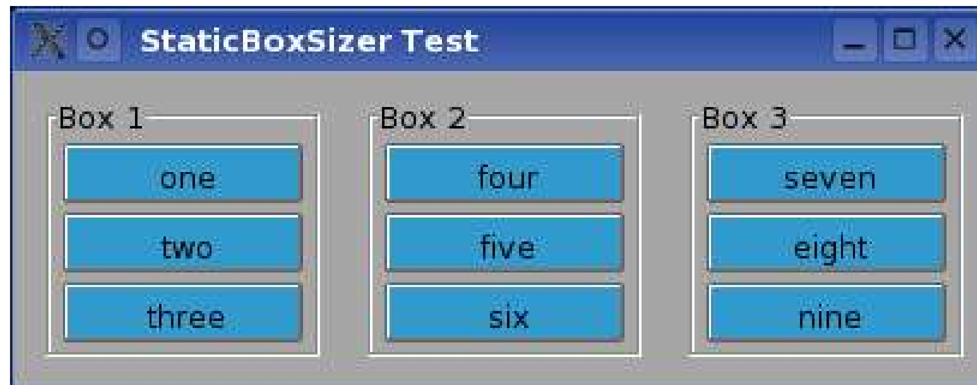


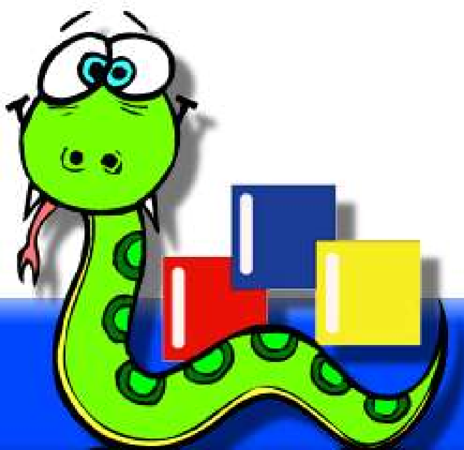
Sizers: wx.StaticBoxSizer

- A `wx.StaticBoxSizer` is exactly the same as a `wx.BoxSizer`, with the addition that a `wx.StaticBox` is positioned such that it acts as the border around the items that are managed by the sizer



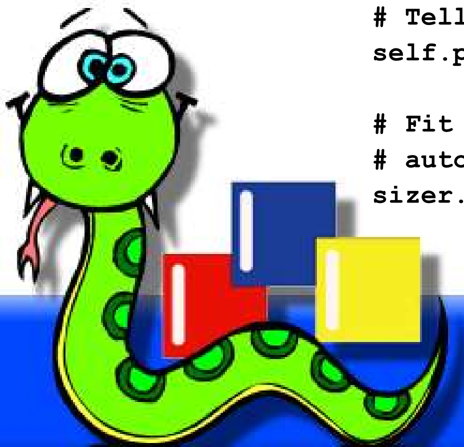
Sizers: wx.StaticBoxSizer





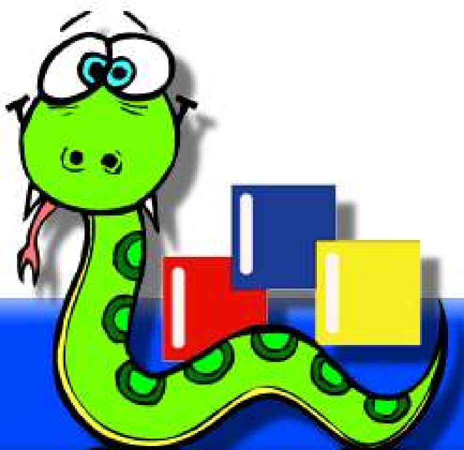
Sizers: wx.StaticBoxSizer

```
class TestFrame(wx.Frame):
    def __init__(self):
        wx.Frame.__init__(self, None, -1, "StaticBoxSizer Test")

        # make a panel this time as wx.StaticBox looks best on one.
        self.panel = wx.Panel(self)

        # make three static boxes with windows positioned inside them
        box1 = self.MakeStaticBoxSizer("Box 1", labels[0:3])
        box2 = self.MakeStaticBoxSizer("Box 2", labels[3:6])
        box3 = self.MakeStaticBoxSizer("Box 3", labels[6:9])

        # We can also use a sizer to manage the placement of other
        # sizers (and therefore the windows and sub-sizers that they
        # manage as well.)
        sizer = wx.BoxSizer(wx.HORIZONTAL)
        sizer.Add(box1, 0, wx.ALL, 10)
        sizer.Add(box2, 0, wx.ALL, 10)
        sizer.Add(box3, 0, wx.ALL, 10)

        # Tell the panel to use the sizer for layout
        self.panel.SetSizer(sizer)

        # Fit the frame to the needs of the sizer.  The frame will
        # automatically resize the panel as needed.
        sizer.Fit(self)
```

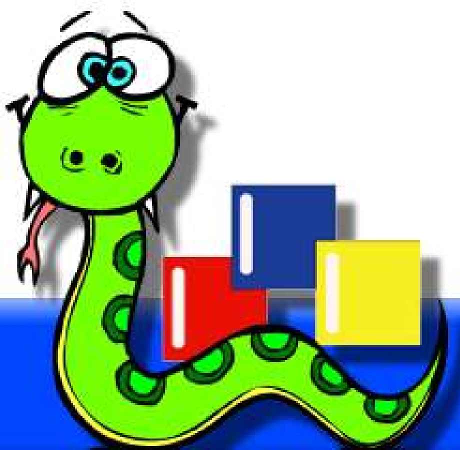


Sizers: wx.StaticBoxSizer

```
def MakeStaticBoxSizer(self, boxlabel, itemlabels):
       # first the static box
       box = wx.StaticBox(self.panel, -1, boxlabel)

       # then the sizer
       sizer = wx.StaticBoxSizer(box, wx.VERTICAL)

       # then add items to it like normal
       for label in itemlabels:
           bw = BlockWindow(self.panel, label=label)
           sizer.Add(bw, 0, wx.ALL, 2)

       return sizer
```

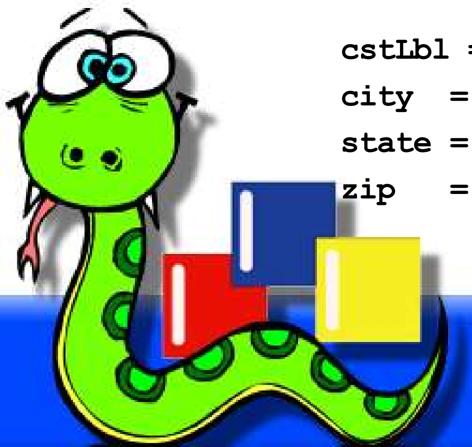


Sizers: real-world example



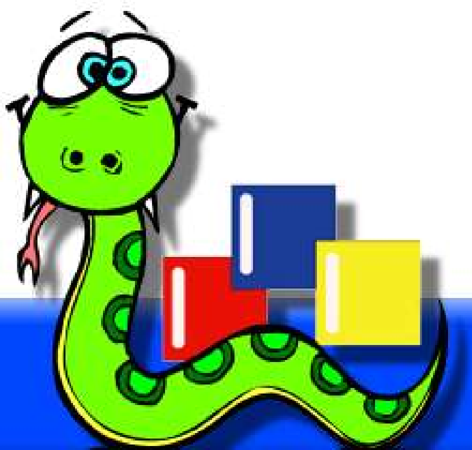
Sizers: real-world example

```
class TestFrame(wx.Frame):
    def __init__(self):
        wx.Frame.__init__(self, None, -1, "Real World Test")
        panel = wx.Panel(self)

        # First create the controls
        topLbl = wx.StaticText(panel, -1, "Account Information")
        topLbl.SetFont(wx.Font(18, wx.SWISS, wx.NORMAL, wx.BOLD))

        nameLbl = wx.StaticText(panel, -1, "Name:")
        name = wx.TextCtrl(panel, -1, "");

        addrLbl = wx.StaticText(panel, -1, "Address:")
        addr1 = wx.TextCtrl(panel, -1, "");
        addr2 = wx.TextCtrl(panel, -1, "");

        cstLbl = wx.StaticText(panel, -1, "City, State, Zip:")
        city  = wx.TextCtrl(panel, -1, "", size=(150,-1));
        state = wx.TextCtrl(panel, -1, "", size=(50,-1));
        zip   = wx.TextCtrl(panel, -1, "", size=(70,-1));
```



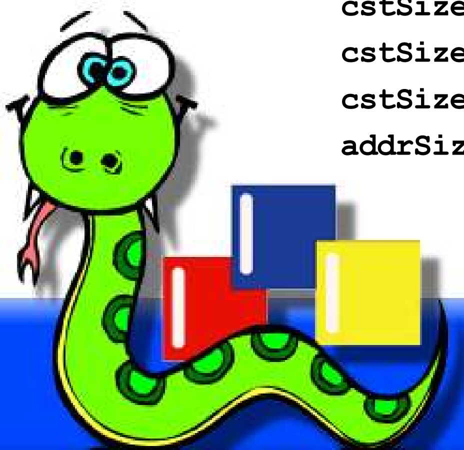
Sizers: real-world example

```
phoneLbl = wx.StaticText(panel, -1, "Phone:")
phone = wx.TextCtrl(panel, -1, "");

emailLbl = wx.StaticText(panel, -1, "Email:")
email = wx.TextCtrl(panel, -1, "");

saveBtn = wx.Button(panel, -1, "Save")
cancelBtn = wx.Button(panel, -1, "Cancel")

# Now do the layout.
# mainSizer is the top-level one that manages everything
mainSizer = wx.BoxSizer(wx.VERTICAL)
mainSizer.Add(topLbl, 0, wx.ALL, 5)
mainSizer.Add(wx.StaticLine(panel), 0, wx.EXPAND|wx.TOP|wx.BOTTOM, 5)
```



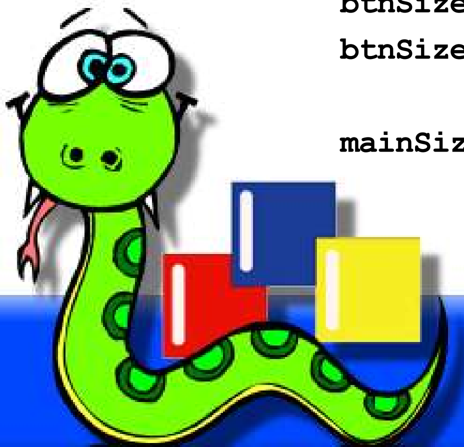
Sizers: real-world example

```
# addrSizer is a grid that holds all of the address info
addrSizer = wx.FlexGridSizer(cols=2, hgap=5, vgap=5)
addrSizer.AddGrowableCol(1)
addrSizer.Add(nameLbl, 0, wx.ALIGN_RIGHT|wx.ALIGN_CENTER_VERTICAL)
addrSizer.Add(name, 0, wx.EXPAND)
addrSizer.Add(addrLbl, 0, wx.ALIGN_RIGHT|wx.ALIGN_CENTER_VERTICAL)
addrSizer.Add(addr1, 0, wx.EXPAND)
addrSizer.Add((10,10)) # some empty space
addrSizer.Add(addr2, 0, wx.EXPAND)

addrSizer.Add(cstLbl, 0, wx.ALIGN_RIGHT|wx.ALIGN_CENTER_VERTICAL)

# the city, state, zip fields are in a sub-sizer
cstSizer = wx.BoxSizer(wx.HORIZONTAL)
cstSizer.Add(city, 1)
cstSizer.Add(state, 0, wx.LEFT|wx.RIGHT, 5)
cstSizer.Add(zip)
addrSizer.Add(cstSizer, 0, wx.EXPAND)
```



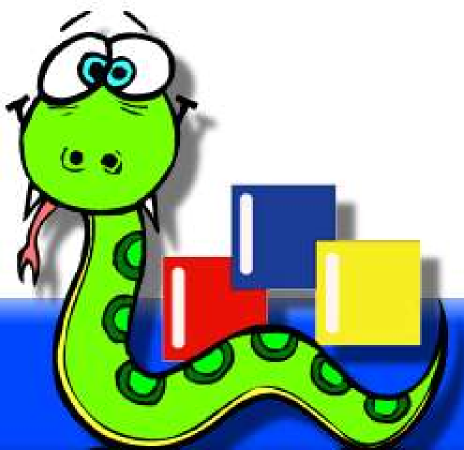
Sizers: real-world example

```
addrSizer.Add(phoneLbl, 0, wx.ALIGN_RIGHT|wx.ALIGN_CENTER_VERTICAL)
addrSizer.Add(phone, 0, wx.EXPAND)
addrSizer.Add(emailLbl, 0, wx.ALIGN_RIGHT|wx.ALIGN_CENTER_VERTICAL)
addrSizer.Add(email, 0, wx.EXPAND)

# now add the addrSizer to the mainSizer
mainSizer.Add(addrSizer, 0, wx.EXPAND|wx.ALL, 10)

# The buttons sizer will put them in a row with resizeable
# gaps between and on either side of the buttons
btnSizer = wx.BoxSizer(wx.HORIZONTAL)
btnSizer.Add((20,20), 1)
btnSizer.Add(saveBtn)
btnSizer.Add((20,20), 1)
btnSizer.Add(cancelBtn)
btnSizer.Add((20,20), 1)

mainSizer.Add(btnSizer, 0, wx.EXPAND|wx.BOTTOM, 10)
```

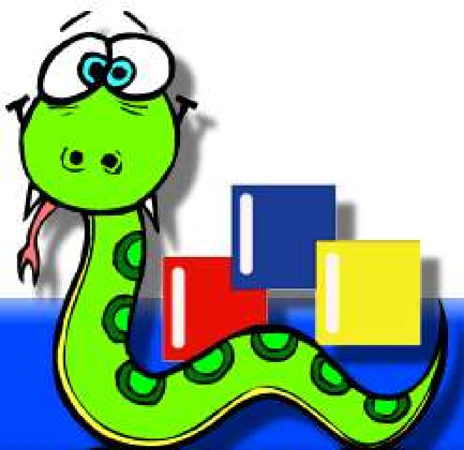


Sizers: real-world example

```
# Finally, tell the panel to use the sizer for layout
panel.SetSizer(mainSizer)

# Fit the frame to the needs of the sizer.  The frame will
# automatically resize the panel as needed.  Also prevent the
# frame from getting smaller than this size.
mainSizer.Fit(self)
self.SetSizeHints(self.GetSize())
```

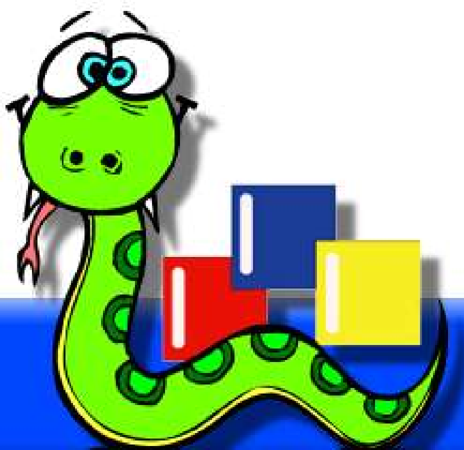


Questions?



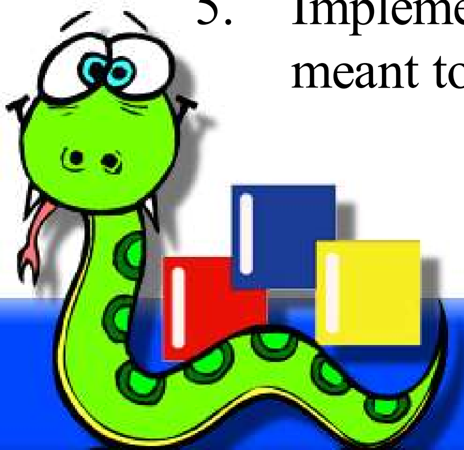
Creating Custom Widgets

- Making custom controls fit well with wxWidgets takes just a few easy steps beyond deriving a new class and implementing behaviors.
- See wx/lib/buttons.py for an example

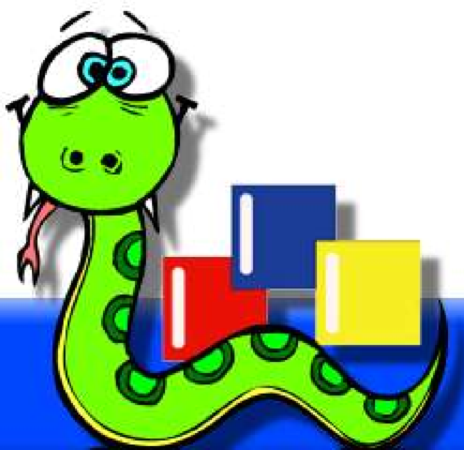


Creating Custom Widgets

1. Derive from `wx.PyControl`. This enables reflection of some pertinent C++ virtual methods to Python methods in the derived class.
2. Call `SetBestFittingSize` from the `__init__` method, passing the size passed to `__init__`. This helps set things up properly for sizers, and also sets the size of the control appropriately (using either the size given or the best size.)
3. Call `InheritAttributes` from the `__init__` method. If the parent has non-standard font or colors then it will set them for your control.
4. Implement a `DoGetBestSize` method, returning the `wx.Size` that would best fit your control based on current font, label or other content, etc.
5. Implement an `AcceptsFocus` method, returning `True` if the control is meant to receive the keyboard focus.

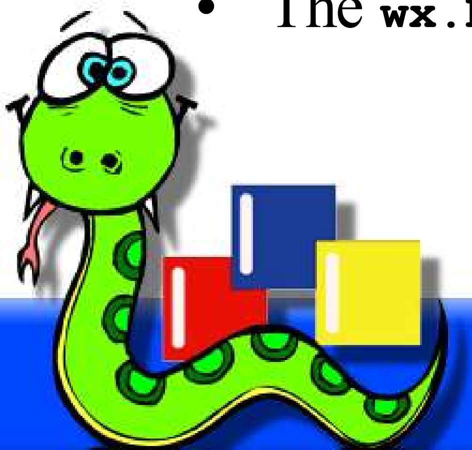


Questions?



Double Buffered Drawing

- When drawing on a window takes several steps, or can be time consuming, you often end up with flicker or other paint artifacts
- Following a simple recipe for “double-buffered drawing” often eliminates flicker and also optimizes the **`EVT_PAINT`** handler.
- You do all your custom drawing to a **`wx.Bitmap`** (the buffer) and then Blit parts of the buffer to the window as needed.
- The **`wx.BufferedDC`** class helps simplify the process



Double Buffered Drawing

- In your class' `__init__` create and initialize the buffer:

```
self.buffer = wx.EmptyBitmap(*size)
dc = wx.BufferedDC(None, self.buffer)
self.DrawBackground(dc)
self.DrawContent(dc)
```

- Bind the **wx.EVT_SIZE** event and in the handler recreate and initialize the buffer:

```
size = self.GetClientSize()
self.buffer = wx.EmptyBitmap(*size)
dc = wx.BufferedDC(wx.ClientDC(self), self.buffer)
self.DrawBackground(dc)
self.DrawContent(dc)
```

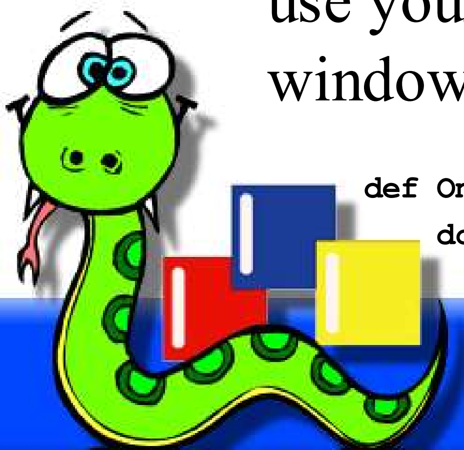
Double Buffered Drawing

- Whenever you need to update the drawing, use a **wx.BufferedDC** with a **wx.ClientDC**. When the buffer DC is dereferenced it will flush its contents to the window via the client DC:

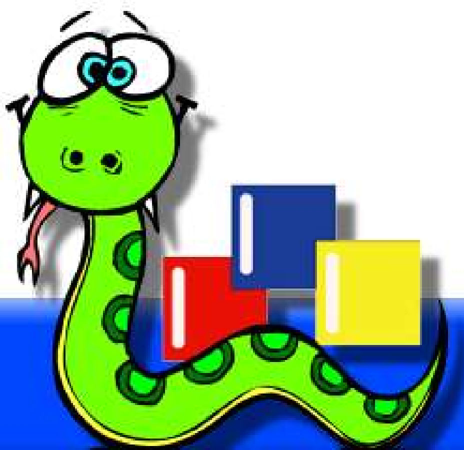
```
dc = wx.BufferedDC(wx.ClientDC(self), self.buffer)
dc.SetPen(wx.Pen(self.GetForeGroundColour()))
dc.DrawLine(old_x, old_y, x, y)
```

- Bind the **wx.EVT_PAINT** event and use **wx.BufferedPaintDC** to use your buffer to refresh the damaged portions of the the window:

```
def OnPaint(self, evt):
    dc = wx.BufferedPaintDC(self, self.buffer)
```



Questions?



Last minute additions

- Slides of this presentation are available at:
  http://wxPython.org/OSCON2004/advanced/