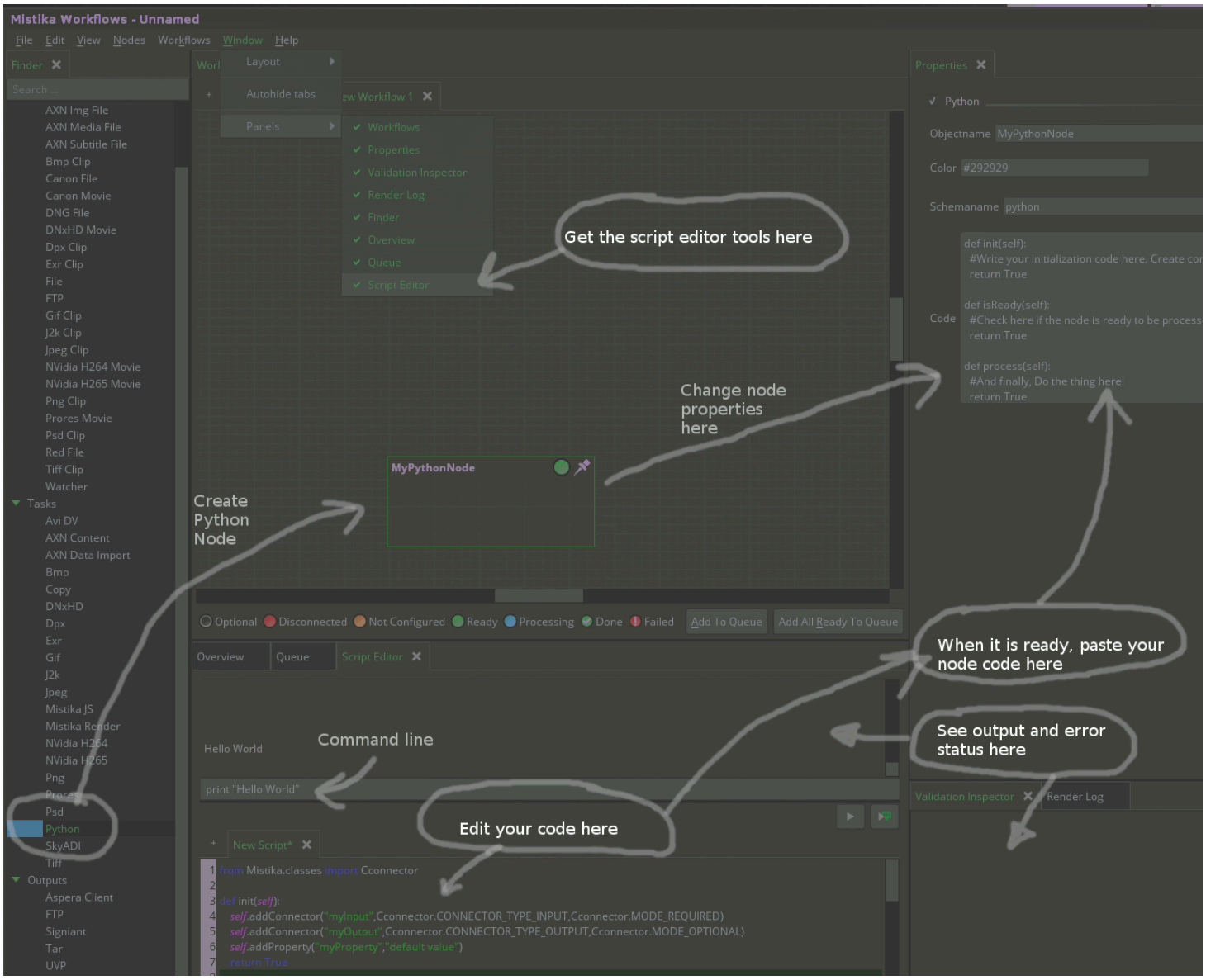
Mistika Workflows

Introduction to the Python script editor

Python scripting tools in Mistika Workflows



Introduction

Mistika Workflows supports **Python 2.7**. There are more modern versions of this language, but **v2.7** is currently the most used version in the media industry, and it has been adopted for that reason.

Mistika Workflows provides Python scripting tools in the **Window->Panels->ScriptEditor,** where you can show and hide all the Python scripting tools described below.

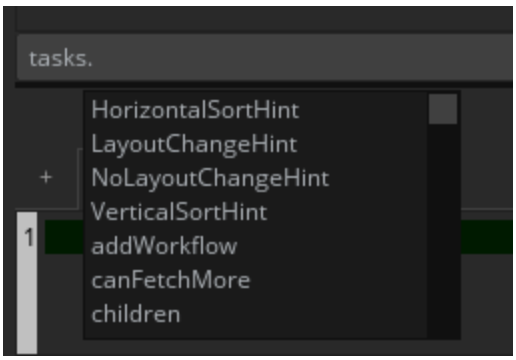
It is recommended to save one layout for programmers with those tools and a different one for normal users. (you can do it with **Window->Layout->Load / Save**)

Script Editor

Python command line (and syntax help): To execute Python command lines, with access to all the data structures of the task nodes. All the GUI functionality is also available as python predefined classes.

The **command line** is also the best way to get help about the syntax of Mistika classes: Whenever you start typing anything that could be matched to a Mistika class it will open a contextual menu automatically, which will show all the possible alternatives to complete the label (you can click on any of them to save typing the full syntax).

As we can see in the next example we have simply typed "**tasks.**" in the command line field, and all the possible functions starting with that prefix have appeared in the contextual list:



You can also use the command line to test interactions with the application (similar to the GUI interface). The following are some command line examples (you can also put them in your node scripts):

Create a Workflow named “TestWorkflow”

```
w=workflows.addWorkflow("TestWorkflow")
```

Add a node of “File” type to the active workflow, and store it in the **f** variable.

```
f=w.addNode("File",0)
```

Connect the “To” connector of the node **f** to the “From” connector of node **c**.

```
f.firstConnectorByName("To").link(c.firstConnectorByName("From"))
```

Note: In that example the “0” can be substituted by a more friendly label, but that will be covered in more complex examples below...

Send the workflow TestWorkflow to the execution queue, and store the handle in the t variable:

```
t=tasks.addWorkflow(“TestWorflow”)
```

Execute that workflow

```
t.start
```

Validation inspector: Provides an output console for Python code execution, including standard output and descriptive error messages.

Python text editor: A text editor with support for Python syntax highlighting.

IMPORTANT NOTE: If you want to copy / paste Python code from external documents it is important to paste it into the **Script Editor** panel, not directly into the Python code field.

This is because the **Code section** of the Python node is not a text editor and it can not deal with external formatting, custom indentations and other hidden codes embedded in the pasted text. Meanwhile the Script Editor provides full conversion and it is Python-syntax aware.

Python task nodes

A “Python” task node that can be used any part of a workflow. It is capable to execute standard Python code, also with access to all the data structures of Mistika Workflows. All the GUI functionality is also available as python predefined procedures. For example, the code contained in just one Python node instance can create multiple workflows with other nodes, send them to the jobs queue and finally execute them.

The Python task node need to have at least 3 functions defined:

Init: Code to be executed when creating a new instance. Return True if no error, or False in case of error.

IsReady: Return True if the node has everything that it needs to run, or not yet (False). As a difference to the init procedure this function is being constantly evaluated to refresh any change in the workflow or any changes in node parameters. The light indicator of each node will also reflect the IsReady state.

process: The actual code to execute when executing the workflow in the job queue. Return True if no error, False in case of errors.

Example: (a “hello world” node):

```
def init(self):
  return True

def.isReady(self):
  return True

def process(self):
  print  "Hello World"
  return True
```

Interface layout of a Python node

You can also control the layout of a python node (how it is drawn in the interface, the properties you add, its color, etc), by putting an xsd scheme file in this folder:

.SGO AppData/shared/scripts/nodeName.xsd

Specifically, if you add properties that you want to be visible to the user you will need to add them to that file (you can also remove the ones that you don't want him to see, for example the source code).

You can use other xsd scheme files as a reference, the basic Python xsd is here:

/home/mistika/SGO AppData/shared/config/schemas/python.xsd

External CLI interface

The Mistika Workflows executable can be passed a "-r script" parameter to open the application and then run a Python script automatically.

workflows -r "Path_To_Python_Script.py"

Once the script is executed the Misitka Workflows application will stay running (so the user can interact with it).

In this case the appearance of the interactive GUI is optional: it will appears by default, but it can be hidden with the "**-s**" parameter.

If the application is executed as a command line then it is important to add this command at the end of the script:

```
workflow.quit
```

Otherwise Mistika Workflows will not realize that the script is finished and needs to exit, so the command will seem to be frozen forever...

Python autoexec file

An autoexec Python script that is executed every time that Mistika WF is executed, which can also be used to automate common needs.

The autoexec script needs to be here: **SGO AppData/shared/scripts/autoexec/autoexec.py**

Forum: Mistika Workflows Python scripts

We recommend to subscribe to this forum as the primary support resource for Python related questions. Only the Python forum is attended by Python programmers (as a difference to the support ticket system and other support channels):

https://forums.sgo.es/forums/index.php?/forum/77-python-scripting/

In the next pages you will find usage examples (typically from the forum).

Basic Python node examples

Example 1: A basic Python node

This is a Python node example from the Mistika Workflows Python scripts forum..

First, let's see the whole example:

```
from Mistika.classes import Cconnector

def init(self):
```

```
self.addConnector("myInput",Cconnector.CONNECTOR_TYPE_INPUT,Cconnector.MODE_REQUIRED)

self.addConnector("myOutput",Cconnector.CONNECTOR_TYPE_OUTPUT,Cconnector.MODE_OPTIONAL
)
    self.addProperty("myProperty","default value")
    return True

def isReady(self):
    return True

def process(self):
    i=self.firstConnectorByName("myInput")
    o=self.firstConnectorByName("myOutput")
    print "input: "+i.url()
    print i.universalPath().files()
    print "myProperty: "+self.myProperty
    o.setUrl(i.url())
    return True
```

Now let's examine de sections:

```
from Mistika.classes import Cconnector
```

included to avoid writing the whole path to access the Cconnector constants in the code.

The initialization function: init(self)

```
def init(self):

self.addConnector("myInput",Cconnector.CONNECTOR_TYPE_INPUT,Cconnector.MODE_REQUIRED)

self.addConnector("myOutput",Cconnector.CONNECTOR_TYPE_OUTPUT,Cconnector.MODE_OPTIONAL
)
    self.addProperty("myProperty","default value")
```

```
    return True
```

This function initializes the node. It can be used to create the connectors and properties to be used in the node. the parameter received is the node itself.
The function to create the connectors is:

addConnector (name, type, mode)

The parameters are:

1. The Connector Name.
2. The connector Type. it can be either Cconnector.CONNECTOR_TYPE_INPUT or Cconnector.CONNECTOR_TYPE_OUTPUT for input or output connectors.
3. The connector mode. It can be Cconnector.MODE_REQUIRED or Cconnector.MODE_OPTIONAL defining if the connector must be connected to enable to workflow, of if it is an optional input to the node.

Properties can also be added with :

addProperty (name, defaultValue)

the parameters are:

1. The parameter Name
2. The default Value. This parameter is optional. If it is not defined, the attribute is initialized to null.

Finally, the init function returns True. Meaning the initialization has been done successfully. The function can return False on error.

The ready function: isReady(self)

```
def isReady(self):
    return True
```

This function is used to define if the node is ready for processing (returning True) or not ready (returning False). It can be used to validate the integrity of the inputs and properties, to be sure the node is not processed if relevant information is missing.

The Process function: process(self)

```
def process(self):
```

```
    i=self.firstConnectorByName("myInput")
    o=self.firstConnectorByName("myOutput")
    print "input: "+i.url()
    print i.universalPath().files()
    print "myProperty: "+self.myProperty
    o.setUrl(i.url())
    return True
```

Finally, the process function executes the task
In this case, we just print in the console the contents of the property and the input files
in order to access the input connector, the function firstConnectorByName is used.
This function returns the first input connector found with the given Name.
the prints show the url defined in the input connection with

```
print "input:"+i.url()
```

then print the list of all the files in the input sequence with

```
print i.universalPath().files()
```

Note: Dealing with Universal Paths is something we will cover in a different post.
then prints the property value with

```
print "myProperty: "+self.myProperty
```

and finally, it sets the output connector with the same sequence found in the input connector with

```
o.setUrl(i.url())
```

That will send the unaltered sequence to the next node.
Returning True to indicate success

Example 2. A copy node with an added property visible in the GUI.

This node just copies the input to the path defined in the property dstFile and sends it to the output for the next node.

```
from shutil import copyfile
```

```
def init(self):                          # initialization function to create in/out
connectors and properties (attributes)
 self.addConnector("in",1,1)         # 1=input. 1=mandatory
 self.addConnector("out",2,0)         # 2=output. 0=optional
 self.addProperty('dstFile')
 return True                 # successful initialization

def isReady(self):                       # check if the node can be processed. in this
example, dstFile needs to be defined
 ifself.dstFile!='':
  return self.critical('myErrorID','dstFile can not be empty')
 return True

def process(self):    #copy in file to dst File and set output connector)
 input=self.firstConnectorByName("in")
 output=self.firstConnectorByName("out")
 copyfile(input.url(),self.dstFile)
 output.setUrl(self.dstFile)
 return True         # performed ok (no error management in this example)
```

In order to make it work correctly, you will need to create an schema file for the node properties. it should be in "SGO AppData/shared/config/schemas" and it should be something like this: (myCopy.xsd)

Noet: This example is just a copy of python.xsd but adding the “dstFile” property:

```
<xsd:schema xmlns:sgo="http://www.sgo.es/2017/sgo.xsd"
            xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<xsd:element sgo:categoryType="categoryCollapsible" name="Python" sgo:visibility="0" sgo:sortOrder="0">
    <xsd:complexType>
        <xsd:sequence>
            <xsd:element name="objectName" type="xsd:string" sgo:visibility="0" sgo:sortOrder="0"/>
            <xsd:element name="color" type="color" sgo:visibility="0" sgo:sortOrder="1"/>
            <xsd:element name="dstFile" type="universalPath" sgo:visibility="0" sgo:sortOrder="2"
sgo:properties="pathNameFilters('All Files (*.*)')"/>
            <xsd:element name="schemaName" type="xsd:anyURI" sgo:visibility="0" sgo:sortOrder="3"
sgo:properties="pathNameFilters('xsd Files (*.xsd)')"/>
```

```
            <xsd:element name="code" type="textEditor" sgo:visibility="0" sgo:sortOrder="4"/>
        </xsd:sequence>
    </xsd:complexType>
</xsd:element>
</xsd:schema>
```

If you want to remove the code and some schema details (once the node is finished and well tested) you can also remove default lines, in the next example we have removed the source code and the scheme file, to keep it out of sight for normal users:

```
<xsd:schema xmlns:sgo="http://www.sgo.es/2017/sgo.xsd"
            xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<xsd:element sgo:categoryType="categoryCollapsible" name="Python" sgo:visibility="0" sgo:sortOrder="0">
    <xsd:complexType>
        <xsd:sequence>
            <xsd:element name="objectName" type="xsd:string" sgo:visibility="0" sgo:sortOrder="0"/>
            <xsd:element name="color" type="color" sgo:visibility="0" sgo:sortOrder="1"/>
            <xsd:element name="dstFile" type="universalPath" sgo:visibility="0" sgo:sortOrder="2"
sgo:properties="pathNameFilters('All Files (*.*)')"/>
        </xsd:sequence>
    </xsd:complexType>
</xsd:element>
</xsd:schema>
```