# Contents

1 Natron Python API reference 1
1.1 NatronEngine .................................................. 1
1.2 NatronGui ....................................................... 93

2 Introduction 115
2.1 Natron plug-in paths ........................................... 115
2.2 Python Auto-declared variables .............................. 115
2.3 Start-up scripts ................................................ 119
2.4 Examples ......................................................... 119
2.5 Natron in command-line ....................................... 120
2.6 Example ........................................................ 123
2.7 Objects hierarchy Overview .................................. 125

3 Getting started 127
3.1 Creating and controlling nodes .............................. 127
3.2 Controlling parameters ....................................... 130
3.3 Parameters expressions ....................................... 132
3.4 Working with groups .......................................... 139
3.5 Using Callbacks ................................................ 153
3.6 Rendering ......................................................... 164
3.7 Using the roto functionalities ............................... 165
3.8 Modal dialogs ................................................... 167
3.9 User menu commands ......................................... 170
3.10 PySide panels .................................................. 171
3.11 Controlling the viewer ....................................... 177

Python Module Index 179
CHAPTER 1

Natron Python API reference

All Python modules of the Natron API are referenced here.

1.1 NatronEngine

1.1.1 Detailed Description

Here are listed all classes being part of NatronEngine module. This module is always loaded by Natron natively, meaning access is granted to these classes in your scripts without importing anything.

AnimatedParam

Inherits Param


Synopsis

This is the base class for all parameters which have the property canAnimate set to True. See the detailed description below

Functions

- def deleteValueAtTime (time[, dimension=0])
- def getCurrentTime ()
- def getDerivativeAtTime (time[, dimension=0])
- def getExpression (dimension)
- def getIntegrateFromTimeToTime (time1, time2[, dimension=0])
- def getIsAnimated ([dimension=0])
- def getKeyIndex (time[, dimension=0])
- def getKeyTime (index, dimension)
- def getNumKeys ([dimension=0])
Natron Documentation, Release 2.0.0

- def removeAnimation ([dimension=0])
- def setExpression (expr, hasRetVariable[, dimension=0])
- def setInterpolationAtTime (time, interpolation[, dimension=0])

Detailed Description

Animating parameters have values that may change throughout the time. To enable animation the parameter should have at least 1 keyframe. Keyframes can be added in the derived class (since function signature is type specific) with the setValueAtTime function. Once 2 keyframes are active on the parameter, the value of the parameter will be interpolated automatically by Natron for a given time. You can control keyframes by adding, removing, changing their values and their interpolation type.

Note that by default new keyframes are always with a Smooth interpolation.

Moreover parameters can have Python expressions set on them to control their value. In that case, the expression takes precedence over any animation that the parameter may have, meaning that the value of the parameter would be computed using the expression provided.

Member functions description

NatronEngine.AnimatedParam.deleteValueAtTime (time[, dimension=0])

Parameters
- time – float
- dimension – int

Removes a keyframe at the given time and dimension for this parameter, if such keyframe exists.

NatronEngine.AnimatedParam.getCurrentTime ()

Return type int

Convenience function: returns the current time on the timeline

NatronEngine.AnimatedParam.getDerivativeAtTime (time[, dimension=0])

Parameters
- time – float
- dimension – int

Return type double

Returns the derivative of the parameter at the given time and for the given dimension. The derivative is computed on the animation curve of the parameter. This function is irrelevant for parameters that have an expression.

NatronEngine.AnimatedParam.getExpression (dimension)

Parameters dimension – int

Return type str

Returns the Python expression set on the parameter at the given dimension. When no expression is set, this function returns an empty string.

NatronEngine.AnimatedParam.getIntegrateFromTimeToTime (time1, time2[, dimension=0])

Parameters
- time1 – float
• \( \text{time2} - \text{float} \)
• \( \text{dimension} - \text{int} \)

Return type \( \text{float} \)

Integrates the value of the parameter over the range \([\text{time1} - \text{time2}]\). This is done using the animation curve of the parameter of the given \( \text{dimension} \). Note that if this parameter has an expression, the return value is irrelevant.

NatronEngine.AnimatedParam.getIsAnimated([\text{dimension}=0])

Parameters \( \text{dimension} - \text{int} \)

Return type \( \text{bool} \)

Returns whether the given \( \text{dimension} \) has an animation or not. This returns true if the underlying animation curve has 1 or more keyframes.

NatronEngine.AnimatedParam.getKeyIndex(time[, \text{dimension}=0])

Parameters

• \( \text{time} - \text{float} \)
• \( \text{dimension} - \text{int} \)

Return type \( \text{int} \)

Returns the index of the keyframe at the given \( \text{time} \) for the animation curve at the given \( \text{dimension} \), or -1 if no such keyframe could be found.

NatronEngine.AnimatedParam.getKeyTime(index, \text{dimension})

Parameters

• \( \text{index} - \text{int} \)
• \( \text{dimension} - \text{int} \)

Return type \( \text{tuple} \)

Returns a tuple [bool, float] where the first member is True if a keyframe exists at the given \( \text{index} \) for the animation curve at the given \( \text{dimension} \). The second float member is the keyframe exact time.

NatronEngine.AnimatedParam.getNumKeys([\text{dimension}=0])

Parameters \( \text{dimension} - \text{int} \)

Return type \( \text{int} \)

Returns the number of keyframes for the animation curve at the given \( \text{dimension} \).

NatronEngine.AnimatedParam.removeAnimation([\text{dimension}=0])

Parameters \( \text{dimension} - \text{int} \)

Removes all animation for the animation curve at the given \( \text{dimension} \). Note that this will not remove any expression set.

NatronEngine.AnimatedParam.setExpression(expr, hasRetVariable[, \text{dimension}=0])

Parameters

• \( \text{expr} - \text{str} \)
• \( \text{hasRetVariable} - \text{bool} \)
• \( \text{dimension} - \text{int} \)

Return type \( \text{bool} \)
Set the Python expression `expr` on the parameter at the given `dimension` If `hasRetVariable` is True, then `expr` is assumed to have a variable `ret` declared. Otherwise, Natron will declare the `ret` variable itself.

```python
NatronEngine.AnimatedParam.setInterpolationAtTime(time, interpolation[, dimension=0])
```

- **param** `time` float
- **param** `interpolation` KeyFrameTypeEnum
- **param** `dimension` int

**Return type** bool

Set the interpolation of the animation curve of the given dimension at the given keyframe. If no such keyframe could be found, this method returns False. Upon success, this method returns True.

Example:

```python
app1.Blur2.size.setInterpolationAtTime(56, NatronEngine.Natron.KeyFrameTypeEnum.eKeyframeTypeConstant, 0)
```

## App

**Inherits** Group

**Inherited by:** GuiApp

## Synopsis

The App object represents one instance of a project. See detailed description...

## Functions

- **def** `addFormat` (formatSpec)
- **def** `createNode` (pluginID[, majorVersion=-1[, group=None]])
- **def** `getAppID` ()
- **def** `getProjectParam` (name)
- **def** `getViewNames` ()
- **def** `render` (effect, firstFrame, lastFrame[, frameStep])
- **def** `render` (tasks)
- **def** `saveTempProject` (filename)
- **def** `saveProject` (filename)
- **def** `saveProjectAs` (filename)
- **def** `loadProject` (filename)
- **def** `resetProject` ()
- **def** `closeProject` ()
- **def** `newProject` ()
- **def** `timelineGetLeftBound` ()

---

Chapter 1. Natron Python API reference
• def timelineGetRightBound()
• def timelineGetTime()
• def writeToScriptEditor(message)

Detailed Description

An App object is created automatically every times a new project is opened. For each instance of Natron opened, there’s a new instance of App. You never create an App object by yourself, instead you can access them with variables that Natron pre-declared for you: The first instance will be named app1, the second app2, etc... See this section for an explanation of auto-declared variables.

When in background mode, (interpreter or render mode) there will always ever be a single App instance, so Natron will make the following assignment before running any other script:

```python
app = app1
```

So you don’t have to bother on which instance you’re in. For Group Python plug-ins exported from Natron, they have a function with the following signature:

```python
def createInstance(app, group):
```

So you don’t have to bother again on which App instance your script is run. You should only ever need to refer to the `app1, app2...` variables when using the Script Editor.

Finally, you can always access the App object of any instance by calling the following function when your script is for command line (background mode):

```python
natron.getInstance(index)
```

Or the following function when you want to use GUI functionalities:

```python
natron.getGuiInstance(index)
```

**Warning:** Note that in both cases, `index` is a 0-based number. So to retrieve `app1` you would need to call the function with `index = 0`.

Creating nodes  The App object is responsible for creating new nodes. To create a node, you need to specify which plug-in you want to instantiate and optionally specify which major version should your node instantiate if the plug-in has multiple versions. For instance we could create a new Reader node this way:

```python
reader = app.createNode("fr.inria.openfx.ReadOIIO")
```

You can also specify the group into which the node should be created, None being the project’s top level:

```python
group = app.createNode("fr.inria.built-in.Group")
reader = app.createNode("fr.inria.openfx.ReadOIIO", -1, group)
```

You find it hard to know what is the plug-in ID of a plug-in? In Natron you can call the following function to get a sequence with all plug-in IDs currently available:

```python
natron.getPluginIDs()
```

You can also get a sub-set of those plug-ins with the `getPluginIDs(filter)` which returns only plug-in IDs containing the given filter (compared without case sensitivity).
Natron Documentation, Release 2.0.0

Accessing the settings of Natron  To modify the parameters in the Preferences of Natron, you can call the getSettings() function to get an object containing all the parameters of the preferences.

Accessing the project settings  You can get a specific parameter of the project settings with the getProjectParam(name) function.

Member functions description

NatronEngine.App.addFormat(formatSpec)

Parameters formatSpec – str

Attempts to add a new format to the project’s formats list. The formatSpec parameter must follow this spec: First the name of the format, without any spaces and without any non Python compliant characters; followed by a space and then the size of the format, in the form width*x*height; followed by a space and then the pixel aspect ratio of the format. For instance:

| HD 1920x1080 1 |

Wrongly formatted format will be omitted and a warning will be printed in the ScriptEditor.

NatronEngine.App.createNode(pluginID[, majorVersion=-1[, group=None ]])

Parameters

- pluginID – str
- majorVersion – int
- group – Group

Return type Effect

Creates a new node instantiating the plug-in specified with the given pluginID at the given majorVersion. If majorVersion is -1, the highest version of the plug-in will be instantiated. The optional group parameter can be used to specify into which group the node should be created, None meaning the project’s top level.

In Natron you can call the following function to get a sequence with all plug-in IDs currently available:

natron.getPluginIDs()

NatronEngine.App.getAppID()

Return type int

Returns the zero-based ID of the App instance. app1 would have the AppID 0, app2 would have the AppID 1, and so on...

NatronEngine.App.getProjectParam(name)

Parameters name – str

Return type Param

Returns a project Param given its name (script-name). See this section for an explanation of script-name vs. label.

NatronEngine.App.getViewNames()

Return type Sequence

Returns a sequence with the name of all the views in the project as setup by the user in the “Views” tab of the Project Settings.

NatronEngine.App.render(effect, firstFrame[, lastFrame[, frameStep ]])

Parameters effect – Effect
**param firstFrame** int

**param lastFrame** int

**param frameStep** int

Starts rendering the given `effect` on the frame-range defined by `[firstFrame, lastFrame]`. The `frameStep` parameter indicates how many frames the timeline should step after rendering each frame. The value must be greater or equal to 1. The `frameStep` parameter is optional and if not given will default to the value of the Frame Increment parameter in the Write node.

For instance:

```py
render(effect, 1, 10, 2)
```

Would render the frames 1, 3, 5, 7, 9

This is a blocking function only in background mode. A blocking render means that this function returns only when the render finishes (from failure or success).

This function should only be used to render with a Write node or DiskCache node.

**NatronEngine.App.render(tasks)**

Parameters:
- **tasks** – sequence

This function takes a sequence of tuples of the form `(effect, firstFrame, lastFrame[, frameStep])`. The `frameStep` is optional in the tuple and if not set will default to the value of the Frame Increment parameter in the Write node.

This is an overloaded function. Same as `render(effect, firstFrame, lastFrame, frameStep)` but all `tasks` will be rendered concurrently.

This function is called when rendering a script in background mode with multiple writers.

This is a blocking call only in background mode.

**NatronEngine.App.timelineGetLeftBound()**

Return type: `int`

Returns the left bound of the timeline, that is, the first member of the project’s frame-range parameter.

**NatronEngine.App.timelineGetRightBound()**

Return type: `int`

Returns the right bound of the timeline, that is, the second member of the project’s frame-range parameter.

**NatronEngine.App.timelineGetTime()**

Return type: `int`

Get the timeline’s current time. In Natron there’s only a single internal timeline and all Viewers are synchronised on that timeline. If the user seeks a specific frames, then all Viewers will render that frame.

**NatronEngine.App.writeToScriptEditor(message)**

Parameters:
- **message** – str

Writes the given `message` to the Script Editor panel of Natron. This can be useful to inform the user of various informations, warnings or errors.

**NatronEngine.App.saveProject(filename)**

Parameters:
- **filename** – str

Return type: `bool`<PySide.QtCore.bool>
Saves the current project under the current project name. Otherwise if the project has never been saved so far, this function asks the user where to save the project in GUI mode and in background mode saves the project to the file indicated by the *filename* parameter. In GUI mode, *filename* is disregarded.

This function returns *True* if it saved successfully, *False* otherwise.

NatronEngine.App.saveProjectAs (*filename*)

**Parameters**

`filename` – str

**Return type**

bool<PySide.QtCore.bool

In GUI mode, prompts the user to save the project at some location. In background mode, the project is saved to *filename*.

This function returns *True* if it saved successfully, *False* otherwise.

NatronEngine.App.saveTempProject (*filename*)

**Parameters**

`filename` – str

**Return type**

:class:`bool<PySide.QtCore.bool`

Saves a copy of the project to the given *filename* without updating project properties such as the project path, last save time etc... This function returns *True* if it saved successfully, *False* otherwise.

NatronEngine.App.loadProject (*filename*)

**Parameters**

`filename` – str

**Return type**

App

Loads the project indicated by *filename*. In GUI mode, this will open a new window only if the current window has modifications. In background mode this will close the current project of this App and open the project indicated by *filename* in it. This function returns the App object upon success, *None* otherwise.

NatronEngine.App.resetProject()

**Return type**

bool<PySide.QtCore.bool

Attempts to close the current project, without wiping the window. In GUI mode, the user is first prompted to saved his/her changes and can abort the reset, in which case this function will return *False*. In background mode this function always succeeds, hence always returns *True*. this always succeed.

NatronEngine.App.closeProject()

**Return type**

bool<PySide.QtCore.bool

Same as *resetProject()* except that the window will close in GUI mode. Also, if this is the last App alive, Natron will close.

NatronEngine.App.newProject()

**Return type**

App

Creates a new App. In GUI mode, this will open a new window. Upon success, the App object is returned, otherwise *None* is returned.

**AppSettings**

**Synopsis**

This class gathers all settings of Natron. You can access them exactly like you would for the Effect class.
Functions

• def `getParam` (scriptId)
• def `getParams` ()
• def `restoreDefaultSettings` ()
• def `saveSettings` ()

Member functions description
NatronEngine.AppSettings. `getParam` (scriptId)

Parameters `scriptId` – str

Return type Param

Returns a Param by its `scriptId`. See this section for a detailed explanation of what is the `scriptId`.

NatronEngine.AppSettings. `getParams` ()

Return type sequence

Returns a sequence with all Param composing the settings.

NatronEngine.AppSettings. `restoreDefaultSettings` ()

Restores all settings to their default value shipped with Natron.

NatronEngine.AppSettings. `saveSettings` ()

Saves all the settings on disk so that they will be restored with their current value on the following runs of Natron.

**BezierCurve**

Inherits ItemBase

Synopsis

A BezierCurve is the class used for beziers, ellipses and rectangles. See detailed description....

Functions

• def `addControlPoint` (x, y)
• def `addControlPointOnSegment` (index, t)
• def `getActivatedParam` ()
• def `getColor` (time)
• def `getColorParam` ()
• def `getCompositingOperator` ()
• def `getCompositingOperatorParam` ()
• def `getFeatherDistance` (time)
• def `getFeatherDistanceParam` ()
• def `getFeatherFallOff` (time)
• def `getFeatherFallOffParam` ()
• def getIsActivated(time)
• def getNumControlPoints()
• def getOpacity(time)
• def getOpacityParam()
• def getOverlayColor()
• def getPointMasterTrack(index)
• def isCurveFinished()
• def moveFeatherByIndex(index, time, dx, dy)
• def moveLeftBezierPoint(index, time, dx, dy)
• def movePointByIndex(index, time, dx, dy)
• def moveRightBezierPoint(index, time, dx, dy)
• def removeControlPointByIndex(index)
• def setActivated(time, activated)
• def setColor(time, r, g, b)
• def setCompositingOperator(op)
• def setCurveFinished(finished)
• def setFeatherDistance(dist, time)
• def setFeatherFallOff(falloff, time)
• def setFeatherPointAtIndex(index, time, x, y, lx, ly, rx, ry)
• def setOpacity(opacity, time)
• def setOverlayColor(r, g, b)
• def setPointAtIndex(index, time, x, y, lx, ly, rx, ry)
• def slavePointToTrack(index, trackTime, trackCenter)

Detailed Description

Almost all functionalities available to the user have been made available to the Python API, although in practise making a shape just by calling functions might be tedious due to the potential huge number of control points and keyframes. You can use the Natron Group node’s export functionality to generate automatically a script from a Roto node within that group.

A bezier initially is in an opened state, meaning it doesn’t produce a shape yet. At this stage you can then add control points using the `addControlPoint(x,y)` function. Once you’re one adding control points, call the function `setCurveFinished(finished)` to close the shape by connecting the last control point with the first.

Once finished, you can refine the bezier curve by adding control points with the `addControlPointOnSegment(index,t)` function. You can then move and remove control points of the bezier.

You can also slave a control point to a track using the `slavePointToTrack(index,trackTime,trackCenter)` function.

A bezier curve has several properties that the API allows you to modify:
• opacity
• color
• feather distance
• feather fall-off
• enable state
• overlay color
• compositing operator

Member functions description
NatronEngine.BezierCurve.CairoOperatorEnum
This enumeration represents the different blending modes of a shape. See the user interface for the different modes, or type help(NatronEngine.BezierCurve.CairoOperatorEnum) to see the different values.

NatronEngine.BezierCurve.addControlPoint(x, y)

Parameters
• x – float
• y – float

Adds a new control point to an opened shape (see isCurveFinished()) at coordinates (x,y). By default the feather point attached to this point will be equivalent to the control point. If the auto-keying is enabled in the user interface, then this function will set a keyframe at the timeline’s current time for this shape.

NatronEngine.BezierCurve.addControlPointOnSegment(index, t)

Parameters
• index – PySide.QtCore.int
• t – PySide.QtCore.double

Adds a new control point to a closed shape (see isCurveFinished()). The index is the index of the bezier segment linking the control points at index and index + 1. t is a value between [0,1] indicating the distance from the control point index the new control point should be. The closer to 1 t is, the closer the new control point will be to the control point at index +1. By default the feather point attached to this point will be equivalent to the control point.

If the auto-keying is enabled in the user interface, then this function will set a keyframe at the timeline’s current time for this shape.

NatronEngine.BezierCurve.getActivatedParam()

Return type BooleanParam
Returns the Param controlling the enabled state of the bezier.

NatronEngine.BezierCurve.getColor(time)

Parameters time – int
Return type ColorTuple
Returns the value of the color parameter at the given time as an [R,G,B,A] tuple. Note that alpha will always be 1.

NatronEngine.BezierCurve.getColorParam()

Return type ColorParam
Returns the Param controlling the color of the bezier.
Return type NatronEngine.BezierCurve.CairoOperatorEnum
Returns the blending mode for this shape. Type help(NatronEngine.BezierCurve.CairoOperatorEnum) to see the different values possible.

NatronEngine.BezierCurve.getCompositingOperatorParam()

Return type NatronEngine.ChoiceParam
Returns the Param controlling the blending mode of the bezier.

NatronEngine.BezierCurve.getFeatherDistance(time)

Parameters time – int
Return type float
Returns the feather distance of this shape at the given time.

NatronEngine.BezierCurve.getFeatherDistanceParam()

Return type NatronEngine.DoubleParam
Returns the Param controlling the feather distance of the bezier.

NatronEngine.BezierCurve.getFeatherFallOff(time)

Parameters time – int
Return type float
Returns the feather fall-off of this shape at the given time.

NatronEngine.BezierCurve.getFeatherFallOffParam()

Return type DoubleParam
Returns the Param controlling the color of the bezier.

NatronEngine.BezierCurve.getIsActivated(time)

Parameters time – int
Return type bool
Returns whether the curve is enabled or not at the given time. When not activated the curve will not be rendered at all in the image.

NatronEngine.BezierCurve.getNumControlPoints()

Return type int
Returns the number of control points for this shape.

NatronEngine.BezierCurve.getOpacity(time)

Parameters time – int
Return type float
Returns the opacity of the curve at the given time.

NatronEngine.BezierCurve.getOpacityParam()

Return type DoubleParam
Returns the Param controlling the opacity of the bezier.

NatronEngine.BezierCurve.getOverlayColor()

Return type ColorTuple
Returns the overlay color of this shape as a [R,G,B,A] tuple. Alpha will always be 1.

NatronEngine.BezierCurve.getPointMasterTrack(index)

Parameters

- **index** – int

Return type DoubleParam

Returns the **Param** of the center of the track controlling the control point at the given **index**.

NatronEngine.BezierCurve.isCurveFinished()

Return type bool

Returns whether the curve is finished or not. A finished curve will have a bezier segment between the last control point and the first control point and the bezier will be rendered in the image.

NatronEngine.BezierCurve.moveFeatherByIndex(index, time, dx, dy)

Parameters

- **index** – int
- **time** – int
- **dx** – float
- **dy** – float

Moves the feather point at the given **index** (zero-based) by the given delta (dx,dy). The **time** parameter is given so that if auto-keying is enabled a new keyframe will be set.

NatronEngine.BezierCurve.moveLeftBezierPoint(index, time, dx, dy)

Parameters

- **index** – int
- **time** – int
- **dx** – float
- **dy** – float

Moves the left bezier point of the control point at the given **index** by the given delta. The **time** parameter is given so that if auto-keying is enabled a new keyframe will be set.

NatronEngine.BezierCurve.movePointByIndex(index, time, dx, dy)

Parameters

- **index** – int
- **time** – int
- **dx** – float
- **dy** – float

Moves the point at the given **index** (zero-based) by the given delta (dx,dy). The **time** parameter is given so that if auto-keying is enabled a new keyframe will be set.

NatronEngine.BezierCurve.moveRightBezierPoint(index, time, dx, dy)

Parameters

- **index** – int
- **time** – int
• `dx` – float
• `dy` – float

Moves the right bezier point at the given `index` (zero-based) by the given delta (dx,dy). The `time` parameter is given so that if auto-keying is enabled a new keyframe will be set.

NatronEngine.BezierCurve.removeControlPointByIndex(index)

**Parameters**

- `index` – int

Removes the control point at the given `index` (zero-based).

NatronEngine.BezierCurve.setActivated(time, activated)

**Parameters**

- `time` – int
- `activated` – bool

Set a new keyframe for the `activated` parameter at the given `time`

NatronEngine.BezierCurve.setColor(time, r, g, b)

**Parameters**

- `time` – int
- `r` – float
- `g` – float
- `b` – float

Set a new keyframe for the `color` parameter at the given `time`

NatronEngine.BezierCurve.setCompositingOperator(op)

**Parameters**

- `op` – NatronEngine.BezierCurve.CairoOperatorEnum

Set the compositing operator for this shape.

NatronEngine.BezierCurve.setCurveFinished(finished)

**Parameters**

- `finished` – bool

Set whether the curve should be finished or not. See `isCurveFinished()`

NatronEngine.BezierCurve.setFeatherDistance(dist, time)

**Parameters**

- `dist` – float
- `time` – int

Set a new keyframe for the `feather distance` parameter at the given `time`

NatronEngine.BezierCurve.setFeatherFallOff(falloff, time)

**Parameters**

- `falloff` – float
- `time` – int

Set a new keyframe for the `feather fall-off` parameter at the given `time`

NatronEngine.BezierCurve.setFeatherPointAtIndex(index, time, x, y, lx, ly, rx, ry)
Parameters

• **index** – int
• **time** – int
• **x** – float
• **y** – float
• **lx** – float
• **ly** – float
• **rx** – float
• **ry** – float

Set the feather point at the given *index* at the position (x,y) with the left bezier point at (lx,ly) and right bezier point at (rx,ry).

The *time* parameter is given so that if auto-keying is enabled a new keyframe will be set.

NatronEngine.BezierCurve.setOpacity(*opacity, time*)

Parameters

• **opacity** – float
• **time** – int

Set a new keyframe for the *opacity* parameter at the given *time*

NatronEngine.BezierCurve.setOverlayColor(*r, g, b*)

Parameters

• **r** – float
• **g** – float
• **b** – float

Set the overlay color of this shape

NatronEngine.BezierCurve.setPointAtIndex(*index, time, x, y, lx, ly, rx, ry*)

Parameters

• **index** – int
• **time** – int
• **x** – float
• **y** – float
• **lx** – float
• **ly** – float
• **rx** – float
• **ry** – float

Set the point at the given *index* at the position (x,y) with the left bezier point at (lx,ly) and right bezier point at (rx,ry).

The *time* parameter is given so that if auto-keying is enabled a new keyframe will be set.

NatronEngine.BezierCurve.slavePointToTrack(*index, trackTime, trackCenter*)
Parameters

- **index** – int
- **trackTime** – int
- **trackCenter** – Double2DParam

Slave the control point at the given `index` to the `trackCenter` parameter. The `trackCenter` must be a 2-dimensional double parameter and must be the parameter called `center` of a track.

Once slaved the control point will move as a relative offset from the track. The offset is initially 0 at the given `trackTime`.

**BooleanParam**

Inherits AnimatedParam

**Synopsis**

A parameter that contains a boolean value. See detailed description below

**Functions**

- `def get ()`
- `def get (frame)`
- `def getDefaultValue ()`
- `def getValue ()`
- `def getValueAtTime (time)`
- `def restoreDefaultValue ()`
- `def set (x)`
- `def set (x, frame)`
- `def setDefaultValue (value)`
- `def setValue (value)`
- `def setValueAtTime (value, time)`

**Detailed Description**

A BooleanParam looks like a checkbox in the user interface.
Member functions description

NatronEngine.BooleanParam.get()  
Return type  bool  
Returns the value of the parameter at the current timeline’s time.

NatronEngine.BooleanParam.get(frame)  
Parameters  frame – float  
Return type  bool  
Returns the value of the parameter at the given frame. This value may be interpolated given the interpolation of the underlying animation curve.

NatronEngine.BooleanParam.getDefaultValue()  
Return type  bool  
Returns the default value for this parameter.

NatronEngine.BooleanParam.getValue()  
Return type  PySide.QtCore.bool  
Same as get()

NatronEngine.BooleanParam.getValueAtTime(time)  
Parameters  time – float  
Return type  bool  
Same as get(frame)

NatronEngine.BooleanParam.restoreDefaultValue()  
Removes all animation and expression set on this parameter and set the value to be the default value.

NatronEngine.BooleanParam.set(x)  
Parameters  x – bool  
Set the value of this parameter to be x. If this parameter is animated (see getIsAnimated(dimension)) then this function will automatically add a keyframe at the timeline’s current time.

NatronEngine.BooleanParam.set(x, frame)  
Parameters  
• x – bool  
• frame – float  
Set a new keyframe on the parameter with the value x at the given frame.

NatronEngine.BooleanParam.setValue(value)  
Parameters  value – bool  
Set the default value for this parameter.

NatronEngine.BooleanParam.setValueAtTime(value, time)  
Parameters  
• value – bool  
Same as set(value)
Parameters

- `value` - bool
- `time` - float

Same as `set(value, time)`

**ButtonParam**

**Inherits** `Param`

**Synopsis**

A button parameter that appears in the settings panel of the node.

To insert code to be executed upon a user click of the button, register a function to the `onParamChanged` callback on the node.

**Functions**

- `def setIconFilePath(icon)`
- `def trigger()`

**Member functions description**

NatronEngine.ButtonParam.setIconFilePath(icon)

**Parameters**

- `icon` - str

Set here the icon file path for the button. This should be either an absolute path or a file-path relative to a path in the NATRON_PLUGIN_PATH.

NatronEngine.ButtonParam.trigger()

Triggers the button action as though the user had pressed it.

**ChoiceParam**

**Inherits**: AnimatedParam

**Synopsis**

A choice parameter holds an integer value which corresponds to a choice. See `detailed description` below.

Chapter 1. Natron Python API reference
Functions

- `def addOption(option, help)`
- `def get()`
- `def get(frame)`
- `def getDefaultValue()`
- `def getOption(index)`
- `def getNumOptions()`
- `def getOptions()`
- `def getValue()`
- `def getValueAtTime(time)`
- `def restoreDefaultValue()`
- `def set(x)`
- `def set(x, frame)`
- `def set(label)`
- `def setDefaultValue(value)`
- `def setDefaultValue(label)`
- `def setOptions(options)`
- `def setValue(value)`
- `def setValueAtTime(value, time)`

**Detailed Description**

A choice is represented as a drop-down (combobox) in the user interface:

![Filter: Gaussian ▼](image)

You can add options to the menu using the `addOption(option, help)` function. You can also set them all at once using the `setOptions(options)` function.

The value held internally is a 0-based index corresponding to an entry of the menu. The choice parameter behaves much like an `IntParam`.

**Member functions description**

`NatronEngine.ChoiceParam.addOption(option, help)`

**Parameters**

- `option` - str
- `help` - str

Adds a new `option` to the menu. If `help` is not empty, it will be displayed when the user hovers the entry with the mouse.

`NatronEngine.ChoiceParam.get(frame)`
Parameters **frame** – float

**Return type** int

Get the value of the parameter at the given *frame*.

NatronEngine.ChoiceParam.get()

**Return type** int

Get the value of the parameter at the current timeline’s time.

NatronEngine.ChoiceParam.getDefaultValue()

**Return type** int

Get the default value for this parameter.

NatronEngine.ChoiceParam.option(index)

**Parameters** index – int

**Return type** str

Get the menu entry at the given *index*.

NatronEngine.ChoiceParam.getNumOptions()

**Return type** int

Returns the number of menu entries.

NatronEngine.ChoiceParam.getOptions()

**Return type** sequence

Returns a sequence of string with all menu entries from top to bottom.

NatronEngine.ChoiceParam.getValue()

**Return type** int

Same as *get()*

NatronEngine.ChoiceParam.getValueAtTime(time)

**Parameters** time – float

**Return type** float

Same as *get(frame)*

NatronEngine.ChoiceParam.restoreDefaultValue()

Removes all animation and expression set on this parameter and set the value to be the default value.

NatronEngine.ChoiceParam.set(x)

**Parameters** x – int

Set the value of this parameter to be *x*. If this parameter is animated (see *getIsAnimated(dimension)*) then this function will automatically add a keyframe at the timeline’s current time.

NatronEngine.ChoiceParam.set(x, frame)

**Parameters**

• x – int
  
• frame – float
Set a new keyframe on the parameter with the value `x` at the given `frame`.

```
NatronEngine.ChoiceParam.set(label)
```

**Parameters**
- `label` – `str`

Set the value of this parameter given a `label`. The `label` must match an existing option. Strings will be compared without case sensitivity. If not found, nothing happens.

```
NatronEngine.ChoiceParam.setDefaultValue(value)
```

**Parameters**
- `value` – `int`

Set the default `value` for this parameter.

```
NatronEngine.ChoiceParam.setDefaultValue(label)
```

**Parameters**
- `label` – `str`

Set the default value from the `label` for this parameter. The `label` must match an existing option. Strings will be compared without case sensitivity. If not found, nothing happens.

```
NatronEngine.ChoiceParam.setOptions(options)
```

**Parameters**
- `options` – `class::sequence`

Clears all existing entries in the menu and add all entries contained in `options` to the menu.

```
NatronEngine.ChoiceParam.setValue(value)
```

**Parameters**
- `value` – `int`

Same as `set`

```
NatronEngine.ChoiceParam.setValueAtTime(value, time)
```

**Parameters**
- `value` – `int`
- `time` – `int`

Same as `set(time)<NatronEngine.ChoiceParam.set()`

---

ColorParam

Inherits AnimatedParam

**Synopsis**

A color parameter is a RGB[A] value that can be animated throughout the time. See *detailed* description...

**Functions**

- def `get()`
- def `get(frame)`
- def `getDefaultValue([dimension=0])`
- def `getDisplayMaximum(dimension)`
- def `getDisplayMinimum(dimension)`
- def `getMaximum([dimension=0])`
• def `getMinimum` ([dimension=0])
• def `getValue` ([dimension=0])
• def `getValueAtTime` (time[, dimension=0])
• def `restoreDefaultValue` ([dimension=0])
• def `set` (r, g, b, a)
• def `set` (r, g, b, a, frame)
• def `setDefaultValue` (value[, dimension=0])
• def `setDisplayMaximum` (maximum[, dimension=0])
• def `setDisplayMinimum` (minimum[, dimension=0])
• def `setMaximum` (maximum[, dimension=0])
• def `setMinimum` (minimum[, dimension=0])
• def `setValue` (value[, dimension=0])
• def `setValueAtTime` (value, time[, dimension=0])

**Detailed Description**

A color parameter can either be of dimension 3 (RGB) or dimension 4 (RGBA). The user interface for this parameter looks like this:

![Color parameter interface](image)

This parameter type is very similar to a `Double3DParam` except that it can have 4 dimensions and has some more controls.

**Member functions description**

**NatronEngine.ColorParam.get(frame)**

- Parameters: `frame` – float
- Return type: `ColorTuple`

Returns a `ColorTuple` of the color held by the parameter at the given `frame`.

**NatronEngine.ColorParam.get()**

- Return type: `ColorTuple`

Returns a `ColorTuple` of the color held by the parameter at the current timeline’s time.

**NatronEngine.ColorParam.getDefaultValue([dimension=0])**

- Parameters: `dimension` – int
- Return type: float

Returns the default value for this parameter at the given `dimension`.

**NatronEngine.ColorParam.getDisplayMaximum(dimension)**

- Parameters: `dimension` – int
**Return type** float

Returns the display maximum for this parameter at the given *dimension*. The display maximum is the maximum value visible on the slider, internally the value can exceed this range.

NatronEngine.ColorParam.displayMinimum(*dimension*)

**Parameters**

- dimension – int

**Return type** float

Returns the display minimum for this parameter at the given *dimension*. The display minimum is the minimum value visible on the slider, internally the value can exceed this range.

NatronEngine.ColorParam.displayMinimum(*dimension*)

**Parameters**

- dimension – int

**Return type** float

Returns the maximum for this parameter at the given *dimension*. The maximum value cannot be exceeded and any higher value will be clamped to this value.

NatronEngine.ColorParam.displayMinimum(*dimension*)

**Parameters**

- dimension – int

**Return type** float

Returns the minimum for this parameter at the given *dimension*. The minimum value cannot be exceeded and any lower value will be clamped to this value.

NatronEngine.ColorParam.displayMinimum(*dimension*)

**Parameters**

- dimension – int

**Return type** float

Returns the value of this parameter at the given *dimension* at the current timeline’s time.

NatronEngine.ColorParam.getValueAtTime(*time[, dimension=0]*)

**Parameters**

- time – float
- dimension – int

**Return type** float

Returns the value of this parameter at the given *dimension* at the given *time*.

NatronEngine.ColorParam.restoreDefaultValue(*dimension*)

**Parameters**

- dimension – int

Removes all animation and expression set on this parameter and set the value to be the default value.

NatronEngine.ColorParam.set(*r, g, b, a, frame*)

**Parameters**

- r – float
- g – float
- b – float
- a – float
• frame – float

Set a keyframe on each of the 4 animations curves at [r,g,b,a] for the given frame. If this parameter is 3-dimensional, the a value is ignored.

NatronEngine.ColorParam.set(r, g, b, a)

Parameters

• r – float
• g – float
• b – float
• a – float

Set the value of this parameter to be [r,g,*b,*a]. If this parameter is animated (see getIsAnimated(dimension)) then this function will automatically add a keyframe at the timeline’s current time.

NatronEngine.ColorParam.setDefaultValue(value[, dimension=0])

Parameters

• value – float
• dimension – int

Set the default value of this parameter at the given dimension to be value.

NatronEngine.ColorParam.setDisplayMaximum(maximum[, dimension=0])

Parameters

• maximum – float
• dimension – int

Set the display maximum of the parameter to be maximum for the given dimension. See getDisplayMaximum

NatronEngine.ColorParam.setDisplayMinimum(minimum[, dimension=0])

Parameters

• minimum – float
• dimension – int

Set the display minimum of the parameter to be minimum for the given dimension. See getDisplayMinimum

NatronEngine.ColorParam.setMaximum(maximum[, dimension=0])

Parameters

• maximum – float
• dimension – int

Set the maximum of the parameter to be maximum for the given dimension. See getMaximum

NatronEngine.ColorParam.setMinimum(minimum[, dimension=0])

Parameters

• minimum – float
• dimension – int

Set the minimum of the parameter to be minimum for the given dimension. See getMinimum
NatronEngine.ColorParam.setValue(value[, dimension=0])

Parameters
- `value` – float
- `dimension` – int

Set the value of this parameter at the given `dimension` to be `value`. If this parameter is animated (see `getIsAnimated(dimension)` then this function will automatically add a keyframe at the timeline’s current time.

NatronEngine.ColorParam.setValueAtTime(value[, time[, dimension=0]])

Parameters
- `value` – float
- `time` – int
- `dimension` – int

Set a keyframe on each of the animation curve at the given `dimension`. The keyframe will be at the given `time` with the given `value`.

**ColorTuple**

**Synopsis**

Utility class used to return [R,G,B,[A]] values.

**Functions**
- `def __getitem__(arg__1)`

class NatronEngine.ColorTuple

NatronEngine.ColorTuple.g
NatronEngine.ColorTuple.r
NatronEngine.ColorTuple.a
NatronEngine.ColorTuple.b
NatronEngine.ColorTuple.__getitem__(index)

Parameters `arg__1` – int

Return type float

Returns the item at the given index. This is the bracket operator `[]`.

**Double2DParam**

Inherits DoubleParam

Inherited by: Double3DParam
Synopsis

See `DoubleParam` for more informations on this class.

Functions

- `def setUsePointInteract<NatronEngine.Double2DParam.setUsePointInteract() (enabled)
- `def get ()
- `def get (frame)
- `def set (x, y)
- `def set (x, y, frame)

Member functions description

NatronEngine.Double2DParam.setUsePointInteract (enabled)

- **Parameters** enabled – bool
  - When called, the parameter will have its own overlay interact on the viewer as a point that the user can select and drag.

NatronEngine.Double2DParam.get ()

- **Return type** `Double2DTuple`
  - Returns a `Double2DTuple` with the [x,y] values for this parameter at the current timeline’s time.

NatronEngine.Double2DParam.get (frame)

- **Parameters** frame – float
  - **Return type** `Double2DTuple`
  - Returns a `Double2DTuple` with the [x,y] values for this parameter at the given frame.
NatronEngine.Double2DParam.set(x, y, frame)

Parameters
- x – float
- y – float
- frame – float

Same as set(x, frame) but for 2-dimensional doubles.

NatronEngine.Double2DParam.set(x, y)

Parameters
- x – float
- y – float

Same as set(x) but for 2-dimensional doubles.

Double2DTuple

Synopsis

Utility class to return pair of floating point values.

Functions
- def __getitem__(arg__1)

Detailed Description

class NatronEngine.Double2DTuple
NatronEngine.Double2DTuple.x
NatronEngine.Double2DTuple.y
NatronEngine.Double2DTuple.__getitem__(index)

Parameters  index – int

Return type  float

Returns the item at the given index. This is the bracket operator []

Double3DParam

Inherits Double2DParam

Synopsis

See DoubleParam for more informations on this class.
Functions

- def get()
- def get(frame)
- def set(x, y, z)
- def set(x, y, z, frame)

Member functions description

NatronEngine.Double3DParam.get()

    Return type Double3DTuple

Returns a Double3DTuple with the [x,y,z] values for this parameter at the current timeline’s time.

NatronEngine.Double3DParam.get(frame)

    Parameters frame – float
    Return type Double3DTuple

Returns a Double3DTuple with the [x,y,z] values for this parameter at the given frame.

NatronEngine.Double3DParam.set(x,y,z,frame)

    Parameters
    • x – float
    • y – float
    • z – float
    • frame – PySide.QtCore.int

Same as set(x,frame) but for 3-dimensional doubles.

NatronEngine.Double3DParam.set(x,y,z)

    Parameters
    • x – float
    • y – float
    • z – float

Same as set(x) but for 3-dimensional doubles.

Double3DTuple

Synopsis

Utility class to return pair of floating point values.

Functions

- def __getitem__(index)
**Detailed Description**

```python
class NatronEngine.Double3DTuple
NatronEngine.Double3DTuple.x
NatronEngine.Double3DTuple.y
NatronEngine.Double3DTuple.z
NatronEngine.Double3DTuple.__getitem__(index)
```

**Parameters**
- `index` - int

**Return type**
- float

Returns the item at the given index. This is the bracket operator `[]`

**DoubleParam**

Inherits **AnimatedParam**

Inherited by: **Double2DParam, Double3DParam**

**Synopsis**

A double param can contain one or multiple floating point values. See *detailed* description...

**Functions**

- `def get ()`
- `def get (frame)`
- `def getDefaultValue ([dimension=0])`
- `def getDisplayMaximum (dimension)`
- `def getDisplayMinimum (dimension)`
- `def getMaxima ([dimension=0])`
- `def getMinimum ([dimension=0])`
- `def getValue ([dimension=0])`
- `def getValueAtTime (time[, dimension=0])`
- `def restoreDefaultValue ([dimension=0])`
- `def set (x)`
- `def set (x, frame)`
- `def setDefaultValue (value[, dimension=0])`
- `def setDisplayMaximum (maximum[, dimension=0])`
- `def setDisplayMinimum (minimum[, dimension=0])`
- `def setMaximum (maximum[, dimension=0])`
- `def setMinimum (minimum[, dimension=0])`
• def setValue (value[, dimension=0])
• def setValueAtTime (value, time[, dimension=0])

**Detailed Description**

A double param can have 1 to 3 dimensions. (See Double2DParam and Double3DParam). Usually this is used to represent a single floating point value that may animate over time.

The user interface for them varies depending on the number of dimensions.

A 1-dimensional DoubleParam

A 2-dimensional Double2DParam

A 3-dimensional Double3DParam

**Member functions description**

NatronEngine.DoubleParam.get (frame)

Parameters frame – float

Return type float

Returns the value of this parameter at the given frame. If the animation curve has an animation (see getIsAnimated then the value will be interpolated using the interpolation chosen by the user for the curve.

NatronEngine.DoubleParam.get ()

Return type float

Returns the value of this parameter at the given current timeline's time.

NatronEngine.DoubleParam.getDefaultValue ([dimension=0])

Parameters dimension – int

Return type float

Returns the default value for this parameter. dimension is meaningless for the DoubleParam class because it is 1-dimensional, but is useful for inherited classes Double2DParam and Double3DParam

NatronEngine.DoubleParam.getDisplayMaximum (dimension)

Parameters dimension – int

Return type double

Returns the display maximum for this parameter at the given dimension. The display maximum is the maximum value visible on the slider, internally the value can exceed this range.
NatronEngine.DoubleParam.getDisplayMinimum(dimension)

Parameters  

dimension – int

Return type  

float

Returns the display minimum for this parameter at the given dimension. The display minimum is the minimum value visible on the slider, internally the value can exceed this range.

NatronEngine.DoubleParam.getMaximum([dimension=0])

Parameters  

dimension – int

Return type  

float

Returns the maximum for this parameter at the given dimension. The maximum value cannot be exceeded and any higher value will be clamped to this value.

NatronEngine.DoubleParam.getMinimum([dimension=0])

Parameters  

dimension – int

Return type  

float

Returns the minimum for this parameter at the given dimension. The minimum value cannot be exceeded and any lower value will be clamped to this value.

NatronEngine.DoubleParam.getValue([dimension=0])

Parameters  

dimension – int

Return type  

float

Returns the value of this parameter at the given dimension at the current timeline’s time.

NatronEngine.DoubleParam.getValueAtTime(time[, dimension=0])

Parameters  

• time – float

• dimension – int

Return type  

float

Returns the value of this parameter at the given dimension at the given time.

If the animation curve has an animation (see getIsAnimated then the value will be interpolated using the interpolation chosen by the user for the curve.

NatronEngine.DoubleParam.restoreDefaultValue([dimension=0])

Parameters  

dimension – int

Returns the value of this parameter at the given dimension at the given time.

NatronEngine.DoubleParam.set(x, frame)

Parameters  

• x – float

• frame – float

Set a new keyframe on the parameter with the value x at the given frame.
Set the value of this parameter to be $x$. If this parameter is animated (see `getIsAnimated(dimension)` then this function will automatically add a keyframe at the timeline’s current time.

```python
NatronEngine.DoubleParam.setDefaultValue(value[, dimension=0])
```

**Parameters**
- `value` – float
- `dimension` – int

Set the default value for this parameter at the given dimension.

```python
NatronEngine.DoubleParamsetDisplayMaximum(maximum[, dimension=0])
```

**Parameters**
- `maximum` – float
- `dimension` – int

Set the display maximum of the parameter to be `maximum` for the given dimension. See `getDisplayMaximum`

```python
NatronEngine.DoubleParamsetDisplayMinimum(minimum[, dimension=0])
```

**Parameters**
- `minimum` – float
- `dimension` – int

Set the display minimum of the parameter to be `minimum` for the given dimension. See `getDisplayMinimum`

```python
NatronEngine.DoubleParam.setMaximum(maximum[, dimension=0])
```

**Parameters**
- `maximum` – float
- `dimension` – int

Set the maximum of the parameter to be `maximum` for the given dimension. See `getMaximum`

```python
NatronEngine.DoubleParam.setMinimum(minimum[, dimension=0])
```

**Parameters**
- `minimum` – float
- `dimension` – int

Set the minimum of the parameter to be `minimum` for the given dimension. See `getMinimum`

```python
NatronEngine.DoubleParam.setValue(value[, dimension=0])
```

**Parameters**
- `value` – float
- `dimension` – int

Same as `set(value, dimension)`

```python
NatronEngine.DoubleParam.setValueAtTime(value, time[, dimension=0])
```

**Parameters**
- `value` – float
- `time` – float
- **dimension** – int

Same as `set(value, time, dimension)`

**Effect**

**Inherits:** Group, UserParamHolder

**Synopsis**

This object represents a single node in Natron, that is: an instance of a plug-in. See Detailed Description

**Functions**

- `def addUserPlane(planeName,channels)`
- `def endChanges()`
- `def beginChanges()`
- `def canConnectInput(inputNumber, node)`
- `def connectInput(inputNumber, input)`
- `def createChild()`
- `def destroy([autoReconnect=true])`
- `def disconnectInput(inputNumber)`
- `def getAvailableLayers()`
- `def getBitDepth()`
- `def getColor()`
- `def getCurrentTime()`
- `def getFrameRate()`
- `def getInput(inputNumber)`
- `def getLabel()`
- `def getInputLabel(inputNumber)`
- `def getMaxInputCount()`
- `def getParam(name)`
- `def getParams()`
- `def getPluginID()`
- `def getPosition()`
- `def getPremult()`
- `def getPixelAspectRatio()`
- `def getRegionOfDefinition(time,view)`
- `def getRotoContext()`
- `def getScriptName()`

1.1. NatronEngine
Detailed Description

The Effect object can be used to operate with a single node in Natron. To create a new Effect, use the `app.createNode(pluginID)` function.

Natron automatically declares a variable to Python when a new Effect is created. This variable will have a script-name determined by Natron as explained in the Python Auto-declared variables section.

Once an Effect is instantiated, it declares all its Param and inputs. See how to manage user parameters below.

To get a specific Param by script-name, call the `getParam(name)` function.

Input effects are mapped against a zero-based index. To retrieve an input Effect given an index, you can use the `getInput(inputNumber)` function.

To manage inputs, you can connect them and disconnect them with respect to their input index with the `connectInput(inputNumber,input)` and then `disconnectInput(inputNumber)` functions.

If you need to destroy permanently the Effect, just call `destroy()`.

For convenience some GUI functionalities have been made accessible via the Effect class to control the GUI of the node (on the node graph):

- Get/Set the node position with the `setPosition(x,y)` and `getPosition()` functions
- Get/Set the node size with the `setSize(width,height)` and `getSize()` functions
- Get/Set the node color with the `setColor(r,g,b)` and `getColor()` functions

Creating user parameters  See this section

Member functions description

NatronEngine.Effect.addUserPlane` (planeName, channels)

- **Parameters**
  - `planeName` – str
  - `channels` – sequence

- **Return type** bool
Adds a new plane to the Channels selector of the node in its settings panel. When selected, the end-user can choose to output the result of the node to this new custom plane. The `planeName` will identify the plane uniquely and must not contain spaces or non-python compliant characters. The `channels` are a sequence of channel names, e.g:

```
```

**Note:** A plane cannot contain more than 4 channels and must at least have 1 channel.

This function returns `True` if the layer was added successfully, `False` otherwise.

**NatronEngine.Effect.beginChanges()**

Starts a begin/End bracket, blocking all evaluation (=renders and callback onParamChanged) that would be issued due to a call to `setValue` on any parameter of the Effect.

Similarly all input changes will not be evaluated until `endChanges()` is called.

Typically to change several values at once we bracket the changes like this:

```
node.beginChanges()
param1.setValue(...)  
param2.setValue(...)  
param3.setValue(...)  
param4.setValue(...)  
node.endChanges()  # This triggers a new render
```

A more complex call:

```
node.beginChanges()  
        node.connectInput(0, otherNode)  
        node.connectInput(1, thirdNode)  
        param1.setValue(...)  
        node.endChanges()  # This triggers a new render
```

**NatronEngine.Effect.endChanges()**

Ends a begin/end bracket. If the begin/end bracket recursion reaches 0 and there were calls made to `setValue` this function will effectively compress all evaluations into a single one. See `beginChanges()`.

**NatronEngine.Effect.canConnectInput (inputNumber, node)**

**Parameters**

- `inputNumber` - int
- `node` - Effect

**Return type**  bool

Returns whether the given `node` can be connected at the given `inputNumber` of this Effect. This function could return False for one of the following reasons:

- The Effect already has an input at the given `inputNumber`
- The `node` is None
- The given `inputNumber` is out of range
- The `node` cannot have any node connected to it (such as a BackDrop or an Output)
- This Effect or the given `node` is a child of another node (for trackers only)
- Connecting `node` would create a cycle in the graph implying that it would create infinite recursions

**NatronEngine.Effect.connectInput (inputNumber, input)**

**Parameters**

- `inputNumber` - int
• input – Effect

     Return type  bool

Connects `input` to the given `inputNumber` of this Effect. This function calls internally `canConnectInput()` to
determine if a connection is possible.

NatronEngine.Effect.createChild()

     Return type  Effect

If this Effect is a multi-instance node (currently only the Tracker node is a multi-instance) then this function will create
a new instance of the same plug-in as a child of this node. This is used for Trackers to create new tracks: internally
each track is in fact a separate node on its own.

NatronEngine.Effect.destroy([`autoReconnect=true`])

     Parameters  autoReconnect – bool

Removes this Effect from the current project definitively. If `autoReconnect` is True then any nodes connected to this
node will try to connect their input to the input of this node instead.

NatronEngine.Effect.disconnectInput(inputNumber)

     Parameters  inputNumber – int

Removes any input Effect connected to the given `inputNumber` of this node.

NatronEngine.Effect.getAvailableLayers()

     Return type  dict

Returns the layer available on this node. This is a dict with a ImageLayer as key and Effect as value. The Effect
is the closest node in the upstream tree (including this node) that produced that layer.

For example, in a simple graph Read → Blur, if the Read node has a layer available named “Render-
Layer.combined” but Blur is set to process only the color layer (RGBA), then calling this function on the Blur
will return a dict containing for key “RenderLayer.combined” the Read node, whereas the dict will have for the
key “RGBA” the Blur node.

NatronEngine.Effect.getBitDepth()

     Return type  ImageBitDepthEnum

Returns the bit-depth of the image in output of this node.

NatronEngine.Effect.getColor()

     Return type  tuple

Returns the color of this node as it appears on the node graph as [R,G,B] 3-dimensional tuple.

NatronEngine.Effect.getCurrentTime()

     Return type  int

Returns the current time of timeline if this node is currently rendering, otherwise it returns the current time at which
the node is currently rendering for the caller thread.

NatronEngine.Effect.getFrameRate()

     Return type  float

Returns the frame-rate of the sequence in output of this node.

NatronEngine.Effect.getInput(inputNumber)

     Parameters  inputNumber – int
Return type  Effect

Returns the node connected at the given `inputNumber`.

NatronEngine.Effect.getLabel()

Return type  str

Returns the label of the node. See this section for a discussion of the label vs the script-name.

NatronEngine.Effect.getInputLabel(inputNumber)

param inputNumber  int

Return type  str

Returns the label of the input at the given `inputNumber`. It corresponds to the label displayed on the arrow of the input in the node graph.

NatronEngine.Effect.getMaxInputCount()

Return type  int

Returns the number of inputs for the node. Graphically this corresponds to the number of arrows in input.

NatronEngine.Effect.getParam(name)

Parameters  name – str

Return type  Param

Returns a parameter by its script-name or None if no such parameter exists.

NatronEngine.Effect.getParams()

Return type  sequence

Returns all the parameters of this Effect as a sequence.

NatronEngine.Effect.getPluginID()

Return type  str

Returns the ID of the plug-in that this node instantiate.

NatronEngine.Effect.getPosition()

Return type  tuple

Returns the current position of the node on the node-graph. This is a 2 dimensional [X,Y] tuple. Note that in background mode, if used, this function will always return [0,0] and should NOT be used.

NatronEngine.Effect.getPremult()

Return type  ImagePremultiplicationEnum

Returns the alpha premultiplication state of the image in output of this node.

NatronEngine.Effect.getPixelAspectRatio()

Return type  float

Returns the pixel aspect ratio of the image in output of this node.

NatronEngine.Effect.getRegionOfDefinition(time, view)

Parameters

•  time – float
Returns the bounding box of the image produced by this effect in canonical coordinates. This is exactly the value displayed in the “Info” tab of the settings panel of the node for the “Output”. This can be useful for example to set the position of a point parameter to the center of the region of definition.

NatronEngine.Effect.getRotoContext()

Returns the roto context for this node. Currently only the Roto node has a roto context. The roto context is in charge of maintaining all informations relative to Beziers and Layers. Most of the nodes don’t have a roto context though and this function will return None.

NatronEngine.Effect.getScriptName()

Returns the script-name of this Effect. See this section for more information about the script-name.

NatronEngine.Effect.getSize()

Returns the size of this node on the node-graph as a 2 dimensional [Width,Height] tuple. Note that calling this function will in background mode will always return [0,0] and should not be used.

NatronEngine.Effect.getUserPageParam()

Convenience function to return the user page parameter if this Effect has one.

NatronEngine.Effect.isUserSelected()

Returns true if this node is selected in its containing nodegraph.

NatronEngine.Effect.setColor(r, g, b)

Parameters

- r – float
- g – float
- b – float

Set the color of the node as it appears on the node graph. Note that calling this function will in background mode will do nothing and should not be used.

NatronEngine.Effect.setLabel(name)

Parameters

name – str

Set the label of the node as it appears in the user interface. See this section for an explanation of the difference between the label and the script-name.

NatronEngine.Effect.setPosition(x, y)

Parameters

- x – float
• \( y \) – float

Set the position of the node as it appears on the node graph. Note that calling this function will in background mode will do nothing and should not be used.

NatronEngine.Effect.setScriptName\( (\text{scriptName}) \)

Parameters \text{scriptName} – str

Return type bool

Set the script-name of the node as used internally by Natron. See this section for an explanation of the difference between the label and the script-name.

**Warning:** Using this function will remove any previous variable declared using the old script-name and will create a new variable with the new script name if valid.

If your script was using for instance a node named:

app1.Blur1

and you renamed it BlurOne, it should now be available to Python this way:

app1.BlurOne

but using app1.Blur1 would report the following error:

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'Blur1' is not defined
```

NatronEngine.Effect.setSize\( (w, h) \)

Parameters
• \( w \) – float
• \( h \) – float

Set the size of the node as it appears on the node graph. Note that calling this function will in background mode will do nothing and should not be used.

NatronEngine.Effect.setSubGraphEditable\( (\text{editable}) \)

Parameters \text{editable} – bool

Can be called to disable editing of the group via Natron’s graphical user interface. This is handy to prevent users from accidentally breaking the sub-graph. This can always be reverted by editing the python script associated. The user will still be able to see the internal node graph but will not be able to unlock it.

NatronEngine.Effect.setPagesOrder\( (\text{pages}) \)

Parameters \text{pages} – sequence

Given the string list \text{pages} try to find the corresponding pages by their-script name and order them in the given order.

**FileParam**

Inherits StringParamBase
**Synopsis**

This parameter is used to specify an input file (i.e: a file that already exist).

**Functions**

- `def openFile ()`

- `def setSequenceEnabled (enabled)`

**Member functions description**

NatronEngine.FileParam.openFile()

When called in GUI mode, this will open a file dialog for the user. Does nothing in background mode.

NatronEngine.FileParam.setSequenceEnabled (enabled)

**Parameters** enabled `– bool`

Determines whether the file dialog opened by `openFile()` should have support for file sequences or not.

**Group**

**Inherited by:** Effect, App, GuiApp

**Synopsis**

Base class for Effect and App. See detailed description below.

**Functions**

- `def getChildren ()`

- `def getNode (fullyQualifiedName)`

**Detailed Description**

This is an abstract class, it is derived by 2 different classes:

- App which represents an instance of Natron, or more specifically the current project.
- Effect which represents a node in the node graph.

The `getNode(fullyQualifiedName)` can be used to retrieve a node in the project, although all nodes already have an auto-declared variable by Natron.

**Member functions description**

NatronEngine.Group.getChildren()

**Return type** sequence

Returns a sequence with all nodes in the group. Note that this function is not recursive and you’d have to call getChildren() on all sub-groups to retrieve their children, etc...

NatronEngine.Group.getNode (fullyQualifiedName)

**Parameters** fullySpecifiedName `– str`
**Return type** Effect

Retrieves a node in the group with its *fully qualified name*. The fully qualified name of a node is the *script-name* of the node prefixed by all the group hierarchy into which it is, e.g:

```plaintext
Blur1  # the node is a top level node
```

```plaintext
Group1.Group2.Blur1  # the node is inside Group2 which is inside Group1
```

Basically you should never call this function because Natron already pre-declares a variable for each node upon its creation. If you were to create a new node named “Blur1”, you could the access it in the Script Editor the following way:

```plaintext
app1.Blur1
```

**GroupParam**

Inherits Param

**Synopsis**

A group param is a container for other parameters. See *detailed* description.

**Functions**

- def *addParam* (param)
- def *getIsOpened* ()
- def *setAsTab* ()
- def *setOpened* (opened)

**Detailed Description**

A group param does not hold any relevant value. Rather this is a purely graphical element that is used to gather multiple parameters under a group. On the graphical interface a GroupParam looks like this:

When a Param is under a group, the *getParent()* will return the group as parent.
Member functions description

NatronEngine.GroupParam.addParam(param)

Parameters

param – Param

Adds param into the group.

Warning: Note that this function cannot be called on groups that are not user parameters (i.e: created either by script or by the “Manage user parameters” user interface)

Warning: Once called, you should call refreshUserParamsGUI() to update the user interface.

NatronEngine.GroupParam.getIsOpened()

Return type

bool

Returns whether the group is currently expanded (True) or folded (False).

NatronEngine.GroupParam.setAsTab()

Set this group as a tab. When set as a tab, it will be inserted into a special TabWidget of the Effect. For instance, on the following screenshot, to and from are 2 groups on which :func:`setAsTab()` has been called.

NatronEngine.GroupParam.setOpened(opened)

Parameters

opened – bool

Set this group to be expanded (opened = True) or folded (opened = False)

ImageLayer

Synopsis

A small object representing a layer of an image. For example, the base image layer is the color layer, or sometimes called “RGBA”. Some other default layers include ForwardMotion, BackwardMotion, DisparityLeft, DisparityRight, etc….

See detailed description...
Functions

- `def ImageLayer (layerName,componentsPrettyName,componentsName)
- `def isColorPlane ()
- `def getNumComponents ()
- `def getLayerName ()
- `def getComponentsNames ()
- `def getComponentsPrettyName ()
- `def getNoneComponents ()
- `def getRGBAComponents ()
- `def getRGBComponents ()
- `def getAlphaComponents ()
- `def getBackwardMotionComponents ()
- `def getForwardMotionComponents ()
- `def getDisparityLeftComponents ()
- `def getDisparityRightComponents ()

Detailed Description

A Layer is constituted of a layer name and a set of channel names (also called components). You can get a sequence with all the channels in the layer with the function `getComponentsNames ()`. For some default layers, the components may be represented by a prettier name for the end-user, such as `DisparityLeft` instead of XY. When the ImageLayer does not have a pretty name, its pretty name will just be a concatenation of all channel names in order.

There is one special layer in Natron: the color layer. It be represented as 3 different types: RGBA, RGB or Alpha. If the ImageLayer is a color layer, the method `isColorPlane ()` will return True

Member functions description

`NatronEngine.ImageLayer.ImageLayer (layerName, componentsPrettyName, componentsName)`

Parameters

- `layerName` - str
  
  Make a new image layer with the given layer name, optional components pretty name and the set of channels (also called components) in the layer.

Return type

- `bool`

  Returns True if this layer is a color layer, i.e: it is RGBA, RGB or alpha. The color layer is what is output by default by all nodes in Natron.

`NatronEngine.ImageLayer.getNumComponents ()`

Return type

- `int`

  Returns the number of channels in this layer. Can be between 0 and 4 included.

`NatronEngine.ImageLayer.getLayerName ()`

Return type

- `str`
Returns the layer name
NatronEngine.ImageLayer.getComponentsNames()

Return type Sequence
Returns a sequence with all channels in this layer in order

NatronEngine.ImageLayer.getComponentPrettyName()

Return type str
Returns the channels pretty name. E.g: DisparityLeft instead of XY

NatronEngine.ImageLayer.getNoneComponents()

Return type ImageLayer
Returns the default “none” layer

NatronEngine.ImageLayer.getRGBAComponents()

Return type ImageLayer
Returns the default “RGBA” layer

NatronEngine.ImageLayer.getRGBComponents()

Return type ImageLayer
Returns the default “RGB” layer

NatronEngine.ImageLayer.getAlphaComponents()

Return type ImageLayer
Returns the default “Alpha” layer

NatronEngine.ImageLayer.getBackwardMotionComponents()

Return type ImageLayer
Returns the default “Backward” layer

NatronEngine.ImageLayer.getForwardMotionComponents()

Return type ImageLayer
Returns the default “Forward” layer

NatronEngine.ImageLayer.getDisparityLeftComponents()

Return type ImageLayer
Returns the default “DisparityLeft” layer

NatronEngine.ImageLayer.getDisparityRightComponents()

Return type ImageLayer
Returns the default “DisparityRight” layer

Int2DParam

Inherits IntParam

Inherited by: Int3DParam
Synopsis

See IntParam for more details.

Functions

• def get ()
• def get (frame)
• def set (x, y)
• def set (x, y, frame)

Detailed Description

NatronEngine.Int2DParam.get ()

Return type
class Int2DTuple

Returns a Int2DTuple containing the [x,y] value of this parameter at the timeline’s current time.

NatronEngine.Int2DParam.get (frame)

param float

Return type
class Int2DTuple

Returns a Int2DTuple containing the [x,y] value of this parameter at the given frame.

NatronEngine.Int2DParam.set (x,y)

Parameters

• x – int
• y – int

Same as set (x) but for 2-dimensional integers.

NatronEngine.Int2DParam.set (x,y,frame)

Parameters

• x – int
• y – int
• frame – float

Same as set (x,frame) but for 2-dimensional integers.

Int2DTuple

Synopsis

Utility class to return pair of integers values.
Functions
  • def __getitem__(index)

Detailed Description

class NatronEngine.Int2DTuple
NatronEngine.Int2DTuple.x
NatronEngine.Int2DTuple.y
NatronEngine.Int2DTuple.__getitem__(index)

Parameters index – int

Return type PyObject

Returns the item at the given index. This is the bracket operator []

Int3DParam

Inherits Int2DParam

Synopsis

See IntParam for more details.

Functions
  • def set ()
  • def set (frame)
  • def set (x, y, z)
  • def set (x, y, z, frame)

Detailed Description

NatronEngine.Int3DParam.get ()

Return type <Int3DTuple>

Returns a Int3DTuple containing the [x,y,z] value of this parameter at the timeline’s current time.

NatronEngine.Int3DParam.get(frame)

param frame float

Return type <Int3DTuple>

Returns a Int3DTuple containing the [x,y,z] value of this parameter at the given frame

NatronEngine.Int3DParam.set (x, y, z)

Parameters
  • x – int
• y – int
• z – int

Same as `set(x)` but for 3-dimensional integers.

```python
NatronEngine.Int3DParam.set(x, y, z, frame)
```

**Parameters**

- x – int
- y – int
- z – int
- frame – float

Same as `set(x, frame)` but for 3-dimensional integers.

### Int3DTuple

#### Synopsis

Utility class to return pair of integers values.

#### Functions

- def `__getitem__(index)`

#### Detailed Description

```python
class NatronEngine.Int3DTuple
NatronEngine.Int3DTuple.x
NatronEngine.Int3DTuple.y
NatronEngine.Int3DTuple.z
NatronEngine.Int3DTuple.__getitem__(index)
```

**Parameters**

- index – PySide.QtCore.int

**Return type**

PyObject

Returns the item at the given index. This is the bracket operator `[]`

### IntParam

* Inherits* AnimatedParam

* Inherited by:* Int2DParam, Int3DParam

#### Synopsis

An IntParam can contain one or multiple int values. See detailed description...
Functions

- def `get()`
- def `get(frame)`
- def `getDefaultValue([dimension=0])`
- def `getDisplayMaximum(dimension)`
- def `getDisplayMinimum(dimension)`
- def `getMaximum([dimension=0])`
- def `getMinimum([dimension=0])`
- def `getValue([dimension=0])`
- def `getValueAtTime(time[, dimension=0])`
- def `restoreDefaultValue([dimension=0])`
- def `set(x)`
- def `set(x, frame)`
- def `setDefaultValue(value[, dimension=0])`
- def `setDisplayMaximum(maximum[, dimension=0])`
- def `setDisplayMinimum(minimum[, dimension=0])`
- def `setMaximum(maximum[, dimension=0])`
- def `setMinimum(minimum[, dimension=0])`
- def `setValue(value[, dimension=0])`
- def `setValueAtTime(value, time[, dimension=0])`

Detailed Description

An int param can have 1 to 3 dimensions. (See Int2DParam and Int3DParam). Usually this is used to represent a single integer value that may animate over time.

The user interface for them varies depending on the number of dimensions. Screenshots are the same than for the :doc:`DoubleParam` because the user interface is the same

A 1-dimensional DoubleParam

A 2-dimensional Double2Dparam

A 3-dimensional Double3DParam
Member functions description

NatronEngine.IntParam.get(frame)

Parameters frame – float

Return type int

Returns the value of this parameter at the given frame. If the animation curve has an animation (see getIsAnimated then the value will be interpolated using the interpolation chosen by the user for the curve.

NatronEngine.IntParam.get()

Return type int

Returns the value of this parameter at the given current timeline’s time.

NatronEngine.IntParam.getDefaultValue([dimension=0])

Parameters dimension – int

Return type int

Returns the default value for this parameter. dimension is meaningless for the IntParam class because it is 1-dimensional, but is useful for inherited classes Int2DParam and Int3DParam.

NatronEngine.IntParam.getDisplayMaximum(dimension)

Parameters dimension – int

Return type int

Returns the display maximum for this parameter at the given dimension. The display maximum is the maximum value visible on the slider, internally the value can exceed this range.

NatronEngine.IntParam.getDisplayMinimum(dimension)

Parameters dimension – int

Return type int

Returns the display minimum for this parameter at the given dimension. The display minimum is the minimum value visible on the slider, internally the value can exceed this range.

NatronEngine.IntParam.getMaximum([dimension=0])

Parameters dimension – int

Return type int

Returns the maximum for this parameter at the given dimension. The maximum value cannot be exceeded and any higher value will be clamped to this value.

NatronEngine.IntParam.getMinimum([dimension=0])

Parameters dimension – int

Return type int

Returns the minimum for this parameter at the given dimension. The minimum value cannot be exceeded and any lower value will be clamped to this value.

NatronEngine.IntParam.getValue([dimension=0])

Parameters dimension – int

Return type int

Returns the value of this parameter at the given dimension at the current timeline’s time.

NatronEngine.IntParam.getValueAtTime(time[, dimension=0])
Parameters

- **time** – float
- **dimension** – int

Return type int

Returns the value of this parameter at the given `dimension` at the given `time`.

If the animation curve has an animation (see `getIsAnimated`) then the value will be interpolated using the `interpolation` chosen by the user for the curve.

NatronEngine.IntParam.restoreDefaultValue([`dimension=0`])

Parameters **dimension** – int

Returns the value of this parameter at the given `dimension` at the given `time`.

NatronEngine.IntParam.set(x, frame)

Parameters

- **x** – int
- **frame** – float

Set a new keyframe on the parameter with the value `x` at the given `frame`.

NatronEngine.IntParam.set(x)

Parameters **x** – int

Set the value of this parameter to be `x`. If this parameter is animated (see `getIsAnimated(dimension)`) then this function will automatically add a keyframe at the timeline’s current time.

NatronEngine.IntParam.setDefaultValue(value[, `dimension=0`])

Parameters

- **value** – int
- **dimension** – int

Set the default `value` for this parameter at the given `dimension`.

NatronEngine.IntParamsetDisplayMaximum(maximum[, `dimension=0`])

Parameters

- **maximum** – int
- **dimension** – int

Set the display maximum of the parameter to be `maximum` for the given `dimension`. See `getDisplayMaximum`.

NatronEngine.IntParamsetDisplayMinimum(minimum[, `dimension=0`])

Parameters

- **minimum** – int
- **dimension** – int

Set the display minimum of the parameter to be `minimum` for the given `dimension`. See `getDisplayMinimum`.

NatronEngine.IntParam.setMaximum(maximum[, `dimension=0`])

Parameters

- **maximum** – int

NatronEngine.IntParam.setMaximum(maximum[, `dimension=0`])

Parameters

- **maximum** – int
• \textit{dimension} – int

Set the maximum of the parameter to be \textit{maximum} for the given \textit{dimension}. See \texttt{getMaximum}\nNatronEngine.IntParam.setMinimum(\texttt{minimum[, dimension=0]})

Parameters
  • \textit{minimum} – int
  • \textit{dimension} – int

Set the minimum of the parameter to be \textit{minimum} for the given \textit{dimension}. See \texttt{getMinimum}\nNatronEngine.IntParam.setValue(\texttt{value[, dimension=0]})

Parameters
  • \textit{value} – int
  • \textit{dimension} – int

Same as \texttt{set(value,dimension)}\nNatronEngine.IntParam.setValueAtTime(\texttt{value, time[, dimension=0]})

Parameters
  • \textit{value} – int
  • \textit{time} – float
  • \textit{dimension} – int

Same as \texttt{set(value,time,dimension)}

\textbf{ItemBase}

\textit{Inherited by: BezierCurve, Layer}

\textbf{Synopsis}

This is an abstract class that serves as a base class for both \texttt{Layer} and :doc:`BezierCurve`. See \textit{detailed} description...

\textbf{Functions}

• def \texttt{getLabel}()
  • def \texttt{getLocked}()
  • def \texttt{getLockedRecursive}()
  • def \texttt{getParentLayer}()
  • def \texttt{getParam}(name)
  • def \texttt{getScriptName}()
  • def \texttt{getVisible}()
  • def \texttt{setLabel}(name)
  • def \texttt{setLocked}(locked)
• def setScriptName (name)
• def setVisible (activated)

Detailed Description

This class gathers all common functions to both layers and beziers. An item has both a script-name and label. The script-name uniquely identifies an item within a roto node, while several items can have the same label.

Member functions description

NatronEngine.ItemBase.getLabel()

Return type str
Returns the label of the item, has visible in the table of the settings panel.

NatronEngine.ItemBase.getLocked()

Return type bool
Returns whether this item is locked or not. When locked the item is no longer editable by the user.

NatronEngine.ItemBase.getLockedRecursive()

Return type bool
Returns whether this item is locked or not. Unlike getLocked() this function looks parent layers recursively to find out if the item should be locked.

NatronEngine.ItemBase.getParentLayer()

Return type Layer
Returns the parent layer of the item. All items must have a parent layer, except the base layer.

NatronEngine.ItemBase.getParam (name)

Parameters name – str
Return type Param
Returns a parameter by its script-name or None if no such parameter exists.

NatronEngine.ItemBase.getScriptName()

Return type str
Returns the script-name of the item. The script-name is unique for each items in a roto node.

NatronEngine.ItemBase.getVisible()

Return type bool
Returns whether the item is visible or not. On the user interface, this corresponds to the small eye. When hidden, an item will no longer have its overlay painted on the viewer, but it will still render in the image.

NatronEngine.ItemBase.setLabel (name)

Parameters name – str
Set the item’s label.

NatronEngine.ItemBase.setLocked (locked)

Parameters locked – bool
Set whether the item should be locked or not. See getLocked().
NatronEngine.ItemBase.setScriptName(name)

Parameters name – str

Return type bool

Set the script-name of the item. You should never call it yourself as Natron chooses automatically a unique script-name for each item. However this function is made available for internal technicalities, but be aware that changing the script-name of an item can potentially break other scripts relying on it.

NatronEngine.ItemBase.setVisible(activated)

Parameters activated – bool

Set whether the item should be visible in the Viewer. See getVisible().

Layer

Inherits ItemBase

Synopsis

This class is used to group several shapes together and to organize them so they are rendered in a specific order. See detailed description...

Functions

• def addItem(item)
  • def getChildren()
  • def insertItem(pos, item)
  • def removeItem(item)

Detailed Description

Currently a layer acts only as a group so that you can organize shapes and control in which order they are rendered. To add a new item to the layer, use the addItem(item) or the insertItem(item) function.

To remove an item from the layer, use the removeItem(item) function.

Items in a layer are rendered from top to bottom, meaning the bottom-most items will always be drawn on top of other items.

Member functions description

NatronEngine.Layer.addItem(item)

Parameters item – ItemBase

Add a new item at the bottom of the layer.

NatronEngine.Layer.getChildren()

Return type sequence

Returns a sequence with all items in the layer.

NatronEngine.Layer.insertItem(pos, item)
Parameters

- `pos` - int
- `item` - ItemBase

Inserts a new item at the given `pos` (0 based index) in the layer. If `pos` is out of range, it will be inserted at the bottom of the layer.

NatronEngine.Layer.removeItem(item)

Parameters `item` - ItemBase

Removes the `item` from the layer.

Natron

Detailed Description

This class contains enumerations that are used by some functions of the API to return status that are more complicated than a simple boolean value.


Can have the following values:

- `eStandardButtonNoButton = 0x00000000`,
- `eStandardButtonEscape = 0x00000200`, // obsolete
- `eStandardButtonOk = 0x00000400`,
- `eStandardButtonSave = 0x00000800`,
- `eStandardButtonSaveAll = 0x00001000`,
- `eStandardButtonOpen = 0x00002000`,
- `eStandardButtonYes = 0x00004000`,
- `eStandardButtonYesToAll = 0x00008000`,
- `eStandardButtonNo = 0x00010000`,
- `eStandardButtonNoToAll = 0x00020000`,
- `eStandardButtonAbort = 0x00040000`,
- `eStandardButtonRetry = 0x00080000`,
- `eStandardButtonIgnore = 0x00100000`,
- `eStandardButtonClose = 0x00200000`,
- `eStandardButtonCancel = 0x00400000`,
- `eStandardButtonDiscard = 0x00800000`,
- `eStandardButtonHelp = 0x01000000`,
- `eStandardButtonApply = 0x02000000`,
- `eStandardButtonReset = 0x04000000`,
- `eStandardButtonRestoreDefaults = 0x08000000`
NatronEngine.Natron.ImageComponentsEnum
Can have the following values:
- eImageComponentNone = 0,
- eImageComponentAlpha,
- eImageComponentRGB,
- eImageComponentRGBA

NatronEngine.Natron.ImageBitDepthEnum
Can have the following values:
- eImageBitDepthNone = 0,
- eImageBitDepthByte,
- eImageBitDepthShort,
- eImageBitDepthFloat

NatronEngine.Natron.KeyframeTypeEnum
Can have the following values:
- eKeyframeTypeConstant = 0,
- eKeyframeTypeLinear = 1,
- eKeyframeTypeSmooth = 2,
- eKeyframeTypeCatmullRom = 3,
- eKeyframeTypeCubic = 4,
- eKeyframeTypeHorizontal = 5,
- eKeyframeTypeFree = 6,
- eKeyframeTypeBroken = 7,
- eKeyframeTypeNone = 8

NatronEngine.NatronValueChangedReasonEnum
Can have the following values:
- eValueChangedReasonUserEdited = 0,
  A user change to the param triggered the call, gui will not be refreshed but onParamChanged will be called
- eValueChangedReasonPluginEdited,
  A plugin change triggered the call, gui will be refreshed but onParamChanged not called
- eValueChangedReasonNatronGuiEdited,
  Natron gui called setValue itself, onParamChanged will be called (with a reason of User edited) AND param gui refreshed
- eValueChangedReasonNatronInternalEdited,
  Natron engine called setValue itself, onParamChanged will be called (with a reason of plugin edited) AND param gui refreshed
Natron Documentation, Release 2.0.0

- `eValueChangedReasonTimeChanged`, A time-line seek changed the call, called when timeline time changes
- `eValueChangedReasonSlaveRefresh`, A master parameter ordered the slave to refresh its value
- `eValueChangedReasonRestoreDefault`, The param value has been restored to its defaults

`NatronEngine.Natron.AnimationLevelEnum`
Can have the following values:
- `eAnimationLevelNone = 0`,
- `eAnimationLevelInterpolatedValue = 1`,
- `eAnimationLevelOnKeyframe = 2`

`NatronEngine.Natron.OrientationEnum`
Can have the following values:
- `eOrientationHorizontal = 0x1`,
- `eOrientationVertical = 0x2`

`NatronEngine.Natron.ImagePremultiplicationEnum`
Can have the following values:
- `eImagePremultiplicationOpaque = 0`,
- `eImagePremultiplicationPremultiplied`,
- `eImagePremultiplicationUnPremultiplied`,

`NatronEngine.Natron.StatusEnum`
Can have the following values:
- `eStatusOK = 0`,
- `eStatusFailed = 1`,
- `eStatusReplyDefault = 14`

`NatronEngine.Natron ViewerCompositingOperatorEnum`
Can have the following values:
- `eViewerCompositingOperatorNone`,
- `eViewerCompositingOperatorOver`,
- `eViewerCompositingOperatorMinus`,
- `eViewerCompositingOperatorUnder`,
- `eViewerCompositingOperatorWipe`

`NatronEngine.NatronPlaybackModeEnum`
Can have the following values:
- `ePlaybackModeLoop = 0`,
- `ePlaybackModeBounce`,

Chapter 1. Natron Python API reference
•ePlaybackModeOnce

NatronEngine.Natron.PixmapEnum
See here for potential values of this enumeration.

NatronEngine.Natron.ViewerColorSpaceEnum
Can have the following values:
•eViewerColorSpaceSRGB = 0,
•eViewerColorSpaceLinear,
•eViewerColorSpaceRec709

OutputFileParam

Inherits StringParamBase

Synopsis

This parameter is used to specify an output file

Functions

• def openFile()
• def setSequenceEnabled(enabled)

Member functions description

NatronEngine.OutputFileParam.openFile()
When called in GUI mode, this will open a file dialog for the user. Does nothing in background mode.

NatronEngine.OutputFileParam.setSequenceEnabled(enabled)

Parameters enabled — bool
Determines whether the file dialog opened by openFile() should have support for file sequences or not.

PageParam

Inherits Param

Synopsis

A page param is a container for other parameters. See detailed description.

Functions

• def addParam(param)
Detailed Description

A page param does not hold any relevant value. Rather this is a purely graphical element that is used to gather parameters under a tab. On the graphical interface a PageParam looks like this (e.g: the Controls tab of the panel)

![PageParam Example](image)

**Warning:** All parameters MUST be in a container, being a group or a page. If a Param is not added into any container, Natron will add it by default to the User page.

NatronEngine.PageParam.addParam(param)

**Parameters**

- param – Param

    Adds param into the page.

**Warning:** Note that this function cannot be called on pages that are not user parameters (i.e: created either by script or by the “Manage user parameters” user interface)

**Warning:** Once called, you should call refreshUserParamsGUI() to update the user interface.

Param

Synopsis

This is the base class for all parameters. Parameters are the controls found in the settings panel of a node. See details here.

Functions

- def copy (param[, dimension=-1])
- def curve (time[, dimension=-1])
- def getAddNewLine ()
- def getCanAnimate ()
- def getEvaluateOnChange ()
- def getHelp ()
- def getIsAnimationEnabled ()
- def getIsEnabled ([dimension=0])
- def getIsPersistant ()
- def getIsVisible ()
- def getLabel ()
- def getNumDimensions ()
- def getParent ()
- def getScriptName ()
- def getTypeName ()
- def randomNatronEngine.Param.random () ([min=0., max=1.])
- def randomNatronEngine.Param.random () (seed)
- def randomIntNatronEngine.Param.randomInt () (min, max)
- def randomIntNatronEngine.Param.randomInt () (seed)
- def setAddNewLine (a)
- def setAnimationEnabled (e)
- def setEnabled (enabled[, dimension=0])
- def setEnabledByDefault (enabled)
- def setEvaluateOnChange (eval)
- def setHelp (help)
- def setPersistant (persistant)
- def setVisible (visible)
- def setVisibleByDefault (visible)
- def setAsAlias (otherParam)
- def slaveTo (otherParam, thisDimension, otherDimension)
- def unslave (dimension)
**Detailed Description**

The Param object can be used to control a specific parameter of a node. There are different types of parameters, ranging from the single checkbox (boolean) to parametric curves. Each type of parameter has specific functions to control the parameter according to its internal value type. In this base class, all common functionalities for parameters have been gathered.

**Warning:** Note that since each child class has a different value type, all the functions to set/get values, and set/get keyframes are specific for each class.

A Param can have several functions to control some properties, namely:

- **addNewLine:** When True, the next parameter declared will be on the same line as this parameter
- **canAnimate:** This is a static property that you cannot control which tells whether animation can be enabled for a specific type of parameter
- **animationEnabled:** For all parameters that have canAnimate=True, this property controls whether this parameter should be able to animate (= have keyframes) or not
- **evaluateOnChange:** This property controls whether a new render should be issued when the value of this parameter changes
- **help:** This is the tooltip visible when hovering the parameter with the mouse
- **enabled:** Should this parameter be editable by the user or not. Generally, disabled parameters have their text in painted in black.
- **visible:** Should this parameter be visible in the user interface or not
- **persistent:** If true then the parameter value will be saved in the project
- **dimension:** How many dimensions this parameter has. For instance a `Double3DParam` has 3 dimensions. A `ParametricParam` has as many dimensions as there are curves.

Note that most of the functions in the API of Params take a `dimension` parameter. This is a 0-based index of the dimension on which to operate.

The following table sums up the different properties for all parameters including type-specific properties not listed above.

Note that most of the properties are not dynamic: they need to be set before calling `refreshUserParamsGUI()` which will create the GUI for these parameters.

**Warning:** A non-dynamic property can no longer be changed once `refreshUserParamsGUI()` has been called.

For non user-parameters (i.e: parameters that were defined by the underlying OpenFX plug-in), only their dynamic properties can be changed since `refreshUserParamsGUI()` will only refresh user parameters.

If a Setter function contains a (*) that means it can only be called for user parameters, it has no effect on already declared non-user parameters.
<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Dynamic</th>
<th>Setter</th>
<th>Getter</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>string</td>
<td>no</td>
<td>None</td>
<td>getScriptName</td>
<td>&quot;&quot;</td>
</tr>
<tr>
<td>label</td>
<td>string</td>
<td>no</td>
<td>None</td>
<td>getLabel</td>
<td>&quot;&quot;</td>
</tr>
<tr>
<td>help</td>
<td>string</td>
<td>yes</td>
<td>setHelp(*)</td>
<td>getHelp</td>
<td>&quot;&quot;</td>
</tr>
<tr>
<td>addNewLine</td>
<td>bool</td>
<td>no</td>
<td>setAddNewLine(*)</td>
<td>getAddNewLine</td>
<td>True</td>
</tr>
<tr>
<td>persistent</td>
<td>bool</td>
<td>yes</td>
<td>setPersistant(*)</td>
<td>getIsPersistant</td>
<td>True</td>
</tr>
<tr>
<td>evaluatesOnChange</td>
<td>bool</td>
<td>yes</td>
<td>setEvaluateOnChange(*)</td>
<td>getEvaluateOnChange</td>
<td>True</td>
</tr>
<tr>
<td>animates</td>
<td>bool</td>
<td>no</td>
<td>setAnimationEnabled(*)</td>
<td>getIsAnimationEnabled</td>
<td>See (1)</td>
</tr>
<tr>
<td>visible</td>
<td>bool</td>
<td>yes</td>
<td>setVisible</td>
<td>getIsVisible</td>
<td>True</td>
</tr>
<tr>
<td>enabled</td>
<td>bool</td>
<td>yes</td>
<td>setEnabled</td>
<td>getIsEnabled</td>
<td>True</td>
</tr>
</tbody>
</table>

Properties on IntParam, Int2DParam, Int3DParam, DoubleParam, Double2DParam, Double3DParam, ColorParam only:

<table>
<thead>
<tr>
<th>Property</th>
<th>Type</th>
<th>Dynamic</th>
<th>Setter</th>
<th>Getter</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>min</td>
<td>int/double</td>
<td>yes</td>
<td>setMinimum(*)</td>
<td>getMinimum</td>
<td>INT_MIN</td>
</tr>
<tr>
<td>max</td>
<td>int/double</td>
<td>yes</td>
<td>setMaximum(*)</td>
<td>getMaximum</td>
<td>INT_MAX</td>
</tr>
<tr>
<td>displayMin</td>
<td>int/double</td>
<td>yes</td>
<td>setDisplayMinimum(*)</td>
<td>getDisplayMinimum</td>
<td>INT_MIN</td>
</tr>
<tr>
<td>displayMax</td>
<td>int/double</td>
<td>yes</td>
<td>setDisplayMaximum(*)</td>
<td>getDisplayMaximum</td>
<td>INT_MAX</td>
</tr>
</tbody>
</table>

Properties on ChoiceParam only:

<table>
<thead>
<tr>
<th>Property</th>
<th>Type</th>
<th>Dynamic</th>
<th>Setter</th>
<th>Getter</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>options</td>
<td>list&lt;string&gt;</td>
<td>yes</td>
<td>setOptions/addOption(*)</td>
<td>getOption</td>
<td>empty list</td>
</tr>
</tbody>
</table>

Properties on FileParam, OutputFileParam only:

<table>
<thead>
<tr>
<th>Property</th>
<th>Type</th>
<th>Dynamic</th>
<th>Setter</th>
<th>Getter</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>sequenceDialog</td>
<td>bool</td>
<td>yes</td>
<td>setSequenceEnabled(*)</td>
<td>None</td>
<td>False</td>
</tr>
</tbody>
</table>

Properties on StringParam only:

<table>
<thead>
<tr>
<th>Property</th>
<th>Type</th>
<th>Dynamic</th>
<th>Setter</th>
<th>Getter</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>type</td>
<td>Type-Enum</td>
<td>no</td>
<td>setType(*)</td>
<td>None</td>
<td>eStringTypeDefault</td>
</tr>
</tbody>
</table>

Properties on PathParam only:

<table>
<thead>
<tr>
<th>Property</th>
<th>Type</th>
<th>Dynamic</th>
<th>Setter</th>
<th>Getter</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>multiPathTable</td>
<td>bool</td>
<td>no</td>
<td>setAsMultiPathTable(*)</td>
<td>None</td>
<td>False</td>
</tr>
</tbody>
</table>

Properties on GroupParam only:

<table>
<thead>
<tr>
<th>Property</th>
<th>Type</th>
<th>Dynamic</th>
<th>Setter</th>
<th>Getter</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>isTab</td>
<td>bool</td>
<td>no</td>
<td>setAsTab(*)</td>
<td>None</td>
<td>False</td>
</tr>
</tbody>
</table>

(1): animates is set to True by default only if it is one of the following parameters: IntParam Int2DParam Int3DParam DoubleParam Double2DParam Double3DParam ColorParam

Note that ParametricParam, GroupParam, PageParam, ButtonParam, FileParam, OutputFileParam, PathParam cannot animate at all.

**Member functions description**

NatronEngine.Param.copy**(other[, dimension=-1])**

**Parameters**

- **other** - Param
- **dimension** - int

**Return type** bool

Copies the other parameter values, animation and expressions at the given dimension. If dimension is -1, all dimensions in min(getNumDimensions()), other.getNumDimensions() will be copied.

**Note:** Note that types must be convertible:
IntParam, DoubleParam, ChoiceParam, ColorParam and BooleanParam can convert between types but StringParam cannot.

**Warning:** When copying a parameter, only values are copied, not properties, hence if copying a choice parameter, make sure that the value you copy has a meaning to the receiver otherwise you might end-up with an undefined behaviour, e.g:

If ChoiceParam1 has 3 entries and the current index is 2 and ChoiceParam2 has 15 entries and current index is 10, copying ChoiceParam2 to ChoiceParam1 will end-up in undefined behaviour.

This function returns **True** upon success and **False** otherwise.

NatronEngine.Param.curve(time[, dimension=-1])

**Parameters**
- time – float
- dimension – int

**Return type** float

If this parameter has an animation curve on the given *dimension*, then the value of that curve at the given *time* is returned. If the parameter has an expression on top of the animation curve, the expression will be ignored, i.e.g: the value of the animation curve will still be returned. This is useful to write custom expressions for motion design such as looping, reversing, etc...

NatronEngine.Param.getAddNewLine()

**Return type** bool

Returns whether the parameter is on a new line or not.

NatronEngine.Param.getCanAnimate()

**Return type** bool

Returns whether this class can have any animation or not. This cannot be changed. Calling `setAnimationEnabled(True)` will not enable animation for parameters that cannot animate.

NatronEngine.Param.getEvaluateOnChange()

**Return type** bool

Returns whether this parameter can evaluate on change. A parameter evaluating on change means that a new render will be triggered when its value changes due to a call of one of the setValue functions.

NatronEngine.Param.getHelp()

**Return type** str

Returns the help tooltip visible when hovering the parameter with the mouse on the GUI;

NatronEngine.Param.getIsAnimationEnabled()

**Return type** bool

Returns whether animation is enabled for this parameter. This is dynamic and can be changed by `setAnimationEnabled(bool)` if the parameter can animate.

NatronEngine.Param.getIsEnabled([dimension=0])

**Parameters**
- dimension – int

**Return type** bool
Returns whether the given dimension is enabled or not.
NatronEngine.Param.getIsPersistant()

   Return type  bool

Returns whether this parameter should be persistant in the project or not. Non-persistant parameter will not have their value saved when saving a project.
NatronEngine.Param.getIsVisible()

   Return type  bool

   Returns whether the parameter is visible on the user interface or not.
NatronEngine.Param.getLabel()

   Return type  str

Returns the label of the parameter. This is what is displayed in the settings panel of the node. See this section for an explanation of the difference between the label and the script name.
NatronEngine.Param.getNumDimensions()

   Return type  int

Returns the number of dimensions. For example a Double3DParam has 3 dimensions. A ParametricParam has as many dimensions as there are curves.
NatronEngine.Param.getParent()

   Return type  NatronEngine.Param

If this param is within a group, then the parent will be the group. Otherwise the param’s parent will be the page onto which the param appears in the settings panel.
NatronEngine.Param.getScriptName()

   Return type  str

Returns the script-name of the param as used internally. The script-name is visible in the tooltip of the parameter when hovering the mouse over it on the GUI. See this section for an explanation of the difference between the label and the script name.
NatronEngine.Param.getTypeName()

   Return type  str

Returns the type-name of the parameter.
NatronEngine.Param.random([min=0., max=1.])

   Parameters
     •  min – float
     •  max – float

   Return type  float

Returns a pseudo-random value in the interval [min, max]. The value is produced such that for a given parameter it will always be the same for a given time on the timeline, so that the value can be reproduced exactly.

Note: Note that if you’re calling multiple times random() in the same parameter expression, each call would return a different value, but they would all return the same value again if the expressions is interpreted at the same time, e.g:
# Would always return the same value at a given timeline’s time.
random() - random()

Note that you can ensure that random() returns a given value by calling the overloaded function `random(seed)` instead.

NatronEngine.Param.random(seed)

Parameters
- **seed** – unsigned int

Return type float

Same as `random()` but takes a `seed` in parameter to control the value returned by the function. E.g:

ret = random(2) - random(2)
# ret == 0 always

NatronEngine.Param.randomInt(min, max)

Parameters
- **min** – int
- **max** – int

Return type int

Same as `random(min, max)` but returns an integer in the range `[min, max]`

NatronEngine.Param.randomInt(seed)

Parameters
- **seed** – unsigned int

Return type int

Same as `random(seed)` but returns an integer in the range `[0, INT_MAX]` instead.

NatronEngine.Param.setAddNewLine(a)

Parameters
- **a** – bool

Set whether the parameter should be on a new line or not. See `getAddNewLine()`

NatronEngine.Param.setAnimationEnabled(e)

Parameters
- **e** – bool

Set whether animation should be enabled (= can have keyframes). See `getIsAnimationEnabled()`

NatronEngine.Param.setEnabled(enabled[, dimension=0])

Parameters
- **enabled** – bool
- **dimension** – int

Set whether the given `dimension` of the parameter should be enabled or not. When disabled, the parameter will be displayed in black and the user will not be able to edit it. See `getIsEnabled(dimension)`

NatronEngine.Param.setEnabledByDefault(enabled)

Parameters
- **enabled** – bool

Set whether the parameter should be enabled or not by default. When disabled, the parameter will be displayed in black and the user will not be able to edit it.

NatronEngine.Param.setEvaluateOnChange(eval)
**Parameters**

`eval` - bool

Set whether evaluation should be enabled for this parameter. When True, calling any function that change the value of the parameter will trigger a new render. See `getEvaluateOnChange()`

NatronEngine.Param.setHelp(help)

**Parameters**

`help` - str

Set the help tooltip of the parameter. See `getHelp()`

NatronEngine.Param.setPersistant(persistent)

**Parameters**

`persistent` - bool

Set whether this parameter should be persistent or not. Non persistent parameter will not be saved in the project. See `getIsPersistant`

NatronEngine.Param.setVisible(visible)

**Parameters**

`visible` - bool

Set whether this parameter should be visible or not to the user. See `getIsVisible()`

NatronEngine.Param.setVisibleByDefault(visible)

**Parameters**

`visible` - bool

Set whether this parameter should be visible or not to the user in its default state.

NatronEngine.Param.setAsAlias(otherParam)

**Parameters**

`otherParam` - Param

**Return type**

bool

Set this parameter as an alias of `otherParam`. They need to be both of the same `type` and of the same `dimension`. This parameter will control `otherParam` entirely and in case of a choice param, its drop-down menu will be updated whenever the `otherParam` menu is updated.

This is used generally to make user parameters on groups with the “Pick” option of the “Manage User Parameters” dialog.

NatronEngine.Param.slaveTo(otherParam, thisDimension, otherDimension)

**Parameters**

- `otherParam` - Param
- `thisDimension` - int
- `otherDimension` - int

**Return type**

:class:`bool<PySide.QtCore.bool>`

Set this parameter as a slave of `otherParam`. They need to be both of the same `type` but may vary in dimension, as long as `thisDimension` is valid according to the number of dimensions of this parameter and `otherDimension` is valid according to the number of dimensions of `otherParam`.

This parameter `thisDimension` will be controlled entirely by the `otherDimension` of `otherParam` until a call to `unslave(thisDimension)` is made.

NatronEngine.Param.unslave(dimension)

**Parameters**

`dimension` - int
If the given *dimension* of this parameter was previously slaved, then this function will remove the link between parameters, and the user will be free again to use this parameter as any other.

**Note:** The animation and values that were present before the link will remain.

---

**ParametricParam**

*Inherits* Param

**Synopsis**

A parametric param represents one or more parametric functions as curves. See *detailed* explanation below.

**Functions**

- def *addControlPoint* (dimension, key, value)
- def *deleteAllControlPoints* (dimension)
- def *deleteControlPoint* (dimension, nthCtl)
- def *getCurveColor* (dimension)
- def *getNControlPoints* (dimension)
- def *getNthControlPoint* (dimension, nthCtl)
- def *getValue* (dimension, parametricPosition)
- def *setCurveColor* (dimension, r, g, b)
- def *setNthControlPoint* (dimension, nthCtl, key, value, leftDerivative, rightDerivative)

**Detailed Description**

A parametric parameter has as many dimensions as there are curves. Currently the number of curves is static and you may only specify the number of curves via the *nbCurves* argument of the `createParametricParam(name, label, nbCurves)` function.

Parametric curves work almost the same way that animation curves do: you can add control points and remove them. You can peak the value of the curve at a special *parametric position* with the `getValue(dimension, parametricPosition)` function. The *parametric position* is represented by the X axis on the graphical user interface.

**Member functions description**

NatronEngine.ParametricParam. *addControlPoint* (dimension, key, value)

**Parameters**

- *dimension* – int
- *key* – float
- *value* – float

**Return type** StatusEnum
Attempts to add a new control point to the curve at the given `dimension`. The new point will have the coordinate (key,value). This function returns a NatronEngine.Natron.StatusEnum.eStatusOK upon success, otherwise NatronEngine.Natron.StatusEnum.eStatusFailed is returned upon failure.

NatronEngine.ParametricParam.deleteAllControlPoints

**Parameters**

- `dimension` - int

**Return type** StatusEnum

Removes all control points of the curve at the given `dimension`. This function returns a NatronEngine.Natron.StatusEnum.eStatusOK upon success, otherwise NatronEngine.Natron.StatusEnum.eStatusFailed is returned upon failure.

NatronEngine.ParametricParam.deleteControlPoint

**Parameters**

- `dimension` - int
- `nthCtl` - int

**Return type** StatusEnum

Attempts to remove the `nth` control point (sorted in increasing X order) of the parametric curve at the given `dimension`. This function returns a NatronEngine.Natron.StatusEnum.eStatusOK upon success, otherwise NatronEngine.Natron.StatusEnum.eStatusFailed is returned upon failure.

NatronEngine.ParametricParam.getCurveColor

**Parameters**

- `dimension` - ColorTuple

**Return type** StatusEnum

Returns a ColorTuple with the [R,G,B] color of the parametric curve at the given `dimension` on the graphical user interface.

NatronEngine.ParametricParam.getNControlPoints

**Parameters**

- `dimension` - int

**Return type** int

Returns the number of control points of the curve at the given `dimension`.

NatronEngine.ParametricParam.getNthControlPoint

**Parameters**

- `dimension` - int
- `nthCtl` - int

**Return type** tuple

Returns a tuple containing informations about the `nth` control point (sorted by increasing X order) control point of the curve at the given `dimension`. The tuple is composed of 5 members:

- `status`: StatusEnum, `key`: float, `value`: float, `left derivative`: float, `right derivative`: float

This function returns in the status a NatronEngine.Natron.StatusEnum.eStatusOK upon success, otherwise NatronEngine.Natron.StatusEnum.eStatusFailed is returned upon failure.

NatronEngine.ParametricParam.getValue

**Parameters**

- `dimension` - int
- `parametricPosition` - double
Return type `double`

Returns the Y value of the curve at the given `parametricPosition` (on the X axis) of the curve at the given `dimension`.

`NatronEngine.ParametricParam.setCurveColor(dimension, r, g, b)`

**Parameters**

- `dimension` – int
- `r` – float
- `g` – float
- `b` – float

Set the color of the curve at the given `dimension`.

`NatronEngine.ParametricParam.setNthControlPoint(dimension, nthCtl, key, value, leftDerivative, rightDerivative)`

**Parameters**

- `dimension` – int
- `nthCtl` – int
- `key` – float
- `value` – float
- `leftDerivative` – float
- `rightDerivative` – float

**Return type** `StatusEnum`

Set the value of an existing control point on the curve at the given `dimension`. The `nthCtl` parameter is the (zero based) index of the control point (by increasing X order). The point will be placed at the coordinates defined by (key,value) and will have the derivatives given by `leftDerivative` and `rightDerivative`.


**PathParam**

**Inherits** `StringParamBase`

**Synopsis**

A path param is used to indicate the path to a directory. See `details...`

**Functions**

- `def setAsMultiPathTable()`
By default the user can select a single directory as path, unless `setAsMultiPathTable()` is called in which case a table is presented to the user to specify multiple directories like this:

When using multiple paths, internally they are separated by a `;` and the following characters are escaped as per the XML specification:

- `<` becomes `<
`- `>` becomes `>
`- `&` becomes `&`
- `"` becomes `"`
- `'` becomes `'

Some more characters are escaped, you can see the full function in the source code of Natron [here](https://example.com).

**Member functions description**

`NatronEnginePathParam.setAsMultiPathTable()`  
When called, the parameter will be able to store multiple paths.

**PyCoreApplication**

Inherited by: `PyGuiApplication`

**Synopsis**

This object represents a background instance of Natron. See [detailed description...](https://example.com).

**Functions**

- `def appendToNatronPath(path)`
- `def getSettings()`
• def getBuildNumber ()
• def getInstance (idx)
• def getNatronDevelopmentStatus ()
• def getNatronPath ()
• def getNatronVersionEncoded ()
• def getNatronVersionMajor ()
• def getNatronVersionMinor ()
• def getNatronVersionRevision ()
• def getNatronVersionString ()
• def getNumCpus ()
• def getNumInstances ()
• def getPluginIDs ()
• def getPluginIDs (filter)
• def isBackground ()
• def is64Bit ()
• def isLinux ()
• def isMacOSX ()
• def isUnix ()
• def isWindows ()
• def setOnProjectCreatedCallback (pythonFunctionName)
• def setOnProjectLoadedCallback (pythonFunctionName)

**Detailed Description**

When running Natron there’s a **unique** instance of the PyCoreApplication object. It holds general informations about the process.

Generally, throughout your scripts, you can access this object with the variable `natron` that Natron pre-declared for you, e.g:

```
natron.getPluginIDs()
```

<table>
<thead>
<tr>
<th>Warning: The variable <code>natron</code> belongs to the module <code>NatronEngine</code>, hence make sure to make the following import:</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>from NatronEngine import *</code></td>
</tr>
</tbody>
</table>

Otherwise with a regular `import` you can still access `natron` by prepending the module:

```
NatronEngine.natron
```
Warning: The variable stored in the module NatronEngine contains a reference to a PyCoreApplication. If you need to have the GUI functionalities provided by PyGuiApplication, you must then use the variable natron belonging to the module NatronGui. Hence make sure to make the following import to have access to natron:

```
from NatronGui import *
```

With a regular import you can access it using NatronGui.natron.

Warning: Make sure to not make the 2 following imports, otherwise the natron variable will not point to something expected:

```
#This you should not do!
from NatronEngine import *
from NatronGui import *

#This is OK
import NatronEngine
import NatronGui

#This can also be done for convenience
from NatronEngine import NatronEngine.natron as NE
from NatronGui import NatronGui.natron as NG
```

This class is used only for background (command-line) runs of Natron, that is when you launch Natron in the following ways:

```
Natron -b ...
Natron -t
NatronRenderer
```

For interactive runs of Natron (with the user interface displayed), the derived class PyGuiApplication is used instead, which gives access to more GUI specific functionalities.

You should never need to make a new instance of this object yourself. Note that even if you did, internally the same object will be used and they will all refer to the same Natron application.

In GUI mode, a :doc:`PyGuiApplication` can have several projects opened. For each project you can refer to them with pre-declared variables `app1`, `app2`, etc...

In background mode, there would be only 1 project opened, so Natron does the following assignment for you before calling any scripts:

```
app = app1
```

See `App` to access different opened projects.

**Member functions description**

**class** NatronEngine.PyCoreApplication

Defines a new variable pointing to the same underlying application that the natron variable points to. This is equivalent to calling:

```
myVar = natron
```

NatronEngine.PyCoreApplication.appendToNatronPath(path)

**Parameters** path – str

Adds a new path to the Natron search paths. See this section for a detailed explanation of Natron search paths.
NatronEngine.PyCoreApplication.getSettings()

Return type: AppSettings

Returns an object containing all Natron settings. The settings are what can be found in the preferences of Natron.

NatronEngine.PyCoreApplication.getBuildNumber()

Return type: int

Returns the build-number of the current version of Natron. Generally this is used for release candidates, e.g:

<table>
<thead>
<tr>
<th>Natron version</th>
<th>Build number</th>
</tr>
</thead>
<tbody>
<tr>
<td>Natron v1.0.0-RC1</td>
<td>1</td>
</tr>
<tr>
<td>Natron v1.0.0-RC2</td>
<td>2</td>
</tr>
<tr>
<td>Natron v1.0.0-RC3</td>
<td>3</td>
</tr>
</tbody>
</table>

NatronEngine.PyCoreApplication.getInstance(idx)

Parameters: idx - int

Return type: App

Returns the App instance at the given idx. Note that idx is 0-based, e.g: 0 would return what’s pointed to by app1.

NatronEngine.PyCoreApplication.getNatronDevelopmentStatus()

Return type: str

Returns a string describing the development status of Natron. This can be one of the following values:

- **Alpha**: Meaning the software has unimplemented functionalities and probably many bugs left
- **Beta**: Meaning the software has all features that were planned are implemented but there may be bugs
- **RC**: Meaning the software seems in a good shape and should be ready for release unless some last minute show-stoppers are found
- **Release**: Meaning the software is ready for production

NatronEngine.PyCoreApplication.getNatronPath()

Return type: sequence

Returns a sequence of string with all natron search paths.

NatronEngine.PyCoreApplication.getNatronVersionEncoded()

Return type: int

Returns an int with the version of Natron encoded so that you can compare versions of Natron like this:

```python
if natron.getNatronVersionEncoded() >= 20101:
    ...
```

In that example, Natron’s version would be 2.1.1

NatronEngine.PyCoreApplication.getNatronVersionMajor()

Return type: int

Returns the major version of Natron. If the version is 1.0.0, that would return 1.

NatronEngine.PyCoreApplication.getNatronVersionMinor()

Return type: int

Get the minor version of Natron. If the version is 1.2.0, that would return 2.
Return type int
Returns the revision number of the version. If the version is 1.2.3, that would return 3.

NatronEngine.PyCoreApplication.getNatronVersionString()

Return type str
Returns the version of Natron as a string, e.g.: “1.1.0”

NatronEngine.PyCoreApplication.getNumCpus()

Return type int
Returns the maximum hardware concurrency of the computer. If the computer has 8 hyper-threaded cores, that would return 16.

NatronEngine.PyCoreApplication.getNumInstances()

Return type sequence
Returns the number of :doc:`App` instances currently active.

NatronEngine.PyCoreApplication.getPluginIDs()

Return type sequence
Returns a sequence of strings with all plugin-IDs currently loaded.

NatronEngine.PyCoreApplication.getPluginIDs(filter)

param filter str

Return type sequence
Same as getPluginIDs() but returns only plug-ins containing the given filter. Comparison is done without case-sensitivity.

NatronEngine.PyCoreApplication.isBackground()

Return type bool
Returns True if Natron is executed in background mode, i.e: from the command-line, without any graphical user interface displayed.

NatronEngine.PyCoreApplication.is64Bit()

Return type bool
Returns True if Natron is executed on a 64 bit computer.

NatronEngine.PyCoreApplication.isLinux()

Return type bool
Returns True if Natron is executed on a Linux or FreeBSD distribution.

NatronEngine.PyCoreApplication.isMacOSX()

Return type bool
Returns True if Natron is executed on MacOSX.

NatronEngine.PyCoreApplication.isUnix()

Return type bool
Returns True if Natron is executed on Unix. Basically this is equivalent to:
if natron.isLinux() or natron.isMacOSX():

NatronEngine.PyCoreApplication.isWindows()

Return type  bool

Returns True if Natron is executed on Windows.

NatronEngine.PyCoreApplication.setOnProjectCreatedCallback(pythonFunctionName)

Param  str

Convenience function to set the After Project Created callback. Note that this will override any callback set in the Preferences->Python->After Project created. This is exactly the same as calling:

NatronEngine.settings.afterProjectCreated.set(pythonFunctionName)

Note:  Clever use of this function can be made in the init.py script to do generic stuff for all projects (whether they are new projects or loaded projects). For instance one might want to add a list of Formats to the project. See the example here

NatronEngine.PyCoreApplication.setOnProjectLoadedCallback(pythonFunctionName)

Param  str

Convenience function to set the Default After Project Loaded callback. Note that this will override any callback set in the Preferences->Python->Default After Project Loaded. This is exactly the same as calling:

NatronEngine.settings.defOnProjectLoaded.set(pythonFunctionName)

RectD

Synopsis

A rectangle defined with floating point precision. See detailed description below

Functions

• def area ()
• def bottom ()
• def clear ()
• def contains (otherRect)
• def height ()
• def intersect (otherRect)
• def intersects (otherRect)
• def isInfinite ()
• def isNull ()
• def left ()
• def merge (otherRect)
• def right ()
• def `set `(x1,y1,x2,y2)
• def `set_bottom `(y1)
• def `set_left `(x1)
• def `set_right `(x2)
• def `set_top `(y2)
• def `top`()
• def `translate `(dx,dy)
• def `width`()

**Detailed Description**

A rectangle where x1 < x2 and y1 < y2 such as width() == (x2 - x1) && height() == (y2 - y1) (x1,y1) is the coordinates of the bottom left corner of the rectangle. The last element valid in the y dimension is y2 - 1 and the last valid in the x dimension is x2 - 1. x1,x2,y1 and y2 are with floating point precision.

**Member functions description**

NatronEngine.RectD.`area()`

**Return type** double

Returns the area covered by the rectangle, that is: (y2 - y1) * (x2 - x1)

NatronEngine.RectD.`bottom()`

**Return type** double

Returns the bottom edge, that is the

NatronEngine.RectD.`clear()`

Same as set (0,0,0,0)

NatronEngine.RectD.`contains`(otherRect)

**param otherRect** RectD

**Return type** bool

Returns True if otherRect is contained in or equals this rectangle, that is if:

- otherRect.x1 >= x1 and
- otherRect.y1 >= y1 and
- otherRect.x2 <= x2 and
- otherRect.y2 <= y2

NatronEngine.RectD.`height()`

**Return type** double

Returns the height of the rectangle, that is: y2 - y1

NatronEngine.RectD.`intersect`(otherRect)

**param otherRect** RectD

**Return type** RectD
Returns the intersection between this rectangle and \textit{otherRect}. If the intersection is empty, the return value will have the \texttt{isNull()} function return True.

\begin{verbatim}
NatronEngine.RectD.intersects \texttt{(otherRect)}
    \textbf{param} \texttt{otherRect} RectD

\textbf{Return type} \texttt{bool}
\end{verbatim}

Returns True if rectangle and \textit{otherRect} intersect.

\begin{verbatim}
NatronEngine.RectD.isInfinite()

\textbf{Return type} \texttt{bool}
\end{verbatim}

Returns True if this rectangle is considered to cover an infinite area. Some generator effects use this to indicate that they can potentially generate an image of infinite size.

\begin{verbatim}
NatronEngine.RectD.isNull()

\textbf{Return type} \texttt{bool}
\end{verbatim}

Returns true if \texttt{x2} \texttt{<=} \texttt{x1} or \texttt{y2} \texttt{<=} \texttt{y1}

\begin{verbatim}
NatronEngine.RectD.left()

\textbf{Return type} \texttt{double}
\end{verbatim}

Returns \texttt{x1}, that is the position of the left edge of the rectangle.

\begin{verbatim}
NatronEngine.RectD.merge \texttt{(otherRect)}
    \textbf{Parameters} \texttt{otherRect} – RectD

Unions this rectangle with \textit{otherRect}. In other words, this rectangle becomes the bounding box of this rectangle and \textit{otherRect}.

\begin{verbatim}
NatronEngine.RectD.left()

\textbf{Return type} \texttt{double}
\end{verbatim}

Returns \texttt{x1}, that is the position of the left edge of the rectangle.

\begin{verbatim}
NatronEngine.RectD.right()

\textbf{Return type} \texttt{double}
\end{verbatim}

Returns \texttt{x2}, that is the position of the right edge of the rectangle. \texttt{x2} is considered to be the first element outside the rectangle.

\begin{verbatim}
NatronEngine.RectD.set \texttt{(x1,y1,x2,y2)}
    \textbf{Parameters}
        • \texttt{x1} – \texttt{double}
        • \texttt{y1} – \texttt{double}
        • \texttt{x2} – \texttt{double}
        • \texttt{y2} – \texttt{double}

Set the \texttt{x1}, \texttt{y1}, \texttt{x2}, \texttt{y2} coordinates of this rectangle.

\begin{verbatim}
NatronEngine.RectD.set_bottom \texttt{(y1)}
    \textbf{Parameters} \texttt{y1} – \texttt{double}

Set \texttt{y1}
NatronEngine.RectD.set_left(x1)

Parameters y1 – double

Set x1

NatronEngine.RectD.set_right(x2)

Parameters x2 – double

Set x2

NatronEngine.RectD.set_top(y2)

Parameters y2 – double

Set y2

NatronEngine.RectD.top()

Return type double

Returns y2, that is the position of the top edge of the rectangle. y2 is considered to be the first element outside the rectangle.

NatronEngine.RectD.translate(dx, dy)

Parameters
  • dx – double
  • dy – double

Moves all edges of the rectangle by dx, dy, that is:

\[
\begin{align*}
x1 & += \text{dx}; \\
y1 & += \text{dy}; \\
x2 & += \text{dx}; \\
y2 & += \text{dy};
\end{align*}
\]

NatronEngine.RectD.width()

Return type double

Returns the width of the rectangle, that is \(x2 - x1\).

**RectI**

**Synopsis**

A rectangle defined with integer precision. See detailed description below

**Functions**

- def bottom()
- def clear()
- def contains(otherRect)
- def height()
- def intersect(otherRect)
- def intersects(otherRect)
• def `isInfinite`()
• def `isNull`()
• def `left`()
• def `merge` (otherRect)
• def `right`()
• def `set` (x1,y1,x2,y2)
• def `set_bottom` (y1)
• def `set_left` (x1)
• def `set_right` (x2)
• def `set_top` (y2)
• def `top`()
• def `translate` (dx,dy)
• def `width`()

**Detailed Description**

A rectangle where \( x_1 < x_2 \) and \( y_1 < y_2 \) such as \( \text{width}() = (x_2 - x_1) \) && \( \text{height}() = (y_2 - y_1) \) \((x_1,y_1)\) is are the coordinates of the bottom left corner of the rectangle. The last element valid in the y dimension is \( y_2 - 1 \) and the last valid in the x dimension is \( x_2 - 1 \). \( x_1, x_2, y_1 \) and \( y_2 \) are with integer precision.

**Member functions description**

`NatronEngine.RectI.bottom()`

Return type: `int`

Returns the bottom edge, that is the

`NatronEngine.RectI.clear()`

Same as set \((0,0,0,0)\)

`NatronEngine.RectI.contains(otherRect)`

param `otherRect` `RectI`

Return type: `bool`

Returns True if `otherRect` is contained in or equals this rectangle, that is if:

\[
\text{otherRect.x1} \geq x_1 \text{ and } \\
\text{otherRect.y1} \geq y_1 \text{ and } \\
\text{otherRect.x2} \leq x_2 \text{ and } \\
\text{otherRect.y2} \leq y_2
\]

`NatronEngine.RectI.height()`

Return type: `int`

Returns the height of the rectangle, that is: \( y_2 - y_1 \)

`NatronEngine.RectI.intersect(otherRect)`

param `otherRect` `RectI`
Return type RectI

Returns the intersection between this rectangle and otherRect. If the intersection is empty, the return value will have the isNull() function return True.

NatronEngine.RectI.intersects(otherRect)

   param otherRect RectI

Return type bool

Returns True if rectangle and otherRect intersect.

NatronEngine.RectI.isInfinite()

Return type bool

Returns True if this rectangle is considered to cover an infinite area. Some generator effects use this to indicate that they can potentially generate an image of infinite size.

NatronEngine.RectI.isNull()

Return type bool

Returns true if x2 <= x1 or y2 <= y1

NatronEngine.RectI.left()

Return type int

Returns x1, that is the position of the left edge of the rectangle.

NatronEngine.RectI.merge(otherRect)

Parameters otherRect – RectI

Unions this rectangle with otherRect. In other words, this rectangle becomes the bounding box of this rectangle and otherRect.

NatronEngine.RectI.right()

Return type int

Returns x1, that is the position of the left edge of the rectangle.

NatronEngine.RectI.set(x1, y1, x2, y2)

Parameters

   • x1 – int
   • y1 – int
   • x2 – int
   • y2 – int

Set the x1, y1, x2, y2 coordinates of this rectangle.

NatronEngine.RectI.set_top(y1)
Parameters \( y_1 \) – int

Set \( y_1 \)

NatronEngine.RectI.set_left(\( x_1 \))

Parameters \( y_1 \) – int

Set \( x_1 \)

NatronEngine.RectI.set_right(\( x_2 \))

Parameters \( x_2 \) – int

Set \( x_2 \)

NatronEngine.RectI.set_top(\( y_2 \))

Parameters \( y_2 \) – int

Set \( y_2 \)

NatronEngine.RectI.top()

Return type int

Returns \( y_2 \), that is the position of the top edge of the rectangle. \( y_2 \) is considered to be the first element outside the rectangle.

NatronEngine.RectI.translate(\( dx \), \( dy \))

Parameters

\* \( dx \) – int
\* \( dy \) – int

Moves all edges of the rectangle by \( dx \), \( dy \), that is:

\[
\begin{align*}
  x_1 & += dx; \\
  y_1 & += dy; \\
  x_2 & += dx; \\
  y_2 & += dy;
\end{align*}
\]

NatronEngine.RectI.width()

Return type int

Returns the width of the rectangle, that is \( x_2 - x_1 \).

Roto

Synopsis

This class encapsulates all things related to the roto node. See detailed description below.

Functions

\* def createBezier(\( x \), \( y \), time)
\* def createEllipse(\( x \), \( y \), diameter, fromCenter, time)
\* def createLayer()
\* def createRectangle(\( x \), \( y \), size, time)
Detailed Description

The Roto class is uses for now in Natron exclusively by the roto node, but its functionalities could be re-used for other nodes as well. Its purpose is to manage all layers and shapes. You can create new shapes with the `createBezier(x, y, time)`, `createEllipse(x, y, diameter, fromCenter, time)` and `createRectangle(x, y, size, time)` functions.

To create a new Layer you can use the `createLayer()` function.

As for other `auto-declared` variables, all shapes in the Roto objects can be accessed by their script-name, e.g:

```
Roto1.roto.Layer1.Bezier1
```

Member functions description

**NatronEngine.Roto.createBezier(x, y, time)**

**Parameters**
- `x` – float
- `y` – float
- `time` – int

**Return type** BezierCurve

Creates a new BezierCurve with one control point at position (x,y) and a keyframe at the given `time`.

**NatronEngine.Roto.createEllipse(x, y, diameter, fromCenter, time)**

**Parameters**
- `x` – float
- `y` – float
- `diameter` – float
- `fromCenter` – bool
- `time` – int

**Return type** BezierCurve

Creates a new ellipse. This is a convenience function that uses `createBezier(x, y, time)` to create a new BezierCurve and then adds 3 other control points to the Bezier so that it forms an ellipse of the given `diameter`. A new keyframe will be set at the given `time`. If `fromCenter` is true, then (x,y) is understood to be the coordinates of the center of the ellipse, otherwise (x,y) is understood to be the position of the top-left point of the smallest enclosing rectangle of the ellipse.

**NatronEngine.Roto.createLayer()**

**Return type** Layer

Creates a new layer.

**NatronEngine.Roto.createRectangle(x, y, size, time)**

**Parameters**
- `x` – float
• y - float
• size - float
• time - int

Return type BezierCurve

Creates a new rectangle. This is a convenience function that uses `createBezier(x, y, time)` to create a new `BezierCurve` and then adds 3 other control points to the Bezier so that it forms a rectangle of the given size on each of its sides. A new keyframe will be set at the given time.

```python
NatronEngine.Roto.getBaseLayer()
```

Return type Layer

Convenience function to access to the base Layer. Note that all shapes should belong to a Layer, the base layer being the top-level parent of all the hierarchy.

```python
NatronEngine.Roto.getItemByName(name)
```

Parameters

- name - str

Return type ItemBase

Returns an item by its script-name. See this section for the details of what is the script-name of an item. E.g:

```python
appl.Roto1.roto.Layer1.Bezier1 = appl.Roto1.roto.getItemByName("Bezier1")
```

### StringParam

Inherits StringParamBase

**Synopsis**

This parameter is used to contain a string. See here for more details.

**Functions**

- def `setType` (type)

**Detailed Description**

A StringParam can have several forms on the user interface, depending on its type

Here are the different types of string parameters:

- A basic string that can be edited by the user
- A non animating label string that the user cannot edit
- A multi-line string that the user can edit and animate
- A multi-line string with rich text support with a subset of html
Input and output informations, press Refresh to update them with current values

Enter text

Label:

Times New Roman
Member functions description
NatronEngine.StringParam \texttt{.\textcolor{blue}{setType}(type)}

\textbf{Parameters} \texttt{type} -- NatronEngine.StringParam.TypeEnum

Set the type of the StringParam. This should be called right away after creation time.

\textbf{Warning:} Once called, you should call \texttt{refreshUserParamsGUI()} to update the user interface.

\section*{StringParamBase}

\textbf{Inherits} AnimatedParam

\textbf{Inherited by:} PathParam, OutputFileParam, FileParam, StringParam

\section*{Synopsis}

This is the base-class for all parameters holding a string. See \textit{here} for more details.

\section*{Functions}

- \texttt{def get ()}
- \texttt{def get(frame)}
- \texttt{def getDefaultValue ()}
- \texttt{def getValue ()}
- \texttt{def getValueAtTime(time)}
- \texttt{def restoreDefaultValue ()}
- \texttt{def set(x)}
- \texttt{def set(x, frame)}
- \texttt{def setDefaultValue(value)}
- \texttt{def setValue(value)}
- \texttt{def setValueAtTime(value, time)}

\section*{Detailed Description}

A string parameter contains internally a string which can change over time. Much like keyframes for value parameters (like IntParam or DoubleParam) keyframes can be set on string params, though the interpolation will remain constant always.

```
Member functions description
NatronEngine.StringParamBase \texttt{.\textcolor{blue}{get}()}

\textbf{Return type} \texttt{str}

Get the value of the parameter at the current timeline’s time

NatronEngine.StringParamBase \texttt{.\textcolor{blue}{get}(frame)}

\textbf{Parameters} \texttt{frame} -- float
```

1.1. NatronEngine
Return type  str
Get the value of the parameter at the given frame.
NatronEngine.StringParamBase.getDefaultValue()

Return type  str
Get the default value for this parameter.
NatronEngine.StringParamBase.getValue()

Return type  str
Same as get()
NatronEngine.StringParamBase.getValueAtTime(time)
Parameters time – float

Return type  str
Same as get(frame)
NatronEngine.StringParamBase.restoreDefaultValue()
Removes all animation and expression set on this parameter and set the value to be the default value.
NatronEngine.StringParamBase.set(x)
Parameters x – str
Set the value of this parameter to be x. If this parameter is animated (see getIsAnimated(dimension) then this function will automatically add a keyframe at the timeline’s current time.
NatronEngine.StringParamBase.set(x, frame)
Parameters

  • x – str
  • frame – float
Set a new keyframe on the parameter with the value x at the given frame.
NatronEngine.StringParamBase.setDefaultValue(value)
Parameters value – str
Set the default value for this parameter.
NatronEngine.StringParamBase.setValue(value)
Parameters value – str
Same as set
NatronEngine.StringParamBase.setValueAtTime(value, time)
Parameters

  • value – str
  • time – float
Same as set(time)<NatronEngine.StringParamBase.set()
UserParamHolder

Inherited by: Effect, PyModalDialog

Synopsis

This is an abstract class that serves as a base interface for all objects that can hold user parameters. See Detailed Description

Functions

- def createBooleanParam(name, label)
- def createButtonParam(name, label)
- def createChoiceParam(name, label)
- def createColorParam(name, label, useAlpha)
- def createDouble2DParam(name, label)
- def createDouble3DParam(name, label)
- def createDoubleParam(name, label)
- def createFileParam(name, label)
- def createGroupParam(name, label)
- def createInt2DParam(name, label)
- def createInt3DParam(name, label)
- def createIntParam(name, label)
- def createOutputFileParam(name, label)
- def createPageParam(name, label)
- def createParametricParam(name, label, nbCurves)
- def createPathParam(name, label)
- def createStringParam(name, label)
- def removeParam(param)
- def refreshUserParamsGUI()  

Detailed Description

To create a new user parameter on the object, use one of the createXParam function. To remove a user parameter created, use the removeParam(param) function. Note that this function can only be used to remove user parameters and cannot be used to remove parameters that were defined by the OpenFX plug-in.

Once you have made modifications to the user parameters, you must call the refreshUserParamsGUI() function to notify the GUI, otherwise no change will appear on the GUI.
Member functions description

NatronEngine.UserParamHolder.createBooleanParam(name, label)

Parameters

• name – str
• label – str

Return type: BooleanParam

Creates a new user parameter with the given name and label. See here for an explanation of the difference between the name and label. This function will return a new parameter of type boolean which will appear in the user interface as a checkbox.

Warning: After calling this function you should call refreshUserParamsGUI() to refresh the user interface. The refreshing is done in a separate function because it may be expensive and thus allows you to make multiple changes to user parameters at once while keeping the user interface responsive.

NatronEngine.UserParamHolder.createButtonParam(name, label)

Parameters

• name – str
• label – str

Return type: ButtonParam

Creates a new user parameter with the given name and label. See here for an explanation of the difference between the name and label. This function will return a new parameter of type button which will appear as a push button. Use the onParamChanged callback of the Effect to handle user clicks.

Warning: After calling this function you should call refreshUserParamsGUI() to refresh the user interface. The refreshing is done in a separate function because it may be expensive and thus allows you to make multiple changes to user parameters at once while keeping the user interface responsive.

NatronEngine.UserParamHolder.createChoiceParam(name, label)

Parameters

• name – str
• label – str

Return type: ChoiceParam

Creates a new user parameter with the given name and label. See here for an explanation of the difference between the name and label. This function will return a new parameter of type choice which will appear as a dropdown combobox.

Warning: After calling this function you should call refreshUserParamsGUI() to refresh the user interface. The refreshing is done in a separate function because it may be expensive and thus allows you to make multiple changes to user parameters at once while keeping the user interface responsive.

NatronEngine.UserParamHolder.createColorParam(name, label, useAlpha)

Parameters

• name – str
• label – str
• useAlpha – bool
Return type ColorParam

Creates a new user parameter with the given name and label. See here for an explanation of the difference between the name and label. This function will return a new parameter of type color.

**Warning:** After calling this function you should call `refreshUserParamsGUI()` to refresh the user interface. The refreshing is done in a separate function because it may be expensive and thus allows you to make multiple changes to user parameters at once while keeping the user interface responsive.

NatronEngine.UserParamHolder.createDouble2DParam(name, label)

Parameters

• name – str
• label – str

Return type Double2DParam

Creates a new user parameter with the given name and label. See here for an explanation of the difference between the name and label. This function will return a new parameter of type double with 2 dimensions.

**Warning:** After calling this function you should call `refreshUserParamsGUI()` to refresh the user interface. The refreshing is done in a separate function because it may be expensive and thus allows you to make multiple changes to user parameters at once while keeping the user interface responsive.

NatronEngine.UserParamHolder.createDouble3DParam(name, label)

Parameters

• name – str
• label – str

Return type Double3DParam

Creates a new user parameter with the given name and label. See here for an explanation of the difference between the name and label. This function will return a new parameter of type double with 3 dimensions.

**Warning:** After calling this function you should call `refreshUserParamsGUI()` to refresh the user interface. The refreshing is done in a separate function because it may be expensive and thus allows you to make multiple changes to user parameters at once while keeping the user interface responsive.

NatronEngine.UserParamHolder.createDoubleParam(name, label)

Parameters

• name – str
• label – str

Return type DoubleParam

Creates a new user parameter with the given name and label. See here for an explanation of the difference between the name and label. This function will return a new parameter of type double with single dimension. A double is similar to a floating point value.

**Warning:** After calling this function you should call `refreshUserParamsGUI()` to refresh the user interface. The refreshing is done in a separate function because it may be expensive and thus allows you to make multiple changes to user parameters at once while keeping the user interface responsive.
NatronEngine.UserParamHolder.createFileParam(name, label)

**Parameters**
- name – str
- label – str

**Return type** FileParam

Creates a new user parameter with the given name and label. See here for an explanation of the difference between the name and label. This function will return a new parameter of type double with 2 dimensions.

**Warning:** After calling this function you should call refreshUserParamsGUI() to refresh the user interface. The refreshing is done in a separate function because it may be expensive and thus allows you to make multiple changes to user parameters at once while keeping the user interface responsive.

NatronEngine.UserParamHolder.createGroupParam(name, label)

**Parameters**
- name – str
- label – str

**Return type** GroupParam

Creates a new user parameter with the given name and label. See here for an explanation of the difference between the name and label. This function will return a new parameter of type group. It can contain other children parameters and can be expanded or folded.

**Warning:** After calling this function you should call refreshUserParamsGUI() to refresh the user interface. The refreshing is done in a separate function because it may be expensive and thus allows you to make multiple changes to user parameters at once while keeping the user interface responsive.

NatronEngine.UserParamHolder.createInt2DParam(name, label)

**Parameters**
- name – str
- label – str

**Return type** Int2DParam

Creates a new user parameter with the given name and label. See here for an explanation of the difference between the name and label. This function will return a new parameter of type integer with 2 dimensions.

**Warning:** After calling this function you should call refreshUserParamsGUI() to refresh the user interface. The refreshing is done in a separate function because it may be expensive and thus allows you to make multiple changes to user parameters at once while keeping the user interface responsive.

NatronEngine.UserParamHolder.createInt3DParam(name, label)

**Parameters**
- name – str
- label – str

**Return type** Int3DParam

Creates a new user parameter with the given name and label. See here for an explanation of the difference between the name and label. This function will return a new parameter of type integer with 2 dimensions.

**Warning:** After calling this function you should call refreshUserParamsGUI() to refresh the user interface. The refreshing is done in a separate function because it may be expensive and thus allows you to make multiple changes to user parameters at once while keeping the user interface responsive.
Creates a new user parameter with the given name and label. See here for an explanation of the difference between the name and label. This function will return a new parameter of type integer with 3 dimensions.

**Warning:** After calling this function you should call `refreshUserParamsGUI()` to refresh the user interface. The refreshing is done in a separate function because it may be expensive and thus allows you to make multiple changes to user parameters at once while keeping the user interface responsive.

`NatronEngine.UserParamHolder.createIntParam(name, label)`

**Parameters**
- `name` – str
- `label` – str

**Return type** `IntParam`

Creates a new user parameter with the given name and label. See here for an explanation of the difference between the name and label. This function will return a new parameter of type integer with a single dimension.

**Warning:** After calling this function you should call `refreshUserParamsGUI()` to refresh the user interface. The refreshing is done in a separate function because it may be expensive and thus allows you to make multiple changes to user parameters at once while keeping the user interface responsive.

`NatronEngine.UserParamHolder.createOutputFileParam(name, label)`

**Parameters**
- `name` – str
- `label` – str

**Return type** `OutputFileParam`

Creates a new user parameter with the given name and label. See here for an explanation of the difference between the name and label. This function will return a new parameter of type string dedicated to specify paths to output files.

**Warning:** After calling this function you should call `refreshUserParamsGUI()` to refresh the user interface. The refreshing is done in a separate function because it may be expensive and thus allows you to make multiple changes to user parameters at once while keeping the user interface responsive.

`NatronEngine.UserParamHolder.createPageParam(name, label)`

**Parameters**
- `name` – str
- `label` – str

**Return type** `PageParam`

Creates a new user parameter with the given name and label. See here for an explanation of the difference between the name and label. This function will return a new parameter of type page. A page is a tab within the settings panel of the node.

**Warning:** After calling this function you should call `refreshUserParamsGUI()` to refresh the user interface. The refreshing is done in a separate function because it may be expensive and thus allows you to make multiple changes to user parameters at once while keeping the user interface responsive.

`NatronEngine.UserParamHolder.createParametricParam(name, label, nbCurves)`
Parameters

- name – str
- label – str
- nbCurves – int

Return type: ParametricParam

Creates a new user parameter with the given name and label. See here for an explanation of the difference between the name and label. This function will return a new parameter of type parametric. A parametric parameter is what can be found in the ColorLookup node or in the Ranges tab of the ColorCorrect node.

Warning: After calling this function you should call refreshUserParamsGUI() to refresh the user interface. The refreshing is done in a separate function because it may be expensive and thus allows you to make multiple changes to user parameters at once while keeping the user interface responsive.

NatronEngine.UserParamHolder.createPathParam(name, label)

Parameters

- name – str
- label – str

Return type: PathParam

Creates a new user parameter with the given name and label. See here for an explanation of the difference between the name and label. This function will return a new parameter of type string. This parameter is dedicated to specify path to single or multiple directories.

Warning: After calling this function you should call refreshUserParamsGUI() to refresh the user interface. The refreshing is done in a separate function because it may be expensive and thus allows you to make multiple changes to user parameters at once while keeping the user interface responsive.

NatronEngine.UserParamHolder.createStringParam(name, label)

Parameters

- name – str
- label – str

Return type: StringParam

Creates a new user parameter with the given name and label. See here for an explanation of the difference between the name and label. This function will return a new parameter of type string.

Warning: After calling this function you should call refreshUserParamsGUI() to refresh the user interface. The refreshing is done in a separate function because it may be expensive and thus allows you to make multiple changes to user parameters at once while keeping the user interface responsive.

NatronEngine.UserParamHolder.removeParam(param)

Parameters: param – Param

Return type: bool

Removes the given param from the parameters of this Effect. This function works only if param is a user parameter and does nothing otherwise. This function returns True upon success and False otherwise.
Warning: After calling this function you should call `refreshUserParamsGUI()` to refresh the user interface. The refreshing is done in a separate function because it may be expensive and thus allows you to make multiple changes to user parameters at once while keeping the user interface responsive.

NatronEngine.UserParamHolder.refreshUserParamsGUI()

This function must be called after new user parameter were created or removed. This will re-create the user interface for the parameters and can be expensive.

1.2 NatronGui

1.2.1 Detailed Description

Here are listed all classes being part of NatronEngine module. This module is loaded by Natron natively in GUI mode only. In that case, access is granted to these classes in your scripts without importing anything. Scripts that want to operate both in command line background mode and in GUI mode should poll the `NatronEngine.PyCoreApplication.isBackground()` function on the `natron` object before calling functions dependent on the module `NatronGui`. E.g:

```python
if not NatronEngine.natron.isBackground():
    # do GUI only stuff here
```

GuiApp

Inherits App

Synopsis

This class is used for GUI application instances. See detailed description...

Functions

- `def createModalDialog ()`
- `def getFilenameDialog (filters[, location=None])`
- `def getSequenceDialog (filters[, location=None])`
- `def getDirectoryDialog ([location=None])`
- `def getRGBColorDialog ()`
- `def getTabWidget (scriptName)`
- `def getSelectedNodes ([group=None])`
- `def getViewer (scriptIdName)`
- `def getUserPanel (scriptIdName)`
- `def moveTab (tabScriptName,pane)`
- `def saveFilenameDialog (filters[, location=None])`
- `def saveSequenceDialog (filters[, location=None])`
• def **selectNode**(node, clearPreviousSelection)
• def **deselectNode**(node)
• def **setSelection**(nodes)
• def **selectAllNodes**(group=None)
• def **clearSelection**(group=None)
• def **registerPythonPanel**(panel, pythonFunction)
• def **unregisterPythonPanel**(panel)
• def **renderBlocking**(effect, firstFrame, lastFrame, frameStep)

**Detailed Description**

See App for the documentation of base functionalities of this class.

To create a new modal dialog, use the :func:`NatronGui.GuiApp.createModalDialog()` function.

Several functions are made available to pop dialogs to ask the user for filename(s) or colors. See `getFilenameDialog(filters, location)` and `getRGBColorDialog()`.

To create a new custom python panel, there are several ways to do it:

- Sub-class the PyPanel class and make your own PySide widget
- Create a :doc:`PyPanel` object and add controls using user parameters (as done for modal dialogs)

Once created, you can register the panel in the project so that it gets saved into the layout by calling `registerPythonPanel(panel, pythonFunction)`

**Member functions description**

**NatronGui.GuiApp.createModalDialog()**

**Return type** PyModalDialog

Creates a modal dialog: the control will not be returned to the user until the dialog is not closed. Once the dialog is created, you can enrich it with parameters or even raw PySide Qt widgets. To show the dialog call the function `exec()` on the dialog.

**NatronGui.GuiApp.getFilenameDialog(filters[, location=None])**

**Return type** str

Opens-up a file dialog to ask the user for a single filename which already exists.

`filters` is a list of file extensions that should be displayed in the file dialog.

`location` is the initial location the dialog should display, unless it is empty in which case the dialog will display the last location that was opened previously by a dialog.

**NatronGui.GuiApp.getSequenceDialog(filters[, location=None])**

**Return type** str
Return type str

Same as `getFilenameDialog(filters, location)` but the dialog will accept sequence of files.

NatronGui.GuiApp.getDirectoryDialog([location=None])

    param location str

Return type str

Same as `getFilenameDialog(filters, location)` but the dialog will only accept directories as a result.

NatronGui.GuiApp.saveFilenameDialog(filters[, location=None])

    param filters sequence
    param location str

Return type str

Opens-up a file dialog to ask the user for a single filename. If the file already exists, the user will be warned about potential overriding of the file.

`filters` is a list of file extensions that should be displayed in the file dialog.

`location` is the initial location the dialog should display, unless it is empty in which case the dialog will display the last location that was opened previously by a dialog.

NatronGui.GuiApp.saveSequenceDialog(filters[, location=None])

    param filters sequence
    param location str

Return type str

Same as `saveFilenameDialog(filters, location)` but the dialog will accept sequence of files.

NatronGui.GuiApp.RGBColorDialog()

Return type ColorTuple

Opens-up a color dialog to ask the user for an RGB color.

NatronGui.GuiApp.getTabWidget(scriptName)

    param scriptName str

Return type PyTabWidget

Returns the tab-widget with the given `scriptName`. The `scriptName` of a tab-widget can be found in the user interface when hovering with the mouse the “Manage layout” button (in the top left-hand corner of the pane)

NatronGui.GuiApp.moveTab(tabScriptName, pane)

    param tabScriptName str
    param pane PyTabWidget
Attempts to move the tab with the given `tabScriptName` into the given `pane` and make it current in the `pane`. This function returns True upon success or False otherwise.

**Warning:** Moving tabs that are not registered to the application via `registerPythonPanel(panel,pythonFunction)` will not work.

NatronGui.GuiApp.registerPythonPanel(panel, pythonFunction)

**Parameters**
- `panel` – PyPanel
- `scriptName` – str

Registers the given panel into the project. When registered, the panel will be saved into the layout for the current project and a new entry in the “Panes” sub-menu of the “Manage layouts” button (in the top left-hand corner of each tab widget) will appear for this panel. `pythonFunction` is the name of a python-defined function that takes no argument that should be used to re-create the panel.

NatronGui.GuiApp.unregisterPythonPanel(panel)

**Parameters**
- `panel` – PyPanel

Unregisters a previously registered panel.

NatronGui.GuiApp.getSelectedNodes([group = None])

**Return type** sequence

Returns a sequence of `nodes` currently selected in the given `group`. You can pass the `app` object to get the top-level NodeGraph. If passing None, the last user-selected NodeGraph will be used:

```python
topLevelSelection = app.getSelectedNodes()
group = app.createNode("fr.inria.built-in.Group")
groupSelection = app.getSelectedNodes(group)
```
NatronGui.GuiApp.getViewer(scriptName)

Parameters scriptName – str

Returns the viewer with the given scriptName if one can be found.

NatronGui.GuiApp.getUserPanel(scriptName)

Parameters scriptName – str

Returns a user panel matching the given scriptName if there is any.

NatronGui.GuiApp.selectNode(node, clearPreviousSelection)

Parameters

• node – Effect
• clearPreviousSelection – bool

Select the given node in its containing nodegraph. If clearPreviousSelection is set to True, all the current selection will be wiped prior to selecting the node; otherwise the node will just be added to the selection.

NatronGui.GuiApp.deselectNode(node)

Parameters node – Effect

Deselect the given node in its containing nodegraph. If the node is not selected, this function does nothing.

NatronGui.GuiApp.setSelection(nodes)

Parameters nodes – sequence

Set all the given nodes selected in the nodegraph containing them and wipe any current selection.

Note: All nodes must be part of the same nodegraph (group), otherwise this function will fail.

NatronGui.GuiApp.selectAllNodes([group=None])

Parameters group – Group

Select all nodes in the given group. You can pass the app object to get the top-level NodeGraph. If passing None, the last user-selected NodeGraph will be used.

NatronGui.GuiApp.clearSelection([group=None])

Wipe any current selection in the given group. You can pass the app object to get the top-level NodeGraph. If passing None, the last user-selected NodeGraph will be used.

NatronGui.GuiApp.renderBlocking(effect, firstFrame, lastFrame, frameStep)

Parameters

• effect – Effect
• firstFrame – int
• lastFrame – int
• frameStep – int

Starts rendering the given effect on the frame-range defined by [firstFrame,*lastFrame*]. The frameStep parameter indicates how many frames the timeline should step after rendering each frame. The value must be greater or equal to 1. The frameStep parameter is optional and if not given will default to the value of the Frame Increment parameter in the Write node.

For instance:
render(effect,1,10,2)

Would render the frames 1,3,5,7,9

This is a blocking function. A blocking render means that this function returns only when the render finishes (from failure or success).

This function should only be used to render with a Write node or DiskCache node.

NatronGui.GuiApp.renderBlocking(tasks)

Parameters tasks - sequence

This function takes a sequence of tuples of the form (effect,firstFrame,lastFrame[,frameStep]) The frameStep is optional in the tuple and if not set will default to the value of the Frame Increment parameter in the Write node.

This is an overloaded function. Same as render(effect,firstFrame,lastFrame,frameStep) but all tasks will be rendered concurrently.

This function is called when rendering a script in background mode with multiple writers.

This is a blocking call.

**PyGuiApplication**

**Inherits** PyCoreApplication

**Synopsis**

See PyCoreApplication for a detailed explanation of the purpose of this object. This class is only used when Natron is run in GUI mode (with user interface). It gives you access to more GUI functionalities via the GuiApp class.

**Functions**

- def addMenuCommand (grouping, function)
- def addMenuCommand (grouping, function, key, modifiers)
- def getGuiInstance (idx)
- def informationDialog (title, message)
- def warningDialog (title, message)
- def errorDialog (title, message)
- def questionDialog (title, question)

**Member functions description**

**class** NatronGui.PyGuiApplication

See PyCoreApplication()

NatronGui.PyGuiApplication.addMenuCommand (grouping, function)

Parameters grouping - str :param function: str

Adds a new menu entry in the menubar of Natron. This should be used exclusively in the initGui.py initialisation script.
The `grouping` is a string indicating a specific menu entry where each submenu is separated from its parent menu with a `/`:

- File/Do something special
- MyStudio/Scripts/Our special trick

The `function` is the name of a python defined function.

**Warning:** If called anywhere but from the `initGui.py` script, this function will fail to dynamically add a new menu entry.

Example:

```python
def printLala():
    print("Lala")
natron.addMenuCommand("Inria/Scripts/Print lala script","printLala")
```

This registers in the menu `Inria-->Scripts` an entry named `Print lala script` which will print `Lala` to the Script Editor when triggered.

NatronGui.PyGuiApplication.addMenuCommand(grouping, function, key, modifiers)

**Parameters**

- **grouping** – str
- **function** – str
- **key** – PySide.QtCore.Qt.Key
- **modifiers** – PySide.QtCore.Qt.KeyboardModifiers

Same as `addMenuCommand(grouping, function)` excepts that it accepts a default shortcut for the action. See PySide documentation for possible keys and modifiers.

The user will always be able to modify the shortcut from the built-in shortcut editor of Natron anyway.

NatronGui.PyGuiApplication.getGuiInstance(idx)

**Parameters**

- **idx** – int

**Return type** GuiApp

Same as `getInstance(idx)` but returns instead an instance of a GUI project.

Basically you should never call this function as Natron pre-declares all opened projects with the following variables: `app1` for the first opened project, `app2` for the second, and so on...

NatronGui.PyGuiApplication.informationDialog(title, message)

**Parameters**

- **title** – str
- **message** – str

Shows a modal information dialog to the user with the given window `title` and containing the given `message`.

NatronGui.PyGuiApplication.warningDialog(title, message)

**Parameters**

- **title** – str
• message – str

Shows a modal warning dialog to the user with the given window title and containing the given message.

NatronGui.PyGuiApplication.errorDialog(title, message)

Parameters
• title – str
• message – str

Shows a modal error dialog to the user with the given window title and containing the given message.

NatronGui.PyGuiApplication.questionDialog(title, message)

Parameters
• title – str
• message – str

Return type NatronEngine.StandardButtonEnum

Shows a modal question dialog to the user with the given window title and containing the given message. The dialog will be a “Yes” “No” dialog, and you can compare the result to the NatronEngine.StandardButtonEnum members.

PyModalDialog

Inherits UserParamHolder QDialog

Synopsis

A modal dialog to ask informations to the user or to warn about something. See detailed description...

Functions
• def addWidget (widget)
• def getParam(scriptName)
• def insertWidget (index, widget)
• def setParamChangedCallback (callback)

Detailed Description

The modal dialog is a way to ask the user for data or to inform him/her about something going on. A modal window means that control will not be returned to the user (i.e no event will be processed) until the user closed the dialog.

If you are looking for a simple way to just ask a question or report an error, warning or even just a miscellaneous information, use the informationDialog(title, message) function.

To create a new PyModalDialog, just use the createModalDialog() function, e.g:

# In the Script Editor

dialog = appl.createModalDialog()
To show the dialog to the user, use the `exec()` function inherited from `QDialog`.

```python
dialog.exec()
```

Note that once `exec()` is called, no instruction will be executed until the user closed the dialog.

The modal dialog always has `OK` and `Cancel` buttons. To query which button the user pressed, inspect the return value of the `exec()` call:

```python
if dialog.exec_():
    # The user pressed OK
    ...
else:
    # The user pressed Cancel or Escape
```

**Adding user parameters:** You can start adding user parameters using all the `createXParam` functions inherited from the `UserParamHolder` class.

Once all your parameters are created, create the GUI for them using the `refreshUserParamsGUI()` function:

```python
myInteger = dialog.createIntParam("myInt","This is an integer very important")
myInteger.setAnimationEnabled(False)
myInteger.setAddNewLine(False)

# Create a boolean on the same line
myBoolean = dialog.createBooleanParam("myBool","Yet another important boolean")

dialog.refreshUserParamsGUI()

dialog.exec()
```

You can then retrieve the value of a parameter once the dialog is finished using the `getParam(scriptName)` function:

```python
if dialog.exec_():
    intValue = dialog.getParam("myInt").get()
    boolValue = dialog.getParam("myBool").get()
```

**Warning:** Unlike the `Effect` class, parameters on modal dialogs are not automatically declared by Natron, which means you cannot do stuff like `dialog.intValue`

### Member functions description

**NatronGui.PyModalDialog**. `addWidget(widget)`
- **Parameters**
  - `widget` — `PySide.QtGui.QWidget`
  - Append a `QWidget` inherited `widget` at the bottom of the dialog. This allows to add custom GUI created directly using `PySide` that will be inserted after any custom parameter.

**NatronGui.PyModalDialog**. `getParam(scriptName)`
- **Parameters**
  - `scriptId` — `str`
  - **Return type** `Param`
  - Returns the user parameter with the given `scriptId` if it exists or `None` otherwise.

**NatronGui.PyModalDialog**. `insertWidget(index, widget)`
- **Parameters**
• **index**—int
• **widget**—PySide.QtGui.QWidget

Inserts a QWidget inherited widget at the given index of the layout in the dialog. This allows to add custom GUI created directly using PySide. The widget will always be inserted after any user parameter.

NatronGui.PyModalDialog.setParamChangedCallback(callback)

**Parameters**

**callback**—str

Registers the given Python callback to be called whenever a user parameter changed. The callback should be the name of a Python defined function (taking no parameter).

The variable **paramName** will be declared upon calling the callback, referencing the script-name of the parameter that just changed. Example:

```python
def myCallback():
    if paramName == "myInt":
        intValue = thisParam.get()
        if intValue > 0:
            myBoolean.setVisible(False)

dialog.setParamChangedCallback("myCallback")
```

**PyPanel**

**Inherits** PySide.QtGui.QWidget NatronEngine.UserParamHolder

**Synopsis**

A custom PySide pane that can be docked into PyTabWidget. See *detailed* description...

**Functions**

• def **PyPanel** (scriptName,label,useUserParameters,app)
• def **addWidget** (widget)
• def **getPanelLabel** ()
• def **getPanelScriptName** ()
• def **getParam** (scriptName)
• def **getParams** ()
• def **insertWidget** (index,widget)
• def **onUserDataChanged** ()
• def **setParamChangedCallback** (callback)
• def **save** ()
• def **setPanelLabel** (label)
• def **restore** (data)
Detailed Description

The PyPanel class can be used to implement custom PySide widgets that can then be inserted as tabs into tab-widgets.

There are 2 possible usage of this class:

- Sub-class it and create your own GUI using PySide
- Use the API proposed by PyPanel to add custom user parameters as done for PyModalDialog.

Sub-classing: When sub-classing the PyPanel class, you should specify when calling the base class constructor that you do not want to use user parameters, as this might conflict with the layout that you will use:

```python
class MyPanel(NatronGui.PyPanel):
    def __init__(scriptName,label,app):
        NatronGui.PyPanel.__init__(scriptName,label,False,app)
```

You're then free to use all features proposed by PySide in your class, including signal/slots. See the following example.

Using the PyPanel API: You can start adding user parameters using all the createXParam functions inherited from the UserParamHolder class.

Once all your parameters are created, create the GUI for them using the refreshUserParamsGUI() function:

```python
panel = NatronGui.PyPanel("fr.inria.mypanel","My Panel",True,app)
myInteger = panel.createIntParam("myInt","This is an integer very important")
myInteger.setAnimationEnabled(False)
myInteger.setAddNewLine(False)
#Create a boolean on the same line
myBoolean = panel.createBooleanParam("myBool","Yet another important boolean")
panel.refreshUserParamsGUI()
```

You can then retrieve the value of a parameter at any time using the getParam(scriptName) function:

```python
intValue = panel.getParam("myInt").get()
boolValue = panel.getParam("myBool").get()
```

**Warning:** Unlike the Effect class, parameters on panels are not automatically declared by Natron, which means you cannot do stuff like `panel.intValue`

You can get notified when a parameter’s value changed, by setting a callback using the setParamChangedCallback(callback) function that takes the name of a Python-defined function in parameters. The variable `thisParam` will be declared prior to calling the callback, referencing the parameter which just had its value changed.

Managing the panel: Once created, you must add your panel to a PyTabWidget so it can be visible. Use the getTabWidget(scriptName) function to get a particular pane and then use the appendTab(pan) function to add this panel to the pane.

**Warning:** Note that the lifetime of the widget will be by default the same as the project’s GUI because PyPanel is auto-declared by Natron.
panel = NatronGui.PyPanel("fr.inria.mypanel","My Panel",True,app)
...
...
pane = app.getTabWidget("pane")
pane.appendTab(panel)
app.mypanel = panel

If you want the panel to persist in the project so that it gets recreated and placed at its original position when the user loads the project, you must use the registerPythonPanel(panel,function) function.

Note that the function parameter is the name of a Python-defined function that takes no parameter used to create the widget, e.g:

```python
def createMyPanel():
    panel = NatronGui.PyPanel("MyPanel",True,app)
    ...
    #Make it live after the scope of the function
    app.mypanel = panel

app.registerPythonPanel(app.mypanel,"createMyPanel")
```

This function will also add a custom menu entry to the “Manage layout” button (located in the top-left hand corner of every pane) which the user can trigger to move the custom pane on the selected pane.

Saving and restoring state: When the panel is registered in the project using the registerPythonPanel(panel,function) function, you may want to also save the state of your widgets and/or special values.

To do so, you must sub-class PyPanel and implement the save() and restore(data) functions.

Note: User parameters, if used, will be automatically saved and restored, you don’t have to save it yourself. Hence if the panel is only composed of user parameters that you want to save, you do not need to sub-class PyPanel as it will be done automatically for you.

The function save() should return a string containing the serialization of your custom data.
The function `restore(data)` will be called upon loading of a project containing an instance of your panel. You should then restore the state of the panel from your custom serialized data.

Note that the auto-save of Natron occurs in a separate thread and for this reason it cannot call directly your `save()` function because it might create a race condition if the user is actively modifying the user interface using the main-thread.

To overcome this, Natron has an hidden thread-safe way to recover the data you have serialized using the `save()` function. The downside is that you have to call the `onUserDataChanged()` function whenever a value that you want to be persistent has changed (unless this is a user parameter in which case you do not need to call it).

**Warning:** If you do not call `onUserDataChanged()`, the `save()` function will never be called, and the data never serialized.

### Member functions description

**NatronGui.PyPanel**

#### `PyPanel(label, useUserParameters, app)`

**Parameters**

- `label` – str
- `useUserParameters` – bool
- `app` – GuiApp

Make a new PyPanel with the given `label` that will be used to display in the tab header. If `useUserParameters` is True then user parameters support will be activated, attempting to modify the underlying layout in these circumstances will result in undefined behaviour.

**NatronGui.PyPanel**

#### `addWidget(widget)`

**Parameters**

- `widget` – PySide.QtGui.QWidget

Append a QWidget inherited `widget` at the bottom of the dialog. This allows to add custom GUI created directly using PySide that will be inserted after any custom parameter.

**Warning:** This function should be used exclusively when the widget was created using `useUserParameters = True`

**NatronGui.PyPanel**

#### `getParam(scriptName)`

**Parameters**

- `scriptName` – str

**Return type** Param

Returns the user parameter with the given `scriptName` if it exists or `None` otherwise.

**Warning:** This function should be used exclusively when the widget was created using `useUserParameters = True`

**NatronGui.PyPanel**

#### `getParams()`

**Return type** sequence

Returns all the user parameters used by the panel.

**Warning:** This function should be used exclusively when the widget was created using `useUserParameters = True`

**NatronGui.PyPanel**

#### `insertWidget(index, widget)`
Parameters

- **index** – int
- **widget** – PySide.QtGui.QWidget

Inserts a QWidget inherited widget at the given index of the layout in the dialog. This allows to add custom GUI created directly using PySide. The widget will always be inserted after any user parameter.

**Warning:** This function should be used exclusively when the widget was created using `useUserParameters = True`

NatronGui.PyPanel.setParamChangedCallback(callback)

**Parameters**

- **callback** – str

Registers the given Python callback to be called whenever a user parameter changed. The callback should be the name of a Python defined function (taking no parameter).

The variable `paramName` will be declared upon calling the callback, referencing the script name of the parameter that just changed. Example:

```python
def myCallback():
    if paramName == "myInt":
        intValue = thisParam.get()
        if intValue > 0:
            myBoolean.setVisible(False)
panel.setParamChangedCallback("myCallback")
```

**Warning:** This function should be used exclusively when the widget was created using `useUserParameters = True`

NatronGui.PyPanel.setPanelLabel(label)

**Parameters**

- **callback** – str

Set the label of the panel as it will be displayed on the tab header of the PyTabWidget. This name should be unique.

**Return type**

- **str**

Get the label of the panel as displayed on the tab header of the PyTabWidget.

NatronGui.PyPanel.getPanelScriptName()

**Return type**

- **str**

Get the script-name of the panel as used internally. This is a unique string identifying the tab in Natron.

NatronGui.PyPanel.onUserDataChanged()

Callback to be called whenever a parameter/value (that is not a user parameter) that you want to be saved has changed.

**Warning:** If you do not call `onUserDataChanged()`, the `save()` function will never be called, and the data never serialized.

**Warning:** This function should be used exclusively when the widget was created using `useUserParameters = True`

NatronGui.PyPanel.save()
Return type  \texttt{str}

\textbf{Warning:} You should overload this function in a derived class. The base version does nothing.

\textbf{Note:} User parameters, if used, will be automatically saved and restored, you don’t have to save it yourself. Hence if the panel is only composed of user parameters that you want to save, you do not need to sub-class PyPanel as it will be done automatically for you.

Returns a string with the serialization of your custom data you need to be persistent.

\texttt{NatronGui.PyPanel.restore(data)}

\begin{description}
\item[Parameters] \texttt{data} – \texttt{str}
\end{description}

\textbf{Warning:} You should overload this function in a derived class. The base version does nothing.

This function should restore the state of your custom PyPanel using the custom \texttt{data} that you serialized. The \texttt{data} are exactly the return value that was returned from the \texttt{save()} function.

\textbf{PyTabWidget}

\textbf{Synopsis}

A PyTabWidget is one of the GUI pane onto which the user can dock tabs such as the NodeGraph, CurveEditor... See \textit{detailed} description...

\textbf{Functions}

- def \texttt{appendTab} (tab)
- def \texttt{closeCurrentTab} ()
- def \texttt{closeTab} (index)
- def \texttt{closePane} ()
- def \texttt{count} ()
- def \texttt{currentWidget} ()
- def \texttt{floatCurrentTab} ()
- def \texttt{floatPane} ()
- def \texttt{getCurrentIndex} ()
- def \texttt{getScriptName} ()
- def \texttt{getTabLabel} (index)
- def \texttt{insertTab} (index,tab)
- def \texttt{removeTab} (tab)
- def \texttt{removeTab} (index)
- def \texttt{setCurrentIndex} (index)
- def \texttt{setNextTabCurrent} ()
• def `splitHorizontally()`
• def `splitVertically()`

Detailed Description

The `PyTabWidget` class is used to represent panes visible in the user interface:

On the screenshot above, each `PyTabWidget` is surrounded by a red box.
You cannot construct tab widgets on your own, you must call one of the `splitVertically()` or `splitHorizontally()` functions to make a new one based on another existing ones.
By default the GUI of Natron cannot have less than 1 tab widget active, hence you can always split it to make new panes.
To retrieve an existing `PyTabWidget` you can call the `getTabWidget(scriptName)` function of `GuiApp`.

```
panel = app.getTabWidget("Panel")
```

Note that the script-name of a pane can be seen on the graphical user interface by hovering the mouse over the “Manage layout” button (in the top left hand corner of a pane).

Managing tabs To insert tabs in the TabWidget you can call either `appendTab(tab)` or `insertTab(index, tab)`.

**Warning:** Note that to insert a tab, it must be first removed from the tab into which it was.

To remove a tab, use the function `removeTab(tab)` on the parent `PyTabWidget`
For convenience to move tabs around, there is a `moveTab(tab, pane)` function in `GuiApp`.
The function `closeTab(index)` can be used to close permanently a tab, effectively destroying it.
To change the current tab, you can use one of the following functions:

• `setCurrentIndex(index)`<NatronGui.PyTabWidget.setCurrentIndex>
To float the current tab into a new floating window, use the `floatCurrentTab()` function.

Managing the pane  To close the pane permanently, use the `closePane()` function. To float the pane into a new floating window with all its tabs, use the `floatPane()` function.

### Member functions description

**NatronGui.PyTabWidget.**

#### appendTab(tab)

**Parameters**

- **tab** - `PySide.QtGui.QWidget`

Appends a new tab to the tab widget and makes it current.

**NatronGui.PyTabWidget.**

#### closeCurrentTab()

Closes the current tab, effectively destroying it.

**NatronGui.PyTabWidget.**

#### closeTab(index)

Closes the tab at the given `index`, effectively destroying it.

**NatronGui.PyTabWidget.**

#### closePane()

Closes this pane, effectively destroying it. Note that all tabs will not be destroyed but instead moved to another existing pane.

**Warning:** If this pane is the last one on the GUI, this function does nothing.

**NatronGui.PyTabWidget.**

#### count()

**Return type** `int`

Returns the number of tabs in this pane.
NatronGui.PyTabWidget.currentWidget()

Return type PySide.QtGui.QWidget

Returns the current active tab.

NatronGui.PyTabWidget.floatCurrentTab()

Make a new floating window with a single pane and moves the current tab of this pane to the new pane of the floating window.

NatronGui.PyTabWidget.floatPane()

Make a new floating window and moves this pane to the new window (including all tabs).

NatronGui.PyTabWidget getCurrentIndex()

Return type int

Returns the index of the current tab. This is 0-based (starting from the left).

NatronGui.PyTabWidget.getScriptName()

Return type str

Returns the script-name of the pane, as used by the getTabWidget (scriptName) function.

NatronGui.PyTabWidget.getTabLabel(index)

Parameters index – int

Return type str

Returns the name of the tab at the given index if it exists or an empty string otherwise.

NatronGui.PyTabWidget.insertTab(index, tab)

Parameters

• tab – PySide.QtGui.QWidget
• index – int

Inserts the given tab at the given index in this tab-widget.

NatronGui.PyTabWidget.removeTab(tab)

Parameters tab – PySide.QtGui.QWidget

Removes the given tab from this pane if it is found. Note that this function does not destroy the tab, unlike closeTab(index).

This is used internally by moveTab(tab, pane).

NatronGui.PyTabWidget.removeTab(index)

Parameters index – int

Same as removeTab(tab) but the index of a tab is given instead.

NatronGui.PyTabWidget.setCurrentIndex(index)

Parameters index – int

Makes the tab at the given index (0-based) the current one (if the index is valid).

NatronGui.PyTabWidget.setNextTabCurrent()

Set the tab at getCurrentIndex() + 1 the current one. This functions cycles back to the first tab once the last tab is reached.
NatronGui.PyTabWidget.splitHorizontally()

**Return type** PyTabWidget

Splits this pane into 2 horizontally-separated panes. The new pane will be returned.

NatronGui.PyTabWidget.splitVertically()

**Return type** PyTabWidget

Splits this pane into 2 vertically-separated panes. The new pane will be returned.

**PyViewer**

**Synopsis**

A PyViewer is a wrapper around a Natron Viewer. See detailed description...

**Functions**

- def `seek`(frame)
- def `getCurrentFrame`()
- def `startForward`()
- def `startBackward`()
- def `pause`()
- def `redraw`()
- def `renderCurrentFrame`([useCache=True])
- def `setFrameRange`((firstFrame,lastFrame))
- def `getFrameRange`()
- def `setPlaybackMode`(mode)
- def `getPlaybackMode`()
- def `getCompositingOperator`()
- def `setCompositingOperator`(operator)
- def `getAInput`()
- def `setAInput`(index)
- def `getBInput`()
- def `setBInput`(index)
- def `setChannels`(channels)
- def `getChannels`()
- def `setProxyModeEnabled`(enabled)
- def `isProxyModeEnabled`()
- def `setProxyIndex`(index)
- def `setCurrentView`(viewIndex)
• def getCurrentView(channels)

**Detailed Description**

This class is a wrapper around a Natron Viewer, exposing all functionalities available as user interaction to the Python API.

To get a PyViewer, use the `getViewer(scriptName)` function, passing it the `script-name` of a viewer node.

**Member functions description**

**NatronGui.PyTabWidget.seek(frame)**

**Parameters**

- `frame` – `int`
  
  Seek the timeline to a particular frame. All other viewers in the project will be synchronized to that frame.

**NatronGui.PyTabWidget.getCurrentFrame()**

**Return type**

`int`

Returns the current frame on the timeline.

**NatronGui.PyTabWidget.startForward()**

Starts playback, playing the video normally.

**NatronGui.PyTabWidget.startBackward()**

Starts playback backward, like a rewind.

**NatronGui.PyTabWidget.pause()**

Pauses the viewer if the playback is ongoing.

**NatronGui.PyTabWidget.redraw()**

Redraws the OpenGL widget without actually re-rendering the internal image. This is provided for convenience as sometimes the viewer might need refreshing for OpenGL overlays.

**NatronGui.PyTabWidget.renderCurrentFrame([useCache=True])**

**Parameters**

- `useCache` – `bool`
  
  Renders the current frame on the timeline. If `useCache` is False, the cache will not be used and the frame will be completely re-rendered.

**NatronGui.PyTabWidget.setFrameRange(firstFrame, lastFrame)**

**Parameters**

- `firstFrame` – `int`
- `lastFrame` – `int`
  
  Set the frame range on the Viewer to be `[firstFrame, lastFrame]` (included).

**NatronGui.PyTabWidget.getFrameRange()**

**Return type**

Tuple

Returns a 2-dimensional tuple of `int` containing `[firstFrame, lastFrame]`.

**NatronGui.PyTabWidget.setPlaybackMode(mode)**

**Parameters**

  
  Set the playback mode for the Viewer, it can be either **bouncing**, **looping** or **playing once**.
NatronGui.PyTabWidget.getPlaybackMode()

Return type NatronEngine.Natron.PlaybackModeEnum

Returns the playback mode for this Viewer.

NatronGui.PyTabWidget.getCompositingOperator()

Return type NatronEngine.Natron.ViewerCompositingOperatorEnum

Returns the current compositing operator applied by the Viewer.

NatronGui.PyTabWidget.setCompositingOperator(operator)


Set the current compositing operator applied by the Viewer.

NatronGui.PyTabWidget.getAInput()

Return type int

Returns the index of the input (the same index used by getInput(index)) used by the A choice of the Viewer.

NatronGui.PyTabWidget.setAInput(index)

Parameters index – int

Set the index of the input (the same index used by getInput(index)) used by the A choice of the Viewer.

NatronGui.PyTabWidget.getBInput()

Return type int

Returns the index of the input (the same index used by getInput(index)) used by the B choice of the Viewer.

NatronGui.PyTabWidget.setBInput(index)

Parameters index – int

Set the index of the input (the same index used by getInput(index)) used by the B choice of the Viewer.

NatronGui.PyTabWidget.setChannels(channels)

Parameters channels – NatronEngine.Natron.DisplayChannelsEnum

Set the channels to be displayed on the Viewer.

NatronGui.PyTabWidget.getChannels()

Return type NatronEngine.Natron.DisplayChannelsEnum

Returns the current channels displayed on the Viewer.

NatronGui.PyTabWidget.setProxyModeEnabled(enabled)

Parameters enabled – bool

Set the proxy mode enabled.

NatronGui.PyTabWidget.isProxyModeEnabled(enabled)

Return type bool

Returns whether the proxy mode is enabled.

NatronGui.PyTabWidget.setProxyIndex(index)

Parameters index – int
Set the *index* of the proxy to use. This is the index in the combobox on the graphical user interface, e.g.: $index = 0$ will be 2

NatronGui.PyTabWidget.getProxyIndex()

**Return type** int

Returns the *index* of the proxy in use. This is the index in the combobox on the graphical user interface, e.g.: $index = 0$ will be 2

NatronGui.PyTabWidget.setCurrentView(viewIndex)

**Parameters**

`viewIndex` - int

Set the view to display the given *viewIndex*. This is the index in the multi-view combobox visible when the number of views in the project settings has been set to a value greater than 1.

NatronGui.PyTabWidget.getCurrentView()

**Parameters**

`viewIndex` - int

Returns the currently displayed view index. This is the index in the multi-view combobox visible when the number of views in the project settings has been set to a value greater than 1.
This section covers the basic principles for scripting in Python in Natron.

2.1 Natron plug-in paths

When looking for startup scripts or Python group plug-ins, Natron will look into the following search paths in order:

- The bundled plug-ins path. There are 2 kinds of plug-ins: PyPlugs and OpenFX plug-ins.
  The bundled OpenFX plug-ins are located in Plugins/OFX/Natron in your Natron installation and the bundled PyPlugs in the directory Plugins/PyPlugs.
- The standard location for non OpenFX plug-ins (i.e PyPlugs): that is the directory .Natron in the home directory, e.g:
  - On Windows that would be:
    ```
    C:\Users\<username>\.Natron
    ```
  - On OS X & Linux that would be:
    ```
    /home/<username>/.Natron
    ```
- All the paths indicated by the NATRON_PLUGIN_PATH environment variable. This environment variable should contain the separator ; between each path, such as:
  ```
  /home/<username>/NatronPluginsA;/home/<username>/NatronPluginsB
  ```
- The user extra search paths in the Plug-ins tab of the Preferences of Natron.

If the setting “Prefer bundled plug-ins over system-wide plug-ins” is checked in the preferences then Natron will first look into the bundled plug-ins before checking the standard location. Otherwise, Natron will check bundled plug-ins as the last location.

Note that if the “User bundled plug-ins” setting in the preferences is unchecked, Natron will not attempt to load any bundled plug-ins.

2.2 Python Auto-declared variables

A lot of Python variables are pre-declared by Natron upon the creation of specific objects. This applies currently to the following objects:
The idea is that it is simpler to access a simple variable like this:

```python
node = app1.Blur1
```

rather than call a bunch of functions such as:

```python
node = app1.getNode("app1.Blur1")
```

To achieve this, auto-declared objects must be named with a correct syntax in a python script. For instance, the following variable would not work in Python:

```python
my variable = 2
```

```
File "<stdin>", line 1
my variable = 2
    ^
SyntaxError: invalid syntax
```

But the following would work:

```python
myVariable = 2
```

To overcome this issue, all auto-declared variables in Natron have 2 names:

1. A script-name: The name that will be used to auto-declare the variable to Python. This name cannot be changed and is set once by Natron the first time the object is created. This name contains only alphabetic characters and does not start with a digit.

2. A label: The label is what is displayed on the graphical user interface. For example the node label is visible in the node graph. This label can contain any character without any restriction.

Basically there can only ever be one instance of an object with a script-name (so it is identified uniquely) but several instances of this object could have the same label.

Generally when calling a function which takes an object name in parameter, you pass it always the script-name of the object. See for example `getParam(name)`.

### 2.2.1 Knowing the script-name of a node:

The script-name of a node is visible in the graphical-user interface in the tool-tip of the widget to edit the label of the node:

For children nodes (like tracks) you can access their script-name from the table of the Tracker node:

In command-line mode or via the Script Editor, you can also get the script-name of the node with the `getScriptName()` function of the Effect class.
2.2.2 Knowing the script-name of a parameter:

In the settings panel of a node, the script-name of a parameter is visible in **bold** in the tooltip displayed when hovering a parameter with the mouse:

In command-line mode or via the *Script Editor* you can also get the script-name of the parameter with the `getScriptName()` function of the *Param* class.

2.2.3 Knowing the script-name of an item of a Roto node:

In the settings panel of a roto node, the script-name of an item is visible in the *Script* column of the table:

In command-line mode or via the *Script Editor* you can also get the script-name of an item with the `getScriptName()` function of the *ItemBase* class.

2.2.4 Knowing the script-name of a tab-widget:

The script-name of a pane can be seen on the graphical user interface by hovering the mouse over the “Manage layout” button (in the top left hand corner of a pane).

2.2.5 Knowing the script-name of a viewer:

The script-name of a viewer is the *script-name* of the node associated to it, e.g:

```
appl panel Viewer1
```
2.2.6 Knowing the script-name of a PyPanel:

The script-name of a PyPanel can be retrieved with the `getPanelScriptName()` function of the class.

2.3 Start-up scripts

On start-up Natron will run different start-up scripts to let you setup anything like callbacks, menus, etc...

There are 2 different initialization scripts that Natron will look for in the search paths.

- **init.py**
  
  This script is always run and should only initialize non-GUI stuff. You may not use it to initialize e.g new menus or windows. Generally this is a good place to initialize all the callbacks that you may want to use in your projects.

- **initGui.py**
  
  This script is only run in GUI mode (that is with the user interface). It should initialize all gui-specific stuff like new menus or windows.

All the scripts with the above name found in the search paths will be run in the order of the search paths.

**Warning:** This is important that the 2 scripts above are named `init.py` and `initGui.py` otherwise they will not be loaded.

**Warning:** These scripts are run well before any application instance (i.e: project) is created. You should therefore not run any function directly that might rely on the `app` variable (or `app1`, etc...). However you’re free to define classes and functions that may rely on these variable being declared, but that will be called only later on, when a project will actually be created.

2.4 Examples

2.4.1 initGui.py

A complete example of a `initGui.py` can be found [here](#).

2.4.2 init.py

Here is an example of a `init.py` script, featuring:

- Formats addition to the project
- Modifications of the default values of parameters for nodes
- PyPlug search paths modifications

/* This Source Code Form is subject to the terms of the Mozilla Public License, v. 2.0. If a copy of the MPL was not distributed with this file, You can obtain one at http://mozilla.org/MPL/2.0/. */

#This Source Code Form is subject to the terms of the Mozilla Public License, v. 2.0. If a copy of the MPL was not distributed with this file, You can obtain one at http://mozilla.org/MPL/2.0/. */

#Created by Alexandre GAUTHIER-FOICHAT on 01/27/2015.
#To import the variable "natron"

```python
import NatronEngine
```

def addFormats(app):
    app.addFormat("720p 1280x720 1.0")
    app.addFormat("2k_185 2048x1108 1.0")

def afterNodeCreatedCallback(thisNode, app, userEdited):
    # Turn-off the Clamp black for new grade nodes
    if thisNode.getPluginID() == "net.sf.openfx.GradePlugin":
        thisNode.clampBlack.setDefaultValue(False)
    # Set the blur size to (3,3) upon creation
    elif thisNode.getPluginID() == "net.sf.cimg.CImgBlur":
        thisNode.size.setDefaultValue(3,0)
        thisNode.size.setDefaultValue(3,1)

    # This will set the After Node Created callback on the project to tweak default values for parameters
    def setNodeDefaults(app):
        app.afterNodeCreated.set("afterNodeCreatedCallback")

    def myCallback(app):
        addFormats(app)
        setNodeDefaults(app)

    # Set the After Project Created/Loaded callbacks
    NatronEngine.natron.setOnProjectCreatedCallback("myCallback")
    NatronEngine.natron.setOnProjectLoadedCallback("myCallback")

    # Add this path to the Natron search paths so that our PyPlug can be found.
    # Note that we also set this from the NATRON_PLUGIN_PATH environment variable
    # or even in the Preferences panel, Plug-ins tab, with the "Pyplugs search path"
    NatronEngine.natron.appendToNatronPath("/Library/Natron/PyPlugs")

2.5 Natron in command-line

Natron has 3 different execution modes:

- The execution of Natron projects (.ntp)
- The execution of Python scripts that contain commands for Natron
- An interpreter mode where commands can be given directly to the Python interpreter

2.5.1 General options:

[--background] or [-b] enables background mode rendering. No graphical interface will be shown. When using `NatronRenderer` or the -t option this argument is implicit and you don’t need to use it. If using Natron and this
option is not specified then it will load the project as if opened from the file menu.

[--interpreter] or [-t] [optional] <python script file path> enables Python interpreter mode. Python commands can be given to the interpreter and executed on the fly. An optional Python script filename can be specified to source a script before the interpreter is made accessible. Note that Natron will not start rendering any Write node of the sourced script, you must explicitly start it. NatronRenderer and Natron will do the same thing in this mode, only the init.py script will be loaded.

2.5.2 Options for the execution of Natron projects:

Natron <project file path>

[--writer] or [-w] <Writer node script name> [optional] <filename> [optional] <frameRange> specifies a Write node to render. When in background mode, the renderer will only try to render with the node script name following this argument. If no such node exists in the project file, the process will abort. Note that if you don’t pass the --writer argument, it will try to start rendering with all the writers in the project.

After the writer node script name you can pass an optional output filename and pass an optional frame range in the format firstFrame-lastFrame (e.g: 10-40).

**Warning:** You may only specify absolute file paths with the -i option, things like:

NatronRenderer -i MyReader ~/pictures.png -w MyWriter rendered###.exr

would not work. This would work on the other hand:

NatronRenderer -i MyReader /Users/me/Images/pictures.png -w MyWriter /Users/me/Images/rendered###.exr

Note that several -w options can be set to specify multiple Write nodes to render.

**Warning:** Note that if specified, then the frame range will be the same for all Write nodes that will render.

[ –reader ]* or **[-i] <reader node script name> <filename> : Specify the input file/sequence/video to load for the given Reader node. If the specified reader node cannot be found, the process will abort.

**Warning:** You may only specify absolute file paths with the -i option, things like:

NatronRenderer -i MyReader ~/pictures.png -w MyWriter rendered###.exr

would not work. This would work on the other hand:

NatronRenderer -i MyReader /Users/me/Images/pictures.png -w MyWriter /Users/me/Images/rendered###.exr

[--onload] or [-l] <python script file path> specifies a Python script to be executed after a project is created or loaded. Note that this will be executed in GUI mode or with NatronRenderer and it will be executed after any Python function set to the callback onProjectLoaded or onProjectCreated. The same rules apply to this script as the rules below on the execution of Python scripts.

[ –render-stats] or [-s] Enables render statistics that will be produced for each frame in form of a file located next to the image produced by the Writer node, with the same name and a .stats.txt extension.

The breakdown contains informations about each nodes, render times etc... This option is useful for debugging purposes or to control that a render is working correctly. **Please note** that it does not work when writing video files.

Some examples of usage of the tool:

2.5. Natron in command-line
Example of a script passed to –onload:

```python
import NatronEngine

#Create a writer when loading/creating a project
writer = app.createNode("fr.inria.openfx.WriteOIIO")
```

### 2.5.3 Options for the execution of Python scripts:

Natron <Python script path>

Note that the following does not apply if the -t option was given.

The script argument can either be the script of a Group that was exported from the graphical user interface or an exported project or even a script written by hand.

When executing a script, Natron first looks for a function with the following signature:

```python
def createInstance(app,group):
```

If this function is found, the script will be imported as a module and it will be executed.

**Warning:** Note that when imported, the script will not have access to any external variable declared by Natron except the variable passed to the createInstance function.

If this function is not found the whole content of the script will be interpreted as though it were given to Python natively.

**Note:** In this case the script can have access to the external variables declared by Natron.

Either cases, the “app” variable will always be defined and pointing to the correct application instance. Note that if you are using Natron in GUI mode, it will source the script before creating the graphical user interface and will not start rendering. When in command-line mode (-b option or NatronRenderer) you must specify the nodes to render. If nothing is specified, all Write nodes that were created in the Python script will be rendered.

You can render specific Write nodes either with the -w option as described above or with the following option:

[-output] or [-o] <filename> <frameRange> specifies an Output node in the script that should be replaced with a Write node.

The option looks for a node named Output1 in the script and will replace it by a Write node much like when creating a Write node in the user interface.

A filename must be specified, it is the filename of the output files to render. Also a frame range must be specified if it was not specified earlier.
This option can also be used to render out multiple Output nodes, in which case it has to be used like this:

`[-output1]` or `[-o1]` looks for a node named `Output1`

`[-output2]` or `[-o2]` looks for a node named `Output2`

`-c` or `[-cmd]` “PythonCommand” : Execute custom Python code passed as a script prior to executing the Python script passed in parameter. This option may be used multiple times and each python command will be executed in the order they were given to the command-line.

Some examples of usage of the tool:

```
Natron /Users/Me/MyNatronScripts/MyScript.py
Natron -b -w MyWriter /Users/Me/MyNatronScripts/MyScript.py
NatronRenderer -w MyWriter /Users/Me/MyNatronScripts/MyScript.py
NatronRenderer -o /FastDisk/Pictures/sequence###.exr 1-100 /Users/Me/MyNatronScripts/MyScript.py
NatronRenderer -o1 /FastDisk/Pictures/sequence###.exr -o2 /FastDisk/Pictures/test###.exr 1-100 /Users/Me/MyNatronScripts/MyScript.py
NatronRenderer -w MyWriter -o /FastDisk/Pictures/sequence###.exr 1-100 /Users/Me/MyNatronScripts/MyScript.py
NatronRenderer -w MyWriter /FastDisk/Pictures/sequence.mov 1-100 /Users/Me/MyNatronScripts/MyScript.py -e "print "Now executing MyScript.py...\""
```

2.5.4 Options for the execution of the interpreter mode:

```
Natron -t [optional] <Python script path>
```

Natron will first source the script passed in argument, if any and then return control to the user. In this mode, the user can freely input Python commands that will be interpreted by the Python interpreter shipped with Natron.

Some examples of usage of the tool:

```
Natron -t
NatronRenderer -t
NatronRenderer -t /Users/Me/MyNatronScripts/MyScript.py
```

2.6 Example

A typical example would be to convert an input image sequence to another format. There are multiple ways to do it from the command-line in Natron and we are going to show them all:

- Passing a .ntp file to the command line and passing the correct arguments
- Passing a Python script file to the command-line to setup the graph and render

2.6.1 With a Natron project (.ntp) file

2.6.2 With a Python script file

We would write a customized Python script that we pass to the command-line:
```python
reader = app.createNode("fr.inria.openfx.ReadOIIIO")
writer = app.createNode("fr.inria.openfx.WriteFFmpeg")

# The node will be accessible via app.MyWriter after this call
# We do this so that we can reference it from the command-line arguments
writer.setScriptName("MyWriter")

# The node will be accessible via app.MyReader after this call
reader.setScriptName("MyReader")

# Set the input sequence value
fileparam = reader.getParam("filename")
fileparam.setValue("/Users/Toto/Sequences/Sequence__####.exr");

# Set the output video
outfileparam = writer.getParam("filename")
outfileparam.setValue("/Users/Toto/Sequences/Sequence.mov")

# Set the format type parameter of the Write node to Input Stream Format so that the video
# is written to the size of the input images and not to the size of the project
formatType = writer.getParam("formatType")
formatType.setValue(0)

# Connect the Writer to the Reader
writer.connectInput(0, reader)

# When using Natron (Gui) then the render must explicitly be requested.
# Otherwise if using NatronRenderer or Natron ~b the render will be automatically started
# using the command-line arguments

# To use with Natron (Gui) to start render
# app.render(writer, 10, 20)
```

To launch this script in the background, you can do it like this:
```
NatronRenderer /path/to/myStartupScript.py -w MyWriter 10-20
```

For now the output filename and the input sequence are static and would need to be changed by hand to execute this script on another sequence.

We can customize the Reader filename and Writer filename parameters using the command-line arguments:
```
NatronRenderer /path/to/myStartupScript.py -i MyReader /Users/Toto/Sequences/AnotherSequence__####.exr -w MyWriter /Users/Toto/Sequences/mySequence.mov 10-20
```

Let’s imagine that now we would need to also set the frame-rate of the video in output and we would need it to vary for each different sequence we are going to transcode. This is for the sake of this example, you could also need to modify other parameters in a real use-case.

Since the fps cannot be specified from the command-line arguments, we could do it in Python with:
```
MyWriter.getParam("fps").set(48)
```

And change the value in the Python script for each call to the command-line, but that would require manual intervention.

That’s where another option from the command-line comes into play: the `c` option (or `-cmd`): It allows to pass custom Python code in form of a string that will be executed before the actual script:

```
MyWriter.getParam("fps").set(48)
```
To set the fps from the command-line we could do as such now:

NatronRenderer /path/to/myStartupScript.py -c "fpsValue=60" -w MyWriter 10-20

Which would require the following modifications to the Python script:

MyWriter.getParam("fps").set(fpsValue)

We could also set the same way the Reader and Writer file names:

NatronRenderer /path/to/myStartupScript.py -c "fpsValue=60; readFileName="/Users/Toto/Sequences/AnotherSequence__####.exr"; writeFileName="/Users/Toto/Sequences/mySequence.mov""

And modify the Python script to take into account the new readFileName and writeFileName parameters:

```python
... fileparam.setValue(readFileName) ... outfileparam.setValue(writeFileName)
```

The -c option can be given multiple times to the command-line and each command passed will be executed once, in the order they were given.

### 2.6.3 With a Natron project file:

Let’s suppose the user already setup the project via the GUI as such:

MyReader—>MyWriter

We can then launch the render from the command-line this way:

NatronRenderer /path/to/myProject.ntp -w MyWriter 10-20

We can customize the Reader filename and Writer filename parameters using the command-line arguments:

NatronRenderer /path/to/myProject.ntp -i MyReader /Users/Toto/Sequences/AnotherSequence__####.exr -w MyWriter /Users/Toto/Sequences/mySequence.mov 10-20

### 2.7 Objects hierarchy Overview

When running Natron, several important objects are created automatically and interact at different levels of the application.

The main object in Natron is the PyCoreApplication class which represents the unique instance of the process. It is available directly via the variable:

```
natron
```

Basically it handles all application-wide informations about plug-ins, environment, application settings...

but also can hold one or multiple application instance which are made available to the global variables via the following variables:

```
app1 # References the first instance of the application (the first opened project)
app2 # The second project
...
```

Note that in background command-line mode, there would always be a single opened project so Natron does the following assignment for you:
app = appl

**Warning:** Note that when running scripts in the *Script Editor*, the application is running in GUI mode hence the `app` variable is not declared.

The **App** object is responsible for managing all informations relative to a project. This includes all the **nodes**, project settings and render controls. See *this section* to create and control nodes.

Each node can have **parameters** which are the controls found in the settings panel of the node.

The same **Param** class is also used for the project settings and the application settings (preferences).
Getting started

This section covers basic functionalities a Python script can do in Natron.

3.1 Creating and controlling nodes

3.1.1 Creating a new node:

To create a node in Natron, you would do so using the app instance via the function `createNode(pluginId, majorVersion, group)` like this:

```python
app1.createNode("fr.inria.openfx.ReadOIIO")
```

In this line we specify that we want the first opened project to create a node instantiating the plug-in ReadOIIO. Note that if we were in background mode we could just write the following which would be equivalent:

```python
app.createNode("fr.inria.openfx.ReadOIIO")
```

Since in command-line there is only a single project opened, Natron does the following assignment:

```python
app = app1
```

If we were to create the node into a specific group, we would do so like this:

```python
group = app.createNode("fr.inria.built-in.Group")
reader = app.createNode("fr.inria.openfx.ReadOIIO", -1, group)
```

Note that when passed the number -1, it specifies that we want to load the highest version of the plug-in found. This version parameter can be useful to load for example a specific version of a plug-in.

The `pluginID` passed to this function is a unique ID for each plug-in. If 2 plug-ins were to have the same ID, then Natron will create separate entries for each version.

You can query all plug-ins available in Natron this way:

```python
allPlugins = natron.getPluginIDs()
```

You can also filter out plug-ins that contain only a given `filter` name:

```python
# Returns only plugin IDs containing ".inria" in it
filteredPlugins = natron.getPluginIDs(".inria.")
```
In the user interface, the plug-in ID can be found when pressing the ? button located in the top right-hand corner of the settings panel:

![User Interface with ? button](image)

### 3.1.2 Connecting a node to other nodes:

To connect a node to the input of another node you can use the `connectInput(inputNumber, input)` function. The `inputNumber` is a 0-based index specifying the input on which the function should connect the given input `Effect`. You can query the input name at a specific index with the following function:

```python
print(node.getInputLabel(i))
```

Here is a small example where we would create 3 nodes and connect them together:

```python
# Create a write node
writer = app.createNode("fr.inria.openfx.WriteOIIO")

# Create a blur
blur = app.createNode("net.sf.cimg.CImgBlur")

# Create a read node
reader = app.createNode("fr.inria.openfx.ReadOIIO")

# Connect the write node to the blur
writer.connectInput(0, blur)

# Connect the blur to the read node
blur.connectInput(0, reader)
```
Note that the following script would do the same since nodes are auto-declared variables.

```python
node = app.createNode("fr.inria.openfx.WriteOIIO")
print(node.getScriptName()) # prints WriteOIIO1

# The write node is now available via its script name app.WriteOIIO1

node = app.createNode("net.sf.cimg.CImgBlur")
print(node.getScriptName()) # prints CImgBlur1

# The blur node is now available via its script name app.BlurCImg1

node = app.createNode("fr.inria.openfx.ReadOIIO")
print(node.getScriptName()) # prints ReadOIIO1

# The ReadOIIO node is now available via its script name app.ReadOIIO1

app.WriteOIIO1.connectInput(0, app.BlurCImg1)
app.BlurCImg1.connectInput(0, app.ReadOIIO1)
```

Note that not all connections are possible, and sometimes it may fail for some reasons explained in the documentation of the `connectInput(inputNumber, input)` function.

You should then check for errors this way:

```python
if not app.WriteOIIO1.connectInput(0, app.BlurCImg1):
    # Handle errors
```

You can check beforehand whether a subsequent `connectInput` call would succeed or not by calling the `canConnectInput(inputNumber, input)` which basically checks whether is is okay to do the connection or not. You can then safely write the following instructions:

3.1. Creating and controlling nodes
if app.WriteOIIO1.canConnectInput(0, app.BlurCImg1):
    app.WriteOIIO1.connectInput(0, app.BlurCImg1)
else:
    # Handle errors

Note that internally `connectInput` calls `canConnectInput` to validate whether the connection is possible.

To disconnect an existing connection, you can use the `disconnectInput(inputNumber)` function.

### 3.2 Controlling parameters

#### 3.2.1 Accessing a node’s parameters:

As for nodes, **parameters are auto-declared objects**. You can access an existing parameter of a node by its **script-name**

```python
app.BlurCImg1.size
```

Note that you can also access a parameter with the `getParam(scriptName)` function:

```python
param = app.BlurCImg1.getParam("size")
```

but you should not ever need it because Natron pre-declared all variables for you.

The **script-name** of a parameter is visible in the user interface when hovering the parameter in the settings panel with the mouse. This is the name in **bold**:

![Parameter Script-Name](image)

#### 3.2.2 Parameters type:

Each parameter has a type to represent internally different data-types, here is a list of all existing parameters:

- **IntParam** to store 1-dimensional integers
- **Int2DParam** to store 2-dimensional integers
- **Int3DParam** to store 3-dimensional integers
- **DoubleParam** to store 1-dimensional floating point
- **Double2DParam** to store 2-dimensional floating point
- **Double3DParam** to store 3-dimensional floating point
- **BooleanParam** to store 1-dimensional boolean (checkbox)
- **ButtonParam** to add a push-button
- **ChoiceParam** a 1-dimensional drop-down (combobox)
• **StringParam** to store a 1-dimensional string
• **FileParam** to specify an input-file
• **OutputFileParam** to specify an output-file param
• **PathParam** to specify a path to a single or multiple directories
• **ParametricParam** to store N-dimensional parametric curves
• **GroupParam** to graphically gather parameters under a group
• **PageParam** to store parameters into a page

### 3.2.3 Retrieving a parameter’s value:

Since each underlying type is different for parameters, each sub-class has its own version of the functions.

To get the value of the parameter at the timeline’s current time, call the `get()` or `getValue()` function.

If the parameter is animated and you want to retrieve its value at a specific time on the timeline, you would use the `get(frame)` or `getValueAtTime(frame, dimension)` function.

Note that when animated and the given `frame` time is not a time at which a keyframe exists, Natron will interpolate the value of the parameter between surrounding keyframes with the interpolation filter selected (by default it is `smooth`).

### 3.2.4 Modifying a parameter’s value:

You would set the parameter value by calling the `set(value)` or `setValue(value)` function. If the parameter is animated (= has 1 or more keyframe) then calling this function would create (or modify) a keyframe at the timeline’s current time.

To add a new keyframe the `set(value, frame)` or `setValueAtTime(value, frame, dimension)` function can be used.

To remove a keyframe you can use the `deleteValueAtTime(frame, dimension)` function. If you want to remove all the animation on the parameter at a given `dimension`, use the `removeAnimation(dimension)` function.

**Warning:** Note that the dimension is a 0-based index referring to the dimension on which to operate. For instance a **Double2DParam** has 2 dimensions x and y. To set a value on x you would use `dimension = 0`, to set a value on y you would use `dimension = 1`.

### 3.2.5 Controlling other properties of parameters:

See the documentation for the **Param** class for a detailed explanation of other properties and how they affect the parameter.

### 3.2.6 Creating new parameters:

In Natron, the user has the possibility to add new parameters, called **User parameters**. They are pretty much the same than the parameters defined by the underlying OpenFX plug-in itself.

In the Python API, to create a new user parameter, you would need to call one of the `createXParam(name, label, ...)` of the **Effect** class.
These parameters can have their default values and properties changed as explained in the documentation page of the Param class.

To remove a user created parameter you would need to call the `removeParam(param)` function of the Effect class.

**Warning:** Only user parameters can be removed. Removing parameters defined by the OpenFX plug-in will not work.

### 3.3 Parameters expressions

The value of a parameter can be set by Python expressions. An expression is a line of code that can either reference the value of other parameters or apply mathematical functions to the current value.

The expression will be executed every times the value of the parameter is fetched from a call to `getValue(dimension)` or `get()`.

**Warning:** Note that when an expression is active, all animation is ignored and only the result of the expression will be used to return the value of the parameter.

When executing an expression, the expression itself has a scope. The scope of the expression defines all nodes and parameters that are possible to use in the expression in order to produce the output value.

Any node in the scope can have a variable declared corresponding to its script-name:

```
Blur1
```

You would then access a parameter of Blur1 also by its script-name:

```
Blur1.size
```

```
Group1.Blur1.size
```

**Warning:** Referencing the value of the same parameter which expression is being edited can lead to an infinite recursion which Python should warn you about.

In fact this is exactly like referencing auto-declared nodes via the Script Editor except that the app prefix was removed for nodes in the scope.

See this section to learn how to determine the `script-name` of a node.

See this section to learn how to determine the `script-name` of a parameter.

By default a parameter’s expression can only refer to parameters of nodes belonging to the same Group, or to parameters belonging to the parent Group node.

Parameters of a Group node are also granted in the scope the parameters contained within that group.

For instance if your graph hierarchy looks like this:

```
Read1
Blur1
Group1/
    Input1
    Blur1
    Convolve1
    Roto1
```
A parameter of Read1 would be able to reference any parameter of Read1, Blur1, Group1, Viewer1 but could not reference any parameter of the nodes within Group1.

Similarly, a parameter of Group1.Blur1 would be able to reference any parameter of Group1, Group1.Input1, Group1.Blur1, Group1.Convolve1, Group1.Roto1, Group1.Output1 but would not be able to reference any top-level node (Read1, Blur1, Viewer1) except the Group1 node.

A parameter of Group1 would on the other hand be able to reference any parameter in top-level nodes and in the nodes of Group1.

The scope was introduced to deal with problems where the user would write expressions referencing parameters that would probably no longer be referable when loading the script again in another project.

**Warning:** Note that you would still be able to reach any node or parameter in the project using the `app1` (or `app` prefix in command-line mode) but is not recommended to do so:

```python
app1.Blur1.size
```

All functions available in the Python API are made available to expressions. Also for convenience the `math` Python module has been made available by default to expressions.

### 3.3.1 Setting an expression:

To create an expression from the user interface, right click a parameter and choose *Set Expression*...

Note that for multi-dimensional parameters such as `ColorParam`, the *Set Expression*... entry will only set an expression for the right-clicked dimension.

The *Set Expression (all dimensions)* entry will on the other hand set the same expression on all dimensions of the parameter at once.

A dialog will open where you can write the expression:

By default you do not have to assign any variable as the result of the expression, Natron will do it by itself.
# Expression for Blur1.size

```
Transform1.translate.get[0]
```

# Will be expanded automatically by Natron to

```
ret = Transform1.translate.get[0]
```

However if you were to write an expression that spans over multiple lines you would need to specifically set the `ret` variable yourself and toggle-on the `multi-line` button:

```
a = acos(Transform1.translate.get[0])
b = sin(Transform1.rotate.get())
ret = (tan(a * b) / pi) + Group1.customParam.get()
```

You can also set an expression from a script using the `setExpression(expr, hasRetVariable, dimension)` function of `AnimatedParam`.

### 3.3.2 Writing an expression:

For convenience the following variables have been declared to Python when executing the expression:

- **thisNode**: It references the node holding the parameter being edited
- **thisGroup**: It references the group containing `thisNode`
- **thisParam**: It references the param being edited
- **dimension**: Defined only for multi-dimensional parameters, it indicates the dimension (0-based index) of the parameter on which the expression has effect.
- **frame**: It references the current time on the timeline
3.3. Parameters expressions
• The app variable will be set so it points to the correct application instance.

To reference the value of another parameter use the `get()` function which retrieves the value of the parameter at the current timeline’s time. If the parameter is multi-dimensional, you need to use the subscript operator to retrieve the value of a particular dimension.

The `getValue(dimension)` does the same thing but takes a `dimension` parameter to retrieve the value of the parameter at a specific `dimension`. The following is equivalent:

```python
ColorCorrect1.MasterSaturation.get()[dimension]
ColorCorrect1.MasterSaturation.getValue(dimension)
```

Note that for 1-dimensional parameter, the `:func:` `get()<>` function cannot be used with subscript, e.g:

```python
Blur1.size.get()
```

To retrieve the value of the parameter at a specific `frame` because the parameter is animated, you can use the `get(frame)` function.

Again the `getValueAtTime(frame,dimension)` does the same thing but takes a `dimension` parameter to retrieve the value of the parameter at a specific `dimension`. The following lines are equivalent to the 2 lines above:

```python
ColorCorrect1.MasterSaturation.get(frame)[dimension]
ColorCorrect1.MasterSaturation.getValueAtTime(frame,dimension)
```

We ask for the value of the `MasterSaturation` parameter of the `ColorCorrect1` node its value at the current `frame` and at the current `dimension`, which is the same as calling the `get()` function without a `frame` in parameter.

### 3.3.3 Copying another parameter through expressions:

If we want the value of the parameter `size` of the node `BlurCImg1` to copy the parameter `mix` of the node `DilateCImg1`, we would set the following expression on the `size` parameter of the node `BlurCImg1` (see setting an expression):

```python
DilateCImg1.mix.get()
```

If mix has an animation and we wanted to get the value of the mix at the previous `frame`, the following code would work:

```python
DilateCImg1.mix.get(frame - 1)
```

Note that when choosing the Link to... option when right-clicking a parameter, Natron writes automatically an expression to copy the parameter to link to for you.

### 3.3.4 Using random in expressions:

Sometimes it might be useful to add a random generator to produce noise on a value. However the noise produced must be reproducible such that when rendering multiple times the same frame or when loading the project again it would use the same value.

We had to add a specific random function in Natron that takes into account the state of a parameter and the current time on the timeline as a seed function to random.

**Warning:** We advise against using the functions provided by the module random.py of the Python standard library, as the values produced by these functions will not be reproducible between 2 runs of Natron for the same project.
The Natron specific random functions are exposed in the :ref:`Param<Param>` class.

When executing an expression, Natron pre-declares the `random()` function so that you do not have to do stuff like:

```python
thisParam.random()
```

Instead you can just type the following in your expression:

```python
myOtherNode.myOtherNodeParam.get() * random()
```

The `random(min = 0.,max = 1.)` function also takes 2 optional arguments indicating the range into which the return value should fall in. The range is defined by `[min,max]`. When using this function in your expression, make sure to add a `.` after the numbers so that the interpreter understands your using `random(min,max)` and not `random(seed)`

```python
#Returns a random floating point value in the range [1., 10.]
random(1.,10.)
```

For integers, use the `randomInt(min,max)` function instead:

```python
#Returns a random integer in the range [1,100]
randomInt(1,100)
```

```python
#Using the randomInt function with a given seed
seed = 5
randomInt(seed)
```

### 3.3.5 Expressions persistence

If you were to write a group plug-in and then want to have your expressions persist when your group will be instantiated, it is important to prefix the name of the nodes you reference in your expression by the `thisGroup` prefix. Without it, Natron thinks you’re referencing a top-level node, i.e: a node which belongs to the main node-graph, however, since you’re using a group, all your nodes are no longer top-level and the expression will fail.

### 3.3.6 Examples

**Setting the label of a Node so it displays the value of a parameter on the node-graph:**

For instance, we may want to have on the shuffle node, the values of the output RGBA channels so we don’t have to open the settings panel to understand what the node is doing.

To do so, we set an expression on the “Label” parameter located in the “Node” tab of the settings panel.

Set the following expression on the parameter

```python
thisNode.outputR.getOption(thisNode.outputR.get()) + "\n" + thisNode.outputG.getOption(thisNode.outputG.get()) + "\n" + thisNode.outputB.getOption(thisNode.outputB.get()) + "\n" + thisNode.outputA.getOption(thisNode.outputA.get())
```

**Generating custom animation for motion editing:**

In this example we will demonstrate how to perform Loop,Negate and Reverse effects on an animation even though this is already available as a preset in Natron.

To do be able to do this we make use of the `curve(frame,dimension)` function of the `Param` class. This function returns the value of the animation curve (of the given dimension) at the given time.

If we were to write the following expression:
curve(frame)

The result would be exactly the animation curve of the parameter.

On the other hand if we write:

curve(-frame)

We have just reversed the curve, meaning that the actual result at the frame F will be in fact the value of the curve at the frame -F.

In the same way we can apply a negate effect:

-\text{curve(frame)}

The loop effect is a bit more complicated and needs to have a frame-range in parameter:

\begin{verbatim}
firstFrame = 0
lastFrame = 10
curve(((frame - firstFrame) \% (lastFrame - firstFrame + 1)) + firstFrame)
\end{verbatim}

3.4 Working with groups

Groups in Natron are a complete sub-nodegraph into which the user can manage nodes exactly like in the main nodegraph, but everything in that sub-group will be referenced as 1 node in the hierarchy above, e.g:

A group can be created like any other node in Natron and by default embeds already 2 nodes: The Output node and one Input node.
The **Output** node is used to reference what would be the output of the internal graph of the group. In Natron, a node has necessarily a single output, hence if you add several **Output** nodes to a group, only the first **Output** node will be taken into account.

Note that you can also add **Output** nodes to the top-level graph of Natron (the main Node Graph). They are useful if you need to export your project as a group.

When used in the top-level graph, there can be multiple **Output** nodes, which can then be used when launching Natron from the command-line to render the script, e.g:

```
NatronRenderer -o1 /FastDisk/Pictures/sequence###.exr -o2 /FastDisk/Pictures/test###.exr 1-100 /Users/Me/MyNatronScripts/MyScript.py
```

Where each argument `o1`, `o2` expand respectively the nodes **Output1** and **Output2**.

**Warning:** You should never attempt to change the script name of output nodes, otherwise Natron has no way to match the given command line arguments to the output nodes. In fact Natron will completely ignore your request if you explicitly try to set the script name of an **Output** node.

The **Input** node is not necessarily unique and represents 1 input arrow of the group node. You can also specify in the settings panel of the **Input** node whether this input should be considered as a mask or whether it should be optional.

**Note:** Note that the OpenFX standard specifies that Mask inputs must be optionals so when checking the mask parameter, this will automatically check the optional parameter.

You can freely rename an **Input** node, effectively changing the label attached to the arrow on the group node.
3.4.1 Parameters expressions and groups

A common task is to add parameters to the group node itself which directly interact to nodes parameters used internally by this group.

You can add a new parameter to the group node by clicking the “Settings and presets” button and clicking “Manage user parameters...”:

A dialog will popup on which you can manage all the parameters that you added. By default a page is added automatically that will contain user parameters.

To create a new parameter, click the add button, this brings up a new dialog:

In this dialog you can configure all the properties of the parameter exactly like you would do using the Python API. Once created, the new parameter can be found in the “User” page of the settings panel:

We can then set for instance an expression on the internal blur size parameter to copy the value of the blur size parameter we just added to the group node:

The expression is now visible in a green-ish color on the parameter in the settings panel and the node on the node-graph has a green “E” indicator.

3.4.2 Exporting a group

Once your group is setup correctly, you can export it as a Python script that Natron will generate automatically. We call them PyPlugs.
Script name: blurSize
Label: Blur size
Type: Integer
Animates: X
Render on change: X
Tooltip: This is the size of the blur produced by this group
Minimum: 0.0
Maximum: 100.0
Page: userNatron

Blur size: 0
3.4. Working with groups
Chapter 3. Getting started
To do so, click the Export as Python plug-in button in the “Node” page of the settings panel of the Group node.

Exporting a group as a plug-in, means that it will create a Python script that will be able to re-create the group entirely and that will be loaded on startup like any other plug-in. That means that the group will also appear in the left toolbar of Natron and can potentially have an icon too.

The Label is the name of the plug-in as it will appear in the user interface. It should not contain spaces or non Python friendly characters as it is going to be used as variable names in several places.

The Grouping is the tool-button under which the plug-in should appear. It accepts sub-menus notation like this: “Inria/StereoGroups”

The Icon relative path is the filepath to an image which should be used as icon for the plug-in. Note that it is a relative path to the location of the python script.

The directory is the location where the script should be written to. For the plug-in to be loaded by Natron, it should be in its search-paths hence if you select a directory that is not yet in the search-paths, it will prompt you to add it.

**Note:** A re-launch of Natron is required to re-scan the plug-ins and build the tool menus

Once restarted, the plug-in should now appear in the user interface and even in the tab menu of the node-graph:

**Note:** The plug-in ID of the group will be exactly the same as the Label you picked when exporting it, hence when creating a node using the group from a Python script, you would do so:

---

### 3.4. Working with groups

---
3.4. Working with groups
app.createNode("MyBlurGroup")

If several plug-ins have the same pluginID, Natron will then sort plug-ins by version.

The version of a plug-in by default when exporting it via Natron is 1.

**Warning:** If 2 plug-ins happen to have the same pluginID and version, Natron will then load the first one found in the search paths.

To change the pluginID and version of your group plug-in, you must implement the 2 following functions in the python script of the group:

```python
# This function should return an int specifying the version of the plug-in
# If not implemented, Natron will use 1 by default
def getVersion():
    return VERSION

# This function should return a string specifying the ID of the plug-in, for example
# "fr.inria.groups.customBlur"
# If not implemented, Natron will use the label as a pluginID
def getPluginID():
    return UNIQUE_ID
```

### 3.4.3 Exporting a project as group

Similarly, Natron allows you to export the top-level node-graph as a Python group plug-in. From the “File” menu, select “Export project as group”.

**Warning:** To be exportable, your project should at least contain 1 output node.

**Note:** While this functionality is made for convenience, you should be cautious, as exporting a project containing Readers will probably not work very well in another project or computer because of file-paths no longer pointing to a valid location.

**Warning:** If you were to write a group plug-in and then want to have your expressions persist when your group will be instantiated, it is important to prefix the name of the nodes you reference in your expression by the this-Group. prefix. Without it, Natron thinks you're referencing a top-level node, i.e: a node which belongs to the main node-graph, however, since you're using a group, all your nodes are no longer top-level and the expression will fail.

### 3.4.4 Moving nodes between groups

You can create a group from the selection in Natron by holding CTRL+SHIFT+G. This will effectively move all nodes selected into a new sub-group.

You can also copy/cut/paste in-between groups and projects.

### 3.4.5 Creating a group by hand

You can also write a group plug-in by hand using the **Python API** of Natron.
To work as a plug-in, your script should implement the following functions:

```python
# This function is mandatory and should return the label of the plug-in as visible on the user interface
def getLabel():
    return LABEL

# This function should return an int specifying the version of the plug-in
# If not implemented, Natron will use 1 by default
def getVersion():
    return VERSION

# This function should return a string specifying the ID of the plug-in, for example
# "fr.inria.groups.customBlur"
# If not implemented, Natron will use the label as a pluginID
def getPluginID():
    return UNIQUE_ID

# This function should return a string specifying the relative file path of an image
# file relative to the location of this Python script.
# This function is optional.
def getIconPath():
    return ICON_PATH

# This function is mandatory and should return the plug-in grouping, e.g:
# "Other/Groups"
def getGrouping():
    return GROUPING

# This function is optional and should return a string describing the plug-in to the user.
# This is the text that will show up when the user press the "?" button on the settings panel.
def getDescription():
    return DESCRIPTION

# This function is mandatory and should re-create all the nodes and parameters state
# of the group.
# The group parameter is a group node that has been created by Natron and that will host all
# the internal nodes created by this function.
# The app parameter is for convenience to have access in a generic way to the app object,
# no matter in which project instance your script is invoked in.
def createInstance(app, group):
    ...
```

The Python group plug-ins generated automatically by Natron are a good start to figure out how to write scripts yourself.

**Warning:** Python group plug-ins should avoid using any functionality provided by the `NatronGui` module because it would then break their compatibility when working in command-line background mode. The reason behind this is that the Python module `NatronGui` is not imported in command-line mode because internally it relies on the QtGui library, which may not be present on some render-farms. Attempts to load PyPlugs relaying on the `NatronGui` module would then fail and the rendering would abort.

**Warning:** Note that PyPlugs are imported by Natron which means that the script will not have access to any external variable declared by Natron except the variables passed to the `createInstance` function or the attributes of the modules imported.
3.4.6 Adding hand-written code (callbacks, etc...)

It is common to add hand-written code to a PyPlug. When making changes to the PyPlug from the GUI of Natron, exporting it again will overwrite any change made to the python script of the PyPlug. In order to help development, all hand-written code can be written in a separate script with the same name of the original Python script but ending with `Ext.py`, e.g:

```python
MyPyPlugExt.py
```

This extension script can contain for example the definition of all callbacks used in the PyPlug. When calling the `createInstance(app, group)` function, the PyPlug will call right at the end of the function the `createInstanceExt(app, group)` function. You can define it in your extension script if you want to apply extra steps to the creation of the group. For example you might want to actually set the callbacks on the group:

```python
# This is in MyPyPlugExt.py

def paramChangedCallback(thisParam, thisNode, thisGroup, app, userEdited):
    print thisParam.getScriptName()

def createInstanceExt(app, group):
    # Note that the callback belongs to the PyPlug to so we use it as prefix
    group.onParamChanged.set("MyPyPlug.paramChangedCallback")
```

**Note:** Note that callbacks don’t have to be registered with the extension module prefix but just with the PyPlug’s name prefix since the “from ... import *” statement is made to import the extensions script.

3.4.7 Starting Natron with a script in command line

Natron can be started with a Python script as argument. When used in background mode (i.e: using NatronRenderer or Natron with the option `-b`) Natron will do the following steps:

- Source the script
- If found, run a function with the following signature `createInstance(app, group)`
- Start rendering the specified writer nodes (with the `-w` option) and/or the Output nodes (with the `-o` option)

This allows to pass a group plug-in to Natron and render it easily if needed. Also, it can take arbitrary scripts which are not necessarily group plug-ins.

When Natron is launched in GUI mode but with a Python script in argument, it will do the following steps:

- Source the script
- If found, run a function with the following signature `createInstance(app, group)`

3.4.8 Toolsets

Toolsets in Natron are a predefined set of actions that will be applied to the node-graph. They work exactly like PyPlugs except that no actual group node will be created, only the content of the `createInstance(app, group)` function will be executed.

This useful to create pre-defined graphs, for example like the Split and Join plug-in in the Views menu.

To be recognized as a toolset, your PyPlug must implement the following function:
def getIsToolset():
    return True

Also the group parameter passed to the createInstance(app,group) function will be None because no group node is actually involved.

### 3.5 Using Callbacks

**Callbacks** are functions that are executed after or before a certain event in Natron. They are Python-defined methods that you declare yourself and then register to Natron in a different manner for each callback.

This document describes the signature that your different callbacks must have in order to work for each event. The parameters of your declaration must match exactly the same signature otherwise the function call will not work.

**Warning:** Note that callbacks will be called in background and GUI modes, hence you should wrap all GUI code by the following condition:

```python
if not NatronEngine.natron.isBackground():
    #...do gui stuff
```

### 3.5.1 Callback persistence

If you want your callback to persist 2 runs of Natron; it is necessary that you define it in a script that is loaded by Natron, that is, either the init.py script (or initGui.py if you want it only available in GUI mode) or the script of a Python group plug-in (or its extension script, see here). See this section for more infos.

Here is the list of the different callbacks:

### 3.5.2 The param changed callback

This function is called every times the value of a parameter changes. This callback is available for all objects that can hold parameters, namely:

- **Effect**
- **PyPanel**
- **PyModalDialog**

The signature of the callback used on the Effect is

```python
callback(thisParam, thisNode, thisGroup, app, userEdited)
```

- **thisParam** : This is a Param pointing to the parameter which just had its value changed.
- **thisNode** : This is a Effect pointing to the effect holding thisParam
- **thisGroup** : This is a Effect pointing to the group holding thisNode or app otherwise if the node is in the main node-graph.
- **app** : This variable will be set so it points to the correct application instance.
- **userEdited** : This indicates whether or not the parameter change is due to user interaction (i.e: because the user changed
the value by herself/himself) or due to another parameter changing the value of the parameter via a derivative of the `setValue(value)` function.

For the param changed callback of `PyPanel` and `PyModalDialog` on the other hand, Natron will define a string variable `paramName` indicating the `script-name` of the parameter which just had its value changed. The signature of the callback is then:

```
callback(paramName, app, userEdited)
```

**Note:** The difference between the callbacks on `PyPanel` and `PyModalDialog` and `Effect` is due to technical reasons: mainly because the parameters of the `PyPanel` class and `PyModalDialog` are not declared as attributes of the object.

### 3.5.3 Registering the param changed callback

To register the param changed callback of an `Effect`, you can do so in the settings panel of the node, in the “Node” tab, by entering the name of your Python function:

You can also set the callback directly from the script: The callback is just another parameter of the node, on which you can call `setValue(value)` to set the name of the callback

```python
def myBlurCallback(thisParam, thisNode, thisGroup, app, userEdited):
    ...

app.BlurCImg1.onParamChanged.set("myBlurCallback")
```

**Note:** If the callback is defined in a separate python file, such as the python script of a python group plug-in, then do not forget the module prefix, e.g:
Example

```
# This simple callback just prints a string when the "size" parameter of the BlurCImg node changes

def myBlurCallback(thisParam, thisNode, thisGroup, app, userEdited):
    if thisParam == thisNode.size:
        print("The size of the blur just changed!")
```

```
app.BlurCImg1.onParamChanged.set("myBlurCallback")
```

### 3.5.4 Using the param changed callback for PyModalDialog and PyModalDialog

To register the callback to the object, use the `setParamChangedCallback` function.

The following example is taken from the `initGui.py` script provided as example in this section.

Example

```
#Callback called when a parameter of the player changes
#The variable paramName is declared by Natron; indicating the name of the parameter which just had it changed.

def myPlayerParamChangedCallback(paramName, app, userEdited):
    viewer = app.getViewer("Viewer1")
    if viewer == None:
        return
    if paramName == "previous":
        viewer.seek(viewer.getCurrentFrame() - 1)
    elif paramName == "backward":
        viewer.startBackward()
    elif paramName == "forward":
        viewer.startForward()
    elif paramName == "next":
        viewer.seek(viewer.getCurrentFrame() + 1)
    elif paramName == "stop":
        viewer.pause()
```

```
def createMyPlayer():
    #...
    app.player.setParamChangedCallback("myPlayerParamChangedCallback")
```

### 3.5.5 The After input changed callback

Similarly to the param changed callback, this function is called whenever an input connection of the node is changed. The signature is:
callback(inputIndex, thisNode, thisGroup, app)

**Note:** This function will be called even when loading a project

- **inputIndex**: This is the input which just got connected/disconnected.
  You can fetch the input at the given index with the `getInput(index)` function of the *Effect* class.
- **thisNode**: This is a *Effect* holding the input which just changed
- **thisGroup**: This is a *Effect* pointing to the group holding *thisNode*. Note that it will be declared only if *thisNode* is part of a group.
- **app**: Points to the correct *application instance*.

### 3.5.6 Registering the input changed callback

To register the input changed callback of an *Effect*, you can do so in the settings panel of the node, in the “Node” tab, by entering the name of your Python function:

![Settings panel with input changed callback](image)

You can also set the callback directly from the script: The callback is just another *parameter* of the node, on which you can call `setValue(value)` to set the name of the callback

```python
def inputChangedCallback(inputIndex, thisNode, thisGroup, app):
    ...

app.Merge1.onInputChanged.set("inputChangedCallback")
```
Example

```python
# This simple callback just prints the input node name if connected or "None" otherwise
# node changes
def inputChangedCallback(inputIndex, thisNode, thisGroup, app):
    inp = thisNode.getInput(inputIndex)
    if not inp is None:
        print("Input ",inputIndex," is ",inp.getScriptName())
    else:
        print("Input ",inputIndex," is None")

app.Merge1.onInputChanged.set("inputChangedCallback")
```

### 3.5.7 The After project created callback

This function is called whenever a new project is created, that is either when launching Natron without loading a project, or when clicking “Create a new project” or “Close project”.

**Note:** Note that this function is never called when a project is loaded either via an auto-save or from user interaction.

The `app` variable will be set so it points to the correct *application instance* being created.

You can set the callback via the `afterProjectCreated` parameter of the settings of Natron.

This is a good place to create custom panels and/or setup the node-graph with node presets.

Example, taken from the initGui.py script provided as example in this section:

```python
def onProjectCreated():
    #Always create our icon viewer on project creation
    createIconViewer()

natron.settings.afterProjectCreated.set("onProjectCreated")
```

### 3.5.8 The After project loaded callback

This function is very similar to the After project created callback but is a per-project callback, called only when a project is loaded from an auto-save or from user interaction. The signature is:

```python
callback(app)
```

- `app` : points to the correct *application instance* being loaded.

You can set this callback in the project settings:

This is a good place to do some checks to opened projects or to setup something:

```python
def onProjectLoaded(app):
    if not natron.isBackground():
        if app.getUserPanel("fr.inria.iconviewer") is None:
```

3.5. Using Callbacks
createIconViewer()

app.afterProjectLoad.set("onProjectLoaded")

Note: You can set a default After project loaded callback for all new projects in the Preferences–>Python tab.

3.5.9 The Before project save callback

This function will be called prior to saving a project either via an auto-save or from user interaction. The signature is:

callback(filename, app, autoSave)

- **filename**: This is the file-path where the project is initially going to be saved.
- **app**: points to the correct application instance being created.
- **autoSave**: This indicates whether the save was originated from an auto-save or from user interaction.

Warning: This function should return the filename under which the project should really be saved.

You can set the callback from the project settings:

```python
def beforeProjectSave(filename, app, autoSave):
    print("Saving project under: ", filename)
    return filename
```

app.beforeProjectSave.set("beforeProjectSave")

Note: You can set a default Before project save callback for all new projects in the Preferences–>Python tab.

3.5.10 The Before project close callback

This function is called prior to closing a project either because the application is about to quit or because the user closed the project. The signature is:
callback(app)

- **app**: points to the correct *application instance* being closed.

This function can be used to synchronize any other device or piece of software communicating with Natron.

You can set the callback from the project settings:

```python
def beforeProjectClose(app):
    print("Closing project")
app.beforeProjectClose.set("beforeProjectClose")
```

**Note:** You can set a default Before project close callback for all new projects in the *Preferences→Python* tab.

### 3.5.11 The After node created callback

This function is called after creating a node in Natron. The signature is:

```python
callback(thisNode, app, userEdited)
```

- **thisNode** points to the *node* that has been created.
- **app** points to the correct *application instance*.
- **userEdited** will be *True* if the node was created by the user (or by a script using the `createNode(pluginID, version, group)` function) or *False* if the node was created by actions such as pasting a node or when the project is loaded.

This is a good place to change default parameters values.

You can set the callback from the project settings:

```python
def onNodeCreated(thisNode, app, userEdited):
    print(thisNode.getScriptName()," was just created")
    if userEdited:
        print(" due to user interaction")
    else:
        print(" due to project load or node pasting")
app.afterNodeCreated.set("onNodeCreated")
```
Note: You can set a default After node created callback for all new projects in the Preferences–>Python tab.

This callback can also be set in the Node tab of any Group node (or PyPlug). If set on the Group, the callback will be invoked for the Group node and all its direct children (not recursively).

### 3.5.12 The Before node removal callback:

This function is called prior to deleting a node in Natron. The signature is:

```python
callback(thisNode, app)
```

- `thisNode`: points to the node about to be deleted.
- `app`: points to the correct application instance.

**Warning:** This function will NOT be called when the project is closing

You can set the callback from the project settings:
def beforeNodeDeleted(thisNode, app):
    print(thisNode.getScriptName(), "is going to be destroyed")

app.beforeNodeRemoval.set("beforeNodeDeleted")

Note: You can set a default Before node removal callback for all new projects in the Preferences->Python tab.

This callback can also be set in the Node tab of any Group node (or PyPlug). If set on the Group, the callback will be invoked for the Group node and all its direct children (not recursively).

### 3.5.13 The Before frame render callback:

This function is called prior to rendering any frame with a Write node. The signature is:

```python
callback(frame, thisNode, app)
```

- `thisNode`: points to the *write node*.
- `app`: points to the correct *application instance*.
- `frame`: The frame that is about to be rendered

To execute code specific when in background render mode or in GUI mode, use the following condition

```python
if natron.isBackground():
    # We are in background mode
```

You can set the callback from the Write node settings panel in the “Python” tab.

This function can be used to communicate with external programs for example.

**Warning:** Any exception thrown in this callback will abort the render

### 3.5.14 The After frame rendered callback:

This function is called after each frame is finished rendering with a Write node. The signature is:

```python
callback(frame, thisNode, app)
```
• **thisNode**: points to the *write node*.

• **app**: points to the correct *application instance*.

• **frame**: The frame that is about to be rendered

To execute code specific when in background render mode or in GUI mode, use the following condition:

```python
if natron.isBackground():
    # We are in background mode
```

You can set the callback from the Write node settings panel in the “Python” tab.

This function can be used to communicate with external programs for example.

**Warning**: Any exception thrown in this callback will abort the render

### 3.5.15 The Before render callback:

This function is called once before starting rendering the first frame of a sequence with the Write node. The signature is:

```python
callback(frame, thisNode, app)
```

• **thisNode**: points to the *write node*.

• **app**: points to the correct *application instance*.

To execute code specific when in background render mode or in GUI mode, use the following condition:

```python
if natron.isBackground():
    # We are in background mode
```

You can set the callback from the Write node settings panel in the “Python” tab.

This function can be used to communicate with external programs for example.

**Warning**: Any exception thrown in this callback will abort the render

### 3.5.16 The After render callback:

This function is called once after the rendering of the last frame is finished with the Write node or if the render was aborted. The signature is:
callback(aborted, thisNode, app)

- **aborted**: True if the rendering was aborted or False otherwise.
- **thisNode**: points to the write node.
- **app**: points to the correct application instance.

To execute code specific when in background render mode or in GUI mode, use the following condition:

```python
if natron.isBackground():
    # We are in background mode
```

You can set the callback from the Write node settings panel in the “Python” tab.

This function can be used to communicate with external programs for example.

### 3.6 Rendering

To start rendering in Natron you need to use the `render(effect, firstFrame, lastFrame, frameStep)` or `render(tasks)` functions of the `App` class. The parameters passed are:

- **writeNode**: This should point to the node you want to start rendering with
- **firstFrame**: This is the first frame to render in the sequence
- **lastFrame**: This is the last frame to render in the sequence
- **frameStep**: This is the number of frames the timeline should step before rendering a new frame
e.g To render frames 1,3,5,7,9, you can use a frameStep of 2

Natron always renders from the firstFrame to the lastFrame. Generally Natron uses multiple threads to render concurrently several frames, you can control this behaviour with the parameters in the settings.

Let’s imagine there’s a node called Write1 in your project and that you want to render frames 20 to 50 included, you would call it the following way:

```
app.render(app.Write1, 20, 50)
```

**Note:** Note that when the render is launched from a GuiApp, it is not blocking, i.e: this function will return immediately even though the render is not finished.

On the other hand, if called from a background application, this call will be blocking and return once the render is finished.

If you need to have a blocking render whilst using Natron Gui, you can use the renderBlocking() function but bear in mind that it will freeze the user interface until the render is finished.

This function can take an optional frameStep parameter:

```
#This will render frames 1,4,7,10,13,16,19
app.render(app.Write1, 1, 20, 3)
```

You can use the after render callback to call code to be run once the render is finished.

For convenience, the App class also have a render(tasks) function taking a sequence of tuples (Effect,int,int) (or (Effect,int,int,int) to specify a frameStep).

Let’s imagine we were to render 2 write nodes concurrently, we could do the following call:

```
app.render([(app.Write1, 1, 10),
            (app.WriteFFmpeg1, 1, 50, 2)])
```

**Note:** The same restrictions apply to this variant of the render function: it is blocking in background mode and not blocking in GUI mode.

When executing multiple renders with the same call, each render is called concurrently from the others.

### 3.6.1 Using the DiskCache node

All the above can be applied to the DiskCache node to pre-render a sequence. Just pass the DiskCache node instead of the Write node to the render function.

### 3.7 Using the roto functionalities

All rotoscopy functionalities are gathered in the Roto class. For now, only the roto node can have a Roto object. The Roto by Natron and can be accessed as an attribute of the roto node:

```
app.Roto1.roto
```

All the objects hierarchy in the Roto object is broken up in 2 classes:

- **BezierCurve:** This class represents a single bezier, may it be an ellipse, rectangle or bezier.
• **Layer**: This is a container for BezierCurves and Layers

Bezier and layers can be accessed via their script-name directly:

```plaintext
app.Roto1.roto.Layer1.Bezier1
```

The script-name of the roto items can be found in the settings panel of the Roto node.

### 3.7.1 Moving items within layers

In Natron, all the items in a layer are rendered from top to bottom, meaning the bottom-most items will always appear on top of the others.

You can re-organize the tree using the functions available in the **Layer** class.

**Warning:** Removing an item from a layer or inserting it in a layer will change the auto-declared variable, e.g:

```plaintext
fromLayer = app.Roto1.roto.Layer1
toLayer = app.Roto1.roto.Layer2
item = app.Roto1.roto.Layer1.Bezier1
toLayer.addItem(item)
#Now item is referenced from app.Roto1.roto.Layer2.Bezier1
```

### 3.7.2 Creating layers

To create a new **BezierCurve**, use the **createLayer()** function made available by the **Roto** class.

### 3.7.3 Creating shapes

To create a new **BezierCurve**, use one of the following functions made available by the **Roto** class:

- `createBezier(x,y,time)`
- `createEllipse(x,y,diameter,fromCenter,time)`
- `createRectangle(x,y,size,time)`

Once created, the bezier will have at least 1 control point (4 for ellipses and rectangles) and one keyframe at the time specified in parameter.

A bezier initially is in an **opened** state, meaning it doesn’t produce a shape yet (unless it is a rectangle or ellipse). At this stage you can then add control points using the `func`-`addControlPoint(x,y)<NatronEngine.BezierCurve.addControlPoint>` function. Once you’re one adding control points, call the function **setCurveFinished(finished)** to close the shape by connecting the last control point with the first.

Once finished, you can refine the bezier curve by adding control points with the `addControlPointOnSegment(index,t)` function. You can then move and remove control points of the bezier.

You can also slave a control point to a track using the **slavePointToTrack(index,trackTime,trackCenter)** function.

A bezier curve has several properties that the API allows you to modify:

- **opacity**
• color
• feather distance
• feather fall-off
• enable state
• overlay color
• compositing operator

Most of them are available via a parameter, e.g:

```python
colorParam = bezier.getColorParam()
bezierColor = colorParam.get(time)
```

### 3.8 Modal dialogs

Modal dialogs are windows (or popup) that inform the user about something or ask for some informations and that does not allow any other action to be performed while the dialog is opened.

This can be used as a quick way to retrieve user inputs.

#### 3.8.1 Simple dialogs

The most simple dialogs in Natron are the information/warning/error/question dialog which basically just take some text in input and may return a reply from the user

```python
natron.informationDialog("Info","Here is a relevant info")
natron.warningDialog("Warning","Warning you might lose everything on your computer")
natron.errorDialog("Error","Something went wrong, oops.")
```

```python
reply = natron.questionDialog("Question","Are you sure you paid the license for Natron ?;)
if reply == NatronEngine.Natron.StandardButtonEnum.eStandardButtonNo:
    ...
    ...
```

#### 3.8.2 More refined dialogs

To create dialogs that may request some informations such as colors, frame range, coordinates or text input, you can create modal dialogs.

Basically you can add user parameters, and retrieve their value afterwards when the user pressed **OK**.

You can start adding user parameters using all the createXParam functions inherited from the UserParamHolder class. See the documentation of the :ref:`PyModalDialog<pyModalDialog>` for more informations:

```python
dialog = app.createModalDialog()
myInteger = dialog.createIntParam("myInt","This is an integer very important")
myInteger.setAnimationEnabled(False)
myInteger.setAddNewLine(False)
```
Here is a relevant info

Warning you might lose everything on your computer
3.8. Modal dialogs

- **Something went wrong, oops.**
  - OK

- **Are you sure you paid the license for Natron ?;)
  - No
  - Yes
#Create a boolean on the same line
myBoolean = dialog.createBooleanParam("myBool","Yet another important boolean")

dialog.refreshUserParamsGUI()

You can also add custom PySide widgets that can be inserted after any user parameter(s) using the `addWidget(widget)` and `insertWidget(index, widget)` functions.

```python
label = QLabel("This is a PySide label")
dialog.addWidget(label)
```

To make the dialog show-up, use the `exec()` function on the dialog. This function will return once the user pressed either “OK” or “Canceled”:

```python
if dialog.exec():
    #User pressed OK
```

You can add a custom callback when a parameter changes, for instance to hide another parameter:

```python
#Callback called when a parameter of changes
#The variable paramName is declared by Natron, indicating the name of the parameter which just had its value changed
def paramChangedCallback():
    if paramName == "myBool":
        myInteger.setVisible(myBoolean.get())

dialog.setParamChangedCallback("paramChangedCallback")
```

### 3.9 User menu commands

In Natron you can add multiple menu commands that will then be available to the user via the menu. You can also assign it a shortcut and the user will be able to modify it via the shortcuts editor.
3.9.1 Project-wide menu commands:

To add a project-wipe menu command to the application’s menu-bar, you need to use the `addMenuCommand(grouping, function, key, modifiers)` of the `PyGuiApplication` class to register it:

```python
def createIconViewer():
    ...

#Add a custom menu entry with a shortcut to create our icon viewer
```

Note that this function is to be called on the whole application via the `natron` variable and is not per-project unlike most functions that are generally called on the `app` object.

**Warning:** This function can only be called in the startup script `init.py` and will have no effect otherwise. This is not a dynamic function and will not create menu entries on the fly.

3.10 PySide panels

To create a non-modal panel that can be saved in the project’s layout and docked into the application’s tab-widgets, there is 2 possible way of doing it:

- Sub-class `PyPanel` and create your own GUI using PySide
- Use the API proposed by `PyPanel` to add custom user parameters as done for `PyModalDialog`.

Generally you should define your panels in the `initGui.py` script (see `startup-scripts`). You can also define the panel in the Script Editor at run-time of Natron, though this will not persist when Natron is closed.

To make your panel be created upon new project created, register a Python callback in the Preferences–>Python tab in the parameter After project created. This callback will not be called for project being loaded either via an auto-save or via a user action.

```python
#This goes in initGui.py
def createMyPanel():
    #Create panel
    ...

def onProjectCreatedCallback():
    createMyPanel()
```
Warning: When the initGui.py script is executed, the app variable (or any derivative such as app1, app2 etc...) does not exist since no project is instantiated yet. The purpose of the script is not to instantiate the GUI per say but to define classes and functions that will be used later on by application instances.

Python panels can be re-created for existing projects using serialization functionalities explained here. See the example below (the whole script is available attached below)

```python
# We override the save() function and save the filename
def save(self):
    return self.locationEdit.text()

# We override the restore(data) function and restore the current image
def restore(self, data):
    self.locationEdit.setText(data)
    self.label.setPixmap(QPixmap(data))
```

The sole requirement to save a panel in the layout is to call the registerPythonPanel(panel, function) function of GuiApp:

```python
app.registerPythonPanel(app.mypanel, "createIconViewer")
```

See the details of the PyPanel class for more explanation on how to sub-class it.

Also check-out the complete example source code below.

### 3.10.1 Using user parameters:

Let’s assume we have no use to make our own widgets and want quick parameters fresh and ready, we just have to use the PyPanel class without sub-classing it:

```python
#Callback called when a parameter of the player changes
#The variable paramName is declared by Natron; indicating the name of the parameter which just had its value changed
def myPlayerParamChangedCallback():
    viewer = app.getViewer("Viewer1")
    if viewer == None:
        return
    if paramName == "previous":
        viewer.seek(viewer.getCurrentFrame() - 1)

def createMyPlayer():
    #Create a panel named "My Panel" that will use user parameters
    app.player = NatronGui.PyPanel("fr.inria.myplayer","My Player",True,app)

    #Add a push-button parameter named "Previous"
    app.player.previousFrameButton = app.player.createButtonParam("previous","Previous")

    #Refresh user parameters GUI, necessary after changes to static properties of parameters.
    #See the Param class documentation
    app.player.refreshUserParamsGUI()

    #Set a callback that will be called upon parameter change
    app.player.setParamChangedCallback("myPlayerParamChangedCallback")
```
Note: For convenience, there is a way to also add custom widgets to python panels that are using user parameters with the `addWidget(widget)` and `insertWidget(index, widget)` functions. However the widgets will be appended after any user parameter defined.

### 3.10.2 Managing panels and panes

Panels in Natron all have an underlying script-name, that is the one you gave as first parameter to the constructor of `PyPanel`.

You can then move the `PyPanel` between the application’s panes by calling the function `moveTab(scriptName, pane)` of `GuiApp`.

Note: All application’s panes are auto-declared by Natron and can be referenced directly by a variable, such as:

```python
app.pane2
```

Panels also have a script-name but only viewers and user panels are auto-declared by Natron:

```python
app.pane2.Viewer1
app.pane1.myPySidePanelScriptName
```

### 3.10.3 Source code of the example initGui.py

```python
#This Source Code Form is subject to the terms of the Mozilla Public License, v. 2.0. If a copy of the MPL was not distributed with this file, You can obtain one at http://mozilla.org/MPL/2.0/. */
#Created by Alexandre GAUTHIER-FOICHAT on 01/27/2015.

from PySide.QtGui import *
from PySide.QtCore import *

#To import the variable "natron"
from NatronGui import *

#Callback called when a parameter of the player changes
#The variable paramName is declared by Natron; indicating the name of the parameter which just had it changed.
def myPlayerParamChangedCallback(paramName, app, userEdited):
    viewer = app.getViewer("Viewer1")
    if viewer == None:
        return
    if paramName == "previous":
        viewer.seek(viewer.getCurrentFrame() - 1)
    elif paramName == "backward":
        viewer.startBackward()
    elif paramName == "forward":
        viewer.startForward()
    elif paramName == "next":
        viewer.seek(viewer.getCurrentFrame() + 1)
    elif paramName == "stop":
        viewer.pause()
```

3.10. PySide panels
def createMyPlayer():

    app.player = NatronGui.PyPanel("fr.inria.myplayer","My Player",True,app)
    app.player.previousFrameButton = app.player.createButtonParam("previous","Previous")
    app.player.previousFrameButton.setAddNewLine(False)

    app.player.playBackwardButton = app.player.createButtonParam("backward","Rewind")
    app.player.playBackwardButton.setAddNewLine(False)

    app.player.stopButton = app.player.createButtonParam("stop","Pause")
    app.player.stopButton.setAddNewLine(False)

    app.player.playForwardButton = app.player.createButtonParam("forward","Play")
    app.player.playForwardButton.setAddNewLine(False)

    app.player.nextFrameButton = app.player.createButtonParam("next","Next")

    app.player.helpLabel = app.player.createStringParam("help","Help")
    app.player.helpLabel.setType(NatronEngine.StringParam.TypeEnum.eStringTypeLabel)
    app.player.helpLabel.set("<br><b>Previous:</b> Seek the previous frame on the timeline<br>
    "<br><b>Rewind:</b> Play backward<br>
    "<br><b>Pause:</b> Pauses the playback<br>
    "<br><b>Play:</b> Play forward<br>
    "<br><b>Next:</b> Seek the next frame on the timeline<br>")

    app.player.refreshUserParamsGUI()
    app.player.setParamChangedCallback("myPlayerParamChangedCallback")

    #Add it to the "pane2" tab widget
    app.pane2.appendTab(app.player);

    #Register the tab to the application, so it is saved into the layout of the project
    #and can appear in the Panes sub-menu of the "Manage layout" button (in top left-hand corner of each tab widget)
    app.registerPythonPanel(app.player,"createMyPlayer")

#A small panel to load and visualize icons/images
class IconViewer(NatronGui.PyPanel):

    #Register a custom signal
    userFileChanged = QtCore.Signal()

    #Slots should be decorated:
    #http://qt-project.org/wiki/Signals_and_Slots_in_PySide

    #This is called upon a user click on the button
    @QtCore.Slot()
    def onButtonClicked(self):
        location = self.currentApp.getFilenameDialog(("jpg","png","bmp","tif"))
        if location:
            self.locationEdit.setText(location)

            #Save the file
            self.onUserDataChanged()

            self.userFileChanged.emit()

    #This is called when the user finish editing of the line edit (when return is pressed or focus out)
```python
@QtCore.Slot()
def onLocationEditEditingFinished(self):
    # Save the file
    self.onUserDataChanged()
    self.userFileChanged.emit()

    # This is called when our custom userFileChanged signal is emitted
@QtCore.Slot()
def onFileChanged(self):
    self.label.setPixmap(QPixmap(self.locationEdit.text()))

def __init__(self, scriptName, label, app):
    # Init base class, important! otherwise signals/slots won't work.
    NatronGui.PyPanel.__init__(self, scriptName, label, False, app)
    # Store the current app as it might no longer be pointing to the app at the time being called
    # when a slot will be invoked later on
    self.currentApp = app

    # Set the layout
    self.setLayout(QVBoxLayout())

    # Create a widget container for the line edit + button
    fileContainer = QWidget(self)
    fileLayout = QHBoxLayout()
    fileContainer.setLayout(fileLayout)

    # Create the line edit, make it expand horizontally
    self.locationEdit = QLineEdit(fileContainer)
    self.locationEdit.setSizePolicy(QSizePolicy.Expanding, QSizePolicy.Preferred)

    # Create a pushbutton
    self.button = QPushButton(fileContainer)
    # Decorate it with the open-file pixmap built-in into Natron
    buttonPixmap = natron.getIcon(NatronEngine.Natron.PixmapEnum.NATRON_PIXMAP_OPEN_FILE)
    self.button.setPixmap(QPixmap(buttonPixmap))

    # Add widgets to the layout
    fileLayout.addWidget(self.locationEdit)
    fileLayout.addWidget(self.button)

    # Use a QLabel to display the images
    self.label = QLabel(self)

    # Init the label with the icon of Natron
    natronPixmap = natron.getIcon(NatronEngine.Natron.PixmapEnum.NATRON_PIXMAP_APP_ICON)
    self.label.setPixmap(natronPixmap)
    # Built-in icons of Natron are in the resources
    self.locationEdit.setText(":/Resources/Images/natronIcon256_linux.png")

    # Make it expand in both directions so it takes all space
    self.label.setSizePolicy(QSizePolicy.Expanding, QSizePolicy.Expanding)

    # Add widgets to the layout
    self.layout().addWidget(fileContainer)
    self.layout().addWidget(self.label)
```

### 3.10. PySide panels

175
# Make signal/slot connections
self.button.clicked.connect(self.onButtonClicked)
self.locationEdit.editingFinished.connect(self.onLocationEditEditingFinished)
self.userFileChanged.connect(self.onFileChanged)

# We override the save() function and save the filename
def save(self):
    return self.locationEdit.text()

# We override the restore(data) function and restore the current image
def restore(self, data):
    self.locationEdit.setText(data)
    self.label.setPixmap(QPixmap(data))

# To be called to create a new icon viewer panel:
# Note that *app* should be defined. Generally when called from onProjectCreatedCallback
# this is set, but when called from the Script Editor you should set it yourself beforehand:
# app = app
# See http://natron.readthedocs.org/en/python/natronobjects.html for more info
def createIconViewer():
    if hasattr(app,"p"):
        # The icon viewer already exists, it we override the app.p variable, then it will destroy the
        # and create a new one but we don't really need it
        # The warning will be displayed in the Script Editor
        print("Note for us developers: this widget already exists!")
        return

    # Create our icon viewer
    app.p = IconViewer("fr.inria.iconViewer","Icon viewer",app)

    # Add it to the "pane2" tab widget
    app.pane2.appendTab(app.p);

    # Register the tab to the application, so it is saved into the layout of the project
    # and can appear in the Panes sub-menu of the "Manage layout" button (in top left-hand corner of each tab widget)
    app.registerPythonPanel(app.p,"createIconViewer")

    # Callback set in the "After project created" parameter in the Preferences-->Python tab of Natron
    # This will automatically create our panels when a new project is created
    def onProjectCreatedCallback(app):
        # Always create our icon viewer on project creation, you must register this call-back in the
        # "After project created callback" parameter of the Preferences-->Python tab.
        createIconViewer()
        createMyPlayer()

        # Add a custom menu entry with a shortcut to create our icon viewer
        natron.addMenuCommand("Inria/Scripts/IconViewer","createIconViewer",QtCore.Qt.Key.Key_L,QtCore.Qt.KeyboardModifier.ShiftModifier)
3.11 Controlling the viewer

Natron exposes all functionalities available to the user in the Python API via the *PyViewer* class.

To retrieve a *PyViewer*, use the *auto-declared* variable:

```python
app.pane2.Viewer1
```

or use the following function `getViewer(scriptName)`, passing it the *script-name* of a viewer node.

You can then control the player, the displayed channels, the current view, the current compositing operator, which are the input A and B, the frame-range, the proxy level and various other stuff.
\textbf{n}

NatronEngine, 1
NatronGui, 93
Symbols
__getitem__() (NatronEngine.NatronEngine.ColorTuple method), 25
__getitem__() (NatronEngine.NatronEngine.Double2DTuple method), 27
__getitem__() (NatronEngine.NatronEngine.Double3DTuple method), 29
__getitem__() (NatronEngine.NatronEngine.Int2DTuple method), 46
__getitem__() (NatronEngine.NatronEngine.Int3DTuple method), 47

A
a (NatronEngine.NatronEngine.ColorTuple attribute), 25
addControlPoint() (NatronEngine.NatronEngine.BezierCurve method), 11
addControlPoint() (NatronEngine.NatronEngine.ParametricParam method), 66
addControlPointOnSegment() (NatronEngine.NatronEngine.BezierCurve method), 11
addItem() (NatronEngine.NatronEngine.Layer method), 53
addOption() (NatronEngine.NatronEngine.ChoiceParam method), 19
addParam() (NatronEngine.NatronEngine.GroupParam method), 42
addParam() (NatronEngine.NatronEngine.PageParam method), 58
addUserPlane() (NatronEngine.NatronEngine.Effect method), 34

AnimationLevelEnum (NatronEngine.NatronEngine.Natron attribute), 56
appendTab() (NatronGui.NatronGui.PyTabWidget method), 109
appendToNatronPath() (NatronEngine.NatronEngine.PyCoreApplication method), 72
area() (NatronEngine.NatronEngine.RectD method), 76

B
b (NatronEngine.NatronEngine.ColorTuple attribute), 25
beginChanges() (NatronEngine.NatronEngine.Effect method), 35
bottom() (NatronEngine.NatronEngine.RectD method), 76
bottom() (NatronEngine.NatronEngine.RectI method), 79

C
CairoOperatorEnum (NatronEngine.NatronEngine.BezierCurve attribute), 11
canConnectInput() (NatronEngine.NatronEngine.Effect method), 35
clear() (NatronEngine.NatronEngine.RectD method), 76
clear() (NatronEngine.NatronEngine.RectI method), 79
closeCurrentTab() (NatronGui.NatronGui.PyTabWidget method), 109
closePane() (NatronGui.NatronGui.PyTabWidget method), 109
closeProject() (NatronEngine.NatronEngine.App method), 8
closeTab() (NatronGui.NatronGui.PyTabWidget method), 109
ColorTuple (class in NatronEngine), 25
connectInput() (NatronEngine.NatronEngine.Effect method), 35
contains() (NatronEngine.NatronEngine.RectD method), 76
contains() (NatronEngine.NatronEngine.RectI method), 79
createBezier() (NatronEngine.NatronEngine.Roto method), 82
createBooleanParam() (NatronEngine.NatronEngine.UserParamHolder method), 88
createButtonParam() (NatronEngine.NatronEngine.UserParamHolder method), 88
createChild() (NatronEngine.NatronEngine.Effect method), 36
createChoiceParam() (NatronEngine.NatronEngine.UserParamHolder method), 88
createColorParam() (NatronEngine.NatronEngine.UserParamHolder method), 88
createDouble2DParam() (NatronEngine.NatronEngine.UserParamHolder method), 89
createDouble3DParam() (NatronEngine.NatronEngine.UserParamHolder method), 89
createDoubleParam() (NatronEngine.NatronEngine.UserParamHolder method), 89
createEllipse() (NatronEngine.NatronEngine.Roto method), 82
createFileParam() (NatronEngine.NatronEngine.UserParamHolder method), 89
createGroupParam() (NatronEngine.NatronEngine.UserParamHolder method), 90
createInt2DParam() (NatronEngine.NatronEngine.UserParamHolder method), 90
createInt3DParam() (NatronEngine.NatronEngine.UserParamHolder method), 90
createIntParam() (NatronEngine.NatronEngine.UserParamHolder method), 91
createLayer() (NatronEngine.NatronEngine.Roto method), 82
createNode() (NatronEngine.NatronEngine.App method), 6
createOutputFileParam() (NatronEngine.NatronEngine.UserParamHolder method), 91
createPageParam() (NatronEngine.NatronEngine.UserParamHolder method), 91
createParametricParam() (NatronEngine.NatronEngine.UserParamHolder method), 91
createPathParam() (NatronEngine.NatronEngine.UserParamHolder method), 92
createRectangle() (NatronEngine.NatronEngine.Roto method), 82
createStringParam() (NatronEngine.NatronEngine.UserParamHolder method), 92
curve() (NatronEngine.NatronEngine.Param method), 62

deleteAllControlPoints() (NatronEngine.NatronEngine.ParametricParam method), 68
deleteControlPoint() (NatronEngine.NatronEngine.ParametricParam method), 68
deleteValueAtTime() (NatronEngine.NatronEngine.AnimatedParam method), 2
destroy() (NatronEngine.NatronEngine.Effect method), 36
disconnectInput() (NatronEngine.NatronEngine.Effect method), 36

Double2DTuple (class in NatronEngine), 27
Double3DTuple (class in NatronEngine), 29

destroy() (NatronEngine.NatronEngine.Effect method), 36
disconnectInput() (NatronEngine.NatronEngine.Effect method), 36

Double2DTuple (class in NatronEngine), 27
Double3DTuple (class in NatronEngine), 29

endChanges() (NatronEngine.NatronEngine.Effect method), 35
errorDialog() (NatronGui.NatronGui.PyGuiApplication method), 100

floatCurrentTab() (NatronGui.NatronGui.PyTabWidget method), 110
floatPane() (NatronGui.NatronGui.PyTabWidget method), 110
G

g (NatronEngine.NatronEngine.ColorTuple attribute), 25
get() (NatronEngine.NatronEngine.BooleanParam method), 17
get() (NatronEngine.NatronEngine.ColorParam method), 22
get() (NatronEngine.NatronEngine.Double2DParam method), 26
get() (NatronEngine.NatronEngine.Double3DParam method), 28
get() (NatronEngine.NatronEngine.DoubleParam method), 30
get() (NatronEngine.NatronEngine.Int2DParam method), 45
get() (NatronEngine.NatronEngine.Int3DParam method), 46
get() (NatronEngine.NatronEngine.IntParam method), 49
get() (NatronEngine.NatronEngine.StringParamBase method), 85
getActivatedParam() (NatronEngine.NatronEngine.BezierCurve method), 11
getAInput() (NatronGui.NatronGui.PyTabWidget method), 113
getAlphaComponents() (NatronEngine.NatronEngine.ImageLayer method), 44
getBackwardMotionComponents() (NatronEngine.NatronEngine.ImageLayer method), 44
getBaseLayer() (NatronEngine.NatronEngine.Roto method), 83
getBInput() (NatronGui.NatronGui.PyTabWidget method), 113
getBitDepth() (NatronEngine.NatronEngine.Effect method), 36
getBuildNumber() (NatronEngine.NatronEngine.PyCoreApplication method), 73
getCanAnimate() (NatronEngine.NatronEngine.Param method), 62
getChannels() (NatronGui.NatronGui.PyTabWidget method), 113
getChildren() (NatronEngine.NatronEngine.Group method), 40
getChildren() (NatronEngine.NatronEngine.Layer method), 53
color() (NatronEngine.NatronEngine.BezierCurve method), 11
color() (NatronEngine.NatronEngine.Effect method), 36
getComponentsNames() (NatronEngine.NatronEngine.ImageLayer method), 44
getComponentsPrettyName() (NatronEngine.NatronEngine.ImageLayer method), 44
compositingOperator() (NatronGui.NatronGui.PyTabWidget method), 113
curIndex() (NatronGui.NatronGui.PyTabWidget method), 112
curIndex() (NatronGui.NatronGui.PyTabWidget method), 110
curTime() (NatronEngine.NatronEngine.AnimatedParam method), 2
curTime() (NatronEngine.NatronEngine.Effect method), 36
curView() (NatronGui.NatronGui.PyTabWidget method), 114
curveColor() (NatronEngine.NatronEngine.ParametricParam method), 68
defaultValue() (NatronEngine.NatronEngine.BooleanParam method), 17
defaultValue() (NatronEngine.NatronEngine.ColorParam method), 22
defaultValue() (NatronEngine.NatronEngine.DoubleParam method), 30
defaultValue() (NatronEngine.NatronEngine.IntParam method), 49
defaultValue() (NatronEngine.NatronEngine.StringParamBase method), 53
getMaximum() (NatronEngine.NatronEngine.IntParam method), 31
getMaxInputCount() (NatronEngine.NatronEngine.Effect method), 37
getMinimum() (NatronEngine.NatronEngine.DoubleParam method), 31
getMinimum() (NatronEngine.NatronEngine.IntParam method), 49
getNatronDevelopmentStatus() (NatronEngine.NatronEngine.PyCoreApplication method), 73
getNatronPath() (NatronEngine.NatronEngine.PyCoreApplication method), 73
getNatronVersionEncoded() (NatronEngine.NatronEngine.PyCoreApplication method), 73
getNatronVersionMajor() (NatronEngine.NatronEngine.PyCoreApplication method), 73
getNatronVersionMinor() (NatronEngine.NatronEngine.PyCoreApplication method), 73
getNatronVersionRevision() (NatronEngine.NatronEngine.PyCoreApplication method), 73
getNatronVersionString() (NatronEngine.NatronEngine.PyCoreApplication method), 74
getNControlPoints() (NatronEngine.NatronEngine.ParametricParam method), 68
getNode() (NatronEngine.NatronEngine.Group method), 40
getNoneComponents() (NatronEngine.NatronEngine.ImageLayer method), 44
getNthControlPoint() (NatronEngine.NatronEngine.ParametricParam method), 68
getNumComponents() (NatronEngine.NatronEngine.ImageLayer method), 43
getNumControlPoints() (NatronEngine.NatronEngine.BezierCurve method), 12
getNumCpus() (NatronEngine.NatronEngine.PyCoreApplication method), 74
getNumInstances() (NatronEngine.NatronEngine.Param method), 63
getParam() (NatronEngine.NatronEngine.AppSettings method), 9
getParam() (NatronEngine.NatronEngine.Effect method), 37
getParam() (NatronEngine.NatronEngine.ItemBase method), 52
getParam() (NatronEngine.NatronEngine.PyModalDialog method), 101
getParam() (NatronEngine.NatronEngine.PyPanel method), 105
getParams() (NatronEngine.NatronEngine.AppSettings method), 9
getParams() (NatronEngine.NatronEngine.Effect method), 37
getParams() (NatronEngine.NatronEngine.PyPanel method), 105
getParent() (NatronEngine.NatronEngine.Param method), 63
getParentLayer() (NatronEngine.NatronEngine.ItemBase method), 52
getPixelAspectRatio() (NatronEngine.NatronEngine.Effect method), 37
getPlaybackMode() (NatronEngine.NatronGui.PyTabWidget method), 112
getPluginID() (NatronEngine.NatronEngine.Effect method), 37
getPluginIDs() (NatronEngine.NatronEngine.PyCoreApplication method), 74
getPointMasterTrack() (NatronEngine.NatronEngine.BezierCurve method), 13
getPosition() (NatronEngine.NatronEngine.Effect method),
Int2DTuple (class in NatronEngine), 46
Int3DTuple (class in NatronEngine), 47
intersect() (NatronEngine.NatronEngine.RectD method), 76
intersect() (NatronEngine.NatronEngine.RectI method), 79
intersects() (NatronEngine.NatronEngine.RectD method), 77
intersects() (NatronEngine.NatronEngine.RectI method), 80
is64Bit() (NatronEngine.NatronEngine.PyCoreApplication method), 74
isBackground() (NatronEngine.NatronEngine.PyCoreApplication method), 74
isColorPlane() (NatronEngine.NatronEngine.ImageLayer method), 43
isCurveFinished() (NatronEngine.NatronEngine.BezierCurve method), 13
isInfinite() (NatronEngine.NatronEngine.RectD method), 77
isInfinite() (NatronEngine.NatronEngine.RectI method), 80
isLinux() (NatronEngine.NatronEngine.PyCoreApplication method), 74
isMacOSX() (NatronEngine.NatronEngine.PyCoreApplication method), 74
isNull() (NatronEngine.NatronEngine.RectD method), 77
isNull() (NatronEngine.NatronEngine.RectI method), 80
isProxyModeEnabled() (NatronGui.NatronGui.PyTabWidget method), 113
isUnix() (NatronEngine.NatronEngine.PyCoreApplication method), 74
isUserSelected() (NatronEngine.NatronEngine.Effect method), 38
isWindows() (NatronEngine.NatronEngine.PyCoreApplication method), 75

M
merge() (NatronEngine.NatronEngine.RectD method), 77
merge() (NatronEngine.NatronEngine.RectI method), 80
moveFeatherByIndex() (NatronEngine.NatronEngine.BezierCurve method), 13
moveLeftBezierPoint() (NatronEngine.NatronEngine.BezierCurve method), 13
movePointByIndex() (NatronEngine.NatronEngine.BezierCurve method), 13
moveTab() (NatronGui.NatronGui.GuiApp method), 95

N
NatronEngine (module), 1, 4, 8, 9, 16, 18, 21, 25, 27–29, 33, 39–42, 44–47, 51, 53, 54, 57, 58, 66, 69, 70, 75, 78, 81, 83, 85, 86
NatronGui (module), 93, 98, 100, 102, 107, 111
newProject() (NatronEngine.NatronEngine.App method), 8

O
openFile() (NatronEngine.NatronEngine.FileParam method), 40
openFile() (NatronEngine.NatronEngine.OutputFileParam method), 57
OrientationEnum (NatronEngine.NatronEngine.Natron attribute), 56

P
pause() (NatronGui.NatronGui.PyTabWidget method), 112
PixmapEnum (NatronEngine.NatronEngine.Natron attribute), 57
PlaybackModeEnum (NatronEngine.NatronEngine.Natron attribute), 56
PyCoreApplication (class in NatronEngine), 72
PyGuiApplication (class in NatronGui), 98

Q

R
r (NatronEngine.NatronEngine.ColorTuple attribute), 25
random() (NatronEngine.NatronEngine.Param method), 63, 64
randomInt() (NatronEngine.NatronEngine.Param method), 64
redraw() (NatronGui.NatronGui.PyTabWidget method), 112
refreshUserParamsGUI() (NatronEngine.NatronEngine.UserParamHolder method), 93
removeAnimation() (NatronEngine.NatronEngine.AnimatedParam method), 3
removeControlPointByIndex() (NatronEngine.NatronEngine.BezierCurve method), 14
removeItem() (NatronEngine.NatronEngine.Layer method), 54
removeParam() (NatronEngine.NatronEngine.UserParamHolder method), 92
removeTab() (NatronGui.NatronGui.PyTabWidget method), 110
render() (NatronEngine.NatronEngine.App method), 6, 7
renderCurrentFrame() (NatronGui.NatronGui.PyTabWidget method), 112
resetProject() (NatronEngine.NatronEngine.App method), 8
restoreDefaultSettings() (NatronEngine.NatronEngine.AppSettings method), 9
restoreDefaultValue() (NatronEngine.NatronEngine.BooleanParam method), 17
restoreDefaultValue() (NatronEngine.NatronEngine.Double2DParam method), 26, 27
restoreDefaultValue() (NatronEngine.NatronEngine.DoubleParam method), 31
restoreDefaultValue() (NatronEngine.NatronEngine.Int2DParam method), 45
restoreDefaultValue() (NatronEngine.NatronEngine.Int3DParam method), 46, 47
restoreDefaultValue() (NatronEngine.NatronEngine.IntParam method), 50
set() (NatronEngine.NatronEngine.BooleanParam method), 17
set() (NatronEngine.NatronEngine.ColorParam method), 31
set() (NatronEngine.NatronEngine.Double2DParam method), 86
set() (NatronEngine.NatronEngine.Double3DParam method), 86
set() (NatronEngine.NatronEngine.DoubleParam method), 86
set() (NatronEngine.NatronEngine.IntParam method), 50
set() (NatronEngine.NatronEngine.StringParamBase method), 86
right() (NatronEngine.NatronEngine.RectD method), 77
right() (NatronEngine.NatronEngine.RectI method), 80
S
saveProject() (NatronEngine.NatronEngine.App method), 7
saveProjectAs() (NatronEngine.NatronEngine.App method), 8
saveSettings() (NatronEngine.NatronEngine.AppSettings method), 9
saveTempProject() (NatronEngine.NatronEngine.App method), 8
seek() (NatronGui.NatronGui.PyTabWidget method), 112
selectAllNodes() (NatronGui.NatronGui.GuiApp method), 97
set() (NatronEngine.NatronEngine.BooleanParam method), 17
set() (NatronEngine.NatronEngine.ChoiceParam method), 20, 21
set() (NatronEngine.NatronEngine.Double2DParam method), 26, 27
set() (NatronEngine.NatronEngine.DoubleParam method), 31
set() (NatronEngine.NatronEngine.Int2DParam method), 45
set() (NatronEngine.NatronEngine.Int3DParam method), 46, 47
set() (NatronEngine.NatronEngine.IntParam method), 50
set() (NatronEngine.NatronEngine.RectD method), 77
set() (NatronEngine.NatronEngine.RectI method), 80
set() (NatronEngine.NatronEngine.RectD method), 77
set() (NatronEngine.NatronEngine.RectI method), 81
set() (NatronEngine.NatronEngine.RectD method), 78
set() (NatronEngine.NatronEngine.RectI method), 81
Natron Documentation, Release 2.0.0

set_top() (NatronEngine.NatronEngine.RectD method), 78
set_top() (NatronEngine.NatronEngine.RectI method), 81
setActivated() (NatronEngine.NatronEngine.BezierCurve method), 14
setAddNewLine() (NatronEngine.NatronEngine.Param method), 64
setAInput() (NatronGui.NatronGui.PyTabWidget method), 113
setAnimationEnabled() (NatronGui.NatronGui.PyTabWidget method), 113
setAsAlias() (NatronEngine.NatronEngine.Param method), 65
setAsMultiPathTable() (NatronEngine.NatronEngine.PathParam method), 70
setAsTab() (NatronEngine.NatronEngine.GroupParam method), 42
setBInput() (NatronGui.NatronGui.PyTabWidget method), 113
setChannels() (NatronGui.NatronGui.PyTabWidget method), 113
setColor() (NatronEngine.NatronEngine.BezierCurve method), 14
setColor() (NatronEngine.NatronEngine.Effect method), 38
setCompositingOperator() (NatronGui.NatronGui.PyTabWidget method), 113
setCurrentIndex() (NatronGui.NatronGui.PyTabWidget method), 110
setCurrentView() (NatronGui.NatronGui.PyTabWidget method), 114
setCurveFinished() (NatronEngine.NatronEngine.BezierCurve method), 14
setDefaultValue() (NatronEngine.NatronEngine.IntParam method), 50
setDefaultValue() (NatronEngine.NatronEngine.StringParamBase method), 86
setDisplayMaximum() (NatronEngine.NatronEngine.DoubleParam method), 32
setDisplayMaximum() (NatronEngine.NatronEngine.IntParam method), 50
setDisplayMinimum() (NatronEngine.NatronEngine.IntParam method), 50
setEnabled() (NatronEngine.NatronEngine.Param method), 64
setEnabledByDefault() (NatronEngine.NatronEngine.Param method), 64
setEvaluateOnChange() (NatronEngine.NatronEngine.Param method), 64
set FeatherDistance() (NatronEngine.NatronEngine.BezierCurve method), 14
set FeatherFallOff() (NatronEngine.NatronEngine.BezierCurve method), 14
set FeatherPointAtIndex() (NatronEngine.NatronEngine.BezierCurve method), 14
set Help() (NatronEngine.NatronEngine.Param method), 65
setIconFilePath() (NatronEngine.NatronEngine.ButtonParam method), 18
setInterpolationAtTime() (NatronEngine.NatronEngine.AnimatedParam method), 4
setLabel() (NatronEngine.NatronEngine.Effect method), 38
setLabel() (NatronEngine.NatronEngine.ItemBase method), 32

Index
method), 52
setLocked() (NatronEngine.NatronEngine.ItemBase method), 52
setMaximum() (NatronEngine.NatronEngine.DoubleParam method), 32
setMaximum() (NatronEngine.NatronEngine.IntParam method), 50
setMinimum() (NatronEngine.NatronEngine.IntParam method), 51
setNextTabCurrent() (NatronGui.NatronGui.PyTabWidget method), 110
setOnProjectCreatedCallback() (NatronEngine.NatronEngine.PyCoreApplication method), 75
setOnProjectLoadedCallback() (NatronEngine.NatronEngine.PyCoreApplication method), 75
setOpened() (NatronEngine.NatronEngine.GroupParam method), 42
setPagesOrder() (NatronEngine.NatronEngine.Effect method), 39
setPersistant() (NatronEngine.NatronEngine.Param method), 65
setPlaybackMode() (NatronGui.NatronGui.PyTabWidget method), 112
setPointAtIndex() (NatronEngine.NatronEngine.BezierCurve method), 15
setPosition() (NatronEngine.NatronEngine.Effect method), 38
setProxyIndex() (NatronGui.NatronGui.PyTabWidget method), 113
setProxyModeEnabled() (NatronGui.NatronGui.PyTabWidget method), 113
setScriptName() (NatronEngine.NatronEngine.Effect method), 39
setScriptName() (NatronEngine.NatronEngine.ItemBase method), 39
setSequenceEnabled() (NatronEngine.NatronEngine.FileParam method), 40
setSize() (NatronEngine.NatronEngine.Effect method), 39
setSubGraphEditable() (NatronEngine.NatronEngine.Effect method), 39
setType() (NatronEngine.NatronEngine.StringParam method), 85
setUsePointInteract() (NatronEngine.NatronEngine.Double2DParam method), 26
setValue() (NatronEngine.NatronEngine.BooleanParam method), 17
setValue() (NatronEngine.NatronEngine.DoubleParam method), 32
setValue() (NatronEngine.NatronEngine.IntParam method), 51
setValue() (NatronEngine.NatronEngine.StringParamBase method), 86
setValueAtTime() (NatronEngine.NatronEngine.BooleanParam method), 17
setValueAtTime() (NatronEngine.NatronEngine.DoubleParam method), 32
setValueAtTime() (NatronEngine.NatronEngine.IntParam method), 51
setValueAtTime() (Natron-
<table>
<thead>
<tr>
<th>Method</th>
<th>Class/Attribute</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>setVisible()</td>
<td>NatronEngine.NatronEngine.ItemBase</td>
<td>53</td>
</tr>
<tr>
<td>setVisible()</td>
<td>NatronEngine.NatronEngine.Param</td>
<td>65</td>
</tr>
<tr>
<td>setVisibleByDefault()</td>
<td>NatronEngine.NatronEngine.Param</td>
<td>65</td>
</tr>
<tr>
<td>slavePointToTrack()</td>
<td>NatronEngine.NatronEngine.BezierCurve</td>
<td>15</td>
</tr>
<tr>
<td>slaveTo()</td>
<td>NatronEngine.NatronEngine.Param</td>
<td>65</td>
</tr>
<tr>
<td>splitHorizontally()</td>
<td>NatronGui.NatronGui.PyTabWidget</td>
<td>110</td>
</tr>
<tr>
<td>splitVertically()</td>
<td>NatronGui.NatronGui.PyTabWidget</td>
<td>111</td>
</tr>
<tr>
<td>StandardButtonEnum</td>
<td>NatronEngine.Natron</td>
<td>54</td>
</tr>
<tr>
<td>startBackward()</td>
<td>NatronGui.NatronGui.PyTabWidget</td>
<td>112</td>
</tr>
<tr>
<td>startForward()</td>
<td>NatronGui.NatronGui.PyTabWidget</td>
<td>112</td>
</tr>
<tr>
<td>StatusEnum</td>
<td>NatronEngine.NatronEngine.Natron</td>
<td>56</td>
</tr>
<tr>
<td>timelineGetLeftBound()</td>
<td>NatronEngine.NatronEngine.App</td>
<td>7</td>
</tr>
<tr>
<td>timelineGetRightBound()</td>
<td>NatronEngine.NatronEngine.App</td>
<td>7</td>
</tr>
<tr>
<td>timelineGetTime()</td>
<td>NatronEngine.NatronEngine.App</td>
<td>7</td>
</tr>
<tr>
<td>top()</td>
<td>NatronEngine.NatronEngine.RectD</td>
<td>78</td>
</tr>
<tr>
<td>top()</td>
<td>NatronEngine.NatronEngine.RectI</td>
<td>81</td>
</tr>
<tr>
<td>translate()</td>
<td>NatronEngine.NatronEngine.RectD</td>
<td>78</td>
</tr>
<tr>
<td>translate()</td>
<td>NatronEngine.NatronEngine.RectI</td>
<td>81</td>
</tr>
<tr>
<td>trigger()</td>
<td>NatronEngine.NatronEngine.ButtonParam</td>
<td>18</td>
</tr>
<tr>
<td>unslave()</td>
<td>NatronEngine.NatronEngine.Param</td>
<td>65</td>
</tr>
<tr>
<td>ViewerColorSpaceEnum</td>
<td>NatronEngine.NatronEngine.Natron</td>
<td>57</td>
</tr>
<tr>
<td>width()</td>
<td>NatronEngine.NatronEngine.RectD</td>
<td>78</td>
</tr>
<tr>
<td>width()</td>
<td>NatronEngine.NatronEngine.RectI</td>
<td>81</td>
</tr>
<tr>
<td>writeToScriptEditor()</td>
<td>NatronEngine.NatronEngine.App</td>
<td>7</td>
</tr>
<tr>
<td>x</td>
<td>NatronEngine.NatronEngine.Double2DTuple</td>
<td>27</td>
</tr>
<tr>
<td>x</td>
<td>NatronEngine.NatronEngine.Double3DTuple</td>
<td>29</td>
</tr>
<tr>
<td>x</td>
<td>NatronEngine.NatronEngine.Int2DTuple</td>
<td>46</td>
</tr>
<tr>
<td>x</td>
<td>NatronEngine.NatronEngine.Int3DTuple</td>
<td>47</td>
</tr>
<tr>
<td>y</td>
<td>NatronEngine.NatronEngine.Double2DTuple</td>
<td>27</td>
</tr>
<tr>
<td>y</td>
<td>NatronEngine.NatronEngine.Double3DTuple</td>
<td>29</td>
</tr>
<tr>
<td>y</td>
<td>NatronEngine.NatronEngine.Int2DTuple</td>
<td>46</td>
</tr>
<tr>
<td>y</td>
<td>NatronEngine.NatronEngine.Int3DTuple</td>
<td>47</td>
</tr>
<tr>
<td>z</td>
<td>NatronEngine.NatronEngine.Double3DTuple</td>
<td>29</td>
</tr>
<tr>
<td>z</td>
<td>NatronEngine.NatronEngine.Int3DTuple</td>
<td>47</td>
</tr>
</tbody>
</table>