Python/Maya: Introductory tutorial

Author: Matthias Baas (mbaas@users.sourceforge.net)
Date: 2006/05/08

Contents

This tutorial introduces the usage of the Python Maya package. It provides several "Hello World" scripts that already demonstrate the basics of using Python within Maya.

Hello World - Take 1: Invoking a Python script

The most simple (and boring) Hello World script one could ever image is the following:

```
# helloworld1.py: The very first Hello World Python script for Maya

print "Hello World!"
```

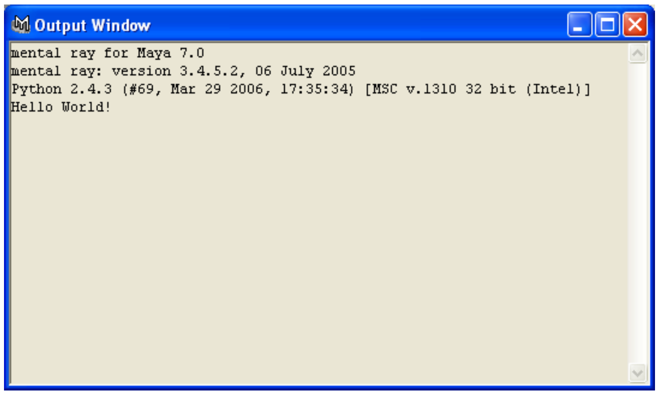
But at least you can use this script to test your Python plugin. So write a file `helloworld1.py` with the above contents and stick it into your Maya directory under `maya/<version>/scripts`. This is the same location where you would put MEL scripts.

Now launch Maya, open the MEL Script Editor and enter:

```
pySource helloworld1;
```

(this assumes you have successfully installed (and loaded) the Maya Python plugin)

To execute this line hit Ctrl+Enter or the Enter key on the numpad. If you have a look at the Output Window (or the console) you should see the text "Hello World!" printed to it. This means the Python interpreter is working and you can execute Python code from within Maya.

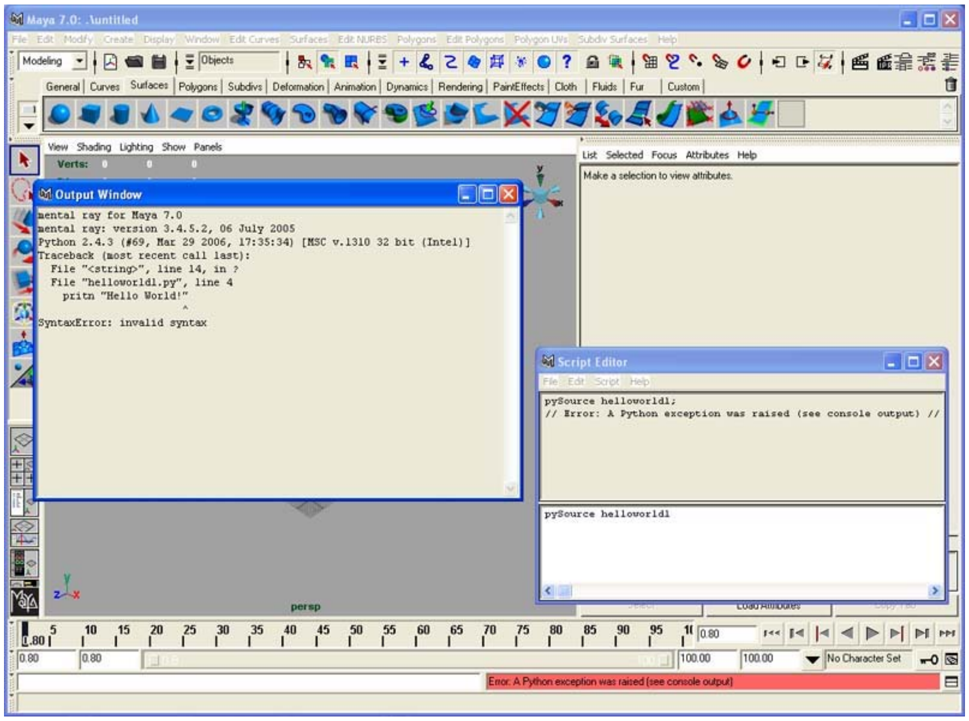


```
mental ray for Maya 7.0
mental ray: version 3.4.5.2, 06 July 2005
Python 2.4.3 (#69, Mar 29 2006, 17:35:34) [MSC v.1310 32 bit (Intel)]
Hello World!
```

The `pySource` command is equivalent to the MEL command `source`. It searches for a Python script at the same locations where MEL scripts can be stored and executes it. The Python plugin defines one additional MEL command, `python`, which you can use to execute Python code given as a string:

```
py "print 'Hello World!'";
```

As you already noticed the output from the print statement is directed to the Output Window. This is also where tracebacks appear when there is an error in your Python script. For example, suppose we misspelled the print statement and executed the script. Then we would see something like this:



```
mental ray for Maya 7.0
mental ray: version 3.4.5.2, 06 July 2005
Python 2.4.3 (#69, Mar 29 2006, 17:35:34) [MSC v.1310 32 bit (Intel)]
Traceback (most recent call last):
  File "<string>", line 14, in ?
  File "helloworld1.py", line 4
    pritn "Hello World!"
                       ^
SyntaxError: invalid syntax
```

```
pySource helloworld1;
// Error: A Python exception was raised (see console output) //
```

```
pySource helloworld1
```

Error: A Python exception was raised (see console output)

When any code executed by `pySource` or `python` raises an exception, the MEL command fails which you will notice in Maya's status line and in the script editor. The details about the error (i.e. the traceback) is printed in the output window (or the console if you are not under Windows). If you are executing MEL code from within your Python script (you will see how this is done in a short moment) and this MEL code produces an error, the situation is just the opposite. You will see a Python exception telling you that executing MEL code failed but more details about this will then be available in the Script Editor. So always keep an eye on both windows whenever an error occurs.

So far so good. Being able to execute Python code is always a pleasing thing to do, but for the ultimate experience it would be desirable if we could somehow communicate with Maya and do stuff with it. That's where the Python package called `maya` comes into play.

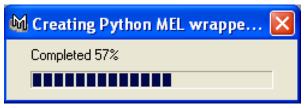
Hello World - Take 2: Using the mel module

Let's rewrite the above script from scratch and do a GUI version of the Hello World script:

```
# helloworld2.py: The second Hello World Python script for Maya

from maya import mel

mel.confirmDialog(message="Hello World!", button=["OK"])
```

(in addition to the Python plugin this also requires the Maya Python package, otherwise you will get an ImportError exception)

This script uses the `mel` module that contains Python wrappers around all MEL commands found in Maya. When you run the script (using pySource) you might notice that there is a little progress window popping up and it takes a few seconds before the actual dialog from our script is shown. This progress window appears the first time the mel module is imported (which might already have been the case when you already did the package initialization):



Creating Python MEL wrappe...

Completed 57%

What actually happens during these few seconds is that the mel module gets written (literally!). Maya is queried for all existing MEL commands and appropriate Python wrappers are created. This can take a few seconds as there are quite a lot of MEL commands in Maya. Try the following line in the MEL Script Editor:

```
py "from maya import mel; print len(dir(mel))";
```

The exact number of functions depends on the plugins you have loaded. The advantage of this dynamic module creation is that you even get wrappers for commands defined in third-party plugins. For example, even the MEL functions defined by the Python plugin itself did get wrapped:

```
py "from maya import mel; print mel.pySource";
```

But back to the example. Eventually, you get the actual output from our example script:

So our Python script just invoked the following MEL command:

```
confirmDialog -message "Hello World!" -button "OK";
```

You can call *any* MEL function this way from Python and do the same things you can do with MEL. Any flag that a MEL command accepts is turned into an optional Python keyword argument. You can still choose between the long version or the short version, so the following line has the same effect than the above code:

```
mel.confirmDialog(m="Hello World!", b=["OK"])
```

Of course, you can also mix long and short names. The button argument of the confirmDialog command is actually a "multi-use" flag, i.e. it can appear more than once in one call. In Python, it is not possible to specify the same keyword argument several times, instead a list of argument values has to be used. That's why the string "OK" has to be put into a list and cannot be passed directly (you could actually specify more than one string to create several buttons).

Some flags don't take any value in MEL, their mere presence already enables a particular option. Such flags must take the value None in Python. This indicates that the flag does not take a value:

```
mysphere = mel.sphere(radius=2.5)
print "Radius:",mel.sphere(mysphere, query=None, radius=None)
```

Also note that the regular arguments must always be specified first and the flags must be last, including the query or edit flag. In MEL, it's just the opposite. But this is because in Python positional arguments must appear before keyword arguments.

Hello World - Take 3: Using the C++ SDK

With the above knowledge you can already do anything you could do with MEL, but why stop here? Maya also provides a C++ SDK which allows some things that cannot be done with MEL (such as defining new nodes, etc.). The classes from the C++ SDK are also available in Python through the `api` module:

```
# helloworld3.py: The third Hello World Python script for Maya

from maya.api import *

MGlobal.displayInfo("Hello World!")
```

When you invoke this script you will see the "Hello World!" message in the status line of Maya (and in the Script Editor). Not very spectacular you might say, but note how this was done. The MGlobal class is actually a C++ class from the Maya SDK that provides a couple of static methods such as displayInfo(). And just as it does this in C++, it also does this in Python.

The classes in the api module were kept as close as possible to their C++ counterparts so that the API documentation that comes with Maya can be applied to the Python version as well. To the experienced Python user this might occasionally look somewhat odd and the "Pythonic" feel sometimes gets lost such as in:

```
from maya.api import *
sel = MSelectionList()
MGlobal.getActiveSelectionList(sel)
print sel
```

The method getActiveSelectionList() returns a list of currently selected nodes. But the list is returned by populating the provided MSelectionList object and the actual return value is a MStatus object (just as in C++). But on the other hand, this makes translating plugins between C++ and Python pretty straightforward as there are less special syntax rules that have to be known. However, in some (rare) cases it was unavoidable to change the name or the signature of a method simply because of the language differences between C++ and Python (see the User Manual).

Warning: Using the classes from the api module requires some knowledge about how Maya works and about what you are allowed to do and what you are not allowed to do. While using the mel module in any way is rather safe, you are not protected against crashes when messing with the api module.

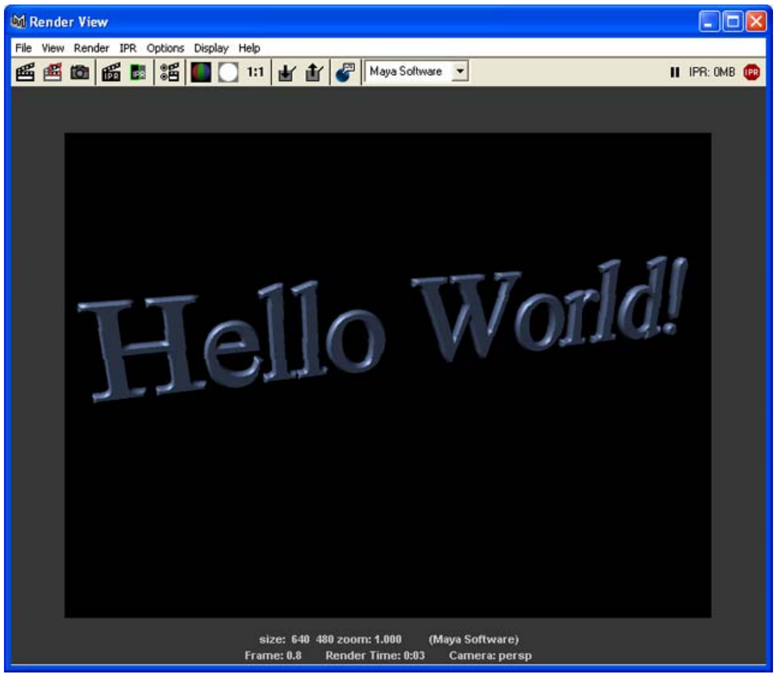
Hello World - Take 4: The showdown

Finally, here is a bit more involved version of a Hello World script:

```
# helloworld4.py: The fourth and fanciest Hello World Python script for Maya

from maya.mel import *

# Delete any existing scene
file(newFile=None, force=None)

# Create the "Hello World" bevelled text...
text = createNode("makeTextCurves")
setAttr(text+".text", "Hello World!", type="string")
setAttr(text+".font", "Times New Roman", type="string")

innerstylecrv = createNode("styleCurve")
outerstylecrv = createNode("styleCurve")
setAttr(innerstylecrv+".style", 0)
setAttr(outerstylecrv+".style", 0)
```

```
bevel = createNode("bevelPlus")
setAttr(bevel+".width", 0.1)
setAttr(bevel+".depth", 0.1)
setAttr(bevel+".extrudeDepth", 0.25)
setAttr(bevel+".capSides", 4)
setAttr(bevel+".numberOfSides", 4)
setAttr(bevel+".tolerance", 0.01)
setAttr(bevel+".normalsOutwards", True)
setAttr(bevel+".polyOutUseChordHeight", False)
setAttr(bevel+".polyOutUseChordHeightRatio", False)
setAttr(bevel+".orderedCurves", True)

meshtransform = createNode("transform", name="Hello_World")
mesh = createNode("mesh", name="Hello_World_Shape", parent=meshtransform)

connectAttr(text+".outputCurve", bevel+".inputCurves")
connectAttr(text+".count", bevel+".count")
connectAttr(text+".position", bevel+".position")
connectAttr(innerstylecrv+".outCurve", bevel+".innerStyleCurve")
connectAttr(outerstylecrv+".outCurve", bevel+".outerStyleCurve")
connectAttr(bevel+".outputPoly", mesh+".inMesh")

# Create and assign a new material...
sg = sets(renderable=True, noSurfaceShader=True, empty=None, name="HW_SG")
sets(mesh, edit=None, forceElement=sg)
blinn = shadingNode("blinn", asShader=None, name="HW")
setAttr(blinn+".color", (0.5, 0.6, 0.8))
connectAttr(blinn+".outColor", sg+".surfaceShader")

# Deselect all objects
select(clear=None)

# Adjust the perspective camera
setAttr("persp.translate", (2.5, 8, 22))
setAttr("persp.rotate", (160, -160, 180))
setAttr("persp.rotateOrder", 0)

# Switch to a single perspective view...
eval('setNamedPanelLayout("Single Perspective View")')
eval('updateToolbox()')
eval('findNewCurrentModelView()')

# Rennder the image...
RenderIntoNewWindow()
```

This script creates a bevelled 3D text, assigns a material, adjusts the perspective view and renders the image.



size: 640 480 zoom: 1.000 (Maya Software)
Frame: 0.8 Render Time: 0:03 Camera: persp

At this point you have now seen some of the possibilities you have with using Python within Maya. There are a couple of things that have not been covered yet, such as:

- Using the Python Script Editor (this is equivalent to the MEL Script Editor)
- Writing Python plugins and using the Python plugin manager (again, writing plugins in Python works almost exactly as in C++).
- Using the special GUI classes from the `gui` module.

But hopefully this tutorial has already been enough to get you going with Python and Maya.