easymodel Documentation

Release 0.4.0

David Zuber

August 09, 2015

Contents

Qt Models and Views made easy with general purpose Model and a Widget delegate.

CHAPTER 1

Documentation

The full documentation is at http://pythonhosted.org/easymodel.

CHAPTER 2

Features

- Easy to use tree model for arbitrary data
- Widgets in views via delegates.

2.1 Welcome to easymodel‘s documentation!

CHAPTER 3

Contents:

3.1 Installation

At the command line either via easy_install or pip:

```
$ easy_install easymodel
$ pip install easymodel
```

Or, if you have virtualenvwrapper installed:

```
$ mkvirtualenv easymodel
$ pip install easymodel
```

3.2 Usermanual

3.2.1 Tutorial

Model

The `easymodel.treemodel` module provides an simple but powerful general purpose treemodel. The model uses `easymodel.treemodel.TreeItem` instances that act like rows.

The tree item itself uses a `easymodel.treemodel.ItemData` instance to provide the data. That way, the structure and logic of the whole model is encapsulated in the model and tree item classes. They take care of indexing, insertion, removal etc.

As a user you only need to subclass `easymodel.treemodel.ItemData` to wrap arbitrary objects. It is pretty easy.

There is also a rudimentary `easymodel.treemodel.ListItemData` that might be enough for simple data.

Root

Each model needs a root item. The root item is the parent of all top level tree items that hold data. It is also responsible for the headers. So in most cases it is enough to simply use a `easymodel.treemodel.ListItemData`:

```
import easymodel.treemodel as treemodel

headers = ['Name', 'Speed', 'Altitude']
```

```
rootdata = treemodel.ListItemData(headers)
# roots do not have a parent
rootitem = treemodel.TreeItem(rootdata, parent=None)
```

Creating a simple model

There are three steps involved. Create a root item, a model and wrap the data. The creation of a model is simple:

```
# use the root item from above
model = treemodel.TreeModel(rootitem)
```

Thats it. The root item might already have children. That way, you can initialize a model with data.

Add Items

Let's assume our data consists of items with 3 values: Name, Velocity, Altitude. The data might describe a vehicle, like an airplane or something like that. Before we create our own `easymodel.treemodel.ItemData` subclasses, we use simple lists, so we can use `easymodel.treemodel.ListItemData`. First create the data:

```
data1 = treemodel.ListItemData(['Cessna', 250, 2000])
data2 = treemodel.ListItemData(['747', 750, 6000])
data3 = treemodel.ListItemData(['Fuel Plane', '730', 5000])
```

Wrap the data in items:

```
# specify the parent to add it directly to the model
item1 = TreeItem(data1, parent=rootitem)
# or add it later
item2 = TreeItem(data2)
item2.set_parent(rootitem)
# use the builtin to_item method
item3 = data3.to_item(item2)
```

The tree items will automatically update the model. No need to emit any signals or call further methods.

Remove Items

Let's say the fuel plane finished its job and landed. You can remove it from the model simply by setting the parent to None:

```
item3.set_parent(None)
```

You could have also used the model's methods to remove it but this way is much easier.

Wrap arbitrary objects

To wrap arbitrary objects in an item data instance, you need to subclass it. Let's assume we have a very simple airplane class:

```
class Airplane(object):
    """This is the data we want to display in a view. An airplane.

    It has a name, a velocity and altitude.
    """
```

```
    def __init__(self, name, speed, altitude):
        self.name = name
        self.speed = speed
        self.altitude = altitude
```

Let's create a item data subclass that has three columns: Name, Speed, Altitude. Speed and Altitude should be editable.

First subclass `easymodel.treemodel.ItemData`. It can store an airplane instance.:

```
class AirplaneItemData(treemodel.ItemData):
    """An item data object that can extract information from an airplane instance.
    """
    def __init__(self, airplane):
        self.airplane = airplane
```

The column count is 3 and we can also give access to the airplane that is stored:

```
def column_count(self,):
    """Return 3. For name, velocity and altitude."""
    return 3

def internal_data(self):
    """Return the airplane instance"""
    return self.airplane
```

By default an item is enabled and selectable. But speed and altitude should be editable. So lets override `easymodel.treemodel.ItemData.flags()`:

```
def flags(self, column):
    """Return flags for enabled and selectable. Speed and altitude are also editable."""
    default = QtCore.Qt.ItemIsEnabled | QtCore.Qt.ItemIsSelectable
    if column == 0:
        return default
    else:
        return default | QtCore.Qt.ItemIsEditable
```

Now we need pass the data to the model. This is pretty simple. Just pass the right attribute for each column:

```
def data(self, column, role):
    """Return the data of the airplane"""
    if role == QtCore.Qt.DisplayRole:
        return (self.airplane.name, self.airplane.speed, self.airplane.altitude)[column]
```

Setting the data is not that complicated. Just set the right attribute for each column:

```
def set_data(self, column, value, role):
    """Set the data of the airplane"""
    if role == QtCore.Qt.EditRole or role == QtCore.Qt.DisplayRole:
        attr = ('speed', 'altitude')[column-1]
        setattr(self.airplane, attr, value)
        return True
    return False
```

Now we can use this class to wrap our own airplanes and add them to a treeitem/model:

```
# create a plane
plane = Airplane('Nimbus 4', 0, 0)
# wrap it in a data object
planedata = AirplaneItemData(plane)
# add it to the model
planeitem = treemodel.TreeItem(planedata, rootitem)
```

Delegate

Sometimes you want to have arbitrary widgets in your views. ItemDelegates of Qt are cool, but it is very hard to get your arbitrary widget into the view.

If the widget changes a lot or you want to use the UI Designer, the regular workflow of styled item delegates is a bit flawed. The `easymodel.widgetdelegate.Widgetdelegate` is there to help.

Let's assume you want have a spin box and a randomize button for the altitude of your planes in a view. The widget might look like this:

```
class RandomSpinBox(QtGui.QWidget):
    """SpinBox plus randomize button
    """

    def __init__(self, parent=None, flags=0):
        super(RandomSpinBox, self).__init__(parent, flags)
        self.main_hbox = QtGui.QHBoxLayout(self)
        self.value_sb = QtGui.QSpinBox(self)
        self.random_pb = QtGui.QPushButton("Randomize")
        self.main_hbox.addWidget(self.value_sb)
        self.main_hbox.addWidget(self.random_pb)

        self.random_pb.clicked.connect(self.randomize)

    def randomize(self, *args, **kwargs):
        v = random.randint(0, 99)
        self.value_sb.setValue(v)
```

To create a delegate for this widget subclass `easymodel.widgetdelegate.Widgetdelegate`:

```
import easymodel.widgetdelegate as widgetdelegate

class RandomSpinBoxDelegate(widgetdelegate.WidgetDelegate):
    """RandomSpinBox delegate"""

    def __init__(self, parent=None):
        super(RandomSpinBoxDelegate, self).__init__(parent)
```

Implement the abstract methods. First reimplement `easymodel.widgetdelegate.Widgetdelegate.create_widget()`. It is used to create the widget that will be rendered in the view:

```
def create_widget(self, parent=None):
    return RandomSpinBox(parent)
```

If your editor should look exactly the same you can reuse this function:

```
def create_editor_widget(self, parent, option, index):
    return self.create_widget(parent)
```

Now you need to implement `easymodel.widgetdelegate.Widgetdelegate.setEditorData()`. It will set the editor in the right state to represent a index in the model. So we take the data of the index and put it in the spinbox:

```
def setEditorData(self, widget, index):
    d = index.data(QtCore.Qt.DisplayRole)
    if d:
```

```
        widget.value_sb.setValue(int(d))
    else:
        widget.value_sb.setValue(int(0))
```

`easymodel.widgetdelegate.Widgetdelegate.set_widget_index()` does the same for the widget that is rendered. Every time an index is painted, the widget has to be set in the right state to represent the index. Because we already did that for the editor we can reuse the function:

```
def set_widget_index(self, index):
    self.setEditorData(self.widget, index)
```

Now all that is left is `easymodel.widgetdelegate.Widgetdelegate.setModelData()`. Here you take the value from the editor and set the data in the model:

```
def setModelData(self, editor, model, index):
    v = editor.value_sb.value()
    model.setData(index, v, QtCore.Qt.EditRole)
```

Done! Now you can use the delegate in any view. But I recommend using one of the views in `easymodel.widgetdelegate`.

You can either use the `WidgetDelegateViewMixin` for your own views or use one of the premade views: `WD_AbstractItemView`, `WD_ListView`, `WD_TableView` `WD_TreeView`.

They will make the user experience better. When the user clicks an widget delegate, it will be set into edit mode and the click will be propagated to the editor. That way it behaves almost like the widget delegate were a regular widget.

Little example app

Let's create a simple widget with a view and controls to add new items into the view. We reuse the code from above.

The window has a view, an add button and 3 edits for name, speed and altitude. When the add button is clicked, a new airplane should be inserted into the model. The parent should be the currently selected index.

First create the widget:

```
class AirplaneAppWidget(QtGui.QWidget):
    def __init__(self, parent=None, flags=0):
        super(AirplaneAppWidget, self).__init__(parent, flags)
        self.main_vbox = QtGui.QVBoxLayout(self)
        self.add_hbox = QtGui.QHBoxLayout()

        self.instruction_lb = QtGui.QLabel("Select Item and click add!", self)
        self.view = widgetdelegate.WD_TreeView(self)

        self.add_pb = QtGui.QPushButton('Add')
        self.add_pb.clicked.connect(self.add_airplane)

        self.name_lb = QtGui.QLabel('Name')
        self.name_le = QtGui.QLineEdit()
        self.speed_lb = QtGui.QLabel('Speed')
        self.speed_sb = QtGui.QSpinBox()
        self.altitude_lb = QtGui.QLabel('Altitude')
        self.altitude_sb = QtGui.QSpinBox()

        self.main_vbox.addWidget(self.instruction_lb)
        self.main_vbox.addWidget(self.view)
        self.main_vbox.addLayout(self.add_hbox)
        self.add_hbox.addWidget(self.add_pb)
```

```
        self.add_hbox.addWidget(self.name_lb)
        self.add_hbox.addWidget(self.name_le)
        self.add_hbox.addWidget(self.speed_lb)
        self.add_hbox.addWidget(self.speed_sb)
        self.add_hbox.addWidget(self.altitude_lb)
        self.add_hbox.addWidget(self.altitude_sb)

        self.delegate1 = RandomSpinBoxDelegate()
        self.view.setItemDelegateForColumn(2, self.delegate1)

        # Now we can build ourselves models
        # First we need a root
        rootdata = treemodel.ListItemData(['Name', 'Velocity', 'Altitude'])
        root = treemodel.TreeItem(rootdata)
        # Create a new model with the root
        model = treemodel.TreeModel(root)

        self.view.setModel(model)
```

Now for the button callback. All we need to do is create an airplane, wrap it in a data/item and parent it under the current index:

```
def add_airplane(self, *args, **kwargs):
    # get parent item
    currentindex = self.view.currentIndex()
    if currentindex.isValid():
        # items are stored in the internal pointer
        # but if you use a proxy model this might not work
        # user the TREEITEM_ROLE instead
        pitem = currentindex.data(treemodel.TREEITEM_ROLE)
    else:
        # nothing selected. Take root as parent
        pitem = self.view.model().root

    # create a new airplane
    name = self.name_le.text()
    speed = self.speed_sb.value()
    altitude = self.altitude_sb.value()
    airplane = Airplane(name, speed, altitude)
    # wrap it in an item data instance
    adata = AirplaneItemData(airplane)
    # create a tree item.
    # because parent is given, the item will
    # automatically be inserted in the model
    treemodel.TreeItem(adata, parent=pitem)
```

The rest of the app code can look like this:

```
app = QtGui.QApplication([], QtGui.QApplication.GuiClient)
app.setStyle(QtGui.QStyleFactory.create("plastique"))
apw = AirplaneAppWidget()
apw.show()
app.exec_()
```

Complete Code

Everything put together:

```
import random

from PySide import QtCore, QtGui

from easymodel import treemodel, widgetdelegate


class Airplane(object):
    """This is the data we want to display in a view. An airplane.

    It has a name, a velocity and altitude.
    """
    def __init__(self, name, speed, altitude):
        self.name = name
        self.speed = speed
        self.altitude = altitude


class AirplaneItemData(treemodel.ItemData):
    """An item data object that can extract information from an airplane instance.
    """
    def __init__(self, airplane):
        self.airplane = airplane

    def data(self, column, role):
        """Return the data of the airplane"""
        if role == QtCore.Qt.DisplayRole:
            return (self.airplane.name, self.airplane.speed, self.airplane.altitude)[column]

    def set_data(self, column, value, role):
        """Set the data of the airplane"""
        if role == QtCore.Qt.EditRole or role == QtCore.Qt.DisplayRole:
            attr = ('name', 'speed', 'altitude')[column]
            setattr(self.airplane, attr, value)
            return True
        return False

    def column_count(self,):
        """Return 3. For name, velocity and altitude."""
        return 3

    def internal_data(self):
        """Return the airplane instance"""
        return self.airplane

    def flags(self, column):
        """Return flags for enabled and selectable. Speed and altitude are also editable."""
        default = QtCore.Qt.ItemIsEnabled | QtCore.Qt.ItemIsSelectable
        if column == 0:
            return default
        else:
            return default | QtCore.Qt.ItemIsEditable


class RandomSpinBox(QtGui.QWidget):
    """SpinBox plus randomize button
    """
```

```
    def __init__(self, parent=None, flags=0):
        super(RandomSpinBox, self).__init__(parent, flags)
        self.main_hbox = QtGui.QHBoxLayout(self)
        self.value_sb = QtGui.QSpinBox(self)
        self.random_pb = QtGui.QPushButton("Randomize")
        self.main_hbox.addWidget(self.value_sb)
        self.main_hbox.addWidget(self.random_pb)

        self.random_pb.clicked.connect(self.randomize)

    def randomize(self, *args, **kwargs):
        v = random.randint(0, 99)
        self.value_sb.setValue(v)


class RandomSpinBoxDelegate(widgetdelegate.WidgetDelegate):
    """RandomSpinBox delegate
    """

    def __init__(self, parent=None):
        super(RandomSpinBoxDelegate, self).__init__(parent)

    def create_widget(self, parent=None):
        return RandomSpinBox(parent)

    def create_editor_widget(self, parent, option, index):
        return self.create_widget(parent)

    def setEditorData(self, widget, index):
        d = index.data(QtCore.Qt.DisplayRole)
        if d:
            widget.value_sb.setValue(int(d))
        else:
            widget.value_sb.setValue(int(0))

    def set_widget_index(self, index):
        self.setEditorData(self.widget, index)

    def setModelData(self, editor, model, index):
        v = editor.value_sb.value()
        model.setData(index, v, QtCore.Qt.EditRole)


class AirplaneAppWidget(QtGui.QWidget):
    def __init__(self, parent=None, flags=0):
        super(AirplaneAppWidget, self).__init__(parent, flags)
        self.main_vbox = QtGui.QVBoxLayout(self)
        self.add_hbox = QtGui.QHBoxLayout()

        self.instruction_lb = QtGui.QLabel("Select Item and click add!", self)
        self.view = widgetdelegate.WD_TreeView(self)

        self.add_pb = QtGui.QPushButton('Add')
        self.add_pb.clicked.connect(self.add_airplane)

        self.name_lb = QtGui.QLabel('Name')
        self.name_le = QtGui.QLineEdit()
        self.speed_lb = QtGui.QLabel('Speed')
```

```
        self.speed_sb = QtGui.QSpinBox()
        self.altitude_lb = QtGui.QLabel('Altitude')
        self.altitude_sb = QtGui.QSpinBox()

        self.main_vbox.addWidget(self.instruction_lb)
        self.main_vbox.addWidget(self.view)
        self.main_vbox.addLayout(self.add_hbox)
        self.add_hbox.addWidget(self.add_pb)
        self.add_hbox.addWidget(self.name_lb)
        self.add_hbox.addWidget(self.name_le)
        self.add_hbox.addWidget(self.speed_lb)
        self.add_hbox.addWidget(self.speed_sb)
        self.add_hbox.addWidget(self.altitude_lb)
        self.add_hbox.addWidget(self.altitude_sb)

        self.delegate1 = RandomSpinBoxDelegate()
        #elf.view.setItemDelegateForColumn(2, self.delegate1)

        # Now we can build ourselves models
        # First we need a root
        rootdata = treemodel.ListItemData(['Name', 'Velocity', 'Altitude'])
        root = treemodel.TreeItem(rootdata)

        # Create a new model with the root
        self.model = treemodel.TreeModel(root)
        self.view.setModel(self.model)

    def add_airplane(self, *args, **kwargs):
        # get parent item
        currentindex = self.view.currentIndex()
        if currentindex.isValid():
            # items are stored in the internal pointer
            # but if you use a proxy model this might not work
            # user the TREEITEM_ROLE instead
            pitem = currentindex.data(treemodel.TREEITEM_ROLE)
        else:
            # nothing selected. Take root as parent
            pitem = self.view.model().root

        # create a new airplane
        name = self.name_le.text()
        speed = self.speed_sb.value()
        altitude = self.altitude_sb.value()
        airplane = Airplane(name, speed, altitude)
        # wrap it in an item data instance
        adata = AirplaneItemData(airplane)
        # create a tree item.
        # because parent is given, the item will
        # automatically be inserted in the model
        treemodel.TreeItem(adata, parent=pitem)

if __name__ == "__main__":
    # Create a view to show what is happening
    app = QtGui.QApplication([], QtGui.QApplication.GuiClient)
    app.setStyle(QtGui.QStyleFactory.create("plastique"))
    apw = AirplaneAppWidget()
    apw.show()
    app.exec_()
```

3.3 Developer's Documentation

Welcome to the developer's documenation. All necessary information for contributors who want to extend the project.

3.4 Reference

Automatic generated Documenation by apidoc and autodoc.

3.4.1 easymodel

Submodules

easymodel.treemodel

easymodel.widgetdelegate

Module contents

3.5 Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

You can contribute in many ways:

3.5.1 Types of Contributions

Report Bugs

Report bugs at https://github.com/storax/easymodel/issues.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with "bug" is open to whoever wants to implement it.

Implement Features

Look through the GitHub issues for features. Anything tagged with "feature" is open to whoever wants to implement it.

Write Documentation

easymodel could always use more documentation, whether as part of the official easymodel docs, in docstrings, or even on the web in blog posts, articles, and such.

Submit Feedback

The best way to send feedback is to file an issue at https://github.com/storax/easymodel/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

3.5.2 Get Started!

Ready to contribute? Here's how to set up *easymodel* for local development.

1. Fork the *easymodel* repo on GitHub.
2. Clone your fork locally:

```
$ git clone git@github.com:your_name_here/easymodel.git
```

3. Create a branch for local development:

```
$ git checkout -b name-of-your-bugfix-or-feature
```

Now you can make your changes locally.

4. When you're done making changes, check that your changes pass style and unit tests, including testing other Python versions with tox:

```
$ tox
```

To get tox, just pip install it.

5. Commit your changes and push your branch to GitHub:

```
$ git add .
$ git commit -m "Your detailed description of your changes."
$ git push origin name-of-your-bugfix-or-feature
```

6. Submit a pull request through the GitHub website.

3.5.3 Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in README.rst.
3. The pull request should work for Python 2.6, 2.7, and 3.3, and for PyPy. Check https://travis-ci.org/storax/easymodel under pull requests for active pull requests or run the `tox` command and make sure that the tests pass for all supported Python versions.

3.5.4 Tips

To run a subset of tests:

```
$ py.test test/test_easymodel.py
```

3.6 Credits

3.6.1 Development Lead

- David Zuber <zuber.david@gmx.de>

3.6.2 Contributors

None yet. Why not be the first?

3.7 History

3.7.1 0.1.0 (2014-08-27)

- First release on PyPI.

3.7.2 0.2.0 (2015-01-04)

- Specialized views that handle click events and propagate them to the editor widget.
- Easier insertion and removal of rows
- Editing supported

3.7.3 0.3.0 (2015-02-10)

- Fix emit signal when calling set_data
- Fix editor resizing
- Add ItemDataRoles to retrieve the internal objects of an index
- Easy conversion from ItemData to TreeItem
- Emit clicks on widgetdelegate via QApplication and to the actual child widget

3.7.4 0.4.0 (2015-08-09)

- python 3 support

CHAPTER 4

Feedback

If you have any suggestions or questions about **easymodel** feel free to email me at zuber.david@gmx.de.

If you encounter any errors or problems with **easymodel**, please let me know! Open an Issue at the GitHub https://github.com/storax/easymodel main repository.

4.1 Indices and tables

- genindex
- modindex
- search

e

E