

---

# XSI Scripting Basics

---

*Herman Tulleken (herman@luma.co.za)*

February 12, 2007

## Contents

<b>1</b>	<b>Introduction</b>	<b>2</b>
<b>2</b>	<b>The XSI SDK</b>	<b>2</b>
<b>3</b>	<b>The Tool Development Environment</b>	<b>2</b>
<b>4</b>	<b>Scene Objects</b>	<b>3</b>
4.1	Accessing scene objects . . . . .	3
4.2	Selected objects and Picking . . . . .	4
4.3	Creating Objects . . . . .	6
4.4	Exercises . . . . .	7
<b>5</b>	<b>Object Properties</b>	<b>7</b>
5.1	Kinematics Object . . . . .	8
5.2	XSI Collections . . . . .	8
5.3	F-Curves . . . . .	9
5.4	Exercises . . . . .	10
<b>6</b>	<b>Application Information</b>	<b>11</b>
<b>7</b>	<b>SDK Documentation</b>	<b>11</b>
7.1	Useful Pages . . . . .	12

# 1 Introduction

Here we will look at the very basics of scripting XSI: getting hold of scene objects, adjusting their properties, driving parameters using f-curves, using XSI collections, and a little bit about using the tool development interface.

# 2 The XSI SDK

The XSI SDK (Software Development Kit) is a huge body of software that allows you to manipulate XSI in a way that cannot be done through the interface. It consists out of several components, of which the Scripting Command API (Application Programming Interface) is one. Other components can be used to interface to XSI through C++, develop shaders, manipulate the dotXSI file format, or develop custom FX operators; these will not concern us here.

The scripting API allows us to do anything in XSI that can be done through the user interface: we can build scenes, animate models, change shortcut keys, and save files. There are about 1800 commands, 500 classes, 1700 methods, and 1700 data attributes.

# 3 The Tool Development Environment

Although XSI does not provide all the rich features of a proper IDE (Integrated Development Environment, that is, a tool for developing software), it does have some features that greatly simplifies scripting. The Tool Development Environment layout (access from menu under View, Layouts) provides, among others, the following panes:

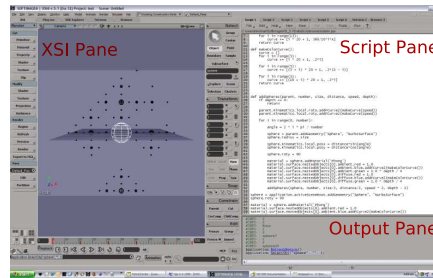


Figure 1: TDE Layout

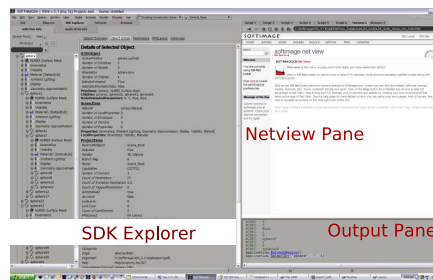


Figure 2: TDE Layout

**Scripting panes:** This is where you can load, write, and run scripts. A scripting pane does not provide syntax highlighting, but it does perform crude automatic indentation. If you select a command, and press F1, you will be taken to a list of relevant topics in the DSK documentation. You can also get syntax help from the context menu (right-click to access).

It is a good idea to always keep your main script in the first pane, and use one of the other panes as a scratch pad where you can try out short snippets of code before you incorporate them in your main script.

You have to set the language to Python for Python scripts to work. To do this, in your script pane, from the menu, select File, Preferences, and select “Python ActiveX Scripting Engine” as your scripting language.

**Output pane:** Output messages (from scripts), warning messages and error messages are printed in this pane. Your actions in the interface are also logged in this pane, for instance, if you select an object named `'sphere'`, the line

```
Application.SelectObj("sphere", "", 1)
```

is logged. This command appears as a hyperlink to the SDK documentation (to enable this functionality, in your script pane, select File, Preferences, and switch “Show Command hyperlinks” on).

**SDK Explorer:** This pane provides information about objects and other XSI elements relevant to scripting, such as their types and attribute values.

**Netview Pane:** This is a built in browser with special XSI capabilities. For examples, if a page contains a link to a script, it can be executed by clicking on the link.

## 4 Scene Objects

### 4.1 Accessing scene objects

There are several ways we can get hold of an XSI object in our scene:

- use its name as a parameter to the function `Application.getValue`;
- use the property `children` of an appropriate object;
- use the property `parent` of an appropriate object;
- query the application for the selected objects, through the property `Application.Selection`; or
- use the return value of the function with which the object was created;

Suppose we have a scene with a sphere (named `'sphere'`) with a child cube (named `'cube'`).

Here are a few examples of how to access the cube:

```

### Accessing an object using its name
cube = Application.GetValue('cube')
Application.LogMessage(cube)

### Accessing an object using the scene tree
cube = Application.activeSceneRoot.children[2].children[0]
Application.LogMessage(cube)

### Accessing an object through selection
cube = Application.Selection[0] #works only if the cube is selected
Application.LogMessage(cube)

```

Obtaining an object through its name is very convenient, especially for quick and dirty jobs. Because of the script's dependence on this name, this method is not well-suited for reusable scripts. It is also not a very good method of handling large sets of objects. We will not be using this method.

## 4.2 Selected objects and Picking

The variable `Application.Selection` gives access to the selected objects in the scene.

Set up your scene as follows:

Now run the following script:

```

selection = Application.Selection

for object in selection:
    Application.LogMessage(object)

```

Select some snippets in your scene, and run the snippet. You will see that the snippet prints the names of all the selected objects.

The following piece of code sets the selection to the children of the selected objects:

```

### Changes the selection to the children of selected object
newSelection = []
selection = Application.Selection

for object in selection:
    newSelection += object.children

if len(newSelection) > 0:
    Application.selectObj(newSelection[0])

    for object in newSelection[1:]:
        Application.addToSelection(object)

```

We might want to develop a tool that works on selected objects, but where the objects play different roles. For instance, suppose we want a tool that adds the local position of an object (the “source”) to another object (the “destination”). The `Application.Selection` object will not work, for we

cannot predict which object will be the source and which the destination. In such cases, we can invoke a pick session, letting the user pick objects in the order we determine. Here is how we would implement the “add location” script through picking:

```
from win32com.client import constants

srcPicked = Application.PickElement(constants.siObjectFilter,
    'Pick source', 'Pick source')

srcObject = srcPicked[2]

destPicked = Application.PickElement(constants.siObjectFilter,
    'Pick source', 'Pick source')

destObject = destPicked[2]

destObject.PosX = destObject.PosX.Value + destObject.PosX.Value
destObject.PosY = destObject.PosY.Value + destObject.PosY.Value
destObject.PosZ = destObject.PosZ.Value + destObject.PosZ.Value
```

The `Application.PickElement` command takes 7 arguments:

**SelFilter**

is of enumeration type `siFilter`, for example `siObjectFilter` or `siModelFilter`. A complete list of filters is given in the documentation under **Scripting Reference/Constants/F/FilterConstant**. See below for an explanation of enumeration types.

! →

**LeftMessage**

Supposedly the status bar message for the left mouse button (I don't see it).

**MiddleMessage**

Supposedly the status bar message for the middle mouse button (again, I don't see it).

**PickedElement**

This is an output argument, and can be ignored (see below for explanation)

**ButtonPressed**

An output argument.

**SelRegionMode**

An integer that specifies the type of selection to perform. The default, 0, uses the current selection mode. Refer to the documentation for the other modes.

**ModifierPressed**

An output argument.

The three output arguments is where you would put variables that would contain the output had you used VBScript. If you use Python, the output is returned as a tuple instead. The `PickElement` command has three output arguments, therefore a tuple with three elements is returned, in this order:

buttonPressed, modifierPressed, pickedElement. The order can be deduced from the examples in the SDK documentation. As you can see, in this example it does not conform to the order given in the commands parameter list.

enumeration type: The first argument is a constant. Groups of constants are often given a name by the XSI documentation, in which case the group of constants is an *enumeration type*. Other languages provides explicit support for enumeration types. In Python, however, we merely use this “type” name to determine which constants are valid in a particular situation. You must import constants from the `win32.client` module to use the constant variables.

You should always check whether the right button has been pressed in a picking session, and halt execution if it has. Here is an improved version of the script above:

```
from win32com.client import constants

def pickElements():
    srcPicked = Application.PickElement(constants.siObjectFilter,
        'Pick source', 'Pick source')

    if srcPicked[0] == 0: #0 is right mouse button
        return

    srcObject = srcPicked[2]

    destPicked = Application.PickElement(constants.siObjectFilter,
        'Pick source', 'Pick source')

    if destPicked[0] == 0:
        return

    destObject = destPicked[2]

    destObject.PosX = destObject.PosX.Value + destObject.PosX.Value
    destObject.PosY = destObject.PosY.Value + destObject.PosY.Value
    destObject.PosZ = destObject.PosZ.Value + destObject.PosZ.Value

pickElements()
```

### 4.3 Creating Objects

There are many methods with which objects can be added to a scene. They basically work the same, so we will look only at one example: `addGeometry`. Other methods for adding objects can be found in the DSK documentation under **Scripting Reference/Objects/X/X3DObject**.

! →

To add geometry as the child of an object (say, the scene root), we call the `addGeometry` method of that object. The method returns the (Python version of) the newly created object, which we can further manipulate through code. The following example adds a cube to the scene root, and prints its name to the output pane:

```
cube = Application.ActiveSceneRoot.addGeometry('Cube',
        'NurbsSurface', 'MyCube')
```

```
Application.LogMessage(cube)
```

The method's first parameter is a string specifying the object's preset; the second parameter is a string specifying the type of geometry. A list of presets and geometry types can be found in the SDK documentation under **Scripting Reference/Other Reference/Primitive Presets**. The final parameter is the name of the new object. XSI automatically adds a counter to the name if it exists.

! →

The following piece of code creates 100 randomly positioned and randomly sized spheres:

```
from random import random

for i in range(100):
    sphere = Application.ActiveSceneRoot.addGeometry('Sphere',
        'NurbsSurface')
    sphere.radius = 2 * random() + 1
    sphere.PosX = random() * 15
    sphere.PosY = random() * 15
    sphere.PosZ = random() * 15
```

#### 4.4 Exercises

1. Write a script that will create the following primitives:
  - (a) a icosahedron;
  - (b) a grid; and
  - (c) a null (Hint: use the AddNull method); and
  - (d) a spiral (Hint: use AddPrimitive).
2. Implement a function that takes a X3DObject as an argument, and adds a sphere as a child to that object. Then use this function in a script that adds objects to all the selected items.
3. Develop a script that inverts the names of two objects the user picks.
4. The PickPosition command is similar to the PickElement command:

```
buttonPressed, x, y, z = Application.PickPosition (
    'Pick position', 'Pick position')
```

Use it in a script that lets the user pick a position, and adds a sphere in that location. Remember to halt execution if the right mouse button has been clicked.

## 5 Object Properties

In the last example, we have accessed some of the sphere's properties to change its size and position. In general, properties are data attributes, and are accessed through the dot notation. Many properties look and behave

like they are of primitive types (such as integers or strings). They can be assigned with these values, they can be used as these values in expressions, and they appear as these when printed.

```
sphere = Application.ActivePrimitive.addGeometry('Speher',
  4,'NurbsSurface')
sphere.PosX = 5
sphere.PosZ = sphere.PosX + sphere.PosY
```

```
Application.LogMessage(sphere.PosZ)
```

However, the following gives an error message:

```
sphere = Application.ActivePrimitive.addGeometry('Sphere',
  'NurbsSurface')
sphere.PosX = 5
sphere.PosX += 5
```

The reason is that this property is *not* in fact a number, but an emulation type that does not support the += operator. To get the actual numeric value, we have to access the `Value` property of `PosX`, like this:

```
sphere = Application.ActivePrimitive.addGeometry('Sphere',
  'NurbsSurface')
sphere.PosX = 5
sphere.PosX.Value += 5
```

The illusion that `PosX` is a number was created intentionally by the developers of the scripting API—to support f-curves and other bells and whistles, `PosX` must be a complex type; letting it emulate a numeric is for our convenience.

## 5.1 Kinematics Object

**Kinematics:** Every object has an attribute *Kinematics*. This object has two attributes, kinematics state: each a *kinematics state* object, which holds the local and global kinematics states. They are called `local` and `global`. Among others, these objects have the following parameters:

`PosX`, `PosY`, `PosZ`  
The position of this object.

`RotX`, `RotY`, `RotZ`  
The rotation of this object. These can be found under the *Ori* group in the explorer.

`SclX`, `SclY`, `SclZ`  
The scale of this object.

`SclrX`, `SclrY`, `SclrZ`  
The shear of this object. These can be found under the *SclOri* group in the explorer.

```
sphere = Application.ActiveSceneRoot.addGeometry('Sphere',
  'NurbsSurface')
```

```

sphere.Kinematics.Local.PosX = 5
sphere.Kinematics.Global.PosY = 1
sphere.Kinematics.Local.RotX = 90
sphere.Kinematics.Local.SclZ = 2
sphere.Kinematics.Local.SclrX = 45

```

You might be wondering why we have used `sphere.posX` before, and not `sphere.Kinematics.Local.PosX`? The shorter notation is equivalent to the longer version. It is provided as a shortcut to the script developer. The attributes of the local kinematics state object have been *promoted* to appear as attributes of `X3DObject`. In the SDK documentation [Scripting Reference/Shortcut Reference](#) gives a list of all shortcuts. Note that the global kinematics attributes cannot be accessed using shortcuts.

## 5.2 XSI Collections

The scripting API uses collections extensively. Parameters (of an object), keys (of an f-curve) and selected items are all stored in specialised XSI collections. While some of these collections provide extra methods or attributes, all emulate sequence types, so that you can use familiar notation to access elements and iterate through them.

```

for object in Application.Selection:
    Application.LogMessage(object)

firstObject = Application.Selection[0]

```

Here is a useful code snippet that prints the names of all the attributes of an object:

```

def printParameters(obj)
    for parm in obj.Parameters:
        Application.LogMessage(parm.FullName)

```

## 5.3 F-Curves

f-curve: A *f-curve* can be added to a property by using the function `addFCurve2`. The function takes a list (or tuple) of alternating frame numbers and values. The following example adds an f-curve to an object's x-position.

```

cube = Application.ActiveSceneRoot.addPrimitive('Cube',
'NurbsSurface')

cube.PosX.addFCurve2([1, 0, 11, 5, 21, 0])

```

If a parameter is already driven by a f-curve, we can use the `source` command of that parameter to manipulate the f-curve using the methods of the class `FCurve`. In the following example, we adjust the times of the keys:

```

cube = Application.ActiveSceneRoot.addPrimitive('Cube',
'NurbsSurface')
cube.PosX.addFCurve2([1, 0, 11, 5, 21, 0])
fCurve = cube.PosX.source

time = 1 #first frame

```

```

for key in fCurve.Keys:
    key.Time = time
    time += 20 # next frame will
                # be 20 frames later

```

We can also see what a parameters value will be at a given frame (not necessarily a key-frame) by evaluating the f-curve at that frame using the Eval method:

```

cube = Application.ActiveSceneRoot.addPrimitive('Cube',
    'NurbsSurface')
cube.PosX.addFCurve2([1, 0, 11, 5, 21, 0])
fCurve = cube.PosX.source

Application.LogMessage('PosX at frame 4 is ' +
    str(fCurve.Eval(4)))

```

## 5.4 Exercises

1. Implement a function that takes four arguments: an X3DObject and three floating point numbers. The function must set the objects local position to the three given numbers.
2. Implement a function that takes four arguments: an object and three floating point numbers. The function must add a Phong material to the given object, and set the diffuse and ambient color of the object to the given RGB values.
3. Implement a function that returns a list of values that can be used to make an f-curve. The function must take the first frame, the last frame, the step size, the beginning value of the parameter, and the end value of the parameter. For instance,

```
makeFCurve(1, 10, 1, 0, 90)
```

must return the list

```
[1, 0, 2, 10, 3, 20, 4, 30, 5,
    40, 6, 50, 7, 60, 8, 70, 9, 80,
    10, 90]
```

4. Modify the function above to take an additional argument. The additional argument is a function, that takes a number as argument, and returns the argument.

```
makeFCurve(1, 10, 1, 0, 90, cos)
```

```
[1, 1.0,
    2, -0.83907152907645244, 3, 0.40808206181339196,
    4, 0.15425144988758405, 5, -0.66693806165226188,
    6, 0.96496602849211333, 7, -0.95241298041515632,
    8, 0.63331920308629985, 9, -0.11038724383904756,
    10, -0.44807361612917013]
```

5. Implement a function that takes four arguments: an X3DObject, an integer (childrenCount), and two floats (radius, minDistance). The

function must add `childrenCount` spheres to the object, with radius `radius` and at a random location at least `minDistance` from the parent, but not further than twice `minDistance`.

6. Modify the function above to take another integer argument `depth`. If this argument is 0 or less, the function should return without doing anything. Otherwise the function must call itself again, but pass each child as parameter, with half the radius and minimum distance, and one less depth.
7. Modify the function above to add a f-curve to the parent object so that the object revolves around one of its axes - one revolution in 100 frames (from 1 to 101).
8. Modify the function above to call your coloring function implemented earlier.
9. Write a script that adds a single sphere to the scene root, color it (using your coloring function), animate it (add a f-curve to it), and call the function above on it. Play around with the parameters until you get a decent effect.
10. Implement a function that takes two `X3DObjects`, and returns a duplicate of the first, with its local position parameters set to halfway between the two objects. (Hint: use the `Application.Duplicate` command). Now write a script that calls this function on two user picked (not selected!) objects.
11. Implement a function similar to the one above, but that will work if the positions of the objects have been animated using f-curves. Use the first frame to do the calculation.
12. Implement a function similar to the one above, except that it should add a f-curve to the object so that the object's position is always the average of the other two objects'.

## 6 Application Information

The global object `Application` provides information about the application's status. Here are a few of its methods:

### `ActiveProject`

The active project. Changing this value changes the active project.

### `ActiveSceneRoot`

Gives the active scene's root model.

### `ActiveToolName`

Gives the name of the currently active tool, i.e. one of `RotationTool`, `ScalingTool` or `TranslateTool`. If the tool is not recognised, an empty string is given.

### `Commands`

A collection of all the available commands, including built-in and custom commands.

### Preferences

An object of type `Preferences` that can be used to manage XSI preferences through scripting.

### Selection

A collection of selected objects.

## 7 SDK Documentation

The SDK documentation can be daunting.

Because XSI supports four scripting languages, the documentation use terms for programming concepts that sometimes differ from those in Python. Here is a list of such terms, with their Python equivalents:

**Array:** Tuple or list. In the case of a return value, it is always a tuple.

**Double:** Float.

**Long:** Integer.

**Property:** Data attribute.

**VARIANT:** used to mean “any type”. This does not actually mean that a method or function can take any type, just that it is not specified. This is not really important for Python programs, since the type of arguments and return values are *never* specified.

return value: The XSI documentation uses the term *return value* to describe the *type* of a data attribute. This usage can be confusing, since data attributes do not return values.

scripting syntax: The *scripting syntax* under a method entry shows how a method should be used, for example:

```
Application.LogMessage (Message, [Severity])
```

shows that `Application.LogMessage` can take two arguments. Arguments in square brackets are optional.

enumeration types: *Enumeration types* are groups of named constants, usually strings or integers. For instance, the second argument of `Application.LogMessage` is of pseudo-type `siSeverity`, which contains among others the constants `siFatal` and `siError`. A constant can be used like any other variable.

command: An XSI *command* is simply a method of the `Application` object. In other languages, commands can be used directly, for example, in VBScript we can use the following:

```
LogMessage "hello"
```

but in Python, we need to call `LogMessage` on `Application`:

```
Application.LogMessage("hello")
```

### 7.1 Useful Pages

The following is short list of useful pages. The path in brackets refers to the table of contents, for example *Answers to Basic Questions* can be found under *Scripting Reference*, under *What's Available*.

- ! → **Answers to Basic Questions: (Scripting Reference/What's Available/)** This section gives a list of links to documents describing some basic concepts of scripting for XSI, including finding the type of an object, the difference between owners and parents, and using collections. You should read all these!
- ! → **Using shortcuts: (Plug-in Developer's Guide/Working with Parameters/ Accessing Parameter Values/)** Provides information about shortcuts.
- ! → **Issues with Python: (Script Development)** Lists and addresses issues specific to Python.
- ! → **Global (Intrinsic) Objects: (Script Development)** Lists the available global objects (such as `Application`) and what they are used for.

## Index

Kinematics, 8

command, 12

constant

    XSI, 12

enumeration type, 6

enumeration types, 12

f-curve, 9

kinematics state, 8

promoted, 9

return value, 12

scripting syntax, 12