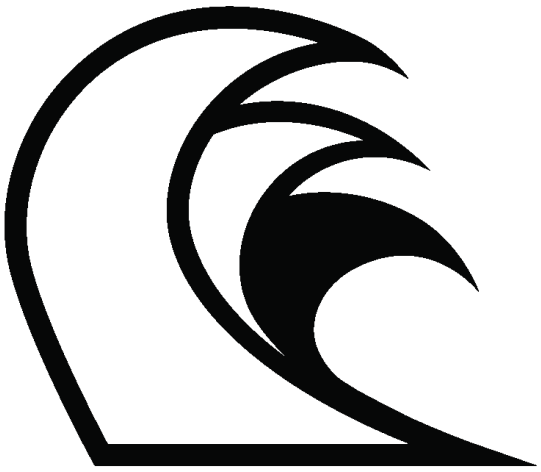
Programming in MotionBuilder || Focusing on Python
Autodesk MotionBuilder 2013

Autodesk Developer Network
May 2012
Module 8: Constraints



AUTODESK®
MOTIONBUILDER

Contents

Programming in MotionBuilder || Focusing on Python

Autodesk Developer Network
Module 8: Constraints

Agenda

- The ‘What’ and ‘Why’ of Constraints?
- Working with Simple Constraints
- Working with Complex Constraints
- Triggering Scripts in Constraints

8.0 The ‘What’ and ‘Why’ of Constraints?

In the real world, we are surrounded by constraints; for example gravity constrains us to the ground, or a dog is constrained by the length of his leash.

To simulate these limits in the 3D animation world, these relationships between objects must be established by the animator.

Constraints are tools used to create relationships between objects. You can use constraints to make a character pick up an object, have a camera follow a character, or use the keyboard or mouse to trigger events.

Another way to think of constraints is that they are a restriction of the translation, rotation, and other data of an object based on the position, translation, rotation, and other data of another object.

The equation x<3 is a mathematical constraint of the variable x, limiting it to any value smaller than 3.

You can use these limits to simulate real-world relationships between objects. For example, the constraint on a dog attached to a five-foot leash is X<5, meaning that x, representing the dog’s area of motion, must be less than 5, which is the full length of the leash. This simple constraint ensures that your dog behaves like it should on a leash.

Similar constraints can be used to restrain a model’s arm motion to an area defined by the shoulder joint, and so on. In fact, the whole Control rig is actually a series of many constraints. The rig’s hand is constrained by the motion of the arm, which is constrained by the elbow joint, which in turn is constrained by the upper arm, and so on.

Each of these connections form complex relationships that work to create a recognizable simulation of the human body in motion.

In the Asset browser, the Constraints folder lets you view and access constraint assets you can add to your scene.

8.1 Working with Simple Constraints

In MotionBuilder here is the list of simple constraints:

- 3 Points
- Aim
- Chain IK
- Mapping
- Multi-Referential
- Parent/Child
- Path
- Position
- Range
- Rigid Body
- Rotation
- Scale

You can find these all in the UI in the Asset Browser here:

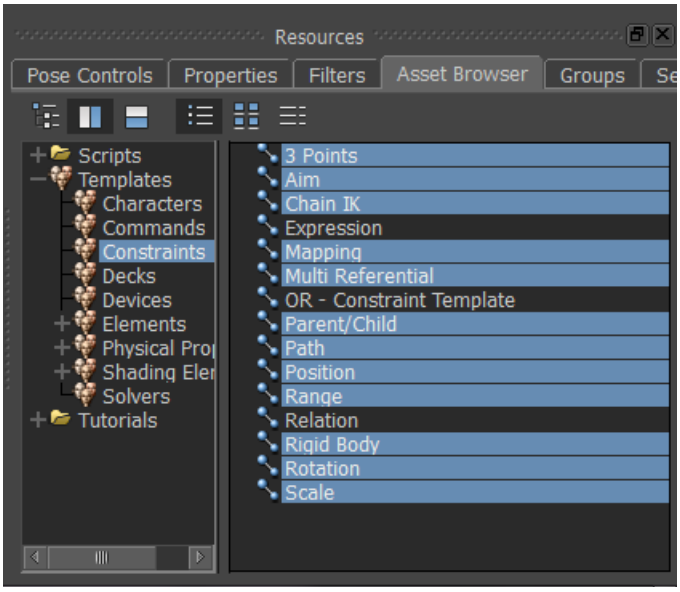


Figure 1: Simple Constraints

Creating your own simple 'Position' constraint in Python

When working with existing constraints we have a utility class called FBConstraintMangaer, this class lets you manage constraints, including relationships between constraints

TypeCreateConstraint() is used to create constraints. Since it expects an int, we need to resolve a constraint type name to its registered index. However, the indices may change due to plug-in constraints, so we need to loop through every index from 0 to *TypeGetCount()* - 1 to see if TypeGetName() returns a match.

Also, *TypeCreateConstraint()* does not add the newly-created constraint to the manager, so you must add it using FBSystem().Scene.Constraints.append.

This script adds a Position Constraint to the scene; we first need to create an FBConstraintManager object, which allows us to access all the available constraints in MotionBuilder, this includes simple, complex and custom constraints, in other words all of these FBConstraintManager exposes all these constraints for creation via Python.

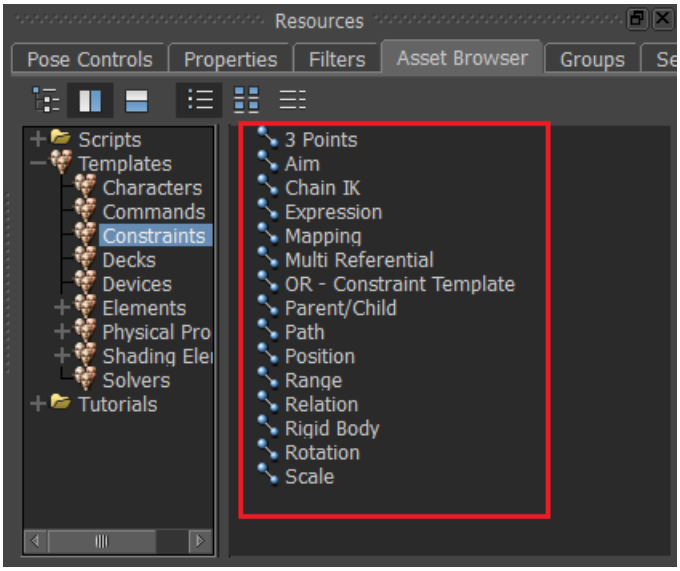


Figure 2: Available Constraints

Let's look at creating them in Python.

For example:

```
from pyfbsdk import *

lMgr = FBConstraintManager()
```

We need to go through all the available constraints and find the one we want to create, so in this case we will go through 18 constraints looking for the one named ‘Position’

```
lIndex = None
for i in range(0, lMgr.TypeGetCount()):
    if lMgr.TypeGetName(i) == 'Position':
        lIndex = i
        break
```

This is the function (TypeCreateConstraint), that creates the constraint and add it into the current scene.

```
lConst = lMgr.TypeCreateConstraint(lIndex)
```

Creating all constraints in Python

```
from pyfbsdk import *

# We create new constraints with thr constaint manager.
lMgr = FBConstraintManager()

# We want to create one constraint of each types that exists.
for lIdx in range( lMgr.TypeGetCount() ):

    # We create the constraint.
    lCnst = lMgr.TypeCreateConstraint( lIdx )

    lCnst.Name = "%s Created by script" % lMgr.TypeGetName( lIdx )

    print "Adding new Constraint Number - %d: '%s'" % (lIdx, lCnst.Name)
```

Finding and Deleting Existing Constraints in the Scene

Currently in our sample scene we have two simple constraints; 3 Points and Aim:

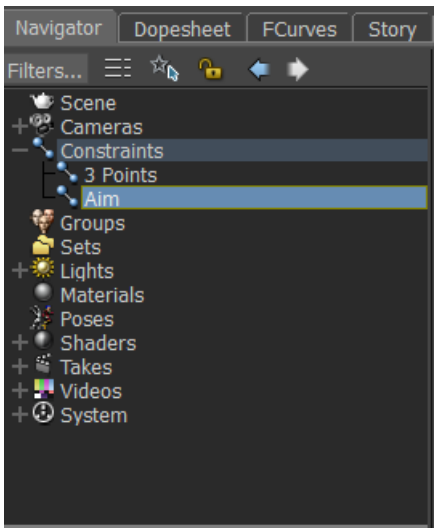


Figure 3: Two Simple Constraints in Scene

To retrieve these existing constraints, we use FBScene from FBSystem; this returns a list of FBConstraint that we then go through them in a for loop.

You need to use FBScene to get the list of constraints in the scene, to append or remove a constraint from the scene, get the count and get a handle to the constraint as this attribute returns a list of FBConstraint. As a reminder you can use standard list functions to work with the list.

Here is how to find the constraints in the scene:

For example:

```
from pyfbsdk import*

lConstraints = FBSystem().Scene.Constraints

for lCon in lConstraints:
    print lCon.Name
```

Here is how to delete the constraint based on name, there are numerous conditions you could delete a constraint based on, not just by name, you could delete it based on type, or property value, and it's really up to your scene requirement to choose what logic you will delete a constraint based on.

For example:

```
from pyfbsdk import*

lConstraints = FBSystem().Scene.Constraints

for lCon in lConstraints:
    if lCon.Name == 'Aim':
```

```
lCon.FBDelete()
```

NOTE: This finding and deleting applies to both simple and complex constraints in Python.

Working with the Constraint Properties

Now that we have accessed the constraint we now want to work with it settings and properties in Python, in some scenarios the properties and settings have the same content but in some places it does not, so don't be confused by the two different views.

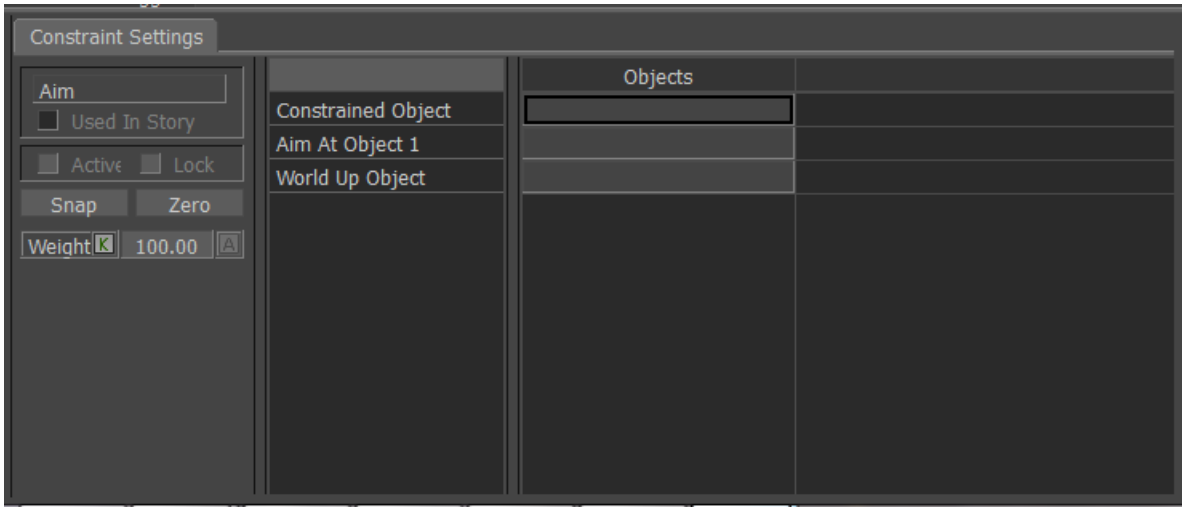


Figure 4: Aim Constraint Settings

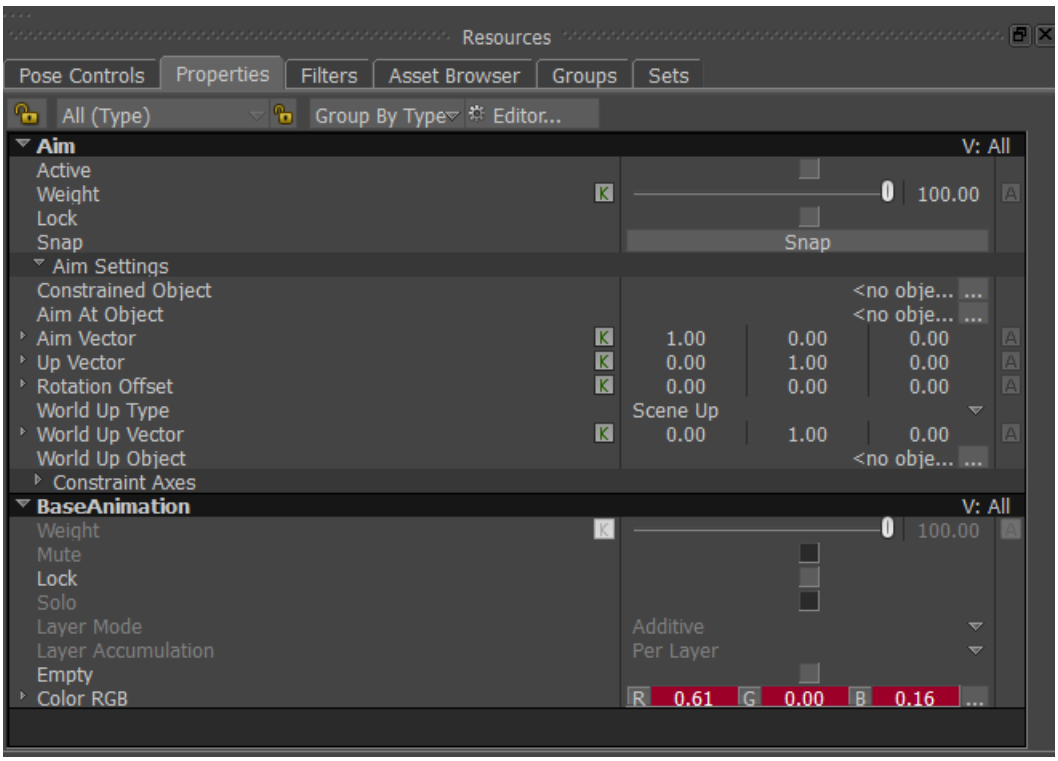


Figure 5: Aim Properties

Like all objects in MotionBuilder we directly expose some of the properties in the class, and in this case it is FBConstraint, however if you do not find the property you are looking for, you can do this using PropertyList.Find (see Module 4 for more details on this).

NOTE: you cannot access the 'Zero' button/functionality in Python it is not exposed at this point.

This is how you can set properties and settings (we haven't set the references yet, but we will do that in a next section).

For example:

```
from pyfbsdk import*

#Create two testing objects
lCam = FBCamera("MyCam")
lCam.Show = True
lCube = FBModelCube("MyCam")
lCube.Show = True

lConstraints = FBSystem().Scene.Constraints

for lCon in lConstraints:
    if lCon.Name == 'Aim':
        # Set the weight of the constraint
        lCon.Weight = 50
```

```
# Lock the state of the constraint
lCon.Lock = True
lCon.PropertyList.Find("Affect X").Data = False
lCon.Active = True
```

The 'FBConstraint' Class

This is the interface for constraints, including custom constraints implemented by plug-ins. This class handles most constraint types, or at least properties common to all constraints. However, some constraints are specialized enough such that full access is not possible through FBConstraint alone. For example, the full functionality of relations constraints can be accessed through FBConstraintRelation, while expression constraints are not yet fully exposed in C++ or Python.

Active is used to activate/deactivate a constraint. This is a read write public attribute.

What is a Reference Groups?

The categories where objects involved in a constraint can be found (e.g. source, constrained object, etc.) are called "reference groups" in FBConstraint. There are several functions for managing reference groups, which let you add/remove objects involved in the constraint.

A reference group can have more than one object. ReferenceGetCount() returns the number of objects currently in the specified group. The maximum count for a specified group can be obtained by ReferenceGroupGetMaxCount(). A maximum count of 0 means the group can handle an unlimited number of objects.

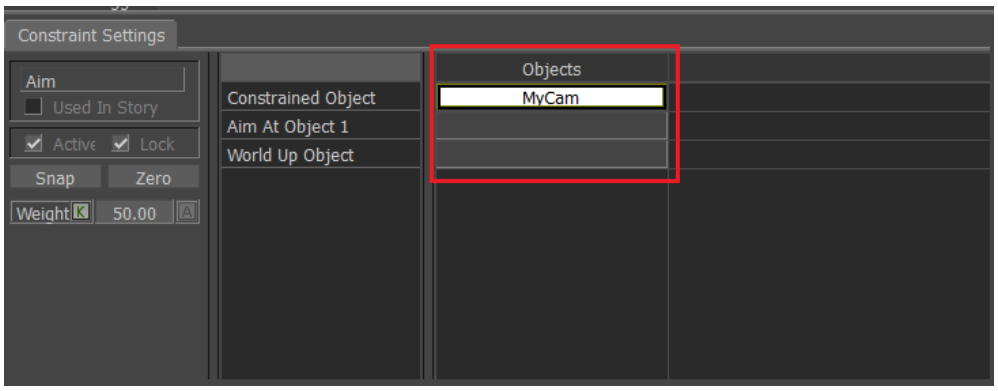


Figure 6: This Constraint has three Reference Groups

ReferenceGroupGetCount() returns the total number of reference groups, in the above screen shot the reference group count is 3.

ReferenceGroupGetName() returns the name of a group given an index, in the above screen shot the reference group names are Constrained Object, Aim At Object and World Up Object.

ReferenceAdd() adds an object to the group represented by the given index. ReferenceRemove() removes the object from the specified group, in the above screen shot, Constrained Object is set to MyCam .

ReferenceGet() obtains an object at the given index in the specified group index, in the above screen shot this would return the object MyCam.

You can also create new groups with *ReferenceGroupAdd()*, where the group name and its maximum object count are specified.

NOTE: There is a refresh issue when adding object to Reference Groups in Python, you need to navigate away from the setting window then go back to see them in the UI.

This is a sample to adding objects to the reference groups.

Steps to run:

1) Drag an Aim Constraint into the scene from the Asset Browser
2) Run this script:

```
from pyfbsdk import*

#Create two testing objects
lCam = FBCamera("MyCam")
lCam.Show = True
lCube = FBModelCube("MyCube")
lCube.Show = True

lConstraints = FBSystem().Scene.Constraints

for lCon in lConstraints:
    if lCon.Name == 'Aim':
        # Set the weight of the constraint
        lCon.Weight = 50
        # Lock the state of the constraint
        lCon.Lock = True
        for i in range(0, lCon.ReferenceGroupGetCount()):
            print lCon.ReferenceGroupGetName(i)
            if lCon.ReferenceGroupGetName(i) == 'Aim At Object':
                lCon.ReferenceAdd(i, lCube)
            elif lCon.ReferenceGroupGetName(i) == 'Constrained Object':
                lCon.ReferenceAdd(i, lCam)
        lCon.Active = True
```

3) In the Viewer go to View > Perspective > MyCam

4) Using the translation tool to move your cube around you should see camera moving with you.

8.2 Working with Complex Constraints

In MotionBuilder here is the list of complex constraints:

- Relation
  - Macro Boxes (not covered here)
- Expression (not covered here)
- Custom OR SDK (not covered here)

NOTE: In Python working with Expression and creating custom constraints from scratch is not exposed fully.

You can find these all in the UI in the Asset Browser here:

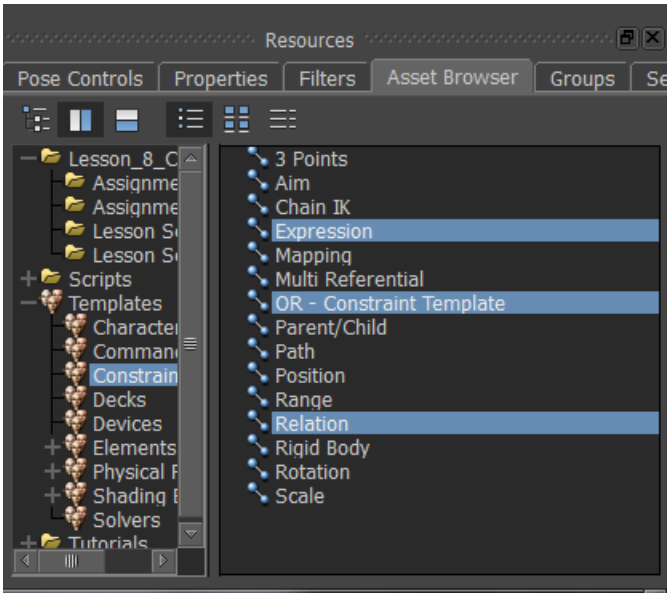


Figure 7: Complex Constraints

Relations constraints refer to constraints you create using a graphical interface, called the Relations pane in a connect-the-dots manner.

Elements of the relations constraint, known as objects, once they are added to the Relations pane, are connected to one another to form an equation. Once these elements are combined, they create a relations constraint that can be applied to a model.

Unlike other constraints, Relations constraints come with mathematical operators that you can use as building blocks to create very specific actions for your models. These building blocks are called Operators.

When a Relations constraint is dragged into the Viewer window, the Constraint settings displays the Relations pane, which is the “drawing board” on which you construct the relation.

Finding existing ‘Relation’ constraint in Python

To be able to location the different types of the constraints you need to use the list returned from FBSystem().Scene.Constraints:

```
from pyfbsdk import *

lConstraints = FBSystem().Scene.Constraints

for lConst in lConstraints:
    if lConst and lConst.Is(FBConstraintRelation_TypeInfo()):
        print lConst.Name
```

Creating your own ‘Relation’ constraint in Python

Creating a relation constraint in Python is the exact same as creating a simple constraint, the part that is complex is working with the connections.

```
from pyfbsdk import *

lConstraint = FBConstraintRelation("Relation_constraint")
```

Components that make up a Relation Constraint

The objects used to create a Relations constraint can be broken down into four types:

- Senders:
  - A Sender can be an input device or a model. Senders are used to transmit data to operations and Receivers. Senders only send data.
- Operators:
  - An Operator is an object that performs mathematical operations, comparisons, or conversions. It is placed between a Sender and a Receiver. Operators receive and send data.
- Receivers
  - A Receiver can be a model or an output device. Receivers receive data transmitted from Operators and Senders.
- Connections
  - This is the data that is being passed around from senders, operators and receivers.

When working inside of a relation constraint the majority of this functionality is provide to you via the 'FBConstraintRelation' class. This is a specialized class, derived from FBConstraint is used for managing relations constraints specifically.

Creating Senders Boxes

To add an object from the scene as a sender box, use SetAsSource() function.

For example:

```
from pyfbsdk import *

cube = FBModelCube('Cube')
cube.Show = True
relConst = FBConstraintRelation('MyConst')
Sender = relConst.SetAsSource(cube)
```

Creating Operator Boxes

To create a fucntion box, including plug-in ones, use CreateFunctionBox(). This is what we would consider an 'operator' in the UI; these can be found in the UI here:

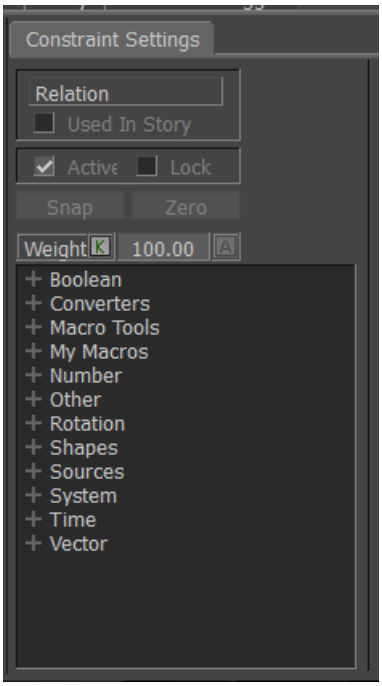


Figure 8: Function Boxes

To add an operator into a relation constraint:

```
from pyfbsdk import *
```

```
lConstraint = FBConstraintRelation("Relation_constraint")
```

Up to this point everything looks familiar, now the next line is where we create the function box. The two parameters it the function accepts corresponds to the UI naming (folder name, operator name):

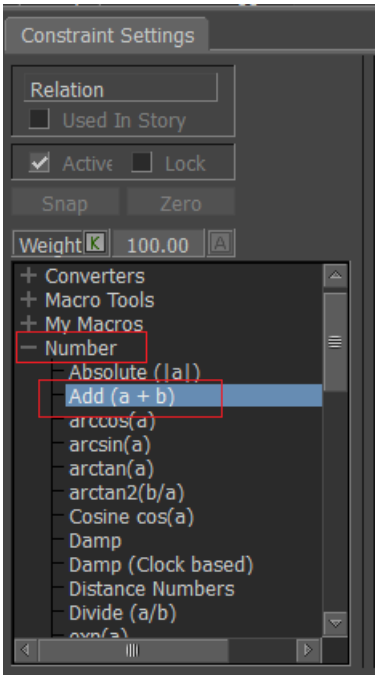


Figure 9: Function Box Naming

```
lBox =lConstraint.CreateFunctionBox('Number','Add (a + b)')
print lBox.Name
```

The next line SetBoxPosition, is where you want to set the function box in the UI pixel space, if you do not call this function it will not show up anywhere.

```
lConstraint.SetBoxPosition(lBox, 200, 80)
```

Just like a simple constraint we need to set the constraint property active for it to work in the scene (you only need to do this once for the whole relation constraint).

```
lConstraint.Active = True
```

NOTE: 200x80 view may not be in focus on your screen depending on the resolution, if you don't see the operator on the relation constraint and click 'a' to bring it into view.

Creating Receiver Boxes

To add an object as a receiver box, in a relation constraint use ConstrainObject() function.

For example:

```
from pyfbsdk import *

cube = FBModelCube('Cube')
cube.Show = True
relConst = FBConstraintRelation('MyConst')
Receiver = relConst.ConstrainObject(cube)
```

Creating Connections

All animatable elements derive in some way from FBBox, such as FBModel, FBDevice, FBConstraint, FBShader, etc. Objects of a class deriving from FBBox can be used as a relations constraint boxes.

The properties on relations constraint boxes can be obtained as animation nodes, using AnimationNodeInGet() for inputs and AnimationNodeOutGet() for outputs.

FBConnect() and FBDisconnect() are standalone functions used for connecting and disconnecting properties, respectively:

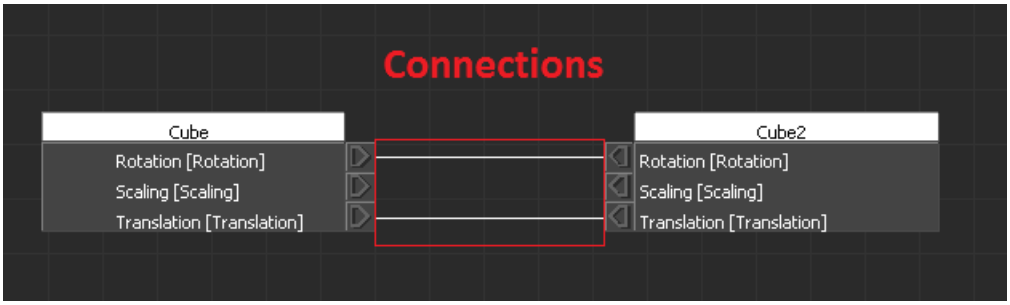


Figure 10: Connections between boxes

For example:

```
from pyfbsdk import *

def FindAnimationNode( pParent, pName ):
    lResult = None
    for lNode in pParent.Nodes:
        if lNode.Name == pName:
            lResult = lNode
            break
```

```
    return lResult

lCube = FBModelCube("Cube")
lCube.Visible=True
lCube.Show=True

lCam = FBCamera ( 'Camera' )
lCam.Show = True

lRelation = FBConstraintRelation("Cube Relation")
lRelation.Active =True

lBoxSenderCube = lRelation.SetAsSource(lCube)

lRelation.SetBoxPosition(lBoxSenderCube,10,10)
print lRelation.GetBoxPosition(lBoxSenderCube)

lBoxReceiverCam = lRelation.ConstrainObject(lCam)
lRelation.SetBoxPosition(lBoxReceiverCam,500,10)
print lRelation.GetBoxPosition(lBoxReceiverCam)

lCubeOut = FindAnimationNode( lBoxSenderCube.AnimationNodeOutGet(), 'Rotation' )
lCamIn = FindAnimationNode( lBoxReceiverCam.AnimationNodeInGet(), 'Translation' )
if lCubeOut and lCamIn:
    FBConnect( lCubeOut, lCamIn )
```

8.3 Triggering Scripts in Constraints

You can setup scripts to be triggered in a relation constraint and based on the value they receive, the script will be triggered. The trigger mechanism is very simple, if it is passed false, or zero, it is not triggered, if it is passed true, or one and greater the script will be executed.

I will go through the steps to set it up and indicate what is programmable and not:

1. Drag and drop script into Viewer, choose ‘Add to scene’ (not programmable)
2. Add a script device to the scene:

```
from pyfbsdk import *

lMouse = FBCreateObject ("Browsing/Templates/Devices", "Script", "myScript")
lSystem = FBSystem()
lSystem.Scene.Devices.append(lMouse)
```

3. Create a relation constraint:

```
relConst = FBConstraintRelation('MyConst')
```

4. Add the device to the Relation Constraint as a Receiver.

```
Receiver = relConst.ConstrainObject(lMouse)
```

This “triggering Scripts in Constraints” functionality can be very powerful when you build up tools in your pipeline, you can image in the relation constraint setting up some threshold boxes depending on different conditions (such as time, scene status, etc.), and depending on the different output values provided by the threshold boxes, different scripts will be triggered and executed. In the scene file attached in this lesson “TriggerScript.fbx”, if the playmode is 0, which means currently transport control stops, createCharacter.py will get executed, if the play mode is 1, which means currently transport control is playing forward, createCube.py will get executed. Note, you will need to activate the constraint and also make the device “online” to see this result.