
ObjectListView Documentation

Release 1.2

Phillip Piper

February 18, 2015

1	Without wasting my time, just tell me what it does!	3
2	OK, I'm interested. What do I do next?	5
3	Bleeding-edge source	7
4	Site contents	9
4.1	What's New?	9
4.2	Features of an ObjectListView	10
4.3	Getting Started	14
4.4	Learning to cook	22
4.5	Editing Cell Values	29
4.6	Using a GroupListView	32
4.7	Using a ListCtrlPrinter	35
4.8	Frequently Asked Questions	40
4.9	Major Classes Reference	41
4.10	Change Log	42

An ObjectListView is a wrapper around the `wx.ListCtrl` that makes the list control easier to use. It also provides some useful extra functionality.

Larry Wall, the author of Perl, once wrote that the three essential character flaws of any good programmer were sloth, impatience and hubris. Good programmers want to do the minimum amount of work (sloth). They want their programs to run quickly (impatience). They take inordinate pride in what they have written (hubris).

ObjectListView encourages the vices of sloth and hubris, by allowing programmers to do far less work but still produce great looking results.

Without wasting my time, just tell me what it does!

OK, here's the bullet point feature list:

- Automatically transforms a collection of model objects into a fully functional wx.ListCtrl.
- Automatically sorts rows.
- Easily *edit the cell values*.
- Supports all ListCtrl views (report, list, large and small icons).
- Columns can be fixed-width, have a minimum and/or maximum width, or be space-filling (*Column Widths*)
- Displays a “*list is empty*” message when the list is empty (obviously).
- Supports *checkboxes in any column*
- Supports *alternate rows background colors*.
- Supports *custom formatting of rows*.
- Supports *searching (by typing) on any column*, even on massive lists.
- Supports custom sorting
- Supports *Filtering* and *batched updates*
- The FastObjectListView version can build a list of 10,000 objects in less than 0.1 seconds.
- The VirtualObjectListView version supports millions of rows through ListCtrl's virtual mode.
- The *GroupListView* version supports arranging rows into collapsible groups.
- Effortlessly produce professional-looking reports using a *ListCtrlPrinter*.

Seriously, after using an ObjectListView, you will never go back to using a plain wx.ListCtrl.

OK, I'm interested. What do I do next?

You can download a [source package of the ObjectListView](#) which includes the ObjectListView module, as well as some demo applications showing ObjectListView in action.

After that, you might want to look at the *Getting Started* and the *Cookbook* sections. Please make sure you have read and understood these sections before asking questions in the Forums.

At some point, you will want to do something with an ObjectListView and it won't be immediately obvious how to make it happen. After dutifully scouring the *Getting Started* and the *Cookbook* sections, you decide that is is still not obvious. The [Forum](#) section is the place to find all your as-yet-unasked questions.

It may even be possible that you might find some undocumented features in the code (also known as bugs). These "features" can be reported to the [project's Issue Tracker](#) and the status of your "feature" report can be [tracked here](#).

If you would like to ask me a question or suggest an improvement, you can contact me here: jppx1@bigfoot.com.

Bleeding-edge source

If you are a very keen developer, you can access the SVN repository directly for this project. The following SVN command will fetch the most recent version from the repository:

```
svn co https://objectlistview.svn.sourceforge.net/svnroot/objectlistview/python/trunk objectlistview
```

There are details on [how to use Subversion here](#) on SourceForge.

Please remember that code within the SVN is bleeding edge. It has not been well-tested and is almost certainly full of bugs. If you just want to play with the ObjectListView, it's better to stay with the official releases, where the bugs are (hopefully) less obvious.

4.1 What's New?

For the (mostly) complete change log, *see [here](#)*.

4.1.1 v1.2 - September 2008

- Big new feature: *ListCtrlPrinter*
- Added `AddObjects()` and `RemoveObjects()` and friends
- Added *Filtering*
- Added *batched updates adapter*
- Made `GroupListView` a subclass of `FastObjectListView`. More speed; less flicker.

Small things

- Correctly handle model objects that cannot be hashed
- Added `CELL_EDIT_STARTED` and `CELL_EDIT_FINISHED` events
- Added *ensureVisible* parameter to `SelectObject()`
- Remove flicker from some more `FastObjectListView` operations

4.1.2 v1.1 - July 2008

- Added *GroupListView*
- Column headers can now have their own images

4.1.3 v1.0.1 - June 2008

- Sorting can now be customised through the `EVT_SORT` event
- Added searching by sort column
- Added binary search

- `VirtualObjectListView` can now be sorted, using the `EVT_SORT` event. By default, they are still not sortable
- Fixed some bugs on Mac and Linux

4.1.4 v1.0 - June 2008

- First true public release.
- Official website up and running
- Added check state support
- Added named image support
- Added more examples

4.1.5 v0.9 - May 2008

- Released on wxWiki – to thunderous silence :-)
- Added cell editing

4.1.6 v0.5 - March 2008

- Converted to use straight wxPython now that wax appears dead
- Added column width management (minimum, maximum, space filling)

4.1.7 v0.1 - November 2006

- First version. Written to work with wax.
- Used internally only

4.2 Features of an ObjectListView

Why take the time to learn how to use an `ObjectListView`? What's the benefit? The return on investment? This page tries to document the useful features of an `ObjectListView`. Not all features are equally useful, but it's better to be aware of what's available so that you can use it when the need arises.

4.2.1 Ease of use

The major goal of an `ObjectListView` is to make your life easier. All common `ListCtrl` tasks should be easier – or at least no more difficult – with an `ObjectListView`. For the investment of creating column definitions, you receive a great deal of convenience and value added functions. See [Getting Started](#) for an introduction to the basics.

4.2.2 Automatically create the ListCtrl from model objects

The major way in which the ObjectListView makes your life easier is by being able to automatically build the ListCtrl from a collection of model objects. Once the columns are defined, an ObjectListView is able to build the rows of the ListCtrl without any other help. It only takes a single method call: *SetObjects()*.

4.2.3 Different flavours of ObjectListView for different purposes

An *ObjectListView* is the plain vanilla version of the control. It accepts a list of model objects, and builds the control from those model objects.

A *FastObjectListView* requires a list of model objects, but it can deal with those objects very quickly. Typically, it can build a list of 10,000 objects in less than 0.1 seconds.

A *GroupListView* also requires a list of model objects, but allows those model objects to be partitioned into groups, and then those groups presented to the user as collapsible collections. Partitioned is normally done using the sort column. When the user clicks on a different sort column, the rows are partitioned into different groups. See [Using a GroupListView](#).

A *VirtualObjectListView* does not require a list of model objects. Instead, it asks for model objects as it requires them. In this way, it can support an unlimited number of rows. A *VirtualObjectListView* must be given an *objectGetter* callable, which is called when the list needs to display a particular model object.

4.2.4 Editing cell values

ListCtrls normally allow only the primary cell (column 0) to be edited. An ObjectListView allows all cells to be edited. This editing knows to use different editors for different data types. It also allows autocompletion based on existing values for that column (pass *autoCompleteCellEditor=True* to a column constructor)

See [Editing Cell Values](#) for more details.

4.2.5 Automatic sorting

Once the column are defined, the ObjectListView will automatically sort the rows when the user clicks on a column header. This sorting understands the data type of the column, so sorting is always correct according to the data type. Sorting does not use the string representation.

4.2.6 Supports all ListCtrl views

An ObjectListView supports all views: report, list, large and small icons. All functions should work equally in all views: editing, check state, icons, selection.

4.2.7 More control over column width

An ObjectListView allows the programmer to have control over the width of columns after the ListCtrl is created.

When a column is defined, it is normally given a width in pixels. This is the width of the column when the ListCtrl is first shown. After creation, the user can resize that column to be something else.

By using the *minimumWidth* and *maximumWidth* attributes, the programmer can control the lower and upper limits of a column. The programmer can also use the *fixedWidth* constructor parameter to indicate that a column should not be resizable by the user.

Finally, the programmer can specify that a column should resize automatically to be wider when the `ListCtrl` is made wider and narrower when the `ListCtrl` is made narrower. This type of column is a space filling column, and is created by passing the *isSpaceFilling* parameter to the `ColumnDefn` constructor.

See these recipes:

- [3. How can I stop the user from making a column too narrow or too wide?](#)
- [4. How can I stop the user from resizing a column?](#)
- [5. How can I make a column get bigger when the `ObjectListView` is made bigger?](#)

4.2.8 Displays a “list is empty” message

An empty `ListCtrl` can be confusing to the user: did something go wrong? Do I need to wait longer and then something will appear?

An `ObjectListView` can show a “this list is empty” message when there is nothing to show in the list, so that the user knows the control is supposed to be empty.

See this recipe: [8. How do I change the message that’s shown when the `ObjectListView` is empty?](#)

4.2.9 Checkboxes in any column

An `ObjectListView` trivially supports checkboxes on rows. In fact, it supports multiple checkboxes per row, if you are really keen. See this recipe for more details: [7. How do I use checkboxes in my `ObjectListView`?](#)

4.2.10 Alternate rows background colors

Having subtly different row colours for even and odd rows can make a `ListCtrl` easier for users to read. `ObjectListView` supports this alternating of background colours. It is enabled by default, and the background colours are controlled by the *evenRowsBackColor* and *oddRowsBackColor* attributes.

4.2.11 Custom row formatting

An `ObjectListView` allows rows to be formatted with custom colours and fonts. For example, you could draw clients with debts in red, or big spending customers could be given a gold background. See here: [6. How can I change the colours of a row?](#)

4.2.12 Model object level operations

The `ObjectListView` allows operations at the level that makes most sense to the application: at the level of model objects. Operations like *SelectObjects()*, *RefreshObjects()*, *GetSelectedObjects()* and *GetCheckedObjects()* provide a high-level interface to the `ListCtrl`.

The `VirtualObjectListView` is an unfortunate exception to these features. It does not know where any given model object is located in the control (since it never deals with the whole list of objects), so these model level operations are not available to it.

4.2.13 Searching on the sort column

When a user types into a normal `ListCtrl`, the control tries to find the first row where the value in cell 0 begins with the character that the user typed. [This feature is not supported by a standard `ListCtrl` on all platforms, but it is supported on all platforms by `ObjectListView`].

`ObjectListView` extends this idea so that the searching can be done on the column by which the control is sorted (the “sort column”). If your music collection is sorted by “Album” and the user presses “z”, `ObjectListView` will move the selection to the first track of the “Zooropa” album, rather than find the next track whose title starts with “z”.

In many cases, this behaviour is quite intuitive. iTunes works in this fashion on its string value columns (e.g. Name, Artist, Album, Genre).

4.2.14 Fast searching on sorted column

When the user types something into a control, the `ObjectListView` will use a binary search (if possible) to find a match for what was typed. A binary search is normally possible if the `ObjectListView` is sorted on a column that shows strings.

A binary search is able to handle very large collections: 10,000,000 rows can be searched in about 24 comparisons. This makes it feasible to search by typing even on large virtual lists.

4.2.15 Filtering

By calling `SetFilter()`, you can dynamically filter the model objects that are presented to the user in the control.

A filter is a callable that accepts a single parameter, which is the list of model objects provided to the `ObjectListView` via the `SetObjects()` method. The filter should return the list of objects that will be presented to the user.

The supplied module `Filter` provides some useful standard filters:

- **`Filter.Predicate(booleanCallable)`** Show only the model objects for which the given callable returns true. The callable must accept a single parameter, which is the model object to be considered.
- **`Filter.Head(n)`** Show only the first N model objects.
- **`Filter.Tail(n)`** Show only the last N model objects. Useful to watching logs.
- **`Filter.TextSearch(objectListView, columns=(), text=“”)`** Show only model objects that contain *text* in one of the given columns. If *columns* is empty, all columns from the `ObjectListView` will be considered.
- **`Filter.Chain(filters)`** Show only model objects which satisfy all of the given filters.

Filtering and performance

Most filters impose a performance penalty on the rebuilding of an `ObjectListView`s contents.

This is because they (normally) examine each model object provided to the `SetObjects()` method and decide if it should be included. Thus, a filter normally has a $O(n)$ performance hit.

However, for a plain vanilla `ObjectListView`, if the filter significantly reduces the number of displayed rows, rebuilding the list may be *faster* with the filter installed, since building $N/2$ rows (for example) is faster than building N rows. This does not apply for `FastObjectListView`s, since it only builds rows when they are displayed.

4.3 Getting Started

4.3.1 Introduction

I often find that I have a collection of objects which I want to present to the user in some sort of tabular format. It could be the list of clients for a business, a list of known FTP servers or even something as mundane as a list of files in a directory. User interface-wise, the ListCtrl is the perfect control for these situations. However, I find myself groaning at the thought of using the ListCtrl and secretly hoping that I can use a ListBox instead.

The reason for wanting to avoid the ListCtrl is all the boilerplate code it needs to work. There is the basic “list building loop” insert the data item, add the subitems, call SetItemData() with some integer that will tell me which data object the row is showing.

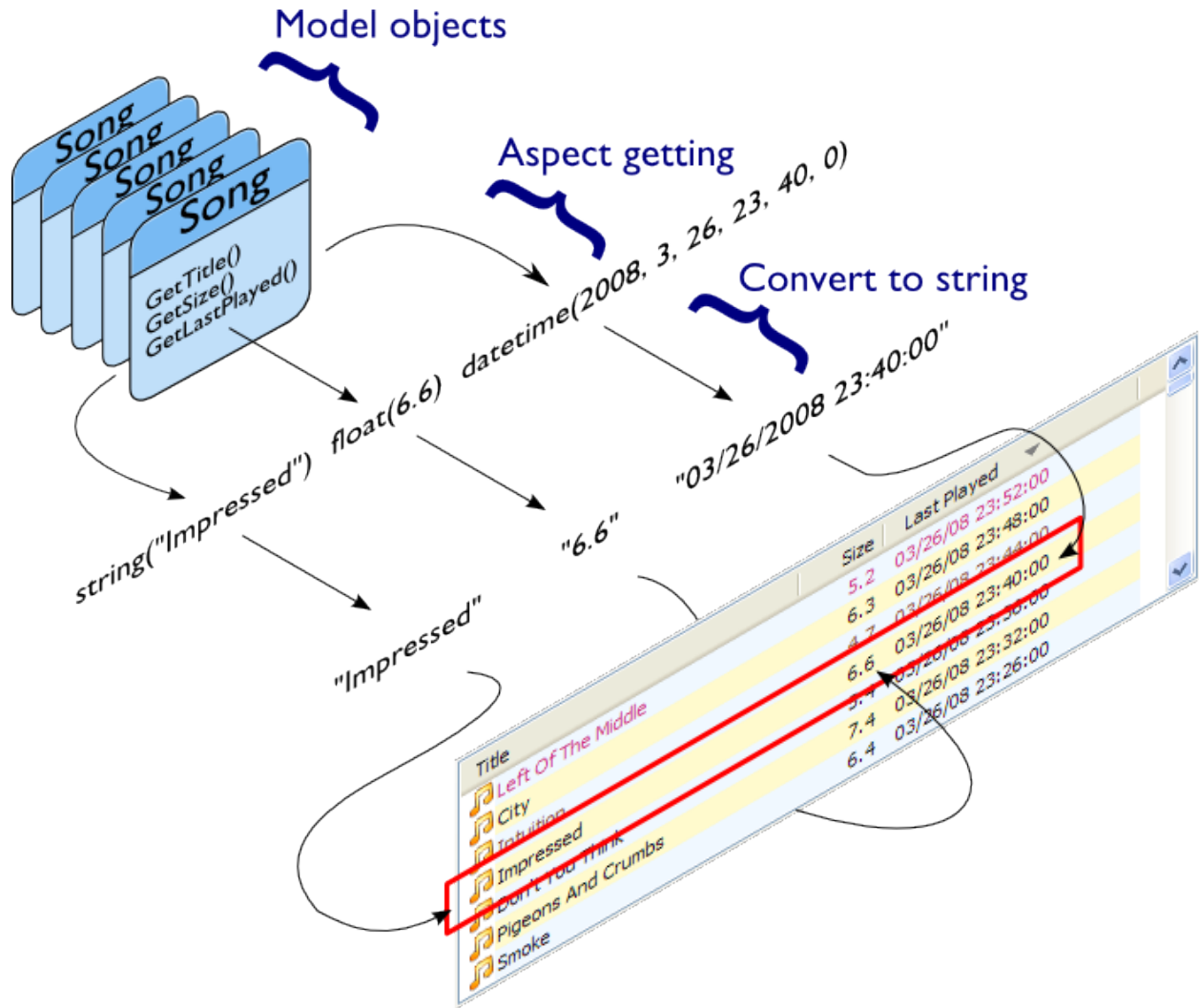
If I want to allow sorting, I can either write everything from scratch (again), or I can do the work of setting up the ‘itemDataMap’ so that ColumnSorterMixin will work.

When the user right clicks on the list, I have to figure out which rows are selected, and then maps those rows back into the model objects I am interested in. Neither of these tasks are obvious if you don’t already know the tricks.

The ObjectListView was designed to take away all these repetitive tasks and so make a ListCtrl much easier to use.

4.3.2 Seeing the big picture

If you are a visual person, the process I’m about to explain is this:



You give the ObjectListView a list of model objects. It extracts aspects from those objects, converts those aspects to strings, and then builds the control with those strings.

Keep this image in mind when reading the following text.

4.3.3 Mental gear shift

This is important. You need to understand this.

ignored

Before trying to use an ObjectListView, you should understand that it is different to a normal ListCtrl. A normal ListCtrl is essentially passive: it sits there, and you poke and prod it and eventually it shows you want. An ObjectListView is much more active. You tell it what you want done and the ObjectListView does it for you.

An ObjectListView is used declaratively: you state what you want the ObjectListView to do (via its configuration), then you give it your collection of model objects, and the ObjectListView does the work of building the ListCtrl for you.

This is a different approach to using a ListCtrl. You must get your mind around this, especially if you have done any ListCtrl programming before (See *Unlearn you must*).

The crucial part of using an ObjectListView is configuring it. Some of this configuration is done on the ObjectListView itself. However, most of this configuration can be done on the columns that are used within the list. Within each column definition, you say what the column should be called, what data it should show, and how that data is to be formatted (there is more, but that's enough to start with).

Once the columns and control are configured, putting it into action is simple. You give it the list of model objects you want it to display, and the ObjectListView will build the ListCtrl for you:

```
self.myFirstOlv.SetObjects(myListOfTracks)
```

4.3.4 Unlearn you must

This section is for those who are familiar with using a ListCtrl.

ignored

For those of you who have struggled with a ListCtrl before, you must unlearn. An ObjectListView is not a drop in replacement for a ListCtrl. If you have an existing project, you cannot simply create an ObjectListView instead of creating a ListCtrl. An ObjectListView needs a different mindset. If you can perform the mind-mangling step of changing your thinking, ObjectListView will be your best friend.

If you find yourself inserting rows, changing subitems, or calling SetItem() or its friends, you need to stop - you are being seduced to the dark side. An ObjectListView does all that work for you. You tell it the aspects of each model object you want to show (via the ColumnDefn objects), specify any formatting and then you give it the list of objects to show.

Resist the temptation to add, edit, remove, or otherwise mess with the rows of an ObjectListView – it will not work.

There is also no need to hide information in each row. Old style ListCtrl programming often required attaching a key of some sort to each row, so that when the user did something with a row, the programmer would know which model object that row was related to. This attaching was often done by creating one or more zero-width columns, or by calling SetItemData() with some integer that uniquely identified the model object.

With an ObjectListView, you do not need to do this anymore. The ObjectListView already knows which model object is behind each row. In many cases, the programmer simply uses the *GetSelectedObjects* method to find out which model objects the user wants to do something to.

4.3.5 This bit goes here, that bit goes there

The first configuration step is to tell each column which bit (called an “aspect”) of your model object it is going to display. This is done through the column's *valueGetter* attribute.

In the most common case, the *valueGetter* attribute of the column is the name of the attribute that the column should display.

Imagine that we were writing an application that managed a MP3 music library. A central part of this application would be the list of tracks available in the library. Your central model class, Track, might look like this:

```
class Track(object):

    def __init__(self, title, artist, album, lastPlayed, sizeInBytes, rating):
        self.title = title
        self.artist = artist
        self.album = album
        self.lastPlayed = lastPlayed
        self.sizeInBytes = sizeInBytes
        self.rating = rating
```

```
def GetSizeInMb(self):
    return self.sizeInBytes / (1024.0*1024.0)
```

To show the title in a column, you would make a `ColumnDefn` like this:

```
titleColumn = ColumnDefn("Title", "left", 120, "title")
```

This says, “Make a left-aligned column, 120 pixels wide, with a title ‘Title’, which shows the aspect ‘title’ from each model object.”

The definitions for the artist and the album are similar:

```
artistColumn = ColumnDefn("Artist", "left", 120, "artist")
albumColumn = ColumnDefn("Album", "left", 120, "album")
```

4.3.6 Converting to strings

OK, we’ve told our first few columns which bits of data they should display. For the title, artist, and album columns, this is all that is necessary. But for the “Size In MBs” and “Last Played” columns, there is something else we need to consider: how to convert our bit of data to a string.

A `ListCtrl` control can only display strings. Everything else - booleans, integers, dates, whatever - has to be converted to a string before it can be given to the `ListCtrl`. By default, the `ObjectListView` converts data to strings using the string format operation, more or less like this: `%s" % aspectValue`

You can use a different format string (instead of the default “%s”) by setting the *stringConverter* attribute on the column definition. If the *stringConverter* attribute is a `string`, its value will be used as the format string instead of “%s”.

So for our “Size In MBs” column, we are only interested in one decimal places, so we would define that column like this:

```
sizeInMbColumn = ColumnDefn("Size (MB)", "center", 100,
                             "GetSizeInMb", stringConverter="%.1f")
```

Notice that here we used a method name (“`GetSizeInMb`”) instead of an attribute name. The column definition can handle either (actually, it can handle quite a bit more than that, but that’s for later).

If the `aspectValue` is a date or time, then instead of using the plain `%` operator, the *stringConverter* is used as the parameter to `strftime` method, like this: `dateTimeAspect.strftime(column.stringConverter)`

So, we would configure our “Last Played” column like this:

```
lastPlayedColumn = ColumnDefn("Last Played", "left", 100,
                              "lastPlayed", stringConverter="%d-%m-%Y")
```

4.3.7 Put it all together and what have you got?

Putting all these columns together, we put them into use via the *SetColumns* method:

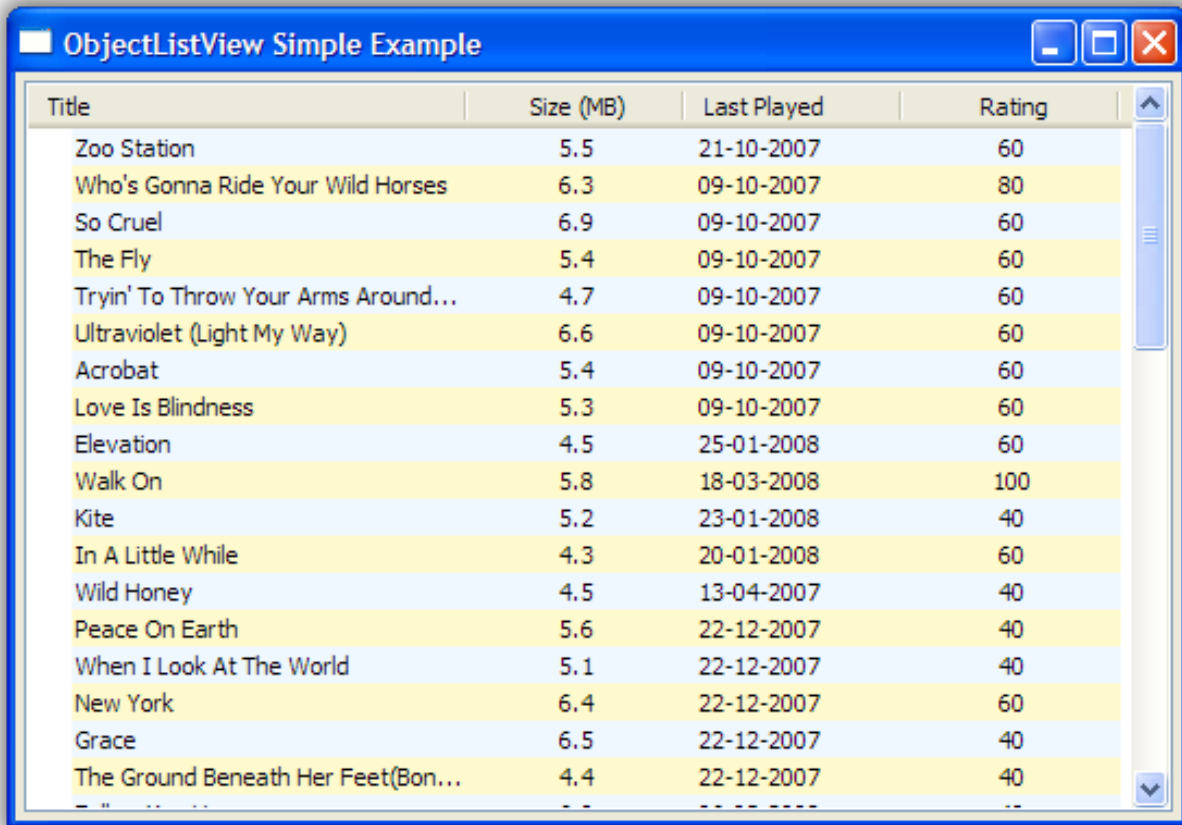
```
self.myFirstOlv.SetColumns([
    ColumnDefn("Title", "left", 120, "title"),
    ColumnDefn("Size (MB)", "center", 100, "GetSizeInMb", stringConverter="%.1f"),
    ColumnDefn("Last Played", "left", 100, "lastPlayed", stringConverter="%d-%m-%Y"),
    ColumnDefn("Rating", "center", 100, "rating")
])
```

[I dropped the Artist and Album columns so that the screen shot below is smaller].

Once we have defined the columns, we set the whole thing into action with *SetObjects*:

```
self.myFirstOlv.SetObjects(self.listOfTracks);
```

And we should get something like this:



Title	Size (MB)	Last Played	Rating
Zoo Station	5.5	21-10-2007	60
Who's Gonna Ride Your Wild Horses	6.3	09-10-2007	80
So Cruel	6.9	09-10-2007	60
The Fly	5.4	09-10-2007	60
Tryin' To Throw Your Arms Around...	4.7	09-10-2007	60
Ultraviolet (Light My Way)	6.6	09-10-2007	60
Acrobat	5.4	09-10-2007	60
Love Is Blindness	5.3	09-10-2007	60
Elevation	4.5	25-01-2008	60
Walk On	5.8	18-03-2008	100
Kite	5.2	23-01-2008	40
In A Little While	4.3	20-01-2008	60
Wild Honey	4.5	13-04-2007	40
Peace On Earth	5.6	22-12-2007	40
When I Look At The World	5.1	22-12-2007	40
New York	6.4	22-12-2007	60
Grace	6.5	22-12-2007	40
The Ground Beneath Her Feet(Bon...	4.4	22-12-2007	40

4.3.8 What have we achieved?

Underwhelmed? Admittedly, it's not much to look at, but be patient. Also, consider how much work you had to do to make it happen: some column definitions and one line of code. It doesn't look so bad now, does it?

With those column definitions, we have a *ListCtrl* that shows the title, date last played, size (in megabytes) and rating of various tracks in your music library. But, actually, we have quite a bit more than that.

Without any extra work, sorting just works. Clicking on the column headers will sort and reverse sort the rows. The sort is accurate for the data type: when sorting by size, a song of 35 megabytes will come after a song of 9 megabytes.

The control also understands 'model object' level operations. So, we can ask the control for the tracks that are selected (via the *GetSelectedObjects* method). We can refresh the information about one or more tracks (using *RefreshObjects* method)

We also have the access to some of the nice builtin functions that *ObjectListView* provides.

4.3.9 Adding some images

OK, that's good, but any real ListCtrl needs to be able to put little icons next to the text. That is our next task.

A ListCtrl can only ever show images that exist in its image list. To make an image available to an ObjectListView, you call *AddImages* method:

```
musicImage = self.myOlv.AddImages(Images.getMusic16Bitmap(), Images.getMusic32Bitmap())
```

This registers two versions of the same image to the *ObjectListView*. The second bitmap is only used when the list is in Large Icon view. If your control is never going to be put into Large Icon view (for example, it's always in Details view), you don't need to register the second image.

The *AddImages* method return an integer that identifies the image. Referring to images via their index is a bit of a pain. It's easier to register named images, via *AddNamedImages* and then use the names of the images:

```
self.myOlv.AddNamedImages("user", Images.getUser16Bitmap(), Images.getUser32Bitmap())
self.myOlv.AddNamedImages("group", Images.getGroup16Bitmap(), Images.getGroup32Bitmap())
```

Once we have an image registered, we can use it in the control. This is done using the *imageGetter* attribute of a column. When a column wants to determine what image should be shown, it uses the *imageGetter* attribute. The simplest case is to assign a constant to this attribute. This will give every cell in the column the same image, like this:

```
titleColumn = ColumnDefn("Title", "left", 120, "title", imageGetter=musicImage)
```

This is easy, but limited. Much more useful is giving a callable as the *imageGetter* (notice that here we're using image names):

```
def artistImageGetter(track):
    soloArtists = ["Nelly Furtado", "Missy Higgins", "Moby", "Natalie Imbruglia",
                  "Dido", "Paul Simon", "Bruce Cockburn"]
    if track.artist in soloArtists:
        return "user"
    else:
        return "group"

artistColumn = ColumnDefn("Artist", "left", 120, "artist", imageGetter=artistImageGetter)
```

If the *imageGetter* is a callable, it must accept a single parameter, which is the model object whose image is being sought. It should return the name or the index of the image to display, or -1 if no image should be shown.

The *imageGetter* can also be given the name of an attribute or a method of your model object. This is useful when the model objects themselves know which image they should use.

4.3.10 Smarter string conversions

Being able to change the *stringConverter* to a different format string is useful, but there are just some cases that need something more substantial.

In our Track class, the actual size of the track is stored as *sizeInBytes*. It would be nice if we could show the size as “360 bytes”, “901 KB”, or “1.1 GB” which ever was more appropriate.

To do this, we can set the *stringConverter* attribute to be a callable, like this:

```
def sizeToNiceString(byteCount):
    for (cutoff, label) in [(1024*1024*1024, "GB"), (1024*1024, "MB"), (1024, "KB")]:
        if byteCount >= cutoff:
            return "%.1f %s" % (byteCount * 1.0 / cutoff, label)
    if byteCount == 1:
        return "1 byte"
```



```

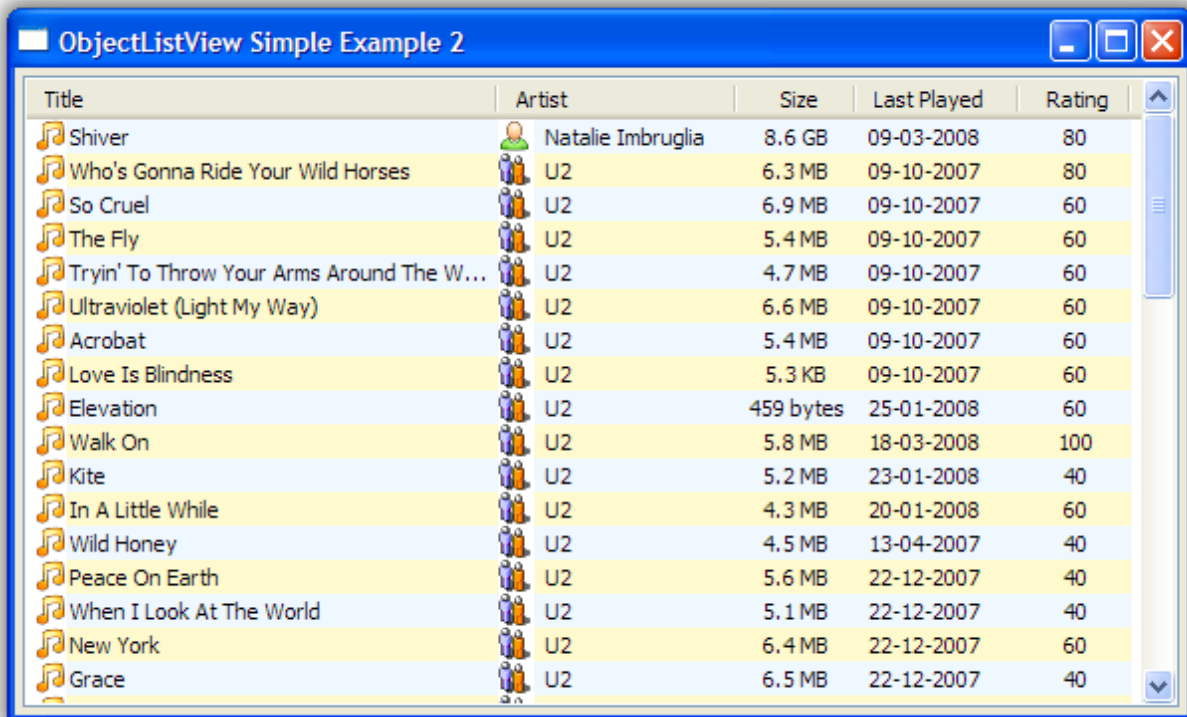
    return "1 byte"
else:
    return "%d bytes" % byteCount

```

```
sizeColumn = ColumnDefn("Size", "center", 100, "sizeInBytes", stringConverter=sizeToNiceString)
```

If *stringConverter* is a callable, it must accept a single parameter, which is the value to be converted (note: it's the value not the model object that is given to the converter).

Putting all these bits together, we now have something that looks like this:



Hey! That's starting to not look too bad.

4.3.11 Playing with groupies

Moving up a gear, we can build on these basic configurations to produce a *GroupListView*, which is a *wx.ListCtrl* which has collapsible groups.

The code for this example is in "GroupExample.py". The first thing to notice is that the code for this example is almost identical to the code from example 2. There is very little extra needed to make a *GroupListView*.

To make a *GroupListView* work, the control needs to know how to collect the model objects into different groups. This is done by getting a "group key" for each object. All objects with the same group key are placed in the same group.

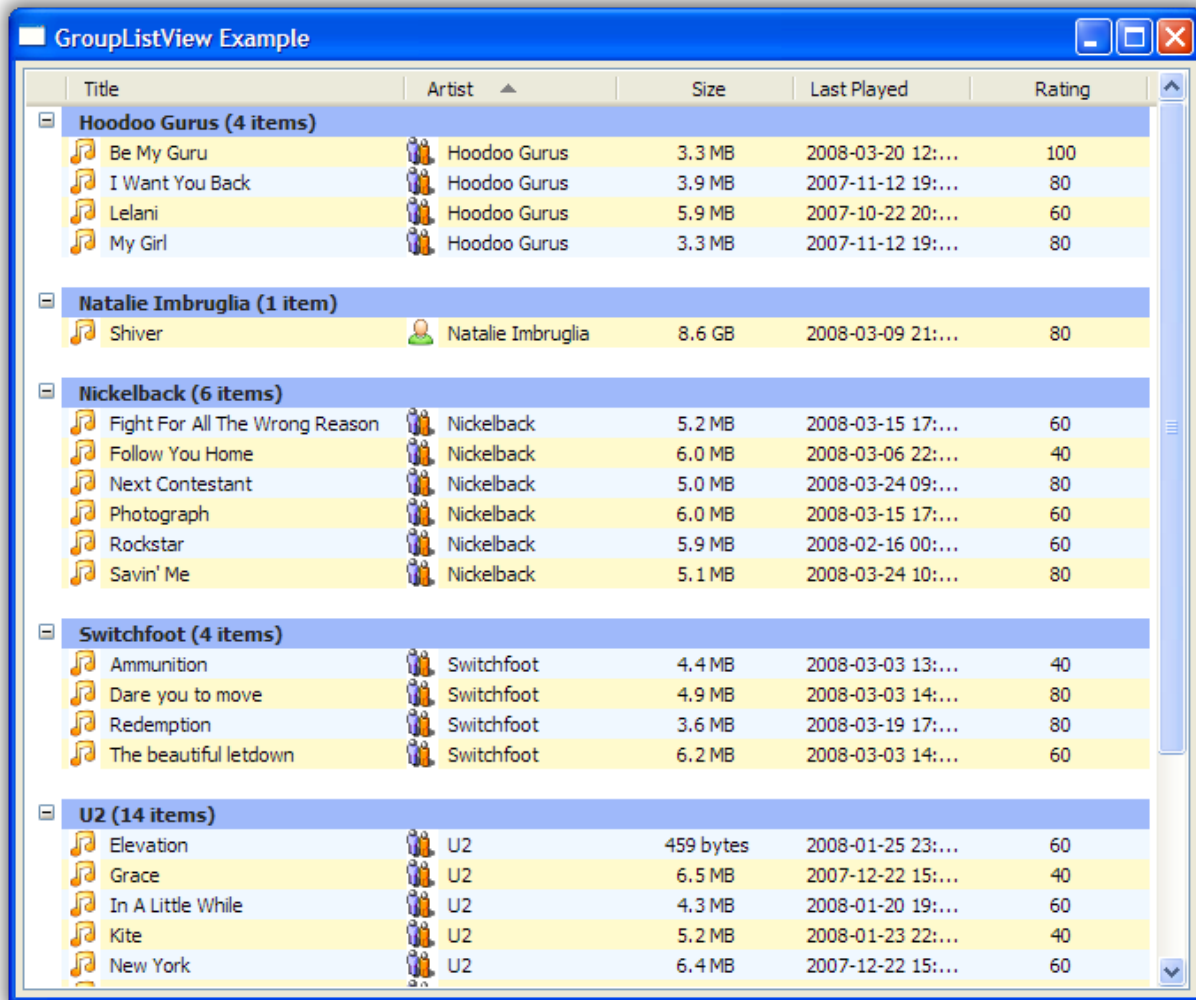
The "group key" for an object is normally whatever value the object has in the "group by" column. So if the group list is being grouped by the "Artist" column, the group key for a track will be, for example, "Coldplay" or "Nickelback". This process can, of course, be customised, using the *groupKeyGetter* attribute of *OLVColumn* – but that's a story for another time. See [Using a GroupListView](#) for full information.

One common pattern is for objects to be grouped by the first letter of a string value, for example on the "Title" column, all tracks starting with "A" would be grouped together. This is so common that there is built-in way to do it, as it shown

in the first column in the example:

```
ColumnDefn("Title", "left", 120, "title", imageGetter=musicImage, useInitialLetterForGroupKey=True)
```

Putting all these bits together, we now have something that looks like this:



Now that's cool!

4.3.12 And they're off and running

Well done! You've made it to the end of the tutorial. You should by now have a reasonable grasp of some of the things an ObjectListView can do, and how to use it in your application.

If you need further help, you can look at the [Cookbook](#) and the [Forum](#) for those questions that just don't have answer anywhere else.

Don't forget: Use The Source Luke! You have all the source code. If you can't figure something out, read the code and see what is actually happening.

4.4 Learning to cook

1. What flavour of *ObjectListView* do I want to use?
2. How do I let the user edit the values shown in the list?
3. How can I stop the user from making a column too narrow or too wide?
4. How can I stop the user from resizing a column?
5. How can I make a column get bigger when the *ObjectListView* is made bigger?
6. How can I change the colours of a row?
7. How do I use checkboxes in my *ObjectListView*?
8. How do I change the message that's shown when the *ObjectListView* is empty?
9. How can I show a list of dictionaries in the *ObjectListView*?
10. How can I reference the column in a *valueGetter* function?
11. How can I change the way group headers look?
12. How can I only redraw the control every *N* seconds?

4.4.1 1. What flavour of *ObjectListView* do I want to use?

There are three flavours of *ObjectListView* (four if you include *GroupListView*):

ObjectListView - Plain Vanilla



Stock standard, but still very nice

A normal *ObjectListView* is capable of anything a *ListCtrl* can do – only in a much easier to use package. Though it will handle a large number of rows, it is better suited for smaller lists, that is, about 1000 rows or less.

If in doubt, start with this flavour. You can always change it to one of the others later on.

VirtualObjectListView - Espresso



Has a slight bitter taste, but makes you capable of anything

When you want to knock your users out with 10 million search results in a ListCtrl, a VirtualObjectListView is for you. It can handle any number of objects.

But it does so at a cost:

- You have to implement your own sorting method.
- You cannot use object level commands like *SelectObject* or *RefreshObject* since the list has no way of knowing where any given model object is in the list.

But for the cost of this “bitterness,” you really can do anything.

FastObjectListView - Red Bull



Also makes you capable of anything, but without the bitterness

When you want speed, speed, and more speed, but you don’t want the bitterness of the VirtualObjectListView, a FastObjectListView is your weapon of choice.

It operates just like a normal ObjectListView – only much faster.

Did I mention the speed?

GroupListView

A *GroupListView* is slightly different type of beast. You would use this creature when you want your users to be able to group your model objects into different “partitions” depending on the column they last clicked.

See [Using a GroupListView](#) for more information.

4.4.2 2. How do I let the user edit the values shown in the list?

This really needs a page to itself: [Cell Editing in an ObjectListView](#)

4.4.3 3. How can I stop the user from making a column too narrow or too wide?

Columns have both *minimumWidth* and *maximumWidth* attributes. By default, these are -1, which means that no limit is enforced. But if they are set to some other value, the column will be limited to the given minimum or maximum width.

For example, this:

```
self.titleColumn.minimumWidth = 30
```

will stop the “Title” column from becoming less than 30 pixels in width. This is useful for preventing users from shrinking columns to 0 width and then not being about to find them again.

4.4.4 4. How can I stop the user from resizing a column?

There are some columns just don’t make sense to be resizable. A column that always shows a 16x16 status icon makes no sense to be resizable. To make a column be fixed width and unresizable by the user, you can create the column with a *fixedWidth* parameter:

```
self.statusColumn = ColumnDefn("", imageGetter=statusImageGetter, fixedWidth=16)
```

Or, if you decide after column creation that the column should be fixed width, you can call *SetColumnFixedWidth*:

```
self.olv1.SetColumnFixedWidth(0, 16) # the first column is fixed to 16 pixel wide
```

4.4.5 5. How can I make a column get bigger when the ObjectListView is made bigger?

On most columns, the column’s width is static, meaning that it doesn’t change by itself. But sometimes it would be useful if a column would resize itself to show more (or less) of itself when the user changed the size of the ListCtrl. For example, the rightmost column of a personnel list might display “Comments” about that person. When the window was made larger, it would be nice if that column automatically expanded to show more of the comments about that person. You can make this happen by setting the *isSpaceFilling* attribute to `True` on that column.

An ObjectListView can have more than one space filling column, and they generally share the available space equally between them (see the *freeSpaceProportion* attribute to change this).

You should be aware that as the ObjectListView becomes smaller, the space filling columns will become smaller too, until they eventually disappear (have zero width). The *minimumWidth* and *maximumWidth* attributes still work for space filling columns. So you can use the *minimumWidth* attribute to make sure that space filling columns don’t disappear.

4.4.6 6. How can I change the colours of a row?

You install a *rowFormatter* on the ObjectListView.

A rowFormatter is a callable that accepts two parameters: a `wx.ListItem`, and the model object for that `wx.ListItem`. The rowFormatter can change the formatting of the list item, or any of its other properties.

To show customers in red when they owe money, you could do this:

```
def rowFormatter(listItem, customer):
    if customer.amountOwed > 0:
        listItem.SetTextColour(wx.RED)

self.objectListView1.rowFormatter = rowFormatter
```

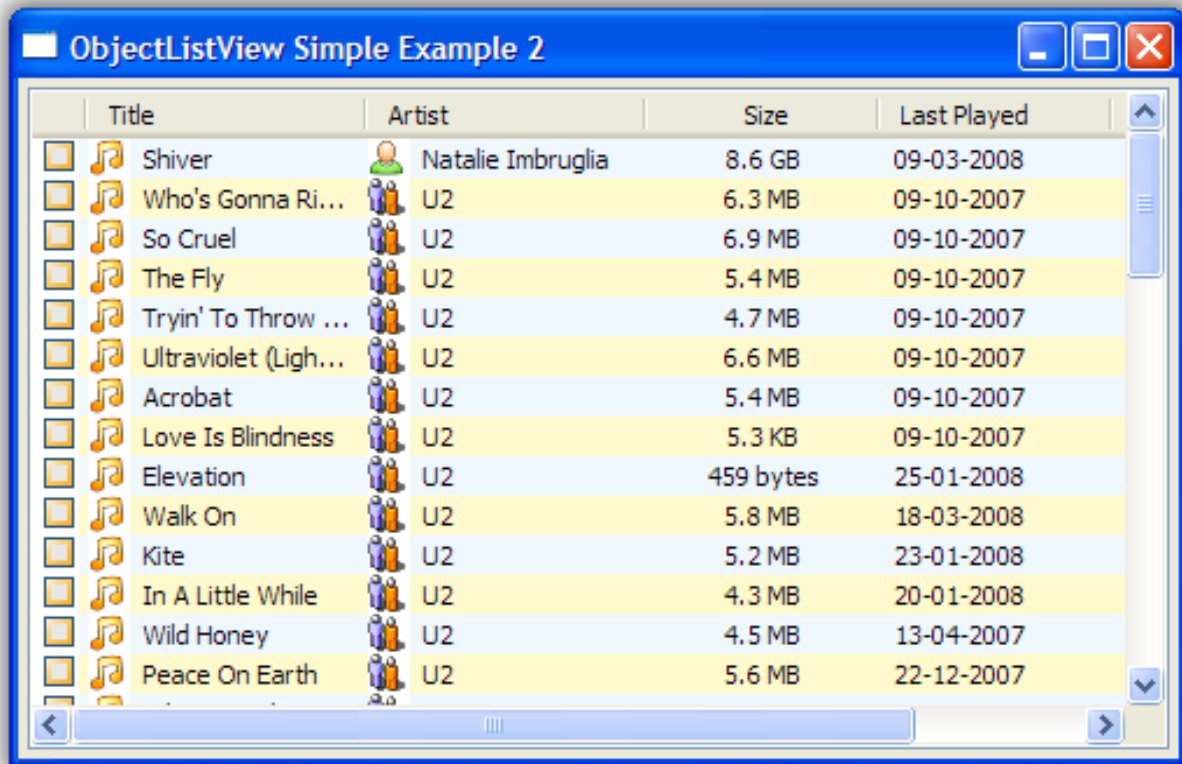
Due to the vaguaries of virtual lists, *rowFormatter* for a *VirtualObjectListView* operates slightly differently. Instead of being given a `wx.ListItem`, it is given a `wx.ListItemAttr` object. These object only support a limited subset of `wx.ListItem` method, specifically those members related to the background color, text colour and font. See `wx.ListItemAttr` for more details of what is supported.

4.4.7 7. How do I use checkboxes in my ObjectListView?

ObjectListView allows several ways of using check boxes. The simplest way is to call *CreateCheckStateColumn*:

```
self.objectListView1.CreateCheckStateColumn()
```

This creates a new column in the control that is solely for the purpose of showing a checkbox for each row. By default, this column is the first column (you can pass an integer to *CreateCheckStateColumn* to create the column at a different position). It results in something that looks like this:

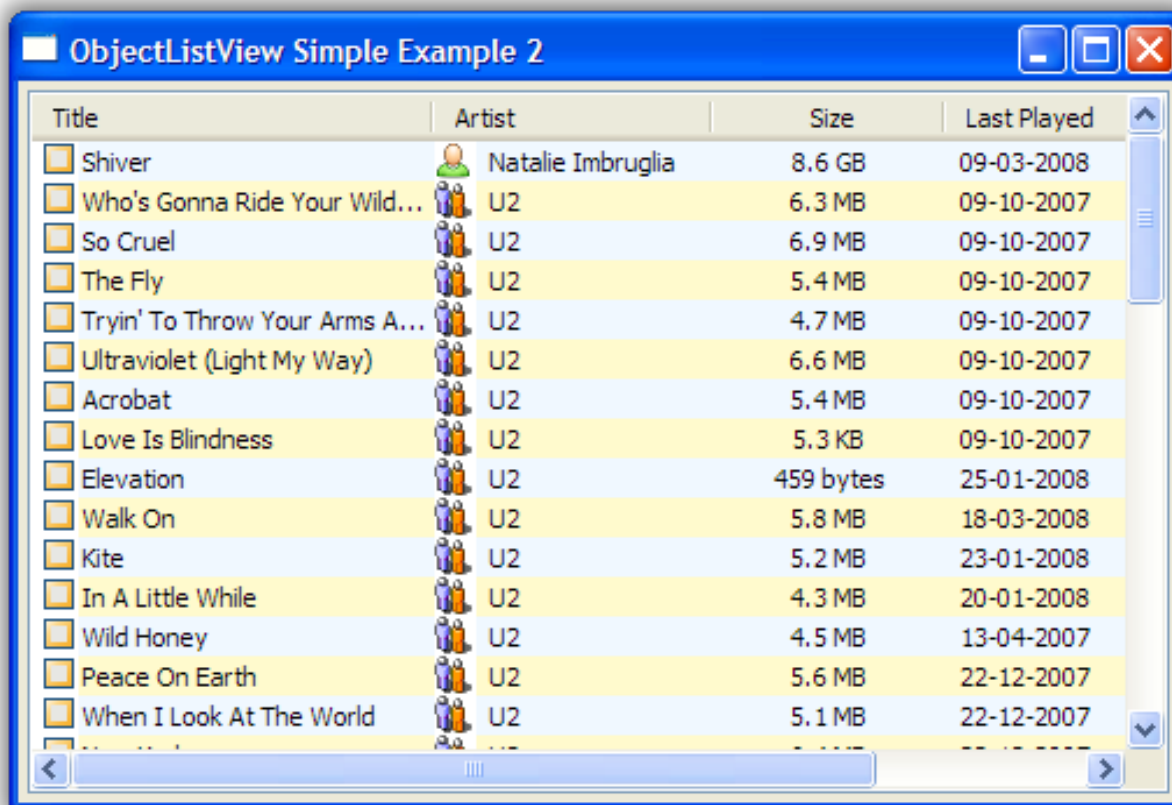


If you don't want to have a specific column just for the checkbox, you can use an existing column as the check box column. To do this, call *InstallCheckStateColumn* with the column defn you want to show the check boxes. Be aware that a column can only have one image, so that column will have the checkbox as its image, and will ignore anything you might have set up with the *imageGetter* attribute.

So, if we installed the "Title" column as a checkbox column:

```
self.objectListView1.InstallCheckStateColumn(self.titleColumn)
```

It would produce something different:



Each track now has the checkbox image instead of the track image.

The user can change checkboxes by clicking on the check box (obviously) or by selecting one or more rows and pressing Space. Pressing Space toggles the values of the selected rows. Actually, it toggles the top-most check box, and the sets all the other rows to have the same value.

Data-based Checkboxes

Both of these methods install checkboxes where the “checked-ness” of the rows are specific to that ObjectListView. So if the same model object was visible in two different lists, it could be checked in one but not in the other.

But sometimes, the “checked-ness” of a row is part of the model object itself. Consider a customer management system, where customers could be marked as active or not. This `isActive` property should be the same regardless of which listview was showing the customer.

To define that checkbox column where the data comes from the model object, you would give that column a *checkStateGetter* attribute:

```
isActiveColumn = ColumnDefn("Active?", fixedWidth=24, checkStateGetter="isActive")
```

Or instead of giving the checkbox its own column, you could combine the checkbox with the customers name:

```
nameColumn = ColumnDefn("Name", valueGetter="name", checkStateGetter="isActive")
```

Though this would stop the “Name” column from having its own image.

ObjectListViews support multiple check box columns.

4.4.8 8. How do I change the message that's shown when the ObjectListView is empty?

When an ObjectListView is empty, it can display a “this list is empty” type message. You alter the text and its font using the following methods:

```
self.objectListView1.SetEmptyListMsg("This database has no rows")
self.objectListView1.SetEmptyListMsgFont(wx.Font(24, wx.DEFAULT, face="Tekton"))
```

4.4.9 9. How can I show a list of dictionaries in the ObjectListView?

I have a list of dictionaries that hold the data I want to show. How can I show them in an ObjectListView?

In your ColumnDefn, set the `valueGetter` to the key of the data you want to display in that column. Everything should just work. As a bonus, your dictionary will be automatically updated when the user edits a cell value (if the ObjectListView is editable).

Example:

```
self.listOfDictionaries = [
    { "title": "Shiver", "artist": "Natalie Imbruglia", "album": "Counting Down the Days" },
    { "title": "Who's Gonna Ride Your Wild Horses", "artist": "U2", "album": "Achtung Baby" },
    { "title": "So Cruel", "artist": "U2", "album": "Achtung Baby" },
    { "title": "The Fly", "artist": "U2", "album": "Achtung Baby" }
]
self.myOlv.SetColumns([
    ColumnDefn("Title", "left", -1, "title"),
    ColumnDefn("Artist", "left", -1, "artist"),
    ColumnDefn("Album", "center", -1, "album")
])
self.myOlv.SetObjects(self.listOfDictionaries)
```

4.4.10 10. How can I reference the column in a valueGetter function?

I've got a super-duper valueGetter function, but it needs to know which column it's being used for. How can I do that?

Normally, `valueGetter` functions don't know which column they are being used for. But there could be cases where you might want to know the column: for example, you might have a central getter function that decides that to do based on which column is being used.

So, imagine our super `valueGetter` looks like this:

```
def MySuperValueGetter(modelObject, columnDefn):
    # Do something clever here
    return value
```

There are (at least) three possible solutions:

1. Use `functools.partial()`:

```
import functools

for column in self.olv1.columns:
    column.valueGetter = functools.partial(MySuperValueGetter, columnDefn=column)
```

This only works with Python 2.5 and later.

2. Use local functions and default parameters:

```
for column in self.olv.columns:
    def myFunc(modelObject, col=column):
        return MySuperValueGetter(modelObject, col)
    column.valueGetter = myFunc
```

3. Subclass ColumnDefn and override GetValue():

```
class MyColumnDefn(ColumnDefn):

    def GetValue(self, modelObject):
        return MySuperValueGetter(modelObject, self)
```

4.4.11 11. How can I change the way group headers look?

Whoever decided on the colour scheme for group headers was an artistic incompetent. I want to use my own snazzy scheme. How do I do that?

The formatting of group headers is strictly limited. wx.ListCtrl's do not support any form of owner drawing, so anything snazzy is currently impossible. Don't even think about trying to do gradient fills or fancy text effects – it's just not possible.

What you can do is:

- change the colour of the text, via the *groupTextColour* variable.
- change the colour of background of the entire row, via the *groupBackgroundColour* variable. You can't change just the group header background. It is the whole row or nothing.
- change the font of the header via the *groupFont* variable. Remember that row height is fixed, so if you make the font too big, the text will be truncated. The header row will *not* become bigger.

4.4.12 12. How can I only redraw the control every N seconds?

I'm writing a network monitor app. In some circumstances, the model objects can be updated 100 times or more each second. But if I try to update the ObjectListView that often, the application grinds to a halt. Is there an easy way to make the ObjectListView not redraw so often?

Yes. You can use a BatchedUpdate adapter. This wraps an ObjectListView such that no matter how often you update it, it will redraw at most once every N seconds (you supply the value of N).

So in your network monitor app, you need to added a line like this some time after the ObjectListView is created and before it is used:

```
self.olv = ObjectListView.BatchedUpdate(self.olv, SECONDS_BETWEEN_UPDATES)
```

This wraps the original *olv* with an Adapter that intercepts some of the model updating commands (*SetObjects()* or *AddObjects()* or *RefreshObjects()*) so that the control only redraws once every SECONDS_BETWEEN_UPDATES seconds (SECONDS_BETWEEN_UPDATES is a constant defined by you somewhere else).

See `Examples/BatchedUpdateExample.py` for a demonstration and [:ref:here](#) for class docs <batchedupdate-class>.

4.5 Editing Cell Values

ListCtrls are normally used for displaying information. The standard ListCtrl allows the value at column 0 (the primary cell) to be edited, but nothing beyond that. ObjectListView allows all cells to be edited. Depending on how the data for a cell is sourced, the edited values can be automatically written back into the model object.

The “editability” of an ObjectListView is controlled by the *cellEditMode* attribute. This attribute can be set to one of the following values:

- **ObjectListView.CELLEDIT_NONE** Cell editing is not allowed on the control This is the default.
- **ObjectListView.CELLEDIT_SINGLECLICK** Single clicking on any subitem cell begins an edit operation on that cell. Single clicking on the primary cell does *not* start an edit operation. It simply selects the row. Pressing F2 edits the primary cell.
- **ObjectListView.CELLEDIT_DOUBLECLICK** Double clicking any cell starts an edit operation on that cell, including the primary cell. Pressing F2 edits the primary cell.
- **ObjectListView.CELLEDIT_F2ONLY** Pressing F2 edits the primary cell. Tab/Shift-Tab can be used to edit other cells. Clicking does not start any editing.

Individual columns can be marked as editable via the *isEditable* attribute (default value is True), though this only has meaning once the ObjectListView itself is editable. If you know that the user should not be allowed to change cells in a particular column, set *isEditable* to False. Be aware, though, that this may create some surprises, resulting in user complaints like “How come I can’t edit this value by clicking on it like I can on all the other cells?”.

Once a cell editor is active, the normal editing conventions apply:

- Enter or Return finishes the edit and commits the new value to the model object.
- Escape cancels the edit.
- Tab commits the current edit, and starts a new edit on the next editable cell.
- Shift-Tab commits the current end, and starts a new edit on the previous editable cell.

4.5.1 Deciding on a cell editor

When a cell is to be edited, we need to decide what sort of editor to use.

There are three ways this decision can be made:

1. Column based decision

Most simply, the column can be configured with a *cellEditorCreator* attribute. When a cell in this column is to be edited, the *cellEditorCreator* will be invoked. This attribute must be set to a “cell editor factory” callable. A “cell editor factory” must be a callable that accepts three parameters:

- the ObjectListView that needs the editor
- the index of the row to be edited
- the index of the subitem to be edited

This factory should return a fully configured widget that can edit the value at that cell.

2. Event based decision

If this is not enough, the programmer can have complete control over the process by listening for a cell editing starting event (*ObjectListView.EVT_CELL_EDIT_STARTING*). Within the handler for this event, the programmer can create and configure any sort of widget they like and then return this widget via the *newEditor* attribute of the event.

If the `shouldConfigureEditor` attribute of the event is `True` (this is the default), the `ObjectListView` will perform all the normal default configuration of the cell editor. This includes setting the controls value, positioning it correctly and hooking up any required events. If `shouldConfigureEditor` is `False`, it is assumed that all configuration has already been done and nothing else will be done to the widget.

3. Registry based decision

Most generally, you can register a “cell editor factory” for a type of object. This is done using `RegisterCreatorFunction` method of the `CellEditorRegistry`.

For example, there is no standard editor for a `wx.Colour`. To handle the editing of colours, we would need a factory callable, and then to register it with the `CellEditorRegistry`. Which might look something like this:

```
def makeColourEditor(olv, rowIndex, subItemIndex):
    odcb = OwnerDrawnEditor.ColourComboBox(olv)
    # OwnerDrawnComboBoxes don't generate EVT_CHAR so look for keydown instead
    odcb.Bind(wx.EVT_KEY_DOWN, olv._HandleChar)
    return odcb
```

```
CellEditorRegistry().RegisterCreatorFunction(type(wx.BLACK), makeColourEditor)
```

By default, the cell registry is configured with editors for the following standard types: `bool`, `int`, `long`, `str`, `unicode`, `float`, `datetime`, `date`, `time`.

You can replace the standard editors with editors of your own devising using the registry. So if someone make a better date-time editor (yes, please!), they could use it to edit all `datetime` values by doing this:

```
import datetime
...
CellEditorRegistry().RegisterCreatorFunction(datetime.datetime, makeBetterDateTimeEditor)
```

4.5.2 Getting and Setting the Editors value

A cell editor must implement both `GetValue` and `SetValue` methods.

Once the cell editor has been created, it is given the cell's value via the control's `SetValue` method.

When the user has finished editing the value, the new value in the editor is retrieved via the `GetValue` method.

4.5.3 Updating the Model Object

Once the user has entered a new value into a cell and pressed `Enter`, the `ObjectListView` tries to store the modified value back into the model object. There are three ways this can happen:

1. `ObjectListView.EVT_CELL_EDIT_FINISHING` Event Handler

You can create an event handler for the `EVT_CELL_EDIT_FINISHING` event (see below). In that handler, you would write the code to get the modified value from the editor, put that new value into the model object, and then call `Veto` on the event, so that the `ObjectListView` knows that it doesn't have to do anything else. You will also need to call at least `RefreshItem()` or `RefreshObject()`, so that the changes to the model object are shown in the `ObjectListView`.

There are cases where this is necessary, but as a general solution, it doesn't fit my philosophy of slothfulness.

2. Via the Column's `valueSetter` attribute

You can set the `valueSetter` attribute on the corresponding `ColumnDefn`. Like `valueGetter`, this attribute is quite flexible:

- It can be a callable that accepts the model object and the new value:

```
def updateSalary(person, newValue):
    person.SetSalary(newValue)
    if person.userId == self.currentUser.userId:
        self.NotifySupervisorOfSalaryChange()

ColumnDefn("Salary", ... valueSetter=updateSalary)
```

- It can be the name of a method to be invoked,. This method must accept the new value as its sole parameter. Example:

```
class Track():
    ...
    def SetDateLastPlayed(self, newValue):
        self.dateLastPlayed = newValue

ColumnDefn("Last Played", ... valueSetter="SetDateLastPlayed")
```

- It can be the name of an attribute to be updated. This attribute will not be created: it must already exist. Example:
- ```
ColumnDefn("Last Played", ... valueSetter="dateLastPlayed")
```
- For dictionary like model objects, it can be the key into the dictionary. The key would commonly be a string, but it doesn't have to be.

## 3. Via the Column's `valueGetter` attribute

Updating the value through the *value-GETTER* attribute seems wrong somehow. In practice, it is neat and commonly used.

If the `valueGetter` attribute is the name of an attribute, or the key into a dictionary, it will very commonly be the same place where any modified value should be written.

So if a value needs to be written back into the model, and there is no `valueSetter` attribute, the `ObjectListView` will try to use the `valueGetter` attribute to decide how to update the model.

### 4.5.4 After the update

If the model is updated, the row will be automatically refreshed to display the new data.

If the user enters a new value, presses `Enter`, and the value in the `ObjectListView` doesn't change, then almost certainly the `ObjectListView` could not automatically update the model object.

In that case, you will need to track down, which of the above three strategies should be being used, and why it is not.

### 4.5.5 How Can You Customise The Editing

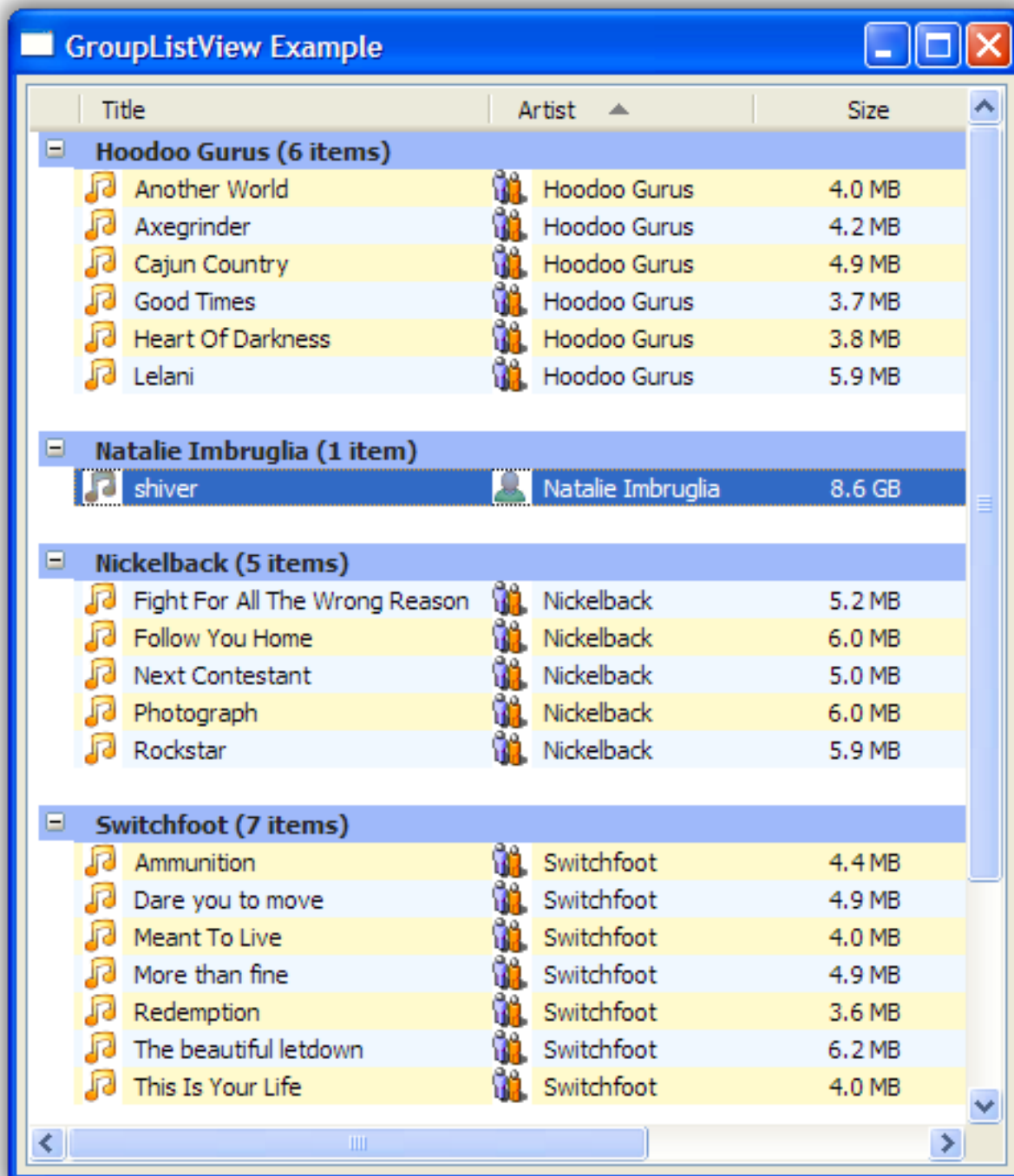
To do something other than the default processing, you can listen for two events: `ObjectListView.EVT_CELL_EDIT_STARTING` and `ObjectListView.EVT_CELL_EDIT_FINISHING`.

```
MORE HERE
```

## 4.6 Using a GroupListView

A flat list is enough in many cases, but sometimes it would be really nice to be able to put the model objects into groups, making it easier for users to see where something belongs. It's nice that our database of songs can be sorted by "Album," but it would be even nicer if the control was able to put all the tracks for an album together under their own title.

This is what a *GroupListView* does. And it looks like this:



### 4.6.1 Understanding the process

To make a `GroupListView` work, the control needs to collect the model objects into different groups. This is done in the following steps:

1. Calculate a “group key” for each object.
2. All objects with the same group key are placed into the same group.
3. The group key is converted into a string, which become the title of the group.

Understanding this simple process is the key to working with a `GroupListView`. Remember this, and you will have conquered the `GroupListView`.

### 4.6.2 Getting the “group key”

The “group key” for an object is normally whatever value the object has in the “group by” column. So if list of tracks is being grouped by the “Artist” column, the group key for a track will be, for example, “Coldplay” or “Nickelback”.

However, that isn’t always the best group key. For example, without any other configuration, if we group our tracks by the “Last Played” column, every track ends up in it’s own group (bonus points if you can explain why).

If we want the tracks to be more usefully grouped, we will need to install a group key getter for the “Last Played” column. In this case, we want all tracks that were played in the same month to be placed into the same group. So for each track, we want to calculate the month it was last played and return that as the group key.

We would do this by creating a function and then installing it as the group key getter for the column:

```
def lastPlayedGroupKey(track):
 # We only want to group tracks by the month in which they were played
 return datetime.date(track.lastPlayed.year, track.lastPlayed.month, 1)
...
ColumnDefn("Last Played", "left", 100, "lastPlayed", groupKeyGetter=lastPlayedGroupKey)
```

The `groupKeyGetter` can be specified in the same ways that a `valueGetter` can be specified:

1. a callable that accepts the model whose group key should be calculated.
2. **a string which will be treated as:**
  - the name of parameter-less instance method
  - the name of an instance variable
  - an index into a dictionary-like object
3. an integer, used as an index into a indexable collection

#### Grouping by initial letter

One common pattern is for objects to be grouped by the first letter of a string value. For example on the “Title” column, all tracks starting with “A” would be grouped together. This is so common that there is a built-in way to do it: set `useInitialLetterForGroupKey` to `True` for a column:

```
ColumnDefn("Title", "left", 120, "title", imageGetter=musicImage, useInitialLetterForGroupKey=True)
```

### 4.6.3 Converting the “group key” to title

Once the group keys have been calculated for each model object, and all the model objects with the same group key have been collected into their respective groups, we are almost ready to present the groups to the user.

The final remaining step is to decide that to call the group. The name of a group is normally its group key converted to a string. This works well when the group key is a string, and reasonably well for other data types, but sometimes you need something different. In those cases, you can install a *groupKeyConverter* on the column:

```
def lastPlayedGroupKeyConverter(groupKey):
 # Convert the given group key (which is a date) into a representation string
 return groupKey.strftime("%B %Y")
...
ColumnDefn("Last Played", "left", 100, "lastPlayed", groupKeyGetter=lastPlayedGroupKey,
 groupKeyConverter=lastPlayedGroupKeyConverter)
```

Here our group key is the first of the month in which the track was last played. Without a *groupKeyConverter*, the title of the groups would look like “2008/05/01”. But with our *groupKeyConverter*, the title of the groups end up like “May 2008”, which is nicer.

The *groupKeyConverter* operates in the same way that a *stringConverter* operates.

### 4.6.4 Using *SetGroups()* directly

All of the above steps are used when you give the *GroupListView* a straight list of model objects, leaving the *GroupListView* to convert the model objects into groups. It is also possible for the programmer to manually create the groups and then tell the *GroupListView* to show the groups that the programmer has created.

Each group is represented by a *ListGroup* object. A *ListGroup* basically consists of a title and a list of model objects that are to be shown in the groups.

Once the programmer has created a list of *ListGroup* objects, they should be given to *SetGroups()* method. The order of the groups in the list, and the order of the model objects in the group are the order in which they will be presented to the user.

If you manually create the groups, you will need to handle sorting yourself, or turn off sorting altogether. This is necessary since the *GroupListView* will not know how to recalculate the groups.

### 4.6.5 Customizing using events

A *GroupListView* triggers several events which allow the programmer to change key behaviours of the control.

|                   |                                                                                                                                                                                                                                                     |
|-------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| EVT_GROUP_CREATED | Triggered when a new collection of groups has been created but not yet displayed to the user. The handler of this event can make any changes they like to the groups, including the names of the group and the members.                             |
| EVT_GROUP_SORTED  | Triggered when the groups need to be sorted. Both the groups themselves and the model objects within each group should be sorted. The handler of this event should call <i>Handled()</i> on the event, otherwise normal sort processing will occur. |
| EVT_EXPANDING     | Triggered when one or more groups is being expanded. The handler of this event can call <i>Veto()</i> to prevent the groups from being expanded.                                                                                                    |
| EVT_EXPANDED      | Triggered after one or more groups have been expanded. This is a notification event only.                                                                                                                                                           |
| EVT_COLLAPSING    | Triggered when one or more groups is being collapsed. The handler of this event can call <i>Veto()</i> to prevent the groups from being collapsed.                                                                                                  |
| EVT_COLLAPSED     | Triggered after one or more groups have been collapsed. This is a notification event only.                                                                                                                                                          |

#### 4.6.6 Other capabilities

A *GroupListView* can stop showing groups and revert to a straight *ObjectListView* by calling *SetShowGroups(False)*.

### 4.7 Using a ListCtrlPrinter

A *ListCtrlPrinter* takes an *ObjectListView* (or plain old `wx.ListCtrl`) and effortlessly turns it into a pretty report. With only two lines of code, you can produce a nice report like this:

Playing with ListCtrl Printing

| Playing with ListCtrl Printing                                                                           |                                                                                                       |                        |              |        |
|----------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------|------------------------|--------------|--------|
| Title                                                                                                    | Artist                                                                                                | Last Played            | Size         | Rating |
|  Lelani                 |  Hoodoo Gurus        | 2007-10-22<br>08:45:00 | 6186598.4    | 60     |
|  Tojo                   |  Hoodoo Gurus        | 2007-10-22<br>08:48:00 | 4299161.6    | 60     |
|  My Girl                |  Hoodoo Gurus        | 2007-11-12<br>07:57:00 | 3460300.8    | 80     |
|  Be My Guru             |  Hoodoo Gurus        | 2008-03-20<br>12:15:00 | 3460300.8    | 100    |
|  I Want You Back        |  Hoodoo Gurus        | 2007-11-12<br>07:42:00 | 4089446.4    | 80     |
|  I Was A Kamikaze Pilot |  Hoodoo Gurus        | 2007-10-22<br>09:00:00 | 4089446.4    | 60     |
|  Bittersweet            |  Hoodoo Gurus        | 2007-10-22<br>09:04:00 | 4928307.2    | 60     |
|  Poison Pen             |  Hoodoo Gurus        | 2007-10-22<br>09:11:00 | 5242880      | 60     |
|  In The Wild           |  Hoodoo Gurus       | 2007-10-22<br>09:14:00 | 4089446.4    | 60     |
|  Whats My Scene?      |  Hoodoo Gurus      | 2007-11-12<br>07:51:00 | 4823449.6    | 100    |
|  Heart Of Darkness    |  Hoodoo Gurus      | 2007-10-22<br>09:21:00 | 3984588.8    | 60     |
|  Good Times           |  Hoodoo Gurus      | 2008-03-20<br>12:18:00 | 3879731.2    | 80     |
|  Cajun Country        |  Hoodoo Gurus      | 2007-10-22<br>09:28:00 | 5138022.4    | 60     |
|  Axe-grinder          |  Hoodoo Gurus      | 2007-10-22<br>09:32:00 | 4404019.2    | 60     |
|  Another World        |  Hoodoo Gurus      | 2008-03-20<br>12:21:00 | 4194304      | 80     |
|  shiver               |  Natalie Imbruglia | 2008-03-09<br>09:51:00 | 9234179686.4 | 80     |
|  Follow You Home      |  Nickelback        | 2008-03-06<br>10:42:00 | 6291456      | 40     |

Bright Ideas Software 08/12/08 23:28:38 1 of 3

#### 4.7.1 When would I want to use it?

Your shiny new application is finished to perfection — and it's a week ahead of schedule (we're imagining here, so let's be completely unrealistic). After your demonstration to The Management, the CEO says, "It's great! I love it! But I want to have all that information as a report. And I want to be able to fiddle with the columns in the report, resize



them, any way I want. It has to be sortable, and groupable, just like we can see on the screen there. And of course, we have to be able to print preview it before we print it. And can we have it finished by tomorrow?" You consider briefly whether you should mention the PrintScreen key, but you doubt he would be enthused with that solution.

Management love reports. Programmers aren't so keen. They often appear as an afterthought to the requirement. Yet producing nice looking reports is not a trivial task. If you using other languages, you can buy a commercial product (if you have Blizzard's budget), but with wxPython, there are not that many options. You normally have to write it all yourself, which can be frustrating. wx's printing scheme can cause even strong programmers to go weak in the knees – even with Robin's wonderful book for reference.

For a lazy and vain programmer like myself, what I really want is something that takes no effort to implement, yet produces wonderful results. I want to be able to go back to my CEO the same day, show him the nice looking reports that do exactly what he wanted, and then remind him about my overdue raise.

The ListCtrlPrinter is designed to be just such a solution.

### 4.7.2 How do I use it in my project?

As always, the goal is for this to be as easy to use as possible. A typical usage should be as simple as:

```
printer = ListCtrlPrinter(self.myListCtrl, "My Report Title")
printer.PrintPreview()
```

This code create a new ListCtrlPrinter, telling it what ListCtrl it should print and what the report should be titled. Then a print preview of the report is opened.

A more complete example might look like this:

```
printer = ListCtrlPrinter(self.lv, "Graviton Collision Statistics")
printer.ReportFormat = ReportFormat.Normal("Lucida Bright")
printer.ReportFormat.IsColumnHeadingsOnEachPage = True

printer.PageHeader("Monthly Bad Science Report")
printer.PageFooter("Bright Ideas Software", "%(date)s", "%(currentPage)d of %(totalPages)d")
printer.Watermark("Work hard!")

printer.PrintPreview(self)
```

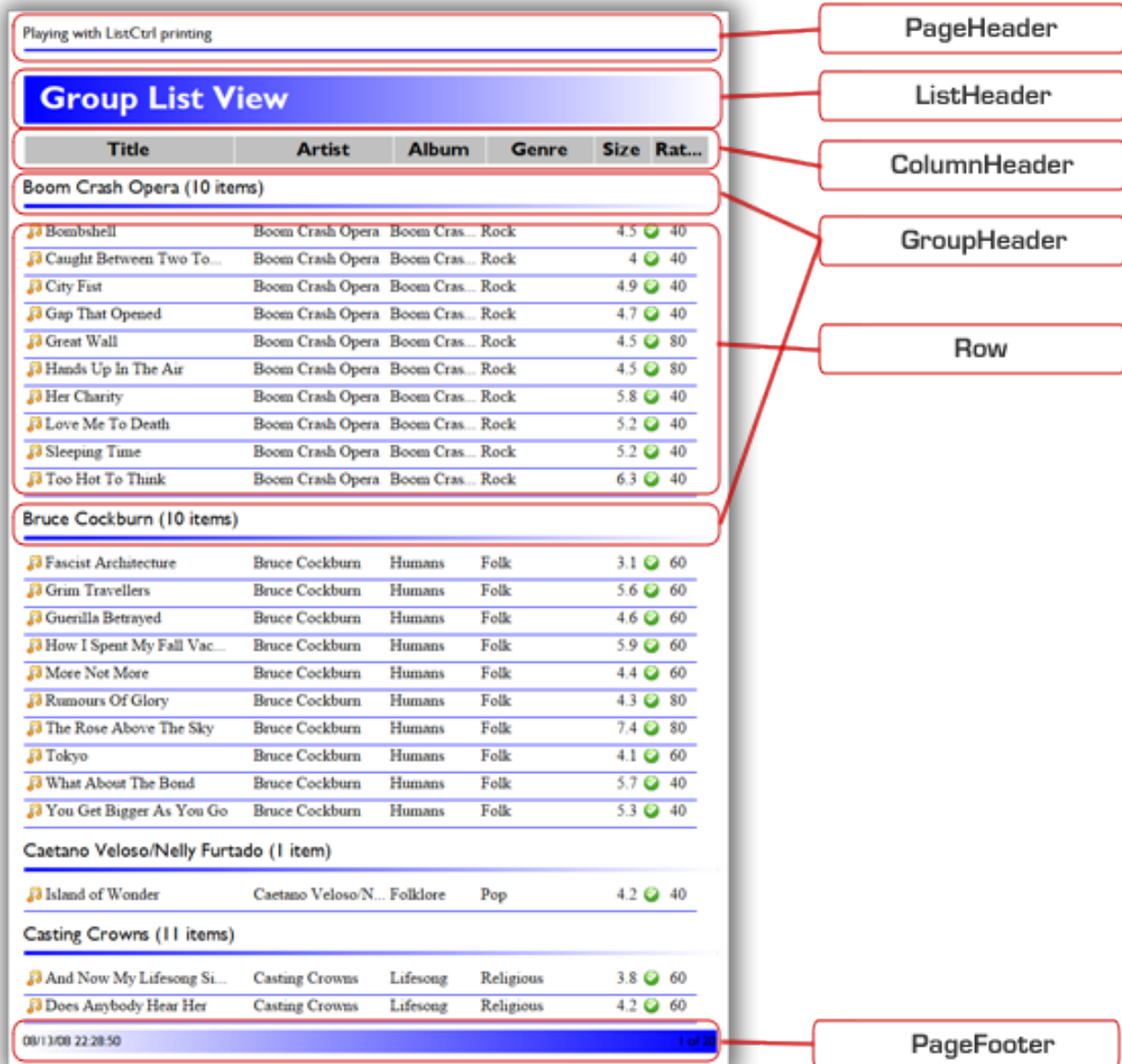
### 4.7.3 Primary commands

The ListCtrlPrinter has commands that match the normal printing commands, which should be hooked into the menu structure of your application.

- **PageSetup()** Opens the normal “Page Setup” dialog that allows the user to choose the page orientation and margins, as well as choose the printer for output.
- **PrintPreview()** Open a print preview dialog, from which the user can also print the report.
- **Print()** Opens the “Print” dialog that lets the user choose which printer, the pages and other details before printing the report.

### 4.7.4 Working in a structured environment

A report consists of several blocks, each of which stacks vertically. The structure of a report is like this:



The PageHeader and PageFooter repeat on each page (obviously). The ColumnHeader repeats on each page, depending on the ReportFormat options.

#### 4.7.5 Controlling the appearance

The formatting of a report is controlled completely by the ReportFormat object of the ListCtrlPrinter. To change the appearance of the report, you change the settings in this object.

These properties control the appearance of the report as a whole:

- **IncludeImages** Should images from the ListCtrl be included in the report?
- **IsColumnHeadingsOnEachPage** If this is True, the column headers will be repeated at the top of each page.
- **IsShrinkToFit** If this is True, the report will be shrunk so that all the column of the ListCtrl can fit

within the width of page.

- **UseListCtrlTextFormat** If this is True, the format (text font and color) of the rows will be taken from the `ListCtrl` itself, rather than the `Cell.BlockFormat` object. Useful if your `ListCtrl` has fancy formatting on the rows that you want to replicate in the printed version.

As was illustrated above, a report consists of various sections (called “blocks”). Each of these blocks has a matching `BlockFormat` object in the `ReportFormat`. To modify the appearance of a block, you modify its matching `BlockFormat` object. So, to modify the format of the page header, you change the `ReportFormat.PageHeader` object.

A `ReportFormat` object has the following properties which control the appearance of the matching sections of the report:

- `PageHeader`
- `ListHeader`
- `ColumnHeader`
- `GroupTitle`
- `Row`
- `ListFooter`
- `PageFooter`

These properties return `BlockFormat` objects, which have the following properties:

- **CanWrap** If the text for this block cannot fit horizontally, should be wrap to a new line (True) or should it be truncated (False)?
- **Font** What font should be used to draw the text of this block
- **Padding** How much padding should be applied to the block before the text or other decorations are drawn? This can be a numeric (which will be applied to all sides) or it can be a collection of the paddings to be applied to the various sides: (left, top, right, bottom).
- **TextAlignment** How should text be aligned within this block? Can be `wx.ALIGN_LEFT`, `wx.ALIGN_CENTER`, or `wx.ALIGN_RIGHT`.
- **TextColor** In what color should be text be drawn?

The blocks that are based on cells (`PageHeader`, `ColumnHeader`, `Row`, `PageFooter`) can also have the following properties set:

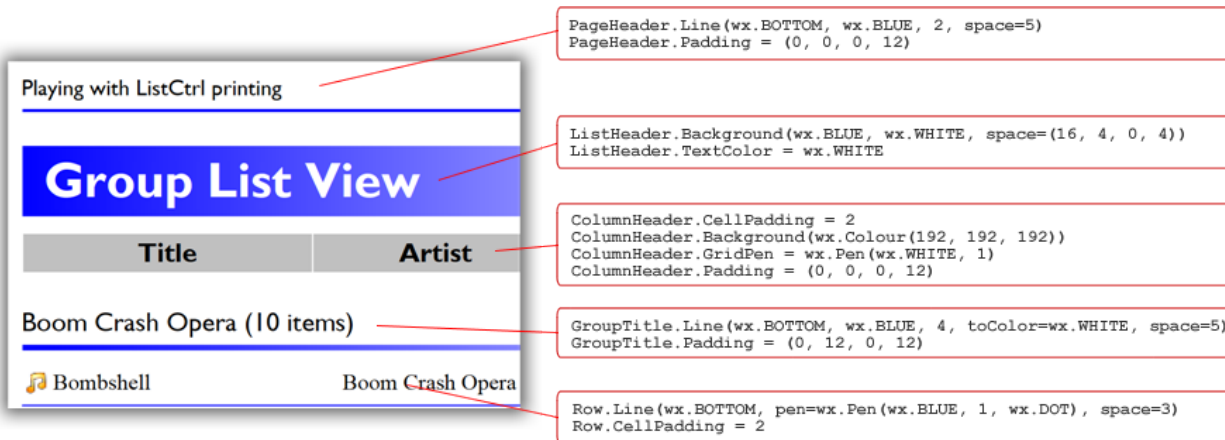
- **AlwaysCenter** Will the text in the cells be center aligned, regardless of other settings?
- **CellPadding** How much padding should be applied to this cell before the text or other decorations are drawn? This can be a numeric (which will be applied to all sides) or it can be a collection of the paddings to be applied to the various sides: (left, top, right, bottom).
- **GridPen** What Pen will be used to draw the grid lines of the cells?

In addition to these properties, there are some methods which add various decorations to the blocks:

- **Background(color=wx.BLUE, toColor=None, space=0)** This gives the block a solid color background (or a gradient background if `toColor` is not None). If `space` is not 0, `space` pixels will be subtracted from all sides from the space available to the block.
- **Frame(pen=None, space=0)** Draw a rectangle around the block in the given pen
- **Line(side=wx.BOTTOM, color=wx.BLACK, width=1, toColor=None, space=0, pen=None)** Draw a line on a given side of the block. If a pen is given, that is used to draw the line (and the other

parameters are ignored), otherwise a solid line (or a gradient line if *toColor* is not None) of *width* pixels is drawn.

### 4.7.6 Can't you just show me what these things do?



### 4.7.7 Understanding the process

Use The Source Luke (at least until I write this part of the docs)

### 4.7.8 Other things to be aware of

- A `ListCtrlPrinter` only works on `ListCtrls` that are in report view. It will ignore any `ListCtrl` that is in any other view.
- You can set the left *Padding* of the *ColumnHeader* format and *Row* format to different values. This results in the column headers not lining up with the rows. This should be understood as a feature.
- For reasons that are still not clear to me, images that come from BMP files will not print on some (most?) printers. Images that come from PNG and other formats work fine.
- The `ListCtrlPrinter` is not designed to be general purpose reporting solution. There are no running totals, macro language, or ODBC data sources. It just prints `ListCtrls`.

## 4.8 Frequently Asked Questions

Some questions and issues surface regularly on the Forums or in emails. This section has several of the most common questions. Please read the questions before asking questions on the Forum. Several people have been known to have blood pressure problems.

### 4.8.1 What platforms does it work on?

ObjectListView has been extensively tested on Windows and somewhat tested on Linux (Ubuntu).

I have no experience on other platforms but would welcome feedback. I'd be especially interested if someone from MacLand could test it, especially with native mode enabled.

### 4.8.2 Can an ObjectListView have rows of different heights? Can it word-wrap?

No.

ObjectListView is a wrapper for the underlying ListCtrl. It makes a ListCtrl much easier to use, but it can't change the control's basic behaviour. One limitation of a ListCtrl is it that cannot have rows of different heights. There is no way to make one row be taller than other rows. It's just not possible. So there is no way to word wrap a long line on just one row either.

If being able to have rows of different heights is essential to you, ObjectListView is not your solution.

### 4.8.3 Why doesn't the ObjectListView auto-update when I change my model objects?

*I have ObjectListView that's showing my model objects. But when I change the values in my model, the ObjectListView doesn't update. What's going wrong here?*

Nothing. That's what it is supposed to do.

The ObjectListView knows nothing about your model objects, and particularly it doesn't know when they have been changed. Only you know that. When you know that a model object has changed and needs to be updated, you can either call `RefreshObject()` to update just one object, or you can call `RepopulateList()` to rebuild everything at once.

### 4.8.4 Why doesn't it do *some-feature-I-really-want*?

It could be that I simply haven't thought of it. Or it could be that I have thought of it but it just isn't possible.

Remember that ObjectListView is just a wrapper around `wx.ListCtrl`. It can make the ListCtrl a little easier to use, and can add some helper functions, but it can't change the basic behaviour of the control.

One thing I would really like to add is owner drawing. But this is not supported by a ListCtrl, so ObjectListView cannot have it either.

### 4.8.5 Why is the text of the first column indented by about 20 pixels?

This shows up when you have a ListCtrl that doesn't have an icon in the first column. The control still leaves space for the icon, even when there isn't one.

If the ListCtrl doesn't have a small image list, this indent disappears. But as soon as the control has a small image list, even an empty one, the text of the first column will be indented. Unfortunately, almost all ObjectListViews have a small image list, since showing sort indicators in the column headers uses the small image list.

So, if you really want to get rid of this indent, make an ObjectListView which isn't sortable (pass "`sortable=False`" to the constructor) and don't add any images to the control. The indent will disappear – but the list will not be sortable.

## 4.9 Major Classes Reference

- `:ref:objectlistview-class`
- `:ref:columnndefn-class`
- `:ref:grouplistview-class`
- `:ref:batchedupdate-class`

## 4.9.1 ObjectListView

---

## 4.9.2 ColumnDefn

---

## 4.9.3 GroupListView

---

## 4.9.4 BatchedUpdate

## 4.10 Change Log

2008-09-04 23:12 (#250) - setup.py

- v1.2

2008-09-04 22:30 (#249) - ObjectListView/ObjectListView.py

- Correct an incomplete comment

2008-09-04 22:30 (#248) - Examples/SqlExample.py

- Correctly locate primary key when there is a WHERE clause

2008-09-04 22:30 (#247) - Examples/BatchedUpdateExample.py

- Nicely format file size column

2008-09-04 22:29 (#246) - docs/whatsnew.rst, docs/index.rst, docs/recipes.rst

- Final changes for v1.2

2008-09-02 23:25 (#245) - docs/features.rst, docs/.static/icon.ico, docs/whatsnew.rst, docs/listCtrlPrinter.rst, docs/conf.py, docs/m

- Updated in preparation for v1.2 release

2008-09-02 23:22 (#244) - Examples/UsingVirtualListExample.py

- Simplified initial insertions (removed executemany)

2008-09-02 23:21 (#243) - ObjectListView/\_\_init\_\_.py

- Added BatchedUpdate

2008-09-02 23:20 (#242) - Examples/BatchedUpdateExample.py

- First version

2008-09-02 23:20 (#241) - ObjectListView/ObjectListView.py

- Added BatchedUpdate adaptor
- Improved speed of selecting and refreshing by keeping a map of objects to indices
- Added GetIndexOf()
- Removed flicker from FastObjectListView.AddObjects() and RefreshObjects()

2008-08-31 23:09 (#240) - Examples/BatchedUpdateExample.py

- Scanning now works

**2008-08-31 20:58 (#239) - Examples/BatchedUpdateExample.py**

- First hand done layout

**2008-08-31 18:07 (#238) - Examples/BatchedUpdateExample.py**

- Initial checkin

**2008-08-28 22:50 (#237) - CHANGELOG.txt, docs/changelog.rst**

- Rebuilt change log

**2008-08-28 22:41 (#235) - test/test\_ObjectListView.py**

- Added filtering tests

**2008-08-28 22:41 (#234) - ObjectListView/ObjectListView.py**

- Added GetObjects() and GetFilteredObjects()
- Added resortNow parameter to SetSortColumn()

**2008-08-28 22:39 (#233) - ObjectListView/Filter.py**

- Added Filter.Chain
- Added text constructor parameter to TextSearch

**2008-08-28 01:22 (#232) - ObjectListView/ObjectListView.py**

- Correct AddObjects() when a filter is in effect
- Made RebuildGroups() public

**2008-08-28 01:21 (#231) - Examples/Demo.py**

- Implement search controls on several tabs

**2008-08-28 01:20 (#230) - ObjectListView/Filter.py**

- Make text search handle non-report views better

**2008-08-27 23:59 (#229) - ObjectListView/ObjectListView.py, ObjectListView/\_\_init\_\_.py, ObjectListView/Filter.py**

- Filters work

**2008-08-25 10:51 (#225) - Examples/Demo.py**

- Use AddObjects() for “Add 1000” commands

**2008-08-25 10:50 (#224) - ObjectListView/ObjectListView.py**

- Added AddObjects()/RemoveObjects() and friends
- Removed duplicate code when building/refreshing/adding objects
- One step closer to secondary sort column support

**2008-08-22 19:38 (#220) - docs/listCtrlPrinter.rst**

- Added formatting picture

**2008-08-20 22:21 (#219) - Examples/Demo.py**

- Changed to use new properties on ListViewPrinter

**2008-08-20 22:20 (#218) - ObjectListView/ListCtrlPrinter.py**

- Consistently use properties on ListCtrlPrinter (ReportFormat, PageFooter, PageHeader, Watermark and PrintData are now all properties)
- Removed ListCtrlPrinter.PageHeader(), ListCtrlPrinter.PageFooter(), ListCtrlPrinter.Watermark(), since they are now replaced with properties (and make more sense that way)

**2008-08-20 00:28 (#217) - docs/.static/icon.ico, docs/images/listctrlprinter-example2.png, docs/images/listctrlprinter-structure.png**

- Added lots of documentation about ListCtrlPrinter

**2008-08-20 00:27 (#216) - ObjectListView/ListCtrlPrinter.py**

- Moved AlwaysCenter and CanWrap to BlockFormat
- Improved docs

**2008-08-18 10:04 (#214) - THANKS.txt**

- Added Werner Bruhin to THANKS

**2008-08-18 10:03 (#213) - ObjectListView/ObjectListView.py, ObjectListView/\_\_init\_\_.py, ObjectListView/OLVEvent.py**

- Handle model objects that cannot be hashed
- Added editing started and finished events

**2008-08-18 10:02 (#212) - Examples/SqlExample.py**

- Reorganized code slightly

**2008-08-18 00:37 (#211) - Examples/SqlExample.py**

- Initial checkin

**2008-08-17 21:47 (#210) - ObjectListView/WordWrapRenderer.py**

- Second attempt at avoid bug in wordwrap module

**2008-08-16 23:31 (#209) - ObjectListView/WordWrapRenderer.py**

- Allow truncated text to be vertically aligned

**2008-08-16 23:24 (#208) - ObjectListView/ListCtrlPrinter.py**

- Use RunningBlockPusher to simplify code
- Allow truncated strings to be vertically aligned

**2008-08-16 22:58 (#207) - ObjectListView/ListCtrlPrinter.py**

- Centralize cell width calculation (again)
- Gracefully handle substitutions that fail

**2008-08-16 22:55 (#206) - Examples/Demo.wxg, Examples/Demo.py**

- All control changes on ListCtrlPrinting now update the preview

**2008-08-16 10:23 (#205) - ObjectListView/WordWrapRenderer.py**

- Avoid bug in wordwrap module
- use DCClipper
- Simplified some code

**2008-08-16 09:47 (#204) - ObjectListView/ListCtrlPrinter.py**



- Column width is now calculated by the column headers only
- Added ListCtrlPrinter.GetPrintData()
- Make sure print data is destroyed after printing
- Remove print statements

**2008-08-16 09:38 (#203) - ObjectListView/ObjectListView.py**

- Added ensureVisible parameter to SelectObject()

**2008-08-13 00:09 (#199) - ObjectListView/ObjectListView.py, ObjectListView/\_\_init\_\_.py, ObjectListView/OLVPrinter.py, Obj**

- Allow text to be vertically aligned in cells
- Improved some docs
- Renamed OLVPrinter to be ListCtrlPrinter

**2008-08-13 00:07 (#198) - Examples/Demo.wxg, Examples/Demo.py**

- Changed OLVPrinter to be ListCtrlPrinter

**2008-08-13 00:06 (#197) - docs/images/listctrlprinter-example1.png, docs/images/grouplist-example1.png, docs/listCtrlPrinter.rs**

- Began documenting ListCtrlPrinter

**2008-08-12 19:40 (#195) - ObjectListView/OLVPrinter.py**

- Added TooMuch() formatting
- Create instance variables normally in ReportFormat – rather than using setattr()
- Simplified scaling of rows
- Changed some method names to better reflect their more generic role
- Changed variable to refer to a listview rather than an objectlistview

**2008-08-08 11:36 (#194) - Examples/Demo.wxg, Examples/Demo.py**

- Completely reworked ListCtrl printing tab

**2008-08-08 11:35 (#193) - test/test\_ObjectListView.py**

- Make adjustments for GroupListView now being virtual

**2008-08-08 11:34 (#192) - ObjectListView/OLVPrinter.py**

- Added ImageDecoration
- Removed report title and footer
- Corrected (and optimized) counting pages and skipping pages

**2008-08-06 20:44 (#191) - ObjectListView/OLVPrinter.py**

- Row height is now calculate for the whole row, not just the current slice
- Separated water format from the watermark text
- Allow blocks to decide not to print themselves
- Use a dummy DC when counting total pages

**2008-08-06 10:14 (#190) - Examples/Demo.wxg, Examples/Demo.py**

- Rearranged Printing panel

- Inline print preview has water mark

**2008-08-05 22:43 (#189) - ObjectListView/ObjectListView.py**

- GroupListView is now implemented as a virtual list
- Moved putBlankLineBetweenGroups into GroupListView (and out of ObjectListView)

**2008-08-05 22:40 (#188) - ObjectListView/\_\_init\_\_.py**

- Added ListGroup into classes exported from the module

**2008-08-05 22:39 (#187) - Examples/Demo.wxg, Examples/Demo.py**

- ListCtrl print previewing now works more or less completely

**2008-08-04 16:43 (#186) - Examples/Demo.wxg, Examples/Demo.py**

- Added List printing example tab (not yet complete)

**2008-08-04 16:43 (#185) - ObjectListView/OLVPrinter.py**

- Header and footers are now ThreeCellBlock
- Added substitutions on text strings
- Print garbage pages to a MemoryDC
- Added IncludeImages and UseListCtrlTextFormat into ReportFormat
- Added ReportFormat.Minimal()
- Column headers can now be repeated on each page

**2008-08-04 16:37 (#184) - ObjectListView/\_\_init\_\_.py**

- Added list printing stuff

**2008-08-02 10:26 (#183) - ObjectListView/OLVPrinter.py**

- Now includes images
- Cells can now be truncated
- Decorations can now be either over or under their block

**2008-08-02 10:24 (#182) - ObjectListView/ObjectListView.py**

- Added putBlankLineBetweenGroups to GroupListView
- Handle None as aspect values

**2008-08-02 10:23 (#181) - ObjectListView/WordWrapRenderer.py**

- Changed to use wx.lib.wordwrap
- Added DrawTruncatedString()

**2008-08-02 10:22 (#180) - Examples/GroupExample.py, Examples/ExampleModel.py, Examples/Demo.py**

- Remove locale dependence from date parsing

**2008-07-31 23:50 (#179) - ObjectListView/OLVPrinter.py**

- Watermarks now work

**2008-07-31 21:38 (#178) - ObjectListView/OLVPrinter.py**

- AlwaysCenterColumnHeader and IsShrinkToFit now work

**2008-07-31 11:51 (#177) - ObjectListView/OLVPrinter.py**

- Margins, scaling and printer boundaries all now work

**2008-07-31 10:49 (#176) - test/test\_OLVPrinter.py**

- Added TextBlock tests

**2008-07-31 10:48 (#175) - ObjectListView/OLVPrinter.py**

- Made work with plain ListCtrls
- Cell decorations and grids now work
- Added gradient lines and backgrounds

**2008-07-30 17:06 (#174) - ObjectListView/ObjectListView.py**

- Removed reference to testing variable ‘\_\_rows’

**2008-07-30 17:05 (#173) - docs/groupListView.rst, docs/index.rst, docs/gettingStarted.rst**

- COrrrected some small mistakes in docs

**2008-07-30 11:46 (#172) - CHANGELOG.txt, docs/changelog.rst, setup.py**

- V1.1 release

**2008-07-28 22:10 (#170) - ObjectListView/OLVPrinter.py**

- Move grid drawing into CellBlock. Removed GridDecoration
- Added Bucket and use them instead of dictionaries
- Correctly handle GroupListView
- Made compatible with plain ListCtrls

**2008-07-28 22:04 (#169) - ObjectListView/WordWrapRenderer.py**

- Made all methods static

**2008-07-27 00:22 (#168) - ObjectListView/OLVPrinter.py**

- Added GridDecoration, FrameDecoration
- Changed technique of page header/footers

**2008-07-26 00:30 (#167) - docs/features.rst, docs/whatsnew.rst, docs/groupListView.rst, docs/.templates/layout.html, docs/conf.py**

- Added documentation about GroupListView

**2008-07-26 00:28 (#166) - Examples/GroupExample.py, Examples/Demo.py, Examples/SimpleExample1.py, Examples/SimpleExample2.py**

- Minor corrections to examples

**2008-07-26 00:27 (#165) - ObjectListView/ObjectListView.py**

- Correctly trigger and handle group related events
- Made EmptyListMsg work under Linux
- Correct location of expand/collapse images under Linux
- Removed some isinstance() and callable() tests

**2008-07-26 00:23 (#164) - ObjectListView/\_\_init\_\_.py**

- Export group related events

**2008-07-26 00:23 (#163) - ObjectListView/OLVEvent.py**

- Complete implementation of group related events

**2008-07-26 00:21 (#162) - ObjectListView/WordWrapRenderer.py**

- Factored out \_CalculateLineHeight()
- Set up a nicer font under Linux

**2008-07-26 00:20 (#161) - test/test\_OLVPrinter.py**

- Initial checkin

**2008-07-25 15:52 (#160) - ObjectListView/WordWrapRenderer.py**

- Initial checkin

**2008-07-25 13:31 (#159) - ObjectListView/OLVPrinter.py**

- Pagination now works correctly
- Correctly calculates total number of pages

**2008-07-24 21:07 (#158) - ObjectListView/OLVPrinter.py**

- Before changing to use ReportEngine

**2008-07-24 10:39 (#157) - docs/groupListView.rst**

- Initial checkin

**2008-07-23 11:26 (#154) - docs/features.rst**

- Included GroupListView in features

**2008-07-23 11:25 (#153) - ObjectListView/OLVPrinter.py**

- More WIP

**2008-07-23 11:24 (#152) - ObjectListView/ObjectListView.py**

- Consistently use GetSortColumn()
- Updated some docs

**2008-07-19 15:57 (#151) - ObjectListView/OLVPrinter.py**

- Work in progress

**2008-07-17 20:40 (#150) - ObjectListView/ObjectListView.py**

- Added ability to turn off groups in GroupListView
- Added ability to lock the group by column
- Changed ObjectListView to use 'innerList'
- SetColumns() can now retain the current model objects
- Optimized sort key getter and munging. 30% faster!

**2008-07-17 20:34 (#147) - Examples/Demo.wgx, Examples/Demo.py**

- Added Group tab to demo

**2008-07-17 20:34 (#146) - Examples/GroupExample.py, Examples/ExampleModel.py, Examples/SimpleExample1.py, Examples/**

- Changed to use ExampleModel.py

**2008-07-17 15:03 (#145) - ObjectListView/ObjectListView.py**

- Refactored VirtualObjectListView and FastObjectListView to have common base class (AbstractVirtualObjectListView). This made FastObjectListView much simpler
- Added GetPrimaryColumn()

**2008-07-17 13:19 (#144) - Examples/GroupExample.py, Examples/Demo.py**

- In Demo.py, give the simple list a separate column for the checkbox
- In GroupExample.py, give the list a checkbox and make the control editable.

**2008-07-17 13:17 (#143) - test/test\_ObjectListView.py**

- Fixed all problems with tests
- GroupListView now passes all general ObjectListView tests

**2008-07-17 13:15 (#142) - ObjectListView/OLVEvent.py**

- Added new group events

**2008-07-17 13:15 (#141) - ObjectListView/ObjectListView.py**

- Allow GroupListView to have checkboxes too
- GroupListView now copy objects to clipboard correctly
- Use native renderer for expand/collapse images
- Added “handleStandardKeys”
- GetSelectedObject() now processes at most 2 rows
- Correctly calculate primary column instead of just assuming column 0
- Correctly handle column images
- Search-by-typing now works in GroupListView
- Don’t allow editing of groups and empty rows
- Added groupTitleSingleItem and groupTitlePluralItems to ColumnDefn

**2008-07-15 15:39 (#140) - Examples/GroupExample.py**

- Example showing capabilities of GroupListView

**2008-07-15 15:38 (#139) - ObjectListView/ObjectListView.py, ObjectListView/\_\_init\_\_.py, ObjectListView/OLVEvent.py**

- First take at groupable ListCtrl

**2008-07-14 20:46 (#138) - ObjectListView/ObjectListView.py**

- Added CopySelectionToClipboard and CopyObjectsToClipboard

**2008-07-08 20:37 (#135) - ObjectListView/ObjectListView.py**

- Headers can now have images
- Fixed Linux specific issues
- Fixed cell editor bug when double clicking out of list bounds

**2008-06-27 22:13 (#134) - ObjectListView/ObjectListView.py**

- Updated docs to match v1.0.1

**2008-06-23 19:50 (#132) - Examples/UsingVirtualListExample.py**

- Replace hardcoded path with wx.StandardPaths

**2008-06-22 22:35 (#128) - ObjectListView/ObjectListView.py**

- Fixed bug where an imageGetter that returned 0 was treated as if it returned -1 (i.e. no image)

**2008-06-20 00:16 (#126) - TODO.txt, setup.py, README.txt**

- Changed feature list
- Changed download location

**2008-06-20 00:15 (#125) - docs/features.rst, docs/whatsnew.rst, docs/.templates/layout.html, docs/conf.py, docs/majorClasses.rst**

- Update to version 1.0.1
- Added “Class Docs” section to menu
- Added new sections to Features and What’s New

**2008-06-20 00:12 (#124) - Examples/SimpleExample1.py**

- Enable logging

**2008-06-20 00:11 (#123) - Examples/Demo.py**

- Added more checkboxes
- Corrected some typing errors

**2008-06-20 00:09 (#122) - ObjectListView/ObjectListView.py, ObjectListView/\_\_init\_\_.py, ObjectListView/OLVEvent.py**

- Allowed for custom sorting, even on virtual lists
- Factored out test for binary search
- Added OLVColumn.useBinarySearch
- Added EVT\_SORT and its friends

**2008-06-20 00:05 (#121) - test/test\_ObjectListView.py**

- Added tests for virtual lists

**2008-06-18 09:48 (#118) - setup.py**

- Change download location
- Change feature list

**2008-06-17 20:44 (#117) - ObjectListView/ObjectListView.py**

- Made binary searching work when column is sorted descending

**2008-06-17 00:53 (#116) - ObjectListView/ObjectListView.py**

- use binary searches when searching on sorted columns
- use MAX\_ROWS\_FOR\_UNSORTED\_SEARCH to limit linear searches when typing

**2008-06-17 00:47 (#115) - docs/.templates/layout.html, docs/faq.rst, docs/index.rst, docs/gettingStarted.rst, docs/recipes.rst**

- Changed download location of source distribution
- Added recipe about referencing columnDefns inside a valueGetter
- Rearranged slightly the getting started section.
- Added FAQ about the indent of text when there is no icon

**2008-06-16 22:43 (#114) - ObjectListView/ObjectListView.py**

- Typing searches sort column complete

**2008-06-15 21:15 (#113) - ObjectListView/ObjectListView.py**

- Added 'sortable' parameter. VirtualObjectListView are now not sortable by default
- Improved management of image lists

**2008-06-15 21:13 (#112) - setup.py, MANIFEST.in**

- Include bmp files in MANIFEST.in
- Correct some details in setup.py

**2008-06-14 22:31 (#111) - ObjectListView/CellEditor.py**

- Changed use to utf-8 encoding

**2008-06-14 22:29 (#110) - ObjectListView/ObjectListView.py**

- Renamed sortColumn to be sortColumnIndex to make it clear
- Allow returns in multiline cell editors
- Only use alternate backcolors in report view, not in the other views

**2008-06-08 21:30 (#109) - ObjectListView/ObjectListView.py**

- Clear the DC before drawing a checkbox. Needed for Linux

**2008-05-30 14:13 (#108) - ObjectListView/ObjectListView.py, test/test\_ObjectListView.py**

- Make ImageList.GetSize(0) work to empty image lists under Linux
- Added more tests, especially for FastObjectListView

**2008-05-29 14:22 (#107) - CHANGELOG.txt, docs/changelog.rst**

- v1.0 Release!

**2008-05-29 14:17 (#106) - docs/features.rst, docs/whatsnew.rst, docs/cellEditing.rst, docs/.static/features-icon.png, docs/.template**

- Finally clean up of documentation before v1.0 release

**2008-05-29 14:16 (#105) - ObjectListView/ObjectListView.py, ObjectListView/\_\_init\_\_.py, ObjectListView/CellEditor.py**

- Used named images internally
- Better handling of missing image lists
- Cleaned up some more documentation

**2008-05-29 00:25 (#104) - ObjectListView/ObjectListView.py, ObjectListView/CellEditor.py**

- Changed to use "isinstance(x, basestring)" rather than "isinstance(x, (str, unicode))"

**2008-05-28 00:22 (#102) - docs/.static/changelog-icon.png, docs/whatsnew.rst, ObjectListView/ObjectListView.py, docs/.static/gla**

- Better documentation in Demo.py
- Tidied up docs for v1.0 release
- Allow sorting by column created by CreateCheckStateColumn()

**2008-05-27 13:38 (#101) - test/test\_CellEditors.py, test/test\_ObjectListView.py, test/test\_OLVColumn.py**

- Added “.” to python path so that ObjectListView will be found even if it hasn’t been installed

**2008-05-27 13:37 (#100) - ObjectListView/ObjectListView.py, CHANGELOG.txt, FAQ.txt, COPYING.txt, ObjectListView/OLV**

- Prepare for v1.0 release

**2008-05-27 13:30 (#99) - docs/.static/faq-icon.png, docs/.static/index-icon.png, docs/.static/initial.css, docs/.static/gettingStarted-i**

- Added images to generated html
- Prepare documentation for v1.0 release

**2008-05-26 17:37 (#98) - Examples/Demo.wxg, Examples/Demo.py**

- Remove “dummy” tab

**2008-05-26 00:39 (#95) - setup.cfg, pylint.rc, AUTHORS.txt, TODO.txt, INSTALL.txt, CHANGELOG.txt, FAQ.txt, COPYING.**

- Did all work to create proper package with distutils (setup.py)

**2008-05-26 00:35 (#93) - Examples/example-images/convertImages.bat, Examples/Demo.py, Examples/example-images/convertI**

- Corrected for new directory structure

**2008-05-26 00:35 (#92) - ObjectListView/ObjectListView.py**

- Fixed pyLint annoyances

**2008-05-26 00:34 (#91) - ObjectListView/OLVEvent.py**

- Fixed pyLint annoyances

**2008-05-26 00:34 (#90) - ObjectListView/CellEditor.py**

- Fixed pyLint annoyances

**2008-05-26 00:33 (#89) - ObjectListView/\_\_init\_\_.py**

- Cleaned up a little

**2008-05-24 01:57 (#67) - docs/source/.static/orange-800x1600.png, docs/source/images/coffee.jpg, docs/source/conf.py, docs/sourc**

- Documentation near completion

**2008-05-24 01:55 (#65) - ObjectListView/ObjectListView.py**

- Added ability to name images
- Used \_ to hide “private” methods
- Improved docs
- Correctly calculate subitem rect when in ICON view
- Implemented HitTestSubItem for all platforms
- Make sure empty list msg is shown on virtual lists

**2008-05-24 01:51 (#64) - ObjectListView/CellEditor.py**

- Change editor style when listctrl is in ICON view

**2008-05-24 01:51 (#63) - ObjectListViewDemo/ObjectListViewDemo.py**

- Made sure all buttons worked



- Uses named images

**2008-05-24 01:49 (#62) - Tests/test\_ObjectListView.py**

- Added tests for checkboxes, SelectAll, DeselectAll, Refresh

**2008-05-19 21:34 (#61) - ObjectListView/ObjectListView.py**

- Added support for checkboxes
- Used “modelObject(s)” name instead of “object(s)”
- Made sure all public methods have docstrings

**2008-05-19 21:32 (#60) - Tests/test\_CellEditors.py, Tests/test\_ObjectListView.py, Tests/test\_OLVColumn.py, ObjectListView/CellEditors.py**

- Added “.” to sys.path to demo and tests
- Added demo for checkboxes
- Added tests for check boxes

**2008-05-19 21:30 (#59) - docs/source/images, docs/source/.static, Examples/images/music16.png, Examples/images/convertImage.py**

- Added Sphinx based documentation (in progress)

**2008-05-12 11:29 (#44) - OwnerDrawnEditor.py, ObjectListViewDemo.py**

- Minor changes and add svn property

**2008-05-12 11:28 (#43) - test\_CellEditors.py, test\_ObjectListView.py, test\_OLVColumn.py**

- Add some svn property

**2008-05-12 11:26 (#41) - ObjectListView/ObjectListView.py**

- Massively improved documentation. Generates reasonable docs using epydoc now.

**2008-04-23 20:13 (#40) - ObjectListView/ObjectListView.py, ObjectListView/\_\_init\_\_.py, ObjectListView/OLVEvent.py, ObjectListView/OLVColumn.py**

- Added \$Id\$

**2008-04-18 22:57 (#39) - ObjectListView/ObjectListView.py, ObjectListView/\_\_init\_\_.py, ObjectListView/OLVEvent.py, ObjectListView/OLVColumn.py**

- Updated documentation

**2008-04-18 00:00 (#38) - ObjectListView/ObjectListView.py**

- Added List Empty msg
- Cleaned up code

**2008-04-17 23:59 (#36) - ObjectListViewDemo.py**

- Added “Clear List” buttons
- Set cell edit mode
- Made more columns non-auto sizing

**2008-04-16 22:54 (#35) - ObjectListView/ObjectListView.py, ObjectListView/\_\_init\_\_.py, ObjectListViewDemo.py, ObjectListView/OLVColumn.py**

- Modularized ObjectListView
- Reorganised code within ObjectListView.py

### 2008-04-14 16:29 (#29) - test\_ObjectListView.py

- Added test for cell editing

### 2008-04-14 16:28 (#27) - ObjectListViewDemo.py

- Added Complex tab
- Made Simple tab to show what is possible with only ColumnDefs
- Give colour and font to model objects

### 2008-04-14 16:26 (#26) - ObjectListView.py

- Allow columns to have a cell editor creator function
- Handle horizontal scrolling when cell editing
- Added cell edit modes
- Handle edit during non-report views
- Correctly update slots with a previous value of None
- First cleanup of cell editing code

### 2008-04-08 00:24 (#25) - ObjectListView.py

- Cell editing finished, including model updating
- Changed manner of rebuilding list to use ListItems
- Unified rowFormatter to use ListItems. Now virtual lists use the same logic
- Improved documentation on ColumnDefn
- Lists can now be used a model objects.
- Removed sortable parameter to ObjectListView

### 2008-04-08 00:18 (#24) - test\_OLVColumn.py

- Added tests for value setting
- Added tests of list accessing
- Reorganized tests

### 2008-04-08 00:17 (#23) - ObjectListViewDemo.py

- Changed to handle new unified rowFormatter
- Allow dateLastPlayed to be updated

### 2008-04-08 00:15 (#22) - OLVEvent.py

- Allow cell value to be changed in FinishingCellEdit event

### 2008-04-08 00:15 (#21) - CellEditor.py

- Validate keys in the numeric editors

### 2008-04-07 11:13 (#20) - ObjectListView.py, ObjectListViewDemo.py

- Made to work under Linux (still needs work)

### 2008-04-07 11:12 (#19) - OLVEvent.py

- Added the source listview as a parameter

### 2008-04-07 11:12 (#18) - CellEditor.py

- Make work under Linux
- Autocomplete no longer choke on large lists

**2008-04-06 01:02 (#17) - ObjectListView.py, ObjectListViewDemo.py**

- Cell editing in progress: F2 triggers, Tabbing works
- Improved docs in ObjectListView.py
- Added example of cell editing events to demo

**2008-04-06 00:59 (#16) - OLVEvent.py**

- Initial check in

**2008-04-06 00:59 (#15) - test\_CellEditors.py, test\_ObjectListView.py, test\_OLVColumn.py**

- Separated column tests from list tests
- Added sorting tests and space filling tests
- Added basic tests for all editors

**2008-04-06 00:57 (#14) - CellEditor.py**

- Initial checkin.
- Editors for all basic types working
- Autocomplete textbox and combobox working
- Editor registry working

**2008-04-02 00:42 (#13) - ObjectListView.py, ObjectListViewDemo.py**

- Added free space filling columns

**2008-03-29 22:44 (#12) - test\_ObjectListView.py, ObjectListView.py, Demo.wxd, ObjectListViewDemo.py**

- Added minimum, maximum and fixed widths for columns
- unified 'stringFormat' and 'stringConverter'
- Added/update unit tests

**2008-03-28 23:54 (#11) - ObjectListView.py, Demo.wxd, ObjectListViewDemo.py**

- Added VirtualObjectListView and FastObjectListView
- Changed sort indicator icons
- Changed demo to use track information, and to show new classes

**2008-03-06 12:20 (#10) - ObjectListViewDemo.py**

- Call SetObjects() after assigning a rowFormatter

**2008-03-06 12:19 (#9) - ObjectListView.py**

- Improved docs
- Removed some duplicate code

**2008-03-02 11:02 (#8) - ObjectListView.py, ObjectListViewDemo.py**

- Added alternate row colors
- Added rowFormatter

**2008-03-02 09:33 (#6) - ObjectListViewDemo.py**

- Added Update Selected
- Added examples of lowercase and Unicode

### 2008-03-02 09:31 (#5) - test\_ObjectListView.py

- Test selections
- Use PySimpleApp

### 2008-03-02 09:30 (#4) - ObjectListView.py

- Added RefreshObject() and friends
- Do sorting within python when possible, rather than using SortItems(). 5-10x faster!
- Optimized RepopulateList()

### 2008-02-29 10:34 (#2) - images/BoxesThree32.bmp, images/BoxesThree16.bmp, images/Group32.bmp, test\_ObjectListView.py, O

- Unit tests in progress
- Demo complete