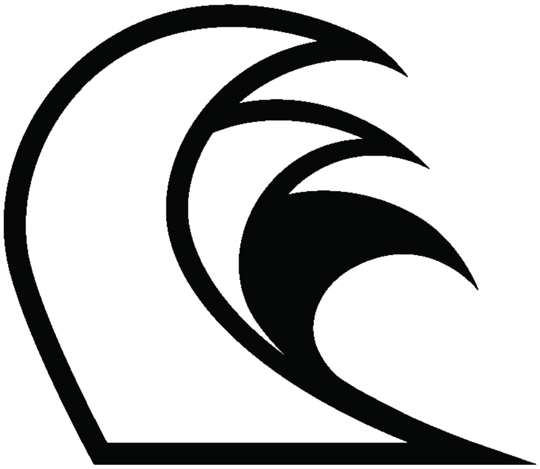
Programming in MotionBuilder || Focusing on Python
Autodesk MotionBuilder 2013

Autodesk Developer Network
May 2012

Module 4: Elements and Properties in the Scene



AUTODESK®
MOTIONBUILDER

Contents

4.0 Exploring Elements in MotionBuilder 4

The ‘FBModel’ Class 4

Mode Properties 5

4.1 Creating New Elements 6

The ‘FBModelCube’ Class 6

The ‘FBCamera’ Class 7

The ‘FBModelNull’ Class 8

The ‘FBLight’ Class 8

The ‘FBCharacter’ Class 8

Global Function: FBCreateObject() 10

Cannot Find Properties Based off the UI Names 10

Setting a Property as Animatable or to Keying the Property (if Available) 11

4.2 Working with Existing Elements 12

The ‘FBSystem’ and ‘FBScene’ Classes 12

The ‘FBComponent’ Class 14

Scene Graph 15

Namespace Management 16

Global Functions 17

4.3 Removing Elements from the Scene 18

Global Function: FBDeleteObjectsByName() 19

4.4 Standard Python Built-in Data Types used in MotionBuilder 20

4.5 MotionBuilder Built-in Data Types 20

The ‘FBVector2d’, ‘FBVector3d’ and ‘FBVector4d’ Class 21

The ‘FBSVector’ Class 21

The ‘FBColor’ Class 21

The ‘FBColorAndAlpha’ Class 22

The ‘FBMatrix’ Class 22

The ‘FBModelList’ Class 23

4.6 Creating Custom Properties on Objects 23

4.7 Deleting Custom Properties on Objects 27

4.8 The ‘FBGroup’ and the ‘FBSet’ Classes 27

Groups vs. Sets 27

Creating a Group/Set 27

Deleting a Group /Set 28
Adding objects to a Group/Set 28
Removing objects from a Group/Set 28

Programming in MotionBuilder || Focusing on Python

Autodesk Developer Network

Module 4: Elements and Properties in the Scene

Agenda

- Exploring Elements in MotionBuilder
- Creating New Elements
- Working with Existing Elements
- Removing Elements
- Standard Python Built-in Data Types used in MotionBuilder
- MotionBuilder Built-in Data Types
- Creating Custom Properties on Objects
- Deleting Custom Properties on Objects
- The ‘FBGroup’ and the ‘FBSet’ Class

4.0 Exploring Elements in MotionBuilder

The ‘FBModel’ Class

This class represents models. It is the parent of several specialized types:

- Cameras (FBCamera)
- Lights (FBLight)
- Cubes (FBModelCube)
- Markers (FBModelMarker)
- Nulls (FBModulNull)
- Optical (FBModelOptical)
- 3D Path (FBModedPath3D)
- Plane (FBModelPlane)
- Model Root (FBModelRoot)
- Skeletons (FBModelSkeleton)

It inherits from FBBox, which inherits from FBComponent, which inherits from FBPlug. FBBox is the class representing relations constraint boxes, and models can be used as boxes.

Parent and Children attribute properties let you traverse the model’s hierarchy.

Access to transformation values is provided through the Translation, Rotation and Scaling properties (local only) or with the GetVector() and SetVector() functions (local and global). You can also obtain the transformation matrix with GetMatrix() and SetMatrix().

Access to animation/keyframe data is through AnimationNode (FBAnimationNode).

Access to geometry data is through Geometry (FBGeometry or its descendant classes: FBMesh, FBNurbs).

Visibility property controls model's visibility, which can be animated.

Another property called "Show" (formerly known as "Visible" prior to 7.5) also controls visibility. Display > Show/Hide Selected/Unselected controls this. This is also used to make the model visible after creating it using code.

For example:

```
from pyfbsdk import *

lCube = FBModelCube('myCube')
lCube.Show = True
```

The attribute Name (inherited FBComponent) and LongName, give the models name as a string. The difference between the two is that LongName includes the namespace name if any, so this can be very key to differentiating two objects of the same name in different namespaces.

Mode Properties

In addition to transformation data, the FBModel class also defines a wide variety of properties, a number of which are described below. For a full list of the FBModel properties, consult the FBModel class documentation.

Property	Description
FBModel.Geometry	A geometric object to be rendered at the model's position in the scene
FBModel.Materials	A list of materials (FBMaterial) to be applied to the model's geometry

FBModel.Textures	A list of textures (FBTexture) to be applied to the model's geometry
FBModel.Shaders	A list of shaders (FBShader) to be applied to the model's geometry
FBModel.ShadingMode	An enumeration value (FBModelShadingMode) used to control the model's shading mode
FBModel.LookAt	An FBModel object to look at in the scene. Useful for pointing cameras and spotlights
FBModel.UpVector	An FBModel object indicating the up vector of the current model. Useful for orienting a camera
FBModel.Show	A boolean value (True, False) indicating whether or not the viewer should show the object according to its visibility value (FBModel.Visibility).
FBModel.Pickable	A boolean value (True, False) indicating whether or not the model can be picked in the viewer.
FBModel.RotationOrder	An enumeration value (FBModelRotationOrder) used to specify the rotation order of the model

4.1 Creating New Elements

This is fundamental lesson to MotionBuilder you need to learn how to access objects, creating and setting attributes, and working with these basic elements.

The 'FBModelCube' Class

This is a basic object, used for testing and markers.

Let's create a cube object:

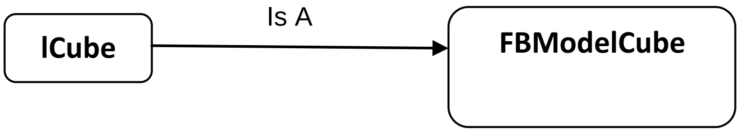
1. First import the pyfbsdk module so we can use classes and functions that are defined in it:

   ```
   from pyfbsdk import *
   ```

2. Next we will generate and instance object, which is just a namespace that gets the classes attributes for free. Here you are passing in a string of the name you want to call your cube, and this call creates the cube…

```
lCube = FBModelCube('myCube')
```

So after we create an instance of FBModelCube, we now have two objects, one instance and a class, but in other words they are two linked namespaces



3. I don't like the name myCube so let's change it, we can easily do this even though FBModelCube has no name attributes but thanks to object oriented programming this should be easy. Because FBModelCube is a child of FBModel, which is a child of FBBox, which is a child of FBComponent which stores the object name for all its children.

```
lCube.Name = "TheAmazingCube"
```

The 'FBCamera' Class

The camera object is commonly used.

Let's create a camera object:

1. Import MotionBuilder module, then create camera named 'Kristine':

```
from pyfbsdk import *
lCam = FBCamera("Kristine")
```

2. Let's set the property ViewShowTimeCode to True because in the UI we can see it is not set:

```
lCam.ViewShowTimeCode = True
```

3. If we want to switch it to whatever the opposite was we could do logic like this:

```
if lCam.ViewShowTimeCode:
   lCam.ViewShowTimeCode = False
else:
   lCam.ViewShowTimeCode = True
```

4. By default your camera is not visible or showing so we should set the if you would like to see the camera model in the Viewer:

```
lCam.Visible = True
lCam.Show = True
```

5. Now I want to set the Interest of the camera, but it says the input type is of FBModel, where do I get an object of type FBModel, let's create it…

The 'FBModelNull' Class

1. FBModelNull is a child of FBModel so it fits the bill for an Camera interest, let's create one so then we can link it to the camera

   ```
   lNull = FBModelNull("Cam Null")
   ```

2. Now will set the Camera interest to the new null:

   ```
   lCam.Interest = lNull
   ```

   Now when we move the null the camera follows.

The 'FBLight' Class

1. Let's now create a light for our camera, so here we call the class constructor and create an instance:

   ```
   from pyfbsdk import *
   lLight = FBLight("myLight")
   ```

2. Now I want to move my light, 10 to the right, so I look in FBLight, no translates there, what about the parent class FBModel, yes there is a Translate attribute let's try this.

   ```
   lLight.Translation = FBVector3d( 10, 0, 0 )
   ```

The 'FBCharacter' Class

A character is the link between a motion source (such as an Actor, a Control rig, or another character) and a character model

The first thing to point out here is this is not a child of FBModel, but rather a child of FBConstraint. We will not get into Constraints at this point, but just think of an Character as a very complex constraint, also please remember this if you are using functions from FBModel on FBCharacter, as they will not work since it is not the child of FBModel like the other classes we have discussed in this section. Characters are fundamental part of MotionBuilder, and can be more complex than we have shown here; we will dive deeper into them in a later lesson, regarding the character workflows.

1. Let's first create a character:

```
#This is the equivalent of dragging and dropping a Character from the Asset Brower >
#Templates > Characters > Character
lChar = FBCharacter("Kristine")
```

2. Now I want to set the property what if I want to only set one of the values on the 3d vector I don't want to create the who thing because I want to keep the values on x and y but wait there is no attributes for translation z only my FBLight class, let's look at the possibilities:
    a. First check all the way up the class hierarchy to see if a parent class let's you do this.

    b. Then we could check if it is undocumented by using the dir python function

    ```
    dir(lChar)
    ```

    c. Let's use the property list, this is inherited from FBComponent, the property list lets you access any of the properties that are on an element in MotionBuilder, so always look for a direct attribute in the documentation but if don't see any then you can use the PropertyList

    ```
    lPropASC = lChar.PropertyList.Find ('Action Space Compensation')
    if lPropASC:
            print "found property"
    ```

3. Now we want to set the propety but were not sure what type of proerty it is, at this point the Find function has returned a lPropASC, which is of type FBProperty, and we can see in this class there is a function GetProperty Type, let's print this out to determine what data type Action Space Compensation is:

    ```
    print lPropASC.GetPropertyType()
    ```

4. Now we know the data type is a double we can set it using the attribute Data:

    ```
    lPropASC.Data = 50
    ```

5. Now let's set the character to active as well, this takes a Boolean value (we know this because it's a check box and check boxes are true or false:

    ```
    lPropActive = lChar.PropertyList.Find("Active" )
    if lPropActive:
        lPropActive.Data = True
    ```

Another thing we could do if we wanted to collect all the animatable properties on a character and put them into a list:

```
from pyfbsdk import *
```

```
lChar = FBCharacter("Kristine")

print lChar.Name

myproplist = list()

for prop in lChar.PropertyList:
    if prop != None and prop.IsAnimatable():
            myproplist.append(prop.Name)

print myproplist
```

Global Function: FBCreateObject()

It is used to create objects from within the Asset Browser UI window. The function signature is:

FBComponent pyfbsdk.FBCreateObject (str pGroupName, str pEntryName, str pName, pData = None, int nth = 0)

- Argument 1: "Browsing/Templates/Devices" - The Asset Browser path. This path must always begin with "Browsing/".
- Argument 2: "Mouse" - The entry name. In the image below, this corresponds to the green circle.
- Argument 3: "MyMouse" - The custom name of the object

When you are a creating primitive element (in Browsing/Templates/Elements/Primitives") such as a torus or a sphere, the FBCreateObject() function requires that the third parameter be identical to the second parameter. The second parameter must also match the name of the primitive you wish to create from the asset browser (e.g. 'Torus').

```
torus = FBCreateObject( 'Browsing/Templates/Elements/Primitives', 'Torus', 'Torus' )
torus.Show = True
```

Cannot Find Properties Based off the UI Names

- One trick that is very helpful is when you can’t find the property based on the UI name is to set the property in the UI to something distinct.
- Then save the scene as ASCII, using a text editor search the ASCII FBX file for the value or name you think it might be.

For example:
For int or float properties, you can set the attribute value to something crazy like 33 and then in the ASCII file just search for 33, then you have your name.

However in the case where it is a Boolean, it can be a bit harder, so if I wanted to set the property 'Finger Solving', I wasn't able to find the property using the UI name, I searched for finger in the ASCII FBX file, and found FingerSolving, I tested it and it works.

- One more rule of thumb is if you can't find the property by using the UI name, generally it won't have any spaces in the property name so that will help when searching the ASCII FBX file or giving it a try before you search the ASCII file

Setting a Property as Animatable or to Keying the Property (if Available)

Let's say you have the property 'Action Space Compensation' on a character, this property has three sub properties, keying, value(double) and animatable….how do you access these, above we already covered the value part, so let's talk about key and animatable properties.

Animatable properties are properties which are represented in numeric values and can be recorded by a curve representing different values at different time. If you see in the UI, there is this "A" button beside the valule of the property, then this property is animatable. You can use FBProperty::IsAnimatable() to check if a property is an animatable property or not.

FBPropertyAnimatable is the base class of all animatable properties. You can call functions of this base class to work with animatable properties. For example, before you set keys on a property, you need to use FBPropertyAnimatable::SetAnimated() to create a FCurve on this property, then you can call FBPropertyAnimatable::Key() to set keys on this property.

```
from pyfbsdk import *
lChar = FBCharacter("TestingCharacter")

lPropShoulder = lChar.PropertyList.Find("Realistic Shoulder Solving")
if lPropShoulder.IsAnimatable():
	if not lPropShoulder.IsAnimated():
		lPropShoulder.SetAnimated(True)
		lPropShoulder.Key()
```

Starting MB2012, properties Translation, Rotation and Scale of objects are not animated by default, you have to set it manually. In previous version you don't need to call SetAnimated explicitly to set them to be animated.

4.2 Working with Existing Elements

You first have to get a handle on the object; there are multiple ways to do this, this list goes from knowing exactly what you want to just searching for everything. In this sample we are going to work with a Camera in the scene, but this applies to any model.

Scenarios where you would use this are if you don't create the objects in the same script that you are looking for them; most of the time you get file that already have the models in them that were created by an end user. This is how you work with them, so don't fear if you did not create the object in your script, you can still access them in many different ways:

1. FBSystem and FBScene
2. FBComponent
3. Scene Graph
4. Namespace Management
5. Global Functions

The 'FBSystem' and 'FBScene' Classes

Here we are introducing two new classes:

1. FBSystem
2. FBScene.

The difference between the two classes is one gets system information and the other get's scene information.

These are very common classes that you will use a lot, as a scripter, so be familiar with them.

What is System Information (FBSystem)?

The FBSystem class exposes many system properties, including the computer name the system time, the process memory, the MotionBuilder installation directoy, and the MotionBuilder version information to name but a few.

It is implemented as a singleton, and also contains references to important objects such as the MotionBuilder scene (FBScene), the current take (FBTake), and the viewport (FBRenderer).

You can think of MotionBuilder application as a system including the underlying operating system. So in other words, it will not get you any information from a scene file (FBX File), if you could open MotionBuilder without a new file, this would be the only information it can access.

You can access things like the Asset Manager, LocalTime, RootModel or SceneRootModel provides the root of the model hierarchy, etc. CurrentTake represents the active take.

You can also obtain system-related info: ComputerName, Version, SystemTime, etc.

```
lSystem = FBSystem()
print 'MotionBuilder version: %f' % lSystem.Version

lCurrentTime = lSystem.LocalTime
print 'The current frame is: ' + lCurrentTime.GetTimeString()
```

What is Scene Information (FBScene)?

FBScene is FBX file specific content, so a good way to think of this is that it is everything that is located in the Navigator. This class allows the user to interact with the underlying scene, as well as a few operating system calls.

The FBScene object is obtained from FBSystem's Scene attribute.

Here you can get information such as Cameras, Materials, Takes Actors, Characters, Groups, Sets, etc, almost everything you see in the Navigator window.

The function Evaluate() forces MotionBuilder to give you up-to-date values when obtaining animated property values at different times. Call this function only when needed.

Locating and working with Cameras:

```
from pyfbsdk import *

lScene = FBSystem().Scene

lSceneCameras = lScene.Cameras
for lSceneCamera in lSceneCameras:
    print lSceneCamera.Name
    if lSceneCamera.Name == "Producer Front":
        lColor = FBColor (0, 1, 0, 0)
    elif lSceneCamera.Name == "Producer Back":
        lColor = FBColor (1, 0, 0, 0)
    else:
```

```
    lColor = FBColor (0, 1, 1, 0)
lSceneCamera.BackGroundColor = lColor
```

Working with Characters:

```
from pyfbsdk import *

lScene = FBSystem().Scene

lNumCharacters = len(lScene.Characters)
if lNumCharacters > 0:
   lChar = lScene.Characters[0]
   print lChar.Name
```

Note: There is only one instance of FBScene available for manipulation at any given time, accessible via FBSystem.Scene. If you try to create an instance of a scene you will get an error.

The ‘FBComponent’ Class

At the root of most classes dealing directly with the MotionBuilder architecture, there is the FBComponent class, which contains all of the creation and management functions for objects integrated into MotionBuilder.

Most Python SDK classes derive from this class. FBComponent classes define common object characteristics, including creation and destruction methods. It also provides a scheme for property management.

This is the base class for most object types and should really be called FBObject, if you think of it like that it may it less confusing.

To access the components of a scene you have to access it through ‘Scene’ through FBSystem, and then access ‘Components’ through FBScene.

What I have explained in this little section simplifies FBComponent as it is much more complex but we will explore this more over the coming week.

You can get all FBComponent objects through the components list and when I mean all the objects it traverses the whole entire scene. This should only be done as a last resort, because as you can imagine, iterating through all the components will be very slow. Also, for some reason, the list gives you components that are NULL/None, which can crash MB if accessed improperly. This is why in samples you see this, to avoid any crashes:

```
if lComp != None and …
```

FBComponent-derived objects contain FBProperty objects as properties/attributes. Certain properties are exposed as public attributes for a class, while others are not.

You can basically search based on any properties you would like type, name, attribute, etc.

Is() lets you identify the precise type. For example, it is used if a given object is an FBCharacter, or an FBModel, etc.

In Python, Is() uses one of several functions of the form <class>_TypeInfo(). For example:

```
lComp = FBSystem().Scene.Components
for lCom in lComp:
   if lComp.Is(FBCharacter_TypeInfo()):
        # etc . . .
```

The Selected property indicates if the object is selected in the Navigator (or scene viewer if applicable). This can also be used to select the object within the code.

As discussed previously PropertyList (FBPropertyManager) contains the object's properties. It can be used to find unexposed properties. As well Find() lets you search a property by name.

In Python, the built-in len() function can be used to find the size of the PropertyList. However, the PropertyList itself is an iterable sequence that can be used in a for-loop.

```
from pyfbsdk import *

for lComp in FBSystem().Scene.Components:
        if lComp != None and lComp.Is(FBModelSkeleton_TypeInfo()):
                lComp.Selected = True
```

or

```
from pyfbsdk import *

for lComp in FBSystem().Scene.Components:
   if lComp != None and lComp.Name.startswith('Control'):
      print lComp.Name
```

Scene Graph

In Motionbuilder, all the scene elements are arranged in a parent/child hierarchy, which is called the Scene Graph. The parent/child relationship is set up by the properties

FBModel.Parent or FBModel.Children. The root model of the scene (FBScene.RootModel) is the entry point into the scene graph. It is analogous to the scene's origin (0, 0, 0). When an instance of FBModel is created, it is automatically added as a child to the root model. The models you create can be assigned as children or parents to other models. This parent/child hierarchy is referred to as the scene graph. An FBModel can only have one parent, but can have multiple children. We can use the following example to traverse all the models under root model and print out their names:

```
from pyfbsdk import *

def printSceneGraph(pModel, pLevel):
    tabs = ''
    for i in range(0, pLevel):
        tabs += '\t'
    print tabs + pModel.Name + ' - ' + pModel.ClassName()

    for child in pModel.Children:
        printSceneGraph(child, pLevel + 1)

scene = FBSystem().Scene
printSceneGraph(scene.RootModel, 0)
```

Namespace Management

With MotionBuilder 2013, we introduced a brand new class FBNamespace to facilitate the use of namespaces among scene elements. From this class, you will be able to get access to what contents are available under this namespace.
Also there are quite a few functions have been added into FBScene and FBFbxOptions to facility working with objects with namespace. For example, you can directly access a list of available namespaces of current scene by going through property FBScene.Namespace, also the following functions give access to the objects under same namespace:
NamespaceGetContentList(): get list of contents under a particular namespace
NamespaceSelectContent(): select all the objects under a particular namespace
NamespaceDeleteContent(): delete all the objects under a particular namespace

Here is an example how you would use FBScene::NamespaceGetContentList():

```
from pyfbsdk import *

#Find all the namespace names in current scene
lScene = FBSystem().Scene
for lNamespace in lScene.Namespaces:
    print lNamespace.LongName

#Get the contents under namespace "NS1"
lcontentList = FBComponentList()
```

```
lScene.NamespaceGetContentList(lcontentList,
"NS2",FBPlugModificationFlag.kFBPlugAllContent)
for lcontent in lcontentList:
   print lcontent.LongName
```

Global Functions

All the following global functions are defined in model pyfbsdk as independent standalone functions independent of any classes.

FBFindObjectsByName(str pNamePattern, tuple pList, bool pIncludeNamespace = True, bool pModelsOnly = False):

This is a standalone function, which returns a list of objects with a particular name pattern. The first parameter indicates the name pattern to search, the second parameter is the returned list, the third parameter indicates whether the search use the complete name with namespace, the last parameter sets if the search is on models or any types of objects. We would use it like this:

```
from pyfbsdk import *
cl = FBComponentList()
Pattern = “*Cube*”
# Find all objects whose name contains "Cube"
FBFindObjectsByName( pattern, cl, True, False )
```

FBFindObjectByFullName(str pObjectFullName):

This standalone function will query the system for an object with its specific full name is “GroupName::NamespaceName:ObjectName” and return the handle onto the object it finds. This function is more straight-forward and limited compared to the previous function.

FBFindModelByLabelName(str pModelLabelName):

This global function replaces the deprecated FBFindModelByName() function, it finds a model in a scene by its label name. Label name is defined as “NamespaceName:ObjectName”, also known as “PrefixName::ObjectName”.

FBFindModelsOfType(tuple pList, int pTypeInfo, FBModel pParent = None)

This standalone function find all models of a certain type in the scene. The first parameter is the returned list that includes all the models which matches the criteria, the second parameter is the model type to look for, the third parameter is the root model to look from. Currently there is a bug with this function in python with MotionBuilder 2013, it does not work properly.

FBGetSelectedModels(tuple pList, FBModel pParent = None, bool pSelected = True, bool pSortBySelectOrder = False)

This global function finds all modes that are selected and adds them to a list of models.

This is used only for selected models, so an object needs to be derived from FBModel to be able to be selected here.

This function is set up a little bit differently than the above function, returns a tuple of all the selected things in the scene. So first we have to create a tuple that can be filled, we cannot use the Python Standard list of lTuple = [], we must use MoBu's built in tuple class:

```
lModelList = FBModelList()
```

Then we can call the FBGetSelectedModels function:

```
FBGetSelectedModels( lModelList,None, True, False )
```

In a real life scenario we could do something like this:

```
from pyfbsdk import *

lLight1 = FBLight("Top Light")
lLight2 = FBLight("Side Light")

#From the class FBModel, we set the Selected attribute to True
lLight1.Selected = True
lLight2.Selected = True

lModelList = FBModelList()

FBGetSelectedModels( lModelList, None, True, False )

#look at each item in the list
for lModel in lModelList:
    print lModel.Name
```

In this example we have touched on some gotchas in the documentation, the FBGetSelectedModels, asks for a parameter of type list, which could lead a programmer to think they can use the Python list data type. In a next lesson we will discuss MoBu built-in data types.

4.3 Removing Elements from the Scene

Global Function: FBDeleteObjectsByName()

FBDeleteObjectsByName (str pNamePattern, str pNameSpace = None, str pGroupName = None)
This function will query the system for objects fulfilling a particular name pattern and delete them. Note that if you don't specify a namespace, deleting all objects with the group name may lead to inconsistent in scene.You need to be careful with using this function in this manner.

Two Options for deleting items

1. FBDelete:
   - This deletes the wrapper (instance) and the object.

2. FBDestroy:
   - This only deletes the wrapper (instance).

Deleting is one of the gotchas that gets most people because you would think you could do this: but you are sporadically missing items

```
from pyfbsdk import *

# Accessing the scene components to view all the characters
lScene = FBSystem().Scene
lChars= lScene.Characters

# Append all the characters in the scene in a list
for lChar in lChars:
   lChar.FBDelete()
```

In this case, it sporadically misses materials:

```
from pyfbsdk import *

for lComp in FBSystem().Scene.Components:

   #Assuming that my naming convention is naming control rigs with the word Material
   if lComp != None and lComp.Name.startswith('Material'):
      #print lComp.Name
      lComp.FBDelete()
```

This is because you are not deleting the last one first, so the internal counting gets messed up when you delete this way.

You need to do something like this, there are multiple variations on this, but the key is ensuring you are not relying on a counter which you are deleting in the above sample or deleting from the bottom up.

```
from pyfbsdk import *

# Create the list to store the objects in
lList = []

# Accessing the scene components to view all the characters
lScene = FBSystem().Scene
lChars= lScene.Characters

# Append all the characters in the scene in a list
for lChar in lChars:
	lList.append(lChar)

# map the the FBDelete class to each item in the list to delete
map( FBComponent.FBDelete, lList )
```

4.4 Standard Python Built-in Data Types used in MotionBuilder

These are the data Types, in which MotionBuilder uses directly from the Python Standard library, they are used just like you would use them in pure Python, so if you need more information on how to create and manipulate them, please see the Python resources listed in the first lesson so that you can learn from their resources.

- object
- bool
- float
- int
- list
- long
- str
- tuple
- Enumeration

4.5 MotionBuilder Built-in Data Types

For functions and attributes that require these values as input or output, you need to create instance of these classes, if your are reading from them you need to just set up an empty constructor like this value = MBType(), however if you are writing to them you need to specify the information, such as value = MBType(0.0, 0.0,0.0). The one exception of writing is FBModelList as you can only read from it.

The ‘FBVector2d’, ‘FBVector3d’ and ‘FBVector4d’ Class

This class creates a list like object, which can be modified using the list protocol method. But unlike lists, its length is fixed: it always contain 2. 3 or 4 floating point values depending on which type you are using. Thus it does not support any list methods that would affect its length. The values within can be changed, usually via the bracket operator.

```
lVector2d = FBVector2d()
```

or

```
lVector3d = FBVector3d()
```

or

```
lVector4d = FBVector4d()
```

or

```
lVector2d = FBVector2d( -25, 25 )
```

or

```
lVector3d = FBVector3d( 50, 5, 50 )
```

or

```
lVector4d = FBVector4d( 50, 5, 50, -30 )
```

or

```
LookAt = FBVector3d(lCamera.Interest.Translation)
```

The ‘FBSVector’ Class

This class represents a three-dimensional scaling vector.

```
lSVector = FBSVector(1.0, 2.0, 3.0)
```

The ‘FBColor’ Class

This class creates a color vector, which is a list like object, which can be modified using the list protocol method. But unlike lists, its length is fixed: it always contain

3floating point values. Thus it does not support any list methods that would affect its length. The values within can be changed, usually via the bracket operator.

```
lColor = FBColor(0, 1, 0)

or

lColor = FBColor(0.0,0.0,0.0)

or

lColor = FBColor(lSceneCamera.BackGroundColor)
```

The 'FBColorAndAlpha' Class

This class creates a color vector with alpha channel. It is a list like object, which can be modified using the list protocol method. But unlike lists, its length is fixed: it always contain 4 floating point values. Thus it does not support the any list methods that would affect its length. The values within can be changed, usually via the bracket operator.

```
lColorAlpha = FBColorAndAlpha(0.0, 0.0, 1.0, 0.5)
```

The 'FBMatrix' Class

This creates a 4 x 4 (double) Matrix. This class creates a list like object, which can be modified using the list protocol method. But unlike lists, its length is fixed: it always contain 16 floating point values. Thus it does not support the any list methods that would affect its length. The values within can be changed, usually via the bracket operator.

```
lMatrix = FBMatrix()

lMatrix [0] = 1
lMatrix [1] = 0
lMatrix[2] = 0
lMatrix[3] = 0
lMatrix[4] = 0
lMatrix[5] = 1
lMatrix[6] = 0
lMatrix[7] = 0
lMatrix[8] = 0
lMatrix[9] = 0
lMatrix[10] = 1
```

```
lMatrix[11] = 0
lMatrix[12] = 50
lMatrix[13] = 50
lMatrix[14] = 50
lMatrix[15] = 1
```

or

```
lMatrix = FBMatrix(FBPose("Mia:Mia_body").GetNodeMatrix(0))
```

or

```
lMatrix = FBMatrix([1, 2.0, 3.3, 4.0, 5.6, 6.4, 7, 8, 9, 10.0, 11, 12., 13.0, 14.1,
15.8, 16])
```

or

```
lMatrix = FBMatrix([ [1,2,3,4], [5.0, 6.2, 7.0, 8], [9, 10, 11, 12], [13, 14, 15, 16] ])
```

The ‘FBModelList’ Class

This class implements a special sort of list that can only contain instances of FBModel objects. To users it behaves as a tuple, since it is not possible to add new objects in the list. Only methods or function that uses FBModelList as argument can insert new objects. Users can query the content of the list with the bracket operator.

```
lList = FBModelList()
```

**The ‘FBTime’ Class is also a built in data class but we will cover this later as it is much more complex to work with.

4.6 Creating Custom Properties on Objects

PropertyCreate() is used for creating and adding custom properties to objects. This function is in the class FBComponent. Anything that you can add properties to in the UI, you can add in Python.

In the class FBPropertyType, we can see the types of custom properties you can create are:

- kFBPT_unknown
- kFBPT_int
- KFBPT_int64
- KFBPT_unit64
- kFBPT_bool

- kFBPT_float
- kFBPT_double
- kFBPT_charptr
- kFBPT_enum
- kFBPT_Time
- KFBPT_TimeCode
- kFBPT_object
- kFBPT_event
- kFBPT_stringlist
- kFBPT_Vector4D
- kFBPT_Vector3D
- kFBPT_ColorRGB
- kFBPT_ColorRGBA
- kFBPT_Action
- kFBPT_Reference
- kFBPT_TimeSpan
- kFBPT_kReference
- kFBPT_Vector2D

Let’s create a cube in the scene, and then create all these custom property types on it:

```
from pyfbsdk import *

lCube = FBFindModelByLabelName( "Cube" )

lPropInt = lCube.PropertyCreate('My Int', FBPropertyType.kFBPT_int, 'Integer', False,
True, None)

lPropFloat = lCube.PropertyCreate('My Float', FBPropertyType.kFBPT_float, 'Number',
False, True, None)

lPropDouble = lCube.PropertyCreate('My Double', FBPropertyType.kFBPT_double,
'Number', False, True, None)

lPropCharptr = lCube.PropertyCreate('My Charptr', FBPropertyType.kFBPT_charptr, "",
False, True, None)

lPropVector4 = lCube.PropertyCreate('My Vector4D', FBPropertyType.kFBPT_Vector4D,
"Vector4", False, True, None)

lPropVector2 = lCube.PropertyCreate('My Vector2D', FBPropertyType.kFBPT_Vector2D,
"Vector", False, True, None)

lPropColour = lCube.PropertyCreate('My Colour', FBPropertyType.kFBPT_ColorRGB,
"Color", True, True, None)
```

```
lPropColourAplpa = lCube.PropertyCreate('My Colour with Alpha', FBPropertyType.kFBPT_ColorRGBA, "ColorAndAlpha", True, True, None)

lPropAction = lCube.PropertyCreate('My Action', FBPropertyType.kFBPT_Action, "Action", True, True, None)

lPropBool = lCube.PropertyCreate('My Bool', FBPropertyType.kFBPT_bool, "Bool", True, True, None)

lPropEnum = lCube.PropertyCreate('My Enum', FBPropertyType.kFBPT_enum, "Enum", True, True, None)
```

How do I know what values go in the third parameter for my PropertyCreate, this is from a list of pre-defined values, from this header file:

fbdata.h defines

```
#define  ANIMATIONNODE_TYPE_NUMBER   "Number"
  Animation node data types.

#define  ANIMATIONNODE_TYPE_TIME   "Time"
  Animation node data types.

#define  ANIMATIONNODE_TYPE_VECTOR   "Vector"
  Animation node data types.

#define  ANIMATIONNODE_TYPE_VECTOR_4   "Vector4"
  Animation node data types.

#define  ANIMATIONNODE_TYPE_ACTION   "Action"
  Animation node data types.

#define  ANIMATIONNODE_TYPE_BOOL   "Bool"
  Animation node data types.

#define  ANIMATIONNODE_TYPE_ENUM   "Enum"
  Animation node data types.

#define  ANIMATIONNODE_TYPE_INTEGER   "Integer"
  Animation node data types.

#define  ANIMATIONNODE_TYPE_OCCLUSION   "Occlusion"
  Animation node data types.

#define  ANIMATIONNODE_TYPE_FIELDOFVIEWX   "FieldOfViewX"
  Animation node data types.

#define  ANIMATIONNODE_TYPE_FIELDOFVIEWY   "FieldOfViewY"
  Animation node data types.

#define  ANIMATIONNODE_TYPE_OPTICALCENTERX   "OpticalCenterX"
```

```
  Animation node data types.

#define  ANIMATIONNODE_TYPE_OPTICALCENTERY   "OpticalCenterY"
  Animation node data types.

#define  ANIMATIONNODE_TYPE_IKREACHTRANSLATION   "IKReachTranslation"
  Animation node data types.

#define  ANIMATIONNODE_TYPE_IKREACHROTATION   "IKReachRotation"
  Animation node data types.

#define  ANIMATIONNODE_TYPE_COLOR   "Color"
  Animation node data types.

#define  ANIMATIONNODE_TYPE_LOCAL_TRANSLATION   "Lcl Translation"
  Animation node data types.

#define  ANIMATIONNODE_TYPE_LOCAL_ROTATION   "Lcl Rotation"
  Animation node data types.

#define  ANIMATIONNODE_TYPE_LOCAL_SCALING   "Lcl Scaling"
  Animation node data types.

#define  ANIMATIONNODE_TYPE_TRANSLATION   "Translation"
  Animation node data types.

#define  ANIMATIONNODE_TYPE_ROTATION   "Rotation"
  Animation node data types.

#define  ANIMATIONNODE_TYPE_SCALING   "Scaling"
  Animation node data types.

#define  ANIMATIONNODE_TYPE_COLOR_RGBA   "ColorAndAlpha"
  Animation node data types.
```

Setting my custom properties:

```python
from pyfbsdk import *

lCube = FBFindModelByLabelName( "Cube" )

lPropInt = lCube.PropertyList.Find('My Int')
lPropInt.SetMin(-100)
lPropInt.SetMax(100)
lPropInt.Data = 45

lPropColour = lCube.PropertyList.Find('My Colour')
lPropColour.Data = FBColor(1,0,0)

if lPropColour.IsAnimatable():
    lPropColourAlpha.SetAnimated(True)
    lPropColourAlpha.Key()

lPropChar = lCube.PropertyList.Find('My Charptr')
lPropChar.Data = " This property is a pointer to a string!”
```

4.7 Deleting Custom Properties on Objects

The easiest thing about custom properties if removing them, all you need to do is call the PropertyRemove function and pass in the property you would like to remove as the parameter, you don’t need to do something different depending on the type, you remove the properties the same on every data type.

```
    lCube.PropertyRemove(lProp)
```

4.8 The ‘FBGroup’ and the ‘FBSet’ Classes

The Groups Class, is called FBGroup and the Sets Class is called FBSet, they let you take any kind of asset or object and associate them into a group or set. You can create groups and sets including models, elements, lights, materials, shaders, markers, nodes, textures, constraints and even other groups and sets.

Groups and sets let you create custom associations between objects or assets. This way, you can select a pre-defined collection of several objects every time you want to t work with them. Once stored in a group or a set, you can manipulate the group or set instead of each individual object.

Groups vs. Sets

Group not exclusive, meaning the same object can appear in many groups. Generally groups are used for the selection and organization of items in the scene. The visibility of groups cannot be animated.

Sets are exclusive, meaning an object can only appear in one set, so you cannot copy an object from one set to another, you need to move it from one set to another. The visibility of sets can be animated, and the animation stores in sets can be cached.

Creating a Group/Set

Let’s create a set in MotionBuilder:

```
from pyfbsdk import *

lSet = FBSet("MySet")
```

Let’s create a group in MotionBuilder:

```
from pyfbsdk import *

lGroup = FBGroup("MyGroup")
```

Deleting a Group /Set

Group

```
lGroup.FBDelete()
```

Set

```
lSet.FBDelete()
```

Adding objects to a Group/Set

```
lCube = FBFindModelByLabelName("Cube")
```

Group

```
lGroup.Items.append(lCube)
```

Set

```
lSet.Items.append(lCube)
```

Removing objects from a Group/Set

Group

```
lGroup.Items.remove(lCube)
```

Set

```
lSet.Items.remove(lCube)
```

For example:

```
from pyfbsdk import *

gApplication = FBApplication()
gSystem = FBSystem()
lScene = FBSystem().Scene

lSphere = FBFindModelByLabelName ( 'Sphere' )
lSphere1 = FBFindModelByLabelName ( 'Sphere 1' )
lSphere2 = FBFindModelByLabelName ( 'Sphere 2' )
lSphere3 = FBFindModelByLabelName ( 'Sphere 3' )
lSphere4 = FBFindModelByLabelName ( 'Sphere 4' )
lSphere5 = FBFindModelByLabelName ( 'Sphere 5' )
lSphere6 = FBFindModelByLabelName ( 'Sphere 6' )

lGroup = FBGroup("3Spheres")
lScene.Groups.append(lGroup)
lGroup.Items.append(lSphere)
lGroup.Items.append(lSphere1)
lGroup.Items.append(lSphere2)
lGroup.Pickable = False

lGroup2 = FBGroup("2Spheres")
lScene.Groups.append(lGroup2)
lGroup2.Items.append(lSphere3)
lGroup2.Items.append(lSphere4)
lGroup2.Show = False

lGroup3 = FBGroup("1Sphere")
lScene.Groups.append(lGroup3)
lGroup3.Items.append(lSphere5)
lGroup3.Transformable = False


for ltest in lScene.Groups:
        if ltest.Name == "3Spheres":
                ltest.Items.remove(lSphere)
        if ltest.Name == "2Spheres":
                ltest.Items.remove(lSphere3)

for ltest2 in lScene.Groups:
        if ltest2.Name == "3Spheres":
                lGroup3Cones = ltest2
        if ltest2.Name == "2Spheres":
                lGroup4Cones = ltest2
        if ltest2.Name == "1Sphere":
                lGroup1Cube = ltest2

lGroup.Pickable = False
lGroup.Show = False
lGroup.Transformable = False


#lGroup.FBDelete()
```