Generating texture list by python:

texture\_types = set(['tif', 'tga', 'png'])

used\_files = set()

for file\_node in [cmds.ls](http://cmds.ls)(type='file') :

   file\_path = cmds.getAttr("%s.fileTextureName" % file\_node)

   if file\_path.lower().rsplit('.', 1)[-1] in texture\_types:

       used\_files.add(file\_path)

print used\_files

logging and maya script editor:

import logging  
import maya.utils  
  
myLogger = logging.getLogger("MyLogger")  
myLogger.propagate = False  
handler = maya.utils.MayaGuiLogHandler()  
handler.setLevel(logging.INFO)  
formatter = logging.Formatter("%(asctime)s %(message)s")  
handler.setFormatter(formatter)  
myLogger.addHandler(handler)  
  
myLogger.warning('is when this event was logged.')

maya debug:

setenv MAYA\_DEBUG\_ENABLE\_CRASH\_REPORTING 1

It enables a dump of the last actions, saved into a file with a .crash suffix.

It a Windows only thing. You can attach Visual Studio to Maya then use  
the following command to print stuff for python into the debug stream.

import win32api;  
win32api.OutputDebugString ('some message I want to see')

MAYA\_CMD\_FILE\_OUTPUT

This variable enables the cmdFileOutput command during startup so the content in the script editor automatically outputs to a designated file.

If no file name is specified, then errors are sent to the standard error output, for example the Terminal window.

Example (for windows):   
maya.exe -log c:\debug.log   
  
There are a couple of other debug options that can be added to   
your .env file.   
  
MAYA\_DEBUG\_ENABLE\_CRASH\_REPORTING is an environmental variable that   
when set to 1 will cause Maya to try dumping debug files to your %temp   
% directory. It dumps stack traces and other things that maybe helpful   
for debugging plugin crashes. As far as I know you can not change the   
path that these files get written to.   
  
MAYA\_CMD\_FILE\_OUTPUT is also an environmental variable. When set to   
the path of a log file it will output everything from the script   
editor window into the log.

[**Embedding a Maya widget into a PyQt UI**](http://nathanhorne.com/?p=381)

import maya.OpenMayaUI as apiUI

from PyQt4 import QtGui, QtCore

import sip

def getMayaWindow():

ptr = apiUI.MQtUtil.mainWindow()

return sip.wrapinstance(long(ptr), QtCore.QObject)

def toQtObject(mayaName):

'''

Given the name of a Maya UI element of any type,

return the corresponding QWidget or QAction.

If the object does not exist, returns None

'''

ptr = apiUI.MQtUtil.findControl(mayaName)

if ptr is None:

ptr = apiUI.MQtUtil.findLayout(mayaName)

if ptr is None:

ptr = apiUI.MQtUtil.findMenuItem(mayaName)

if ptr is not None:

return sip.wrapinstance(long(ptr), QtCore.QObject)

class MayaSubWindow(QtGui.QMainWindow):

def \_\_init\_\_(self, parent=getMayaWindow()):

super(MayaSubWindow, self).\_\_init\_\_(parent)

self.executer = cmds.cmdScrollFieldExecuter(sourceType="python")

qtObj = toQtObject(self.executer)

#Fill the window, could use qtObj.setParent

#and then add it to a layout.

self.setCentralWidget(qtObj)

myWindow = MayaSubWindow()

myWindow.show()

## [Assigning to \_\_main\_\_ in Maya](http://nathanhorne.com/?p=71)

you can add objects to the \_\_main\_\_ namespace (The script editor’s interactive prompt), this is also where python will search when you use the MEL command python();

#import the interactive main module

import \_\_main\_\_

def someCommand():

'''This is a pretty basic function...'''

print 'someCommand called!'

#now assign it to an attribute in \_\_main\_\_

\_\_main\_\_.someCommand = someCommand

python("someCommand()");

// someCommand called!

## [Python API grab frame buffer to image](http://nathanhorne.com/?p=261)

This code will grab the frame buffer from the active viewport (You could also change it to be a specified viewport with little work) and write it to any format that MImage supports:

#Import api modules

import maya.OpenMaya as api

import maya.OpenMayaUI as apiUI

#Grab the last active 3d viewport

view = apiUI.M3dView.active3dView()

#read the color buffer from the view, and save the MImage to disk

image = api.MImage()

view.readColorBuffer(image, True)

image.writeToFile('C:/test.jpg', 'jpg')

## [Maya Python print to output window](http://www.rtrowbridge.com/blog/2008/12/maya-python-print-to-output-window/)

import sys

import maya.mel as mel

import maya.cmds as cmds

dagNodes = cmds.ls( dagObjects=True )

# print nodes in script editor, slow

print "List of nodes in the scene///////////////////////////////////////"

for dag in dagNodes:

print ('Dag Node: ' + dag)

# print nodes in output window, messy

mel.eval('trace "List of nodes in the scene///////////////////////////////////////";')

for dag in dagNodes:

mel.eval('trace (\"' + 'Dag Node: ' + dag + '\");')

# print nodes in output window, using Python sys module

sys.\_\_stdout\_\_.write( 'List of nodes in the scene///////////////////////////////////////\n' )

for dag in dagNodes:

sys.\_\_stdout\_\_.write( 'Dag Node: ' + dag + '\n')

## get selection from screen

import maya.OpenMaya as api

def selectFromScreenApi(x, y, x\_rect=None, y\_rect=None):

#get current selection

sel = api.MSelectionList()

api.MGlobal.getActiveSelectionList(sel)

#select from screen

args = [x, y]

if x\_rect!=None and y\_rect!=None:

api.MGlobal.selectFromScreen(x, y, x\_rect, y\_rect, api.MGlobal.kReplaceList)

else:

api.MGlobal.selectFromScreen(x, y, api.MGlobal.kReplaceList)

objects = api.MSelectionList()

api.MGlobal.getActiveSelectionList(objects)

#restore selection

api.MGlobal.setActiveSelectionList(sel, api.MGlobal.kReplaceList)

#return the objects as strings

fromScreen = []

objects.getSelectionStrings(fromScreen)

return fromScreen

#print selectFromScreenApi(0, 0)

print selectFromScreenApi(0, 0, 640, 480)

# get selection from view

import maya.OpenMaya as om

import maya.OpenMayaUI as omu

view = omu.M3dView.active3dView()

om.MGlobal.selectFromScreen(0, 0, view.portWidth(), view.portHeight(), om.MGlobal.kReplaceList)

## [Python API MTransformationMatrix.getRotation() bug](http://www.rtrowbridge.com/blog/2009/02/python-api-mtransformationmatrixgetrotation-bug/)

I found this out the hard way while working on my 4x4MatrixToTRS plugin. So I thought it would be handy to pass along. There are a few bugs with the MTransformationMatrix class when using Python. You can get around them, you just need to know about them to do so. The main bug seems to be that any function of this class that asks for **RotationOrder &order** will not work with Python. It will always throws an error.

So if you are attempting to get the Euler rotation in a certain order like I was then you need to find a work around. What I found was I just need to use the MTransformationMatrix.rotation() function to get the quaternion then I could use the MQuaternion.asEulerRotation() function and from there I could use the MEulerRotation functions to get what I needed. Here are two examples one that does not work that should and one that does work. I have let Autodesk know about this issue.

If you want to get the rotation value of the MTransformationMatrix class this is the wrong way to do it:

import maya.OpenMaya as OpenMaya

getMatrix = OpenMaya.MMatrix()

matrixList = ( 0.673, 1.545 , 0.0, 0.0,

-0.327, 2.545, 0.0, 0.0,

-0.327, 1.545, 1.0, 0.0,

-0.327, 1.545, 0.0, 1.0)

OpenMaya.MScriptUtil().createMatrixFromList(matrixList, getMatrix)

mTM = OpenMaya.MTransformationMatrix( getMatrix )

rotDoubleArray = OpenMaya.MScriptUtil()

rotDoubleArray.createFromList( [0.0, 0.0, 0.0], 3 )

rotDoubleArrayPtr = rotDoubleArray.asDoublePtr()

rotOrder = OpenMaya.MTransformationMatrix().kXYZ

mTM.getRotation( rotDoubleArrayPtr, rotOrder )

# Error: Wrong number of arguments for overloaded function 'MTransformationMatrix\_getRotation'.

Possible C/C++ prototypes are:

getRotation(double [3],MTransformationMatrix::RotationOrder &)

getRotation(double [3],MTransformationMatrix::RotationOrder &,MSpace::Space)

# Traceback (most recent call last):

# File "", line 1, in

# File "C:\buildforge\Maya\_Main\_Win32\_Build\build\wrk\optim\runTime\Python\Lib\site-packages\maya\OpenMaya.py", line 8044, in getRotation

# NotImplementedError: Wrong number of arguments for overloaded function 'MTransformationMatrix\_getRotation'.

# Possible C/C++ prototypes are:

# getRotation(double [3],MTransformationMatrix::RotationOrder &)

# getRotation(double [3],MTransformationMatrix::RotationOrder &,MSpace::Space)

# #

Here is the right way:

import maya.OpenMaya as OpenMaya

getMatrix = OpenMaya.MMatrix()

matrixList = ( 0.673, 1.545 , 0.0, 0.0,

-0.327, 2.545, 0.0, 0.0,

-0.327, 1.545, 1.0, 0.0,

-0.327, 1.545, 0.0, 1.0)

OpenMaya.MScriptUtil().createMatrixFromList(matrixList, getMatrix)

mTM = OpenMaya.MTransformationMatrix( getMatrix )

rotOrder = OpenMaya.MTransformationMatrix().kXYZ

# Get the rotation as a quaternion then convert to Euler as needed

rotateOrder = 0

mquat = mTM.rotation()

rot = mquat.asEulerRotation()

rot.reorderIt( rotateOrder )

print rot.x, rot.y, rot.z

## [Vectors in Maya Python](http://www.rtrowbridge.com/blog/2008/11/python-and-api-mvectors/)

The MVector is actualy much easier to work with. You do not need to use the MScriptUtil class with MVector’s you can just simply give it a default value and start working with the new object. For example:

# Import the general maya API classes

import maya.OpenMaya as OpenMaya

# Create two MVectors

vectorA = OpenMaya.MVector(1.0, 1.0, 1.0)

vectorB = OpenMaya.MVector(2.0, 2.0, 2.0)

# Do some operations on them

vectorC = (vectorA + vectorB)

vectorD = (vectorA - vectorB)

# vectorC after adding vectorA and vectorB

print vectorC.x

print vectorC.y

print vectorC.z

# vectorD after subtracting vectorA and vectorB

print vectorD.x

print vectorD.y

print vectorD.z

# get the length of vectorC

print vectorC.length()

# normalize vectorC

vectorC.normalize()

# print the new normalized values

print vectorC.x

print vectorC.y

print vectorC.z

## [Matrices in Maya Python](http://www.rtrowbridge.com/blog/2008/11/vectors-and-matrices-in-python/)

I have talked to a few people that think there are no good ways to deal with Vectors or Matrices in Python. I wanted to present an example of how to use the Maya API to create an MMatrix class and introduce how the MScriptUtil class works to deal with API functions that use pointers.

# import mayas general API classes

import maya.OpenMaya as OpenMaya

# the MScriptUtil class is very important to python API users

# it allows python users to pass variable to an API function like it were a pointer

# Called by RTrunPointerExample()

def RTpointerExample( ptr ):

print 'ptr changed using MScriptUtil';

OpenMaya.MScriptUtil.setDouble ( ptr, 2.0 )

def RTrunPointerExample():

# Define a MScriptUtil object

fooba = OpenMaya.MScriptUtil()

# Define the pointer type

# only use the types that end in Ptr

ptr = fooba.asDoublePtr()

# Set a value

OpenMaya.MScriptUtil.setDouble ( ptr, 1.0 )

# Print that value

print 'ptr = ', (OpenMaya.MScriptUtil.getDouble( ptr ))

# Call a function passing it the pointer

RTpointerExample( ptr )

# Print the new value

print 'ptr = ', (OpenMaya.MScriptUtil.getDouble( ptr ))

# execute this

RTrunPointerExample()

# which prints:

>> ptr = 1.0

>> ptr changed using MScriptUtil

>> ptr = 2.0

The above is a simple example of how pass the variable ‘ptr’ as a pointer to RTpointerExample and to change the variable which in turn changes it in the first function RTrunPointerExample. This is sometimes needed to properly interact with API functions since python has no concept of pointers. My goal is not to describe the pointer concept completely but to just show how you interact with the MScriptUtil class.

So here is an example of using the MScriptUtil class to set MMatrix variables.

# import mayas general API classes

import maya.OpenMaya as OpenMaya

# initialize a matrix using a python list

# this is how a matrix can be initialized using a python list

valListA = (1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16)

valListB = (1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16)

matrixA = OpenMaya.MMatrix()

matrixB = OpenMaya.MMatrix()

# Create a matrix using the list

OpenMaya.MScriptUtil.createMatrixFromList( valList, matrixA)

OpenMaya.MScriptUtil.createMatrixFromList( valList, matrixB)

# Multiply matrixA \* matrixB will return a matrix so you dont need to define it

matrixC = matrixA \* matrixB

# print row 1

print matrixC(0, 0)

print matrixC(0, 1)

print matrixC(0, 2)

print matrixC(0, 3)

# print row 2

print matrixC(1, 0)

print matrixC(1, 1)

print matrixC(1, 2)

print matrixC(1, 3)

# print row 3

print matrixC(2, 0)

print matrixC(2, 1)

print matrixC(2, 2)

print matrixC(2, 3)

# print row 4

print matrixC(3, 0)

print matrixC(3, 1)

print matrixC(3, 2)

print matrixC(3, 3)

# Example of setting a certain index of a matrix to a specific value

matrix = OpenMaya.MMatrix()

OpenMaya.MScriptUtil.setDoubleArray(matrix[0], 0, 1.0) # Sets the first row, first column to 1.0

OpenMaya.MScriptUtil.setDoubleArray(matrix[3], 3, 16.0) # Sets the fourth row, fourth column to 16.0

print matrix(0, 0)

print matrix(3, 3)

Hopefully that gives you some ideas of how you might use a MMatrix with one of your python scripts. Note that it is ok to import maya.cmds and also maya.OpenMaya into the same script. Your script does not need to be a plugin to interact with API classes. Once you have a MMatrix variable you can multiple those variables together easily just like you would be able to do if you were using the MMatrix class in C++.

## [Maya API docs demystified for Python users](http://www.rtrowbridge.com/blog/2009/02/maya-api-docs-demystified-for-python-users/)

Class Function Return Data Type

**MStatus** MItMeshPolygon::getConnectedEdges ( MIntArray &edges )

MStatus is what the function will return when executed, in MEL it looks like:

global proc string foo()

The keyword string tells the procedure that it will return a string. In C++ you MUST define a variables type before putting that value into that type. Very much like MEL. So this would error:

string $foo = "value";

$foo = 1;

>> Error: Line 1.13: Invalid redeclaration of variable "$foo" as a different type. //

So Autodesk shows the return data type at the beginning of the function.

MStatus MItMeshPolygon::getConnectedEdges ( MIntArray &edges )

This is important to C++ users, but not always to Python users.

Something to note and this is true 100% of the time. Anytime a Maya API function states that it returns a MStatus or it asks for a MStatus type you should pretend it doesnt exist when using Python for example:

MItMeshPolygon (const MDagPath &polyObject, MObject &component=MObject::kNullObj, MStatus \*ReturnStatus=NULL)

In python you can only do this:

foo = OpenMaya.MItMeshPolygon( polyObject, component )

The MStatus argument is dropped, attempting to pass it a variable will produce an error.

Another note about returned types Python does not require or even allow you to define a parameter name as a type so you can type anything to get the data from a function. So it does not matter as much what it returns. The only time you need to know what the type is, if you need to work with what is returned so you can call that objects functions.

Class Function Name

MStatus **MItMeshPolygon::getConnectedEdges** ( MIntArray &edges )

The first word before the :: means the object class that this function is a part of. You already know what class this function is a part of because you looked up that class and are looking at the page of functions for that class. Things to note though and this can get confusing when coming out of a script language background like MEL. All the functions listed on the class object MItMeshPolygon page may not be all the functions it can call. It might have more. How can this be? Well some classes are derived from other classes. I cant give a MEL example since MEL has not such thing as a class.

But here is a python example:

class fruit():

def \_\_init\_\_(self):

self.amount = 0

class apple(fruit):

def \_\_init\_\_(self, amt):

self.amount = amt

# Return float value of how many pies the amount attribute will make

# Eights apples to make a pie

def bakepie(self):

return (self.amount / 8.0)

class bannana(fruit):

def \_\_init\_\_(self, amt):

self.amount = amt

# Return float value of how many pies the amount attribute will make

# Four bannans to make a pie

def bakepie(self):

return (self.amount / 4.0)

fruitsack = apple( 25 )

print fruitsack.bakepie()

>>3.125

fruitsack = bannana( 25 )

print fruitsack.bakepie()

>>6.25

The apple and bannana classes derives from the fruit class. This means that the apple and bannana classes also has a .amount attribute. Even though they do not have that attribute themselves. In the Maya API many of the C++ classes have this. So sometimes you might be looking for a function that is avaiable within that very class but it is part of its derived functions.

The MFnDoubleArrayData class is a good example of this. It only has about eight functions, but if you click on its “List of all members” it also has functions from the classes MFnBase and MFnData. Go to this address to see it:

<http://download.autodesk.com/us/maya/2009help/API/class_m_fn_double_array_data.html>

Click on the “List of all members link.

Something you should also note is that a function which says “NO SCRIPT SUPPORT.” does not work in Python.  
You can not call those functions. Many documented functions will describe other methods to use when your using Python. Simply click on that function and it will jump you to a description of that function. Sometimes you just have to find another way to do what your trying to do.

Class Function Arguments

MStatus MItMeshPolygon::getConnectedEdges **( MIntArray &edges )**

This is the functions argument list. As I said before if one of the arguments is a MStatus argument. Ignore it and do not pass anything to that argument or you will get an error. In any case that you see a “&” character that means you are passing a variable as a reference. What this means is that the variable passed to that function can be changed in place and this allows a function in C++ to return data to several different variables at once. A function could change data in each of the referenced variables and also return data as well. Most of the time when you pass data as a reference to a function it only returns a MStatus object back to you, which in Python anything that returns a MStatus returns ‘None’ because the MStatus class is never used in the Python API.

In MEL you can do something like this when you pass an array to a procedure. If the procedure adds or removes anything from that array it will edit that array in place. So if you execute the code below:

global proc foobar( string $fooArray[] )

{

for($i=1; $i < 5; $i++)

{

$fooArray[size($fooArray)] = $i;

}

}

string $foo[];

foobar($foo);

print $foo;

>>1

>>2

>>3

>>4

The reason for this is the $foo array was passed as a reference. This is the only case where MEL does this.  
C++ can do this with any variable. You recognize this by looking for the “&” next to an argument in the C++ API documentation.

Sometimes you will see arguments like MStatus with a “\*” this means it is a pointer, which is another C++ thing. Examples of this are the MStatus class and the MString class. Neither of these classes are avalable for use with Python. In the case of a MString you simply should pass a regular Python string to class that lists a MString type argument.

In C++ you need to define a pointer before passing the argument to a reference. Python does not have either of these as part of its programming language. Because of this the MScriptUtil class implements methods for Python to create fake pointers to help it pass data to arugments where it normally could not.

The MScriptUtil class lets you define a pointer variable which you can then send to a Maya API function and then you need to use the MScriptUtil class to retrieve that data from the variable after the API function has done something with it.

# import mayas general API classes

import maya.OpenMaya as OpenMaya

# the MScriptUtil class is very important to python API users

# it allows python users to pass variable to an API function like it were a pointer

# Called by RTrunPointerExample()

def RTpointerExample( ptr ):

print 'ptr changed using MScriptUtil';

OpenMaya.MScriptUtil.setDouble ( ptr, 2.0 )

def RTrunPointerExample():

# Define a MScriptUtil object

fooba = OpenMaya.MScriptUtil()

# Define the pointer type

# only use the types that end in Ptr

ptr = fooba.asDoublePtr()

# Set a value

OpenMaya.MScriptUtil.setDouble ( ptr, 1.0 )

# Print that value

print 'ptr = ', (OpenMaya.MScriptUtil.getDouble( ptr ))

# Call a function passing it the pointer

RTpointerExample( ptr )

# Print the new value

print 'ptr = ', (OpenMaya.MScriptUtil.getDouble( ptr ))

# execute this

RTrunPointerExample()

# which prints:

>> ptr = 1.0

>> ptr changed using MScriptUtil

>> ptr = 2.0

I have a few examples of how to use MScriptUtil on my site, either use my sites search tool or look here:

<http://www.rtrowbridge.com/blog/2009/02/03/python-api-mtransformationmatrixgetrotation-bug/>  
<http://www.rtrowbridge.com/blog/2008/11/07/python-and-api-mvectors/>  
<http://www.rtrowbridge.com/blog/2008/11/05/vectors-and-matrices-in-python/>

To further show how a reference variable is being used the documentation will display [in] and [out] next to its Parameters. This tells you what the intent of the referenced variable is. So if it says [in] you can forget what you passed to the variable. If it says [out] next to the parameter the function is attempting to change the referenced variable you passed to it. So if you passed a function a referenced MVector that was (0,0,0) it will be changed if you look at it after passing it to the function.

Other things to look for when reading the documentation:

Function Overloading

In C++ you can overload a function to accept several different types of data or to accept different amounts of arguments. For example look at the MVector class:

1. MVector ()  
2. MVector (const MVector &)  
3. MVector (const MFloatPoint &)  
4. MVector (const MFloatVector &)  
5. MVector (const MPoint &)  
6. MVector (double xx, double yy, double zz=0.0)  
7. MVector (const double d[3])  
8. ~MVector ()

MVector() anytime a class calls itself as a “function” it is called a constructor in C++. This is how that class initializes itself and creates a object that can be referenced by a variable in Python.

1. MVector() just initializes that class object to default values. In Python it would look like this:

import maya.OpenMaya as OpenMaya  
vec = OpenMaya.MVector()

2. MVector(const MVector &)  
This overloaded version of the MVector constructor asks for another MVector, the Parameters section of this states it is a [in] parameter so you are passing a MVector in and it will return a MVector which is a copy of the MVector which you passed in. Python example:

import maya.OpenMaya as OpenMaya  
vecA = OpenMaya.MVector()  
vecB = OpenMaya.MVector( vecA )

3. MVector (const MFloatPoint &)  
This overloaded version of the MVector constructor asks for a MFloatPoint class object. The parameter states it is passed [in] so you are passing the MFloatPoint class object in to change the MVector you are constructing.

mfpoint = OpenMaya.MFloatPoint()  
vec = OpenMaya.MVector( mfpoint )

4. MVector (const MFloatVector &)  
This overloaded version of the MVector constructor asks for a MFloatVector class object.

mfvector = OpenMaya.MFloatVector()  
vec = OpenMaya.MVector( mfvector )

5. MVector (const MPoint &)  
Now it is asking for a MPoint class object.

mpnt = OpenMaya.MPoint()  
vec = OpenMaya.MVector( mpnt )

6. MVector (double xx, double yy, double zz=0.0)  
Now it is asking for float values.

vec = OpenMaya.MVector( 1.0, 2.0, 3.0 )

7. MVector (const double d[3])  
This is sketchy documentation. It technically must be passed as a reference. Python can not just pass it a  
double array like this:

vec = openMaya.MVector( [1.0, 2.0, 3.0 )

>># Error: UnknownError: #

>># Traceback (most recent call last):

>># line 1 of

>># line 8311 of file C:\buildforge\Maya\_Main\_Win64\_Build\build\wrk\optim\runTime\Python\Lib\site-packages\maya\OpenMaya.py

>>in module \_\_init\_\_

>># NotImplementedError: Wrong number of arguments for overloaded function 'new\_MVector'.

>># Possible C/C++ prototypes are:

>># MVector()

>># MVector(MVector const &)

>># MVector(MFloatPoint const &)

>># MVector(MFloatVector const &)

>># MVector(MPoint const &)

>># MVector(double,double,double)

>># MVector(double const [3]) #

#The correct way is this:

doubleArray = OpenMaya.MScriptUtil()

doubleArray.createFromList( [1.0, 2.0, 3.0], 3 )

doubleArrayPtr = doubleArray.asDoublePtr()

vec = OpenMaya.MVector( doubleArrayPtr )

print vec.x, vec.y, vec.z

8. ~MVector ()  
This is not callable by python, it is called a deconstructor and it is what is called when an object is  
deleted from memory.

Class Operator Overloads

Anytime you see a class object that says operator this is simply C++ way of overloading operators like + – / \* []. Sometimes these are supported for the specific class so Python users can use it. Sometimes it is not. Python can overload operators as well, I have two blog posts explaining how to do this:

<http://www.rtrowbridge.com/blog/2009/02/10/python-class-operator-overloading/>  
<http://www.rtrowbridge.com/blog/2009/02/13/python-class-operator-overloading-part-2/>

Class Function Friends (Are Not Pythons Friend)

Sometimes a class has friend functions. These are functions that in C++ are shared between other classes. So they are kind of like public property, you may use it but no one owns it. Python doesnt have any way of accessing these functions as far as I know.

Class Protected Members

Any function that is in a list under the header “Protected Member Functions” is private to that function and can not be accessed outside of it. Even with C++. An example class that has this is the MFnComponent class. The className() function is protected.

## [Python API Plugin – rt4x4MatrixToTRS](http://www.rtrowbridge.com/blog/2009/02/python-api-plugin-rt4x4matrixtotrs/)

## [Maya Python API – Getting Mesh Data](http://www.rtrowbridge.com/blog/2008/12/maya-python-api-getting-mesh-data/)

Copy and paste this into a Python tab in your script editor and make sure to have a mesh selected so that it has something to work with:

# the OpenMaya module has many of the general Maya classes

# note that some Maya classes require extra modules to use which is noted in the API docs

import maya.OpenMaya as OpenMaya

# This shows how to use the MSelectionList and MGlobal class

# Get the selection and create a selection list of all the nodes meshes

selection = OpenMaya.MSelectionList()

OpenMaya.MGlobal.getActiveSelectionList( selection );

# Create an MItSelectionList class to iterate over the selection

# Use the MFn class to as a filter to filter node types

iter = OpenMaya.MItSelectionList ( selection, OpenMaya.MFn.kGeometric );

# This uses build in functions of the MItSelectionList class to loop through the list of objects

# Note this is not a basic array you must use its built in functions iterate on its objects

# Iterate through selection

while not iter.isDone():

vertexList = []

edgeList = []

polytriVertsList = []

polyList = []

conpolyList = []

# Get MDagPath from current iterated node

dagPath = OpenMaya.MDagPath()

iter.getDagPath( dagPath )

# Get the selection as an MObject

mObj = OpenMaya.MObject()

iter.getDependNode( mObj )

# This shows how to use the MItMeshPolygon class to work with meshes

# Create an iterator for the polygons of the mesh

iterPolys = OpenMaya.MItMeshPolygon( mObj )

# Iterate through polys on current mesh

while not iterPolys.isDone():

# Get current polygons index

polyList.append (iterPolys.index())

# Get current polygons vertices

verts = OpenMaya.MIntArray()

iterPolys.getVertices( verts )

# Append the current polygons vertex indices

for i in range( verts.length() ):

vertexList.append (verts[i])

# Get current polygons edges

edges = OpenMaya.MIntArray()

iterPolys.getEdges( edges )

# Append the current polygons edge indices

for i in range( edges.length() ):

edgeList.append (edges[i])

# Get current polygons connected faces

indexConFaces = OpenMaya.MIntArray()

iterPolys.getConnectedFaces( indexConFaces )

# Append the connected polygons indices

for i in range( indexConFaces.length() ):

conpolyList.append (indexConFaces[i])

# Get current polygons triangles

pntAry = OpenMaya.MPointArray()

intAry = OpenMaya.MIntArray()

space = OpenMaya.MSpace.kObject

# Get the vertices and vertex positions of all the triangles in the current face's triangulation.

iterPolys.getTriangles(pntAry, intAry, space)

# Append vertices that are part of the triangles

for i in range( intAry.length() ):

polytriVertsList.append (intAry[i])

# Move to next polygon in the mesh list

iterPolys.next()

# print data for current node being iterated on

print (dagPath.fullPathName()), '//////////////////////////////////'

print 'Vertex list: ', vertexList

print 'Edge list: ', edgeList

print 'Poly Triangle Vertices: ', polytriVertsList

print 'Polygon index list: ', polyList

print 'Connected Polygons list: ', conpolyList

# Move to the next selected node in the list

iter.next()

# MScriptUtil Class Reference

Utility class for working with pointers and references in Python.

Many of Maya's API methods require that one or more of their parameters be passed as pointers or references. Their return values, too, can be pointers or references.

In Python parameters of class types are passed by reference but parameters of simple types, like integers and floats, are passed by value, making it impossible to call those API methods from Python. The [MScriptUtil](http://download.autodesk.com/us/maya/2011help/API/class_m_script_util.html) class bridges this gap by providing methods which return pointers to values of simple types and which can extract values from such pointers. These pointers can also be used wherever an API method requires a reference to a simple type or an array of a simple type.

For example, [MTransformationMatrix::setShear()](http://download.autodesk.com/us/maya/2011help/API/class_m_transformation_matrix.html#bcedf1325f7c7dec9094e66e8bcf1888) requires as its first parameter an array of three values of simple type 'double'. To call setShear() from Python requires using [MScriptUtil](http://download.autodesk.com/us/maya/2011help/API/class_m_script_util.html) to create the array of doubles:

matrix = maya.OpenMaya.MTransformationMatrix()

util = maya.OpenMaya.MScriptUtil()

util.createFromDouble(1.0, 2.0, 3.0)

ptr = util.asDoublePtr()

m.setShear(ptr, maya.OpenMaya.MSpace.kObject)

Similar code is required to get the shear value using [MTransformationMatrix::getShear()](http://download.autodesk.com/us/maya/2011help/API/class_m_transformation_matrix.html#062d9b4d8b036f391cbd792f6ab19ed5), but in addition [MScriptUtil](http://download.autodesk.com/us/maya/2011help/API/class_m_script_util.html) methods must be used to extract the returned values from the array:

util = maya.OpenMaya.MScriptUtil()

util.createFromDouble(0.0, 0.0, 0.0)

ptr = util.asDoublePtr()

matrix.getShear(ptr, maya.OpenMaya.MSpace.kObject)

shearX = util.getDoubleArrayItem(ptr, 0)

shearY = util.getDoubleArrayItem(ptr, 1)

shearZ = util.getDoubleArrayItem(ptr, 2)

An [MScriptUtil](http://download.autodesk.com/us/maya/2011help/API/class_m_script_util.html) object keeps two sets of values internally: its initial values, and its working values. The initial values are those supplied either in the constructor or by calling one of the object's create\*() methods. The working values are those to which the as\*Ptr() methods return pointers.

Each time a create\*() method is called it completely replaces the old initial values with the new ones. The as\*() methods ([asInt()](http://download.autodesk.com/us/maya/2011help/API/class_m_script_util.html#e42e1236dd42fccf9f98516deebcbe24), [asFloat()](http://download.autodesk.com/us/maya/2011help/API/class_m_script_util.html#0ad0889a68e6f17fad74cea262a52c63), etc, but **not** the as\*Ptr() methods) all return the first internal value, converted to the appropriate type.

Each time an as\*Ptr() method is called it completely replaces the old working values with new ones of the requested type and initializes them from the initial values. If the type of the working values does not match that of the initial values then appropriate conversions are done. For example, if you have initialized an [MScriptUtil](http://download.autodesk.com/us/maya/2011help/API/class_m_script_util.html) object with four doubles via a call to [createFromDouble()](http://download.autodesk.com/us/maya/2011help/API/class_m_script_util.html#f3ab4b15d6c1f24be779389363691760), then a subsequent call to [asBoolPtr()](http://download.autodesk.com/us/maya/2011help/API/class_m_script_util.html#2723730ed3ae6718966e5b335c10b5a1) will allocate space for four boolean working values and initialize them from the initial values by converting non-zero values to true and zero values to false.

The separation of initial and working values means that changes to the working values will not affect the initial values. For example, if you pass the pointer returned by [asBoolPtr()](http://download.autodesk.com/us/maya/2011help/API/class_m_script_util.html#2723730ed3ae6718966e5b335c10b5a1) to an API method which changes the working value it points to, the object's initial value will remain unchanged and its [asInt()](http://download.autodesk.com/us/maya/2011help/API/class_m_script_util.html#e42e1236dd42fccf9f98516deebcbe24) method will continue to return the same value as it did before the API call.

The number of working values returned by the as\*Ptr() methods will match the number of initial values. Thus it is important that you provide a sufficient number of initial values before calling an as\*Ptr() method. If no initial values are provided in the [MScriptUtil](http://download.autodesk.com/us/maya/2011help/API/class_m_script_util.html) constructor and none are subsequently supplied by calls to the create\*() methods, then the pointers returned by the as\*Ptr() methods won't have room to store even a single value, which will likely lead to errors.

As already noted, each call to one of an object's as\*Ptr() methods completely replaces the old working values with new ones, thus rendering invalid any pointers previously returned by that object. This means that if you need multiple pointers for a single API call, you will need multiple [MScriptUtil](http://download.autodesk.com/us/maya/2011help/API/class_m_script_util.html) objects. [MFnMesh::getUV()](http://download.autodesk.com/us/maya/2011help/API/class_m_fn_mesh.html#cc82bf8a12a4d0897e88a2e2f694037b) provides an example of this:

meshFn = maya.OpenMaya.MFnMesh(node)

u\_util = maya.OpenMaya.MScriptUtil()

u\_util.createFromDouble(0.0)

u\_ptr = u\_util.asFloatPtr()

v\_util = maya.OpenMaya.MScriptUtil()

v\_util.createFromDouble(0.0)

v\_ptr = v\_util.asFloatPtr()

meshFn.getUV(0, u\_ptr, v\_ptr)

u = u\_util.getFloat(u\_ptr)

v = v\_util.getFloat(v\_ptr))

When an [MScriptUtil](http://download.autodesk.com/us/maya/2011help/API/class_m_script_util.html) object is destroyed any pointers to its data immediately become invalid. Thus the following code will fail because the [MScriptUtil](http://download.autodesk.com/us/maya/2011help/API/class_m_script_util.html) objects created by getUVPtrs() will go out of scope and be destroyed when the function returns, rendering the pointers it returns invalid:

# WRONG!

def getUVPtrs():

u\_util = maya.OpenMaya.[MScriptUtil](http://download.autodesk.com/us/maya/2011help/API/class_m_script_util.html)()

u\_util.[createFromDouble](http://download.autodesk.com/us/maya/2011help/API/class_m_script_util.html#f3ab4b15d6c1f24be779389363691760)(0.0)

u\_ptr = u\_util.[asFloatPtr](http://download.autodesk.com/us/maya/2011help/API/class_m_script_util.html#a4f99eef076ae7c93947fd0e16c0d33f)()

v\_util = maya.OpenMaya.[MScriptUtil](http://download.autodesk.com/us/maya/2011help/API/class_m_script_util.html)()

v\_util.[createFromDouble](http://download.autodesk.com/us/maya/2011help/API/class_m_script_util.html#f3ab4b15d6c1f24be779389363691760)(0.0)

v\_ptr = v\_util.[asFloatPtr](http://download.autodesk.com/us/maya/2011help/API/class_m_script_util.html#a4f99eef076ae7c93947fd0e16c0d33f)()

return [u\_ptr, v\_ptr]

uvPtrs = getUVPtrs()

meshFn.getUV(0, uvPtrs[0], uvPtrs[1])

To work, getUVPtrs() would have to preserve the [MScriptUtil](http://download.autodesk.com/us/maya/2011help/API/class_m_script_util.html) objects in some way, such as passing them back to the caller as well:

# RIGHT!

def getUVPtrs():

u\_util = maya.OpenMaya.[MScriptUtil](http://download.autodesk.com/us/maya/2011help/API/class_m_script_util.html)()

u\_util.[createFromDouble](http://download.autodesk.com/us/maya/2011help/API/class_m_script_util.html#f3ab4b15d6c1f24be779389363691760)(0.0)

u\_ptr = u\_util.[asFloatPtr](http://download.autodesk.com/us/maya/2011help/API/class_m_script_util.html#a4f99eef076ae7c93947fd0e16c0d33f)()

v\_util = maya.OpenMaya.[MScriptUtil](http://download.autodesk.com/us/maya/2011help/API/class_m_script_util.html)()

v\_util.[createFromDouble](http://download.autodesk.com/us/maya/2011help/API/class_m_script_util.html#f3ab4b15d6c1f24be779389363691760)(0.0)

v\_ptr = v\_util.[asFloatPtr](http://download.autodesk.com/us/maya/2011help/API/class_m_script_util.html#a4f99eef076ae7c93947fd0e16c0d33f)()

return [(u\_ptr, u\_util), (v\_ptr, v\_util)]

uvPtrs = getUVPtrs()

meshFn.getUV(0, uvPtrs[0][0], uvPtrs[1][0])

This class is admittedly cumbersome to use but it provides a way of building parameters and accessing return values for methods which would not normally be accessible from Python.

# [MScriptUtil](http://www.chadvernon.com/blog/resources/maya-api-programming/mscriptutil/)

What I will show are various code samples that demonstrate how to use MScriptUtil in various situations since at the time of this writing, the code examples for MScriptUtil are quite limited.  Luckily, I don’t need to use MScriptUtil often, but when I do encounter it, I will put a snippet on this page to build up a useful reference.

# Pass by Reference

## int

|  |
| --- |
| MStatus MItMeshPolygon::setIndex( int index, int& prevIndex ) |
| itPoly = OpenMaya.MItMeshPolygon(pathShape)  util = OpenMaya.MScriptUtil()  util.createFromInt(0)  pInt = util.asIntPtr()  itPoly.setIndex(faceId, pInt) |

|  |
| --- |
| MStatus MImage::getDepthMapSize( unsigned int &width, unsigned int &height ) const |
| utilWidth = OpenMaya.MScriptUtil()  utilWidth.createFromInt(0)  ptrWidth = utilWidth.asUintPtr()  utilHeight = OpenMaya.MScriptUtil()  utilHeight.createFromInt(0)  ptrHeight = utilHeight.asUintPtr()  mimage.getDepthMapSize(ptrWidth, ptrHeight)  width = OpenMaya.MScriptUtil.getUint(ptrWidth)  height = OpenMaya.MScriptUtil.getUint(ptrHeight) |

## float2

|  |
| --- |
| MStatus MItMeshPolygon::getUVAtPoint ( MPoint &pt, float2 &uvPoint, MSpace::Space space=MSpace::kObject, const MString \*uvSet=NULL ) |
| util = OpenMaya.MScriptUtil()  util.createFromList([0.0, 0.0], 2)  uvPoint = util.asFloat2Ptr()  itPoly.getUVAtPoint(closestPoint, uvPoint, OpenMaya.MSpace.kWorld)  u = OpenMaya.MScriptUtil.getFloat2ArrayItem(uvPoint, 0, 0)  v = OpenMaya.MScriptUtil.getFloat2ArrayItem(uvPoint, 0, 1) |

# Accessing Arrays

|  |
| --- |
| MMatrix::operator[] ( unsigned int row ) |
| #Doesn't work!  matrix[3][1] = 2.2    # Do this instead  OpenMaya.MScriptUtil.setDoubleArray(matrix[3], 1, 2.2) |

|  |
| --- |
| float \* MImage::depthMap ( MStatus \* ReturnStatus = NULL ) const |
| ptrDepthMap = mimage.depthMap()  OpenMaya.MScriptUtil.getFloatArrayItem(ptrDepthMap, index) |

# [Attribute Editor Templates](http://www.chadvernon.com/blog/resources/maya-api-programming/attribute-editor-templates/)

Often when we create a node, we want to customize the attribute editor display of the node to be more user friendly. We do this through attribute editor templates. Attribute editor templates are MEL files describing the attribute editor interface for your node. By default, Maya will automatically arrange the attributes of a node in the attribute editor. Attribute editor templates allow us to customize this display.

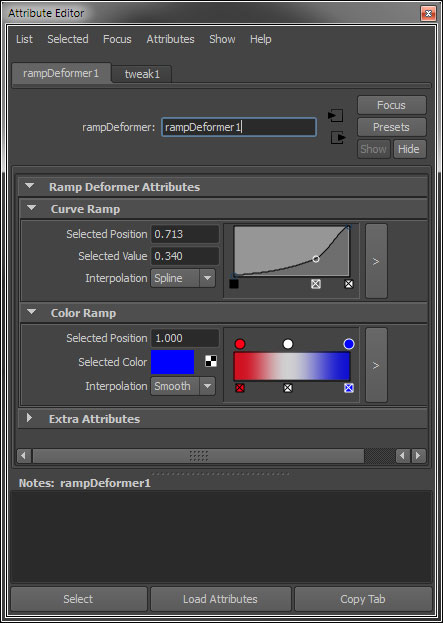
To create an attribute editor template, create a MEL file called AE{nodeName}Template.mel with an AE{nodeName}Template function inside and place the file in your MAYA\_SCRIPT\_PATH. The AE{nodeName}Template function contains editorTemplate commands that instruct the attribute editor how to alter the default layout for the attributes in the node.

Below is a sample attribute editor template for an imaginary node with 4 attributes, one of which is a ramp attribute.

|  |
| --- |
|  |
| global proc AEsampleNodeTemplate( string $nodeName )  {      editorTemplate -beginScrollLayout;      editorTemplate -beginLayout "Sample Node Attributes" -collapse 0;          editorTemplate -addControl "magnitude";          editorTemplate -addControl "offset";          editorTemplate -addControl "distance";          AEaddRampControl ($nodeName + ".rampAttribute");      editorTemplate -endLayout;        AEdependNodeTemplate $nodeName;      editorTemplate -addExtraControls;      editorTemplate -endScrollLayout;  } |

# [MRampAttribute](http://www.chadvernon.com/blog/resources/maya-api-programming/mrampattribute/)

MRampAttribute allows you to create an adjustable curve or color attribute where users can insert and adjust the interpolation of points along the ramp.



To create ramp attributes, we call the convenient classes contained in MRampAttribute:

|  |
| --- |
| Creating Ramp Attributes |
| MStatus RampAttributeDeformer::initialize()  {      // Create the curve ramp attribute      aCurveRamp = MRampAttribute::createCurveRamp( "curveRamp", "cur" );      addAttribute( aCurveRamp );      attributeAffects( aCurveRamp, outputGeom );        // Create the color ramp attribute      aColorRamp = MRampAttribute::createColorRamp( "colorRamp", "cor" );      addAttribute( aColorRamp );      attributeAffects( aColorRamp, outputGeom );        return MS::kSuccess;  } |

To access the ramp attribute values inside a node or deformer:

|  |
| --- |
| Accessing the Ramp Attributes in a Node |
| MStatus RampAttributeDeformer::deform( MDataBlock& data,                            MItGeometry& itGeo,                            const MMatrix &localToWorldMatrix,                            unsigned int geomIndex )  {      MStatus status;        // Get the ramp attributes      MObject oThis = thisMObject();      MRampAttribute curveAttribute( oThis, aCurveRamp, &status );      CHECK\_MSTATUS\_AND\_RETURN\_IT( status );      MRampAttribute colorAttribute( oThis, aColorRamp, &status );      CHECK\_MSTATUS\_AND\_RETURN\_IT( status );        float rampPosition = 0.25f, curveRampValue;      MColor color;        // Get the corresponding value on the curve ramp attribute      curveAttribute.getValueAtPosition( rampPosition, curveRampValue, &status );      CHECK\_MSTATUS\_AND\_RETURN\_IT( status );        // Get the corresponding value on the color ramp attribute      colorAttribute.getColorAtPosition( rampPosition, color, &status );      CHECK\_MSTATUS\_AND\_RETURN\_IT( status );        // Do your calculation with the values        return MS::kSuccess;  } |

You will also need to make sure the attribute is set correctly in your attribute editor template for the node:

|  |
| --- |
| AErampDeformerTemplate.mel |
| global proc AErampDeformerTemplate( string $nodeName )  {      editorTemplate -beginScrollLayout;            editorTemplate -beginLayout "Ramp Deformer Attributes" -collapse 0;              AEaddRampControl( $nodeName + ".curveRamp" );              AEaddRampControl( $nodeName + ".colorRamp" );          editorTemplate -endLayout;        editorTemplate -addExtraControls;      editorTemplate -endScrollLayout;  } |