NodeEditor Documentation

Release 0.9.13

Pavel Křupala

Jul 20, 2022

Contents:

CHAPTER 1

Welcome to PyQtNodeEditor

This package was created from the Node Editor written in PyQt5. The intention was to create a tutorial series describing the path to create a reusable nodeeditor which can be used in different projects. The tutorials are published on youtube for free. The full list of tutorials can be located here: https://www.blenderfreak.com/tutorials/node-editor-tutorial-series/

1.1 Features

- provides full framework for creating customizable graph, nodes, sockets and edges
- full support for undo / redo and serialization into files in a VCS friendly way
- support for implementing evaluation logic
- hovering effects, dragging edges, cutting lines and a bunch more...
- provided 2 examples on how node editor can be implemented

1.2 Requirements

- Python 3.x
- PyQt5 or PySide2 (using wrapper QtPy)

1.3 Installation

```
$ pip install nodeeditor
```

Or directly from source code to get the latest version

```
$ pip install git+https://gitlab.com/pavel.krupala/pyqt-node-editor.git
```

Or download the source code from gitlab:

```
git clone https://gitlab.com/pavel.krupala/pyqt-node-editor.git
```

1.4 Screenshots

1.5 Other links

- Documentation
- Contribute
- Issues
- Merge requests
- Changelog

CHAPTER 2

Event system

Nodeeditor uses its own events (and tries to avoid using `pyqtSignal`) to handle logic happening inside the Scene. If a class does handle some events, they are usually described at the top of the page in this documentation.

Any of the events is subscribable to and the methods for registering callback are called:

```
add<EventName>Listener(callback)
```

You can register to any of these events any time.

2.1 Events used in NodeEditor:

2.1.1 `Scene`

Has Been Modified when something has changed in the *Scene*

Item Selected when *Node* or *Edge* is selected

Items Deselected when deselect everything appears

Drag Enter when something is Dragged onto the *Scene*. Here we do allow or deny the drag

Drop when we Drop something into the *Scene*

2.1.2 `SceneHistory`

History Modified after *History Stamp* has been stored or restored

History Stored after *History Stamp* has been stored

History Restored after *History Stamp* has been restored

CHAPTER 3

Serialization

All of serializable classes derive from *Serializable* class. *Serializable* does create commonly used parameters for our classes. In our case it is just `id` attribute.

Serializable defines two methods which should be overriden in child classes:

- *serialize()*
- *deserialize()*

According to *Coding Standards* we keep these two functions on the bottom of the class source code.

To contain all of the data we use `OrderedDict` instead of regular *dict*. Mainly because we want to retain the order of parameters serialized in files.

Classes which derive from `Serializable`:

- `Scene`
- `Node`
- `QDMNodeContentWidget`
- `Edge`
- `Socket`

CHAPTER 4

Evaluation

TL;DR: The evaluation system uses `eval()` and `evalChildren()`. `eval()` method is supposed to be overriden by your own implementation. The evaluation logic uses Flags for marking the *Nodes* to be *Dirty* and/or *Invalid*.

4.1 Evaluation Functions

There are 2 main methods used for evaluation:

- `eval()`
- `evalChildren()`

These functions are mutually exclusive. That means that `evalChildren` does **not** eval current *Node*, but only children of the current *Node*.

By default the implementation of `eval()` is "empty" and return 0. However it seems logical, that eval (if successfull) resets the *Node* not to be *Dirty* nor *Invalid*. This method is supposed to be overriden by your own implementation. As an example, you can check out the repository's `examples/example_calculator` to have an inspiration how to setup the *Node* evaluation on your own.

The evaluation takes advantage of *Node* flags described below.

4.2 `Node` Flags

Each `Node` has 2 flags:

- `Dirty`
- `Invalid`

The *Invalid* flag has always higher priority. That means when the *Node* is *Invalid* it doesn't matter if it is *Dirty* or not.

To mark a node *Dirty* or *Invalid* there are respective methods `markDirty()` and `markInvalid()`. Both methods take *bool* parameter for the new state. You can mark *Node* dirty by setting the parameter to `True`. Also you can un-mark the state by passing `False` value.

For both flags there are 3 methods available:

- `markInvalid()` - to mark only the *Node*
- `markChildrenInvalid()` - to mark only the direct (first level) children of the *Node*
- `markDescendantsInvalid()` - to mark it self and all descendant children of the *Node*

The same goes for the *Dirty* flag of course:

- `markDirty()` - to mark only the *Node*
- `markChildrenDirty()` - to mark only the direct (first level) children of the *Node*
- `markDescendantsDirty()` - to mark it self and all descendant children of the *Node*

Descendants or Children are always connected to Output(s) of current *Node*.

When a node is marked *Dirty* or *Invalid* event methods `onMarkedInvalid()` `onMarkedDirty()` are being called. By default, these methods do nothing. But still they are implemented in case you would like to override them and use in you own evaluation system.

CHAPTER 5

Coding Standards

The following rules and guidelines are used throughout the nodeeditor package:

5.1 File naming guidelines

- files in the nodeeditor package start with `node_`
- files containing graphical representation (PyQt5 overridden classes) start with `node_graphics_`
- files for window/widget start with `node_editor_`

5.2 Coding guidelines

- methods use Camel case naming
- variables/properties use Snake case naming
- The constructor `__init__` always contains all class variables for the entire class. This is helpful for new users, so they can just look at the constructor and read about all properties that class is using in one place. Nobody wants any surprises hidden in the code later
- nodeeditor uses custom callbacks and listeners. Methods for adding callback functions are usually named `addXYListener`
- custom events are usually named `onXY`
- methods named `doXY` usually *do* certain tasks and also take care of low level operations
- classes always contain methods in this order:
  - `__init__`
  - python magic methods (i.e. `__str__`), setters and getters
  - `initXY` functions

- listener functions
- nodeeditor event fuctions
- nodeeditor `doXY` and `getXY` helping functions
- Qt5 event functions
- other functions
- optionally overridden Qt `paint` method
- `serialize` and `deserialize` methods at the end

CHAPTER 6

Release Notes

Note: Contributors please include release notes as needed or appropriate with your bug fixes, feature additions and tests.

6.1 1.0.0 (unreleased)

- Added first version of library

CHAPTER 7

nodeeditor Package

7.1 node_content_widget Module

7.2 node_edge Module

7.2.1 Edge Validators

Edge Validator can be registered to Edge class using its method `registerEdgeValidator()`.

Each validator callback takes 2 params: *start_socket* and *end_socket*. Validator also needs to return *True* or *False*. For example of validators have a look in `node_edge_validators` module.

Here is an example how you can register the Edge Validator callbacks:

```
from nodeeditor.node_edge_validators import *

Edge.registerEdgeValidator(edge_validator_debug)
Edge.registerEdgeValidator(edge_cannot_connect_two_outputs_or_two_inputs)
Edge.registerEdgeValidator(edge_cannot_connect_input_and_output_of_same_node)
```

7.2.2 Edge Class

7.3 node_edge_dragging Module

7.4 node_edge_intersect Module

7.5 node_edge_rerouting Module

A module containing the Edge Rerouting functionality

class nodeeditor.node_edge_rerouting.**EdgeRerouting**(*grView: QGraphicsView*)
Bases: object

print(**args*)
Helper function to better control debug printing to console for this feature

getEdgeClass()
Helper function to get the Edge class. Using what the Scene class provides

getAffectedEdges() → list
Get a list of all edges connected to the *self.start_socket* where we started the re-routing

Returns List of all edges affected by the rerouting started from this *self.start_socket* Socket

Return type list

setAffectedEdgesVisible(*visibility: bool = True*)
Show/Hide all edges connected to the *self.start_socket* where we started the re-routing

Parameters **visibility** (bool) – True if all the affected Edge (s) should be shown or hidden

resetRerouting()
Reset to default state. Init this feature internal variables

clearReroutingEdges()
Remove the helping dashed edges from the Scene

updateScenePos(*x: float*, *y: float*)
Update position of all the rerouting edges (dashed ones). Called from mouseMove event to update to new mouse position

Parameters

- **x** (float) – new X position
- **y** (float) – new Y position

startRerouting(*socket: Socket*)
Method to start the re-routing. Called from the grView's state machine.

Parameters **socket** (Socket) – Socket where we started the re-routing

stopRerouting(*target: Socket = None*)
Method for stopping the re-routing

Parameters **target** (Socket or None) – Target where we ended the rerouting (usually released mouse button). Provide Socket or None to cancel

7.6 node_edge_snapping Module

7.7 node_edge_validators Module

A module containing the Edge Validator functions which can be registered as callbacks to Edge class.

7.7.1 Example of registering Edge Validator callbacks:

You can register validation callbacks once for example on the bottom of node_edge.py file or on the application start with calling this:

```
from nodeeditor.node_edge_validators import *

Edge.registerEdgeValidator(edge_validator_debug)
Edge.registerEdgeValidator(edge_cannot_connect_two_outputs_or_two_inputs)
Edge.registerEdgeValidator(edge_cannot_connect_input_and_output_of_same_node)
Edge.registerEdgeValidator(edge_cannot_connect_input_and_output_of_different_type)
```

nodeeditor.node_edge_validators.**print_error**(**args*)

Helper method which prints to console if *DEBUG* is set to *True*

nodeeditor.node_edge_validators.**edge_validator_debug**(*input: Socket*, *output: Socket*) → bool

This will consider edge always valid, however writes bunch of debug stuff into console

nodeeditor.node_edge_validators.**edge_cannot_connect_two_outputs_or_two_inputs**(*input: Socket*, *output: Socket*) → bool

Edge is invalid if it connects 2 output sockets or 2 input sockets

nodeeditor.node_edge_validators.**edge_cannot_connect_input_and_output_of_same_node**(*input: Socket*, *output: Socket*) → bool

Edge is invalid if it connects the same node

nodeeditor.node_edge_validators.**edge_cannot_connect_input_and_output_of_different_type**(*input Socke output: Socke* → bool

Edge is invalid if it connects sockets with different colors

7.8 node_editor_widget Module

7.9 node_editor_window Module

7.10 node_graphics_cutline Module

7.11 node_graphics_edge Module

7.11.1 *QDMGraphicsEdge* class

7.12 node_graphics_edge_path Module

7.12.1 *GraphicsEdgePathBase* base class

7.12.2 *GraphicsEdgePathDirect* class

7.12.3 *GraphicsEdgePathBezier* class

7.13 node_graphics_node Module

7.14 node_graphics_scene Module

7.15 node_graphics_socket Module

7.16 node_graphics_view Module

7.16.1 *QDMGraphicsView* class

7.17 node_node Module

7.18 node_scene Module

7.18.1 Events

Has Been Modified when something has changed in the *Scene*

Item Selected when *Node* or *Edge* is selected

Items Deselected when deselect everything appears

Drag Enter when something is Dragged onto the *Scene*. Here we do allow or deny the drag

Drop when we Drop something into the *Scene*

7.18.2 Exceptions

7.18.3 Scene Class

7.19 node_scene_clipboard Module

7.20 node_scene_history Module

7.20.1 Events

History Modified after *History Stamp* has been stored or restored

History Stored after *History Stamp* has been stored

History Restored after *History Stamp* has been restored

7.20.2 SceneHistory Class

7.21 node_serializable Module

A module containing Serializable "Interface". We pretend its an abstract class

class nodeeditor.node_serializable.**Serializable**
Bases: object

Default constructor automatically creates data which are common to any serializable object. In our case we create self.id which we use in every object in NodeEditor.

id
We set this property in the *constructor* because all of NodeEditor's serializable objects use this attribute to unique object identification. It is handy for referencing objects.

serialize() → collections.OrderedDict
Serialization method to serialize this class data into OrderedDict which can be easily stored in memory or file.

Returns data serialized in OrderedDict

Return type OrderedDict

deserialize(*data: dict*, *hashmap: dict = {}*, *restore_id: bool = True*) → bool
Deserialization method which take data in python dict format with helping *hashmap* containing references to existing entities.

Parameters

- **data** (dict) – Dictionary containing serialized data
- **hashmap** (dict) – Helper dictionary containing references (by id == key) to existing objects
- **restore_id** (*bool*) – True if we are creating new Sockets. False is useful when loading existing Sockets of which we want to keep the existing object's *id*.

Returns True if deserialization was successful, otherwise False

Return type bool

7.22 node_socket Module

7.22.1 Socket Class

7.23 utils Module

CHAPTER 8

Indices and tables

- genindex
- modindex
- search

Python Module Index

n

Index