# *ParaView Scripting with Python*

August, 2007

# *ParaView and Python*

ParaView offers rich scripting support through Python. This support is available as part of the ParaView client (paraview), an MPI-enabled batch application (pvbatch), the ParaView python client (pvpython) or any other Python-enabled application. Using Python, users and developers can gain access to the ParaView engine called Server Manager[1].

# *Quick Start - a Tutorial*

## Getting Started

To start interacting with the Server Manager, you have to load the servermanager module. This module can be loaded from any python interpreter as long as the necessary files are in PYTHONPATH. These files are the shared libraries located in the paraview binary directory and python modules in the paraview directory: paraview/servermanager.py, paraview/vtk.py etc. You can also use either pvpython (for stand-alone or client/server execution), pvbatch (for non-interactive, distributed batch processing) or the python shell invoked from Tools -> Python Shell using the ParaView client to execute Python scripts. You do not have to set PYTHONPATH when using these.

In this tutorial, I will be using the python integrated development environment IDLE. My PYTHONPATH is set to the following.

```
/Users/berk/work/paraview3-build/bin:/Users/berk/work/paraview3-build/Utilit
ies/VTKPythonWrapping
```

This is on my Mac and using the build tree. In IDLE, let's start by loading the servermanager module.

```
>>> from paraview import servermanager
```

Note: Importing the paraview module directly is deprecated although still possible for backwards compatibility. This document refers to the servermanager module alone. Next, we need to connect to a server if we are not already connected.

```
>>> if not servermanager.ActiveConnection:
...   connection = servermanager.Connect()

Connection (builtin:5)
```

In this example, we connected to the built-in server. When using pvbatch, this is the only connection mode supported, even when pvbatch is running as an MPI task. When running pvpython or loading servermanager from an external python interpreter, we can also connect to a remote server as shown in the following example

```
>>> connection = servermanager.Connect('localhost')
Connection (localhost:11111)
```

This assumes that you already started the server (pvserver) on the local machine.

---

[1] Server Manager is a library that is designed to make it easy to build distributed client-server applications.

Note: `Connect()` returns a connection object on success and returns `None` on failure. You might want to check the return value to make sure connection succeeded.

```
>>> connection = servermanager.Connect('localhost')
>>> if not connection:
...   raise exceptions.RuntimeError, "Connection to localhost failed."
```

Note: When importing the servermanager module from the application's python shell, Connect() should not be called as a connection already exists.

## Creating a Pipeline

When first loaded, the servermanager module creates several sub-modules that can be used to create a visualization. The most important ones are listed below.

- sources
- filters
- rendering

You can get a list of classes these modules contain by using dir() as shown in the following example.

```
>>> dir(servermanager.sources)
['AVSucdReader', 'ArrowSource', 'Axes', 'CSVReader', 'ConeSource', 'Cube-
Source', 'CylinderSource', 'DEMReader', 'ExodusIIReader', 'ExodusReader',
'FLUENTReader', 'Facet Reader', 'GlyphSource2D', 'HierarchicalFractal', 'Im-
ageMandelbrotSource', 'ImageReader', ...]
```

Let's start by creating a ConeSource object:

```
>>> coneSource = servermanager.sources.ConeSource()
```

The object assigned to `coneSource` is a proxy for the actual vtkConeSource. This proxy provides a set of properties and methods to control the behavior of the underlying VTK object(s). These objects may be in the same process (built-in server) or on one or more server processes (distributed remote server). The proxy will handle communication with the VTK objects in a transparent way. You can get some documentation about the proxy using `help()`.

```
>>> help(coneSource)
Help on ConeSource in module paraview.servermanager object:

class ConeSource(Proxy)
 |  The Cone source can be used to add a polygonal cone to the 3D scene. The
output of the Cone source is polygonal data.
 |
 |  Method resolution order:
 |      ConeSource
 |      Proxy
 |      __builtin__.object
 |
 |  Methods defined here:
```

```
 |
 |  Initialize = aInitialize(self, connection=None)
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors defined here:
 |
 |  Capping
 |      If this property is set to 1, the base of the cone will be capped
with a filled polygon. Otherwise, the base of the cone will be open.
 |
 |  Center
 |      This property specifies the center of the cone.
 |...
```

This gives you a full list of properties. Let's check what the resolution property is set to.

```
>>> coneSource.Resolution
Property name= Resolution value = 6
```

You can increase the resolution as shown below.

```
>>> coneSource.Resolution = 32
```

Alternatively, we could have specified a value for resolution when creating the proxy.

```
>>> coneSource = servermanager.sources.ConeSource(Resolution=32)
```

You can assign values to any number of properties during construction using keyword arguments.

Let's also change the center.

```
>>> coneSource.Center
Property name= Center value = [0.0, 0.0, 0.0]
>>> coneSource.Center[1] = 1
```

Vector properties such as this one support setting and getting of individual elements as well as slices (ranges of elements).

```
>>> coneSource.Center[0:3] = [1, 2, 3]
>>> coneSource.Center
Property name= Center value = [1.0, 2.0, 3.0]
```

Next, let's apply a shrink filter to the cone:

```
>>> shrinkFilter = servermanager.filters.ShrinkFilter(Input=coneSource)
>>> shrinkFilter.Input
Property name= Input value = <paraview.servermanager.ConeSource object at
0x2d00dd90>:0
```

At this point, if you are interested in getting some information about the output of the shrink filter, you can force it to update (which will also cause the execution of the cone source. For details about VTK pipeline model, see one of the VTK books.

```
>>> shrinkFilter.UpdatePipeline()
>>> shrinkFilter.GetDataInformation().GetNumberOfCells()
33L
>>> shrinkFilter.GetDataInformation().GetNumberOfPoints()
128L
```

We will cover the DataInformation class in more detail later.

## Rendering

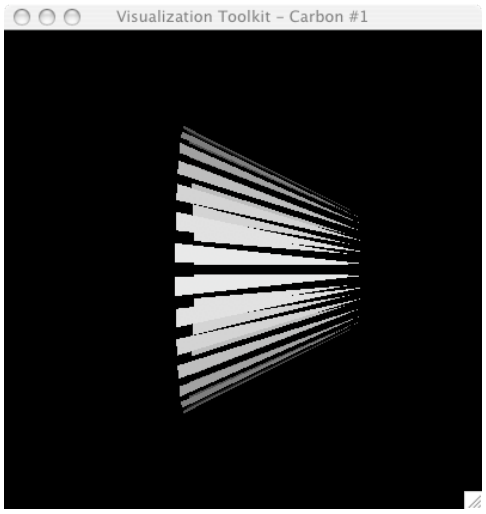
Now that we created a small pipeline, let's render the result. You will need two objects to render the output of an algorithm in a scene: a representation and a view. A representation is responsible for taking a data object and rendering it in a view. A view is responsible for managing a render context and a collection of representations.

```
>>> view = servermanager.CreateRenderView()
>>> rep = servermanager.CreateRepresentation(shrinkFilter, view)
>>> view.StillRender()
```

Oops, nothing is visible. We need to reposition the camera to contain the entire scene.

```
>>> view.ResetCamera()
>>> view.StillRender()
```

Et voila:




`CreateRenderView()` and `CreateRepresentation()` are special methods in the servermanager module to facilitate the creation of representations and views. `CreateRepresentation()` automatically adds the new representation to the view.

```
>>> view.Representations
Property name= Representations value =
[<paraview.servermanager.UnstructuredGridRepresentation object at
0x2d0170f0>]
```

This was a quick introduction to the servermanager module. In the following sections, we will discuss the servermanager in more detail and introduce more advanced concepts.

# *paraview.servermanager Module*

The servermanager module is a ParaView component written using Python on top of the VTK Server Manager C++ library. Its purpose is to make it easier to create ParaView data analysis and visualization pipelines using Python. The servermanager module can be loaded from Python interpreters running in several applications.

- **pvpython:** The pvpython application, distributed with the ParaView application suite, is a Python client to the ParaView servers. It supports interactive execution as well as batch execution.
- **pvbatch:** The pvbatch application, also distributed with the ParaView application suite, is a Python application designed to run batch scripts on distributed servers. When ParaView is compiled with MPI, pvbatch can be launched as an MPI program. In this mode, the first node will load a Python script specified as a command-line argument and execute it using a special built-in connection on all nodes. This application does not support interactive execution.
- **paraview:** Python scripts can be run from the paraview client using the Python shell that is invoked from Tools -> Python Shell. The Python shell supports interactive mode as well as loading of scripts from file.
- **External Python interpreter:** Any Python-capable application can load the paraview.servermanager module if the right environment is configured. For this to work, you either have to install the paraview Python modules (including the right shared libraries) somewhere in sys.path or you have to set PYTHONPATH to point to the right locations.

## Overview

The paraview.servermanager module contains several Python classes designed to be Python-friendly as well as all classes wrapped from the C++ Server Manager library. The most important classes are as follows.

- Proxy
- ProxyManager
- Property and sub-classes

We will cover these classes in detail in following sections.

When first loaded,  the servermanager module creates several sub-modules. The most important ones are as follows.

- sources
- filters
- rendering

These modules are automatically populated (at load time) with various Proxy sub-classes as defined in the Server Manager configuration files (ParaView3/Servers/ServerManager/Resources/*.xml). This allows direct instantiation of actual Proxy sub-classes (for example, SphereSource) instead of having to use

`vtkSMProxyManager::CreateProxy()`. Furthermore, the documentation for each class can be obtained using `help()`.

## Connecting to a Server

Unless you are using the Python shell that is in the paraview application, the first step to any Server Manager Python script is connecting to a server. Even when running in stand-alone mode, you have to connect to a built-in server.

To connect to a server, use `servermanager.Connect()`. This method takes 4 arguments, all of which have default values.

```
def Connect(ds_host=None, ds_port=11111, rs_host=None, rs_port=11111)
```

When connecting to the built-in server and running pvbatch, do not specify any of these arguments. The default values work well.

When connecting to a hybrid server (pvserver), specify only the first 2 arguments. These are the server name (or IP address) and port number.

When connecting to a data-server/render-server pair, you have to specify all four arguments. The first 2 are the host name (or IP address) and port number of the data server, the last 2 those of the render server.

Here are some examples.

```
# Connect to built-in
>>> connection = servermanager.Connect()

# Connect to pvserver running on amber1 (first node of our test cluster)
# using the default port 11111
>>> connection = servermanager.Connect('amber1')

# Connect to pvdataserver running on the amber cluster, pvrenderserver
# running on Berk's desktop
>>> connection = servermanager.Connect('amber1', 12000, 'kamino', 11111)
```

Although not fully implemented yet, the servermanager supports multiple connection. The module keeps track of the active connection with the `ActiveConnection` variable. By default, all communication is sent to the server to which the `ActiveConnection` points. The first time `Connect()` is called, servermanager automatically sets `ActiveConnection`. After the first time, unless `Disconnect()` is called first, `ActiveConnection` is not set. If you want to set the `ActiveConnection` to another connection, use the return value from `Connect()`.

```
>>> servermanager.ActiveConnection = servermanager.Connect('amber1')
```

Note: `Connect()` will return `None` on failure. To be safe, you should check the return value of `Connect()`.

## Getting Help

The servermanager module is well documented. You can access it's online documentation using `help()`.

```
>>> from paraview import servermanager
>>> help(servermanager)
```

You can also access the documentation of all Proxy types using the sources, filters and rendering modules.

```
>>> dir(servermanager.filters)
['AllToN', 'Append', 'AppendAttributes', 'AppendPolyData', 'ArbitrarySource-
Glyph', 'ArbitrarySourceStreamTracer', 'Balance', 'BrownianPoints',
'CTHPart', 'Calculator', 'CellCenters', 'CellDataToPointData', 'CleanPoly-
Data', 'CleanUnstructuredGrid', 'ClientServerMoveData', 'Clip', 'Contour',
'Curvatures', 'Cut', 'D3', 'DataSetSurfaceFilter', 'DataSetTriangleFilter',
'DecimatePro', 'Delaunay2D', ...]
>>> help(servermanager.filters.Calculator)
```

This documentation is automatically generated from the Server Manager configuration files and is identical to the class documentation found under the paraview Help menu.

Beyond this document and the online help, there are a few useful documentation sources.

- The ParaView Guide: http://www.kitware.com/products/paraviewguide.html
- The ParaView source documentation: http://www.paraview.org/doc/
- The ParaView Wiki: http://paraview.org/Wiki/ParaView

If you are interested in learning more about the Visualization Toolkit that is at the foundation of ParaView, visit http://vtk.org.

# *Proxies and Properties*

## Proxies

The VTK Server Manager design uses the Proxy design pattern[2]. Quoting from Wikipedia: "A proxy, in its most general form, is a class functioning as an interface to another thing. The other thing could be anything: a network connection, a large object in memory, a file, or some other resource that is expensive or impossible to duplicate". In the case of Server Manager, a Proxy object acts as a proxy to one or more VTK objects. Most of the time, these are server-side objects and are distributed to the server nodes. Proxy objects allow you to interact with these object as if you directly have access to them, manipulate them and obtain information about them. When creating visualization pipelines, you create proxies instead of VTK objects.

```
>>> sphereSource = vtk.vtkSphereSource() # VTK-Python script

>>> sphereSourceP = servermanager.SphereSource() # servermanager script
```

---

[2] See Design Patterns: Elements of Reusable Object-Oriented Software by Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides for details.

A proxy also provides an interface to modify the properties of the objects it maintains. For example, instead of

```
>>> sphereSource.SetCenter(1.0, 1.0, 0.0)
```

you can write the following.

```
>>> sphereSourceP.Center = [1.0, 1.0, 0.0]
```

Properties are covered in detail in the next section.

In the VTK Server Manager library, proxies are instances of vtkSMProxy or one of it's sub-classes. The servermanager module wraps these classes with a set of Python classes to make them more Python-friendly. Using the Server Manager library, you create a proxy with the help of the Proxy Manager[3] :

```
>>> pxm = servermanager.ProxyManager()
>>> sphereSourceP = pxm.SMProxyManager.NewProxy("sources", "SphereSource")
>>> type(sphereSourceP)
<type 'vtkobject'>
>>> sphereSourceP.GetClassName()
'vtkSMSourceProxy'
```

The proxy groups (for example "sources") and the proxy types that can be created are defined in a set of XML configuration files (e.g., ParaView3/Servers/ServerManager/Resources/*.xml). Using the servermanager module, you can directly create a proxy by instantiating the appropriate class.

```
>>> sphereSourceP = servermanager.sources.SphereSource()
>>> type(sphereSourceP)
<class 'paraview.servermanager.SphereSource'>
>>> sphereSourceP.GetClassName()
'vtkSMSourceProxy'
```

A servermanager proxy object contains a Server Manager proxy object. You can access this object directly using the SMProxy data member:

```
>>> type(sphereSourceP.SMProxy)
<type 'vtkobject'>
```

All unknown attribute requests made to a servermanager proxy are passed to the SMProxy so you normally should not need to access SMProxy directly[4].

## Properties

Property objects are used to read and modify the properties of VTK objects represented by proxies. Each proxy has a list of properties defined in the Server Manager configuration files. The property interface of the Server Manager library is somewhat cumbersome. Here is how you can set the radius property of a sphere source.

---

[3] Covered in detail later.

[4] One common use for accessing SMProxy directly is to get it's attributes using dir(): dir(sphereSourceP.SMProxy).

```
>>> rp = sphereSourceP.GetProperty("Radius")
>>> rp.SetElement(0, 2)
1
>>> sphereSourceP.UpdateProperty("Radius")
```

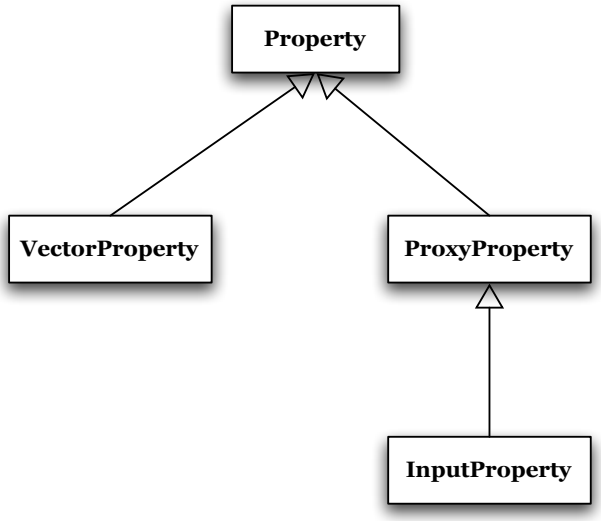
The servermanager module makes property access much easier by defining Python property accessors for property objects:

```
>>> sphereSourceP.Radius = 3
```

Here Radius is a Python property which, when a value is assigned to it, calls `sphereSourceP.SetPropertyWithName(Radius,3)`. The same property returns a servermanager Property object when accessed.

```
>>> type(sphereSourceP.Radius)
<class 'paraview.servermanager.VectorProperty'>
```

The class hierarchy for property classes looks like this.




All Property classes define the following methods.

- `__len__()`
- `__getitem__()`
- `__setitem__()`
- `__getslice__()`
- `__setslice__()`
- `GetData()`
- `SetData()`.

Therefore, all of the following are supported.

```
>>> sphereSourceP.Center
Property name= Center value = [0.0, 0.0, 0.0]
```

```
>>> sphereSourceP.Center[0] = 1
>>> sphereSourceP.Center[0:3] = [1,2,3]
>>> sphereSourceP.Center[0:3]
[1.0, 2.0, 3.0]
>>> sphereSourceP.Center.SetData([2,1,3])
>>> sphereSourceP.Center.GetData()
[2.0, 1.0, 3.0]
>>> len(sphereSourceP.Center)
3
```

ProxyProperty and InputProperty also define

- `append()`
- `__delitem__()`
- `__delslice__()`

to support `del()` and `append()`, similar to Python list objects.

VectorProperty is used for scalars, vectors and lists of integer and floating point numbers as well as strings (vtkSMIntVectorProperty, vtkSMDoubleVectorProperty and vtkSMStringVectorProperty, respectively) . Most properties of this type are simple. Examples include SphereSource.Radius (double scalar), SphereSource.Center (vector of doubles), GlyphSource2D.Filled (boolean), GlyphSource2D.GlyphType (enumeration), VectorText.Text (string) and Contour.ContourValues (list of doubles). Some properties may be more complicated because they map to C++ methods with mixed argument types. Two good examples of this case are Glyph.SelectInputScalars and ExodusIIReader.PointResultArrayStatus.

```
>>> reader =
servermanager.sources.ExodusIIReader(FileName=.../disk_out_ref.ex2")
>>> reader.UpdatePipelineInformation()
# Get information about arrays and whether they will be read. By
# default no arrays are read.
>>> reader.PointResultArrayInfo
Property name= PointResultArrayInfo value = ['Temp', '0', 'V', '0', 'Pres',
'0', 'AsH3', '0', 'GaMe3', '0', 'CH4', '0', 'H2', '0']
# Enable the Temp array
>>> reader.PointResultArrayStatus = ['Temp', '1']
# Force read
>>> reader.UpdatePipeline()
# Now check the output
>>> pdi = reader.GetDataInformation().GetPointDataInformation()
>>> pdi.GetNumberOfArrays()
1
>>> pdi.GetArrayInformation(0).GetName()
'Temp'
```

This example demonstrates the use of ExodusIIReader.PointResultArrayStatus. This is a VectorProperty that represents a list of tuples of strings of 2 elements: (array name, status [on|off]). The underlying C++ function has a signature of `SetPointResultArrayStatus( const char* name, int flag )`. This method is usually called once per array to en-

able or disable it (i.e. to set whether the reader will read a particular array). From the previous case, let's enable reading of `V` and `CH4`.

```
>>> reader.PointResultArrayStatus = [ 'V' , '1', 'CH4', '1' ]
```

Note that the property uses strings for both the array name and the flag (which is actually an integer). Properties do not directly support mixed types, thus forcing us to use strings in this situation.

Glyph.SelectInputScalars is more complicated.

```
>>> sph= servermanager.sources.SphereSource()
# Glyph source producing diamond
>>> source=servermanager.sources.GlyphSource2D(GlyphType=8)
>>> elev=servermanager.filters.ElevationFilter(Input=sph)
# Glyph the points of the sphere with diamonds
>>> glyph=servermanager.filters.Glyph(Input=elev, Source=source)
# Scale the glyph with the Elevation array
>>> glyph.SelectInputScalars = [ '0', '0', '0', '0', 'Elevation' ]
>>> glyph.ScaleMode = 0 # Use scalars
```

Here the property SelectInputScalars maps to `SetInputArrayToProcess( int idx, int port, int connection, int fieldAssociation, const char *name )` which has four integer arguments (some of which are enumeration) and 1 string argument[5].

A ProxyProperty can point to one or more proxies. This is equivalent to a C++ method that takes a pointer to a VTK object. Examples of ProxyProperty include Cut.CutFunction (single proxy) and RenderView.Representations (a list of proxies) .

InputProperty is similar to ProxyProperty; it is used to connect pipeline objects (source proxies). It is covered in more detail in the next section.

Properties are either regular (push) or information (pull) properties. Information properties do not have a VTK method associated with them and are responsible for getting information from the server. A good example of an information property is TimestepValues which returns all time steps available in a file (if the reader supports time).

```
>>> reader = servermanager.sources.ExodusIIReader(FileName=".../can.ex2")
>>> reader.UpdatePropertyInformation()
>>> reader.TimestepValues
Property name= TimestepValues value = [0.0, 0.00010007373930420727,
0.00019990510190837085, 0.00029996439116075635, 0.00040008654468692839,
0.00049991923151537776, 0.00059993512695655227, 0.00070004921872168779,
...]
```

By default, information properties are not pulled. To force a pull, call `UpdatePropertyInformation()` on the proxy.

You can obtain a list of properties a proxy supports by using `help()`. However, this does not allow introspection programmatically. If you need to obtain information about a proxy's properties programmatically, you can use a property iterator:

---

[5] See vtkAlgorithm documentation for details.

```
>>> for prop in glyph:
      print prop

Property name= Input value = <paraview.servermanager.ElevationFilter>:0
Property name= MaximumNumberOfPoints value = 5000
Property name= RandomMode value = 1
Property name= SelectInputScalars value = ['0', '0', '0', '0', 'Elevation']
Property name= SelectInputVectors value = ['1', '', '', '', '']
Property name= Orient value = 1
Property name= ScaleFactor value = 1.0
Property name= ScaleMode value = 0
Property name= Source value = <paraview.servermanager.GlyphSource2D>:0
Property name= UseMaskPoints value = 1
```

As an alternative, you can use a PropertyIterator object.

```
>>> it = s.__iter__()
>>> for i in it:
      print it.GetKey(), it.GetProperty()
```

## Domains (advanced)

The Server Manager provides information about values that are valid for properties. The main use of this information is for the user interface to provide good ranges and choices in enumeration. However, some of this information is also very useful for introspection. For example, enumeration properties look like simple integer properties unless a (value, name) pair is associated with them. The Server Manager uses Domain objects to store this information. The contents of domains may be loaded from xml configuration files or computed automatically. Let's look at an example.

```
>>> s = servermanager.sources.SphereSource()
>>> v = servermanager.CreateRenderView()
>>> r = servermanager.CreateRepresentation(s, v)
>>> r.Representation
Property name= Representation value = 2
# So, what does 2 mean? Let's find out. Find the first domain for
# the Representation property. Usually, there is only one domain per
# property but you still have to use an iterator to get to it.
>>> r.Representation.NewDomainIterator().GetKey()
'enum'
# It is the domain called "enum". This domain gives us information
# about the enumeration values.
>>> d = r.Representation.GetDomain("enum")
# Iterate over the possible values and print them.
>>> for i in range(d.GetNumberOfEntries()):
      print d.GetEntryValue(i), d.GetEntryText(i)

0 Points
1 Wireframe
2 Surface
3 Outline
```

There are many other domain types but most are either not very useful or are difficult to update when used from Python. For more information, see http://www.paraview.org/ParaView3/Doc/Nightly/html/classvtkSMDomain.html.

## Source Proxies

Source proxies are proxies that represent pipeline objects (vtkAlgorithm[6]). They have special properties to connect them as well as special method to query the meta-data of their output.

To connect a source proxy, use one of its input properties. An input property can either accept another source proxy or an OutputPort object. An OutputPort object is a simple object that has two attributes: a source proxy and an output port index. Since most of the VTK algorithms have one output, OutputPort is usually not needed. Use it when you have to connect an algorithm to an output other than the first one.

```
>>> glyph.Input = elev
```

is equivalent to the following.

```
>>> glyph.Input = OutputPort(elev, 0)
```

To connect glyph to the second output of some filter, use the following

```
>>> glyph.Input = OutputPort(elev, 1)
```

Note that `GetData()` and associated methods of an InputProperty always return OutputPort objects.

```
>>> type(glyph.Input.GetData())
<class 'paraview.servermanager.OutputPort'>
```

The SourceProxy class provides several additional methods that are specific to pipelines (See vtkSMSourceProxy documentation for a full list).

- `UpdatePipelineInformation()`: This method calls `UpdateInformation()` on the VTK algorithm. It also calls `UpdatePropertyInformation()` to update any information properties.
- `UpdatePipeline()`: This method calls `Update()` on the VTK algorithm causing a pipeline execution if the pipeline changed. Another way of causing pipeline updates is to render. The render view updates all connected pipelines.
- `GetDataInformation()`: This method is used to obtain meta-data about one output. It is discussed further below.

There are two common ways of getting meta-data information from a proxy: information properties and DataInformation. Information properties are updated automatically every time `UpdatePropertyInformation()` and `UpdatePipelineInformation()` are called. All you have to do is read the data from the property as usual. To get a DataInformation object from a source proxy use `GetDataInformation(port=0)`. By default, this method returns data information for the first output. You can pass an optional port number

[6] For more information about VTK pipelines, see the VTK books: http://vtk.org/buy-books.php

to get information for another output. You can get detailed documentation on DataInformation by using `help()` and by reading online documentation for vtkPVDataInformation[7]. Here are the use of some common methods.

```
>>> di = glyph.GetDataInformation(0)
>>> di
<paraview.servermanager.DataInformation object at 0x2d0920d0>
>>> glyph.UpdatePipeline()
# Make sure to update the data information after updating the
# pipeline. This does not happen automatically
>>> di.Update()
# Get the data type.
>>> di.GetDataSetType()
0
>>> di.GetDataSetTypeAsString()
'vtkPolyData'
# Get information about point data. Use GetCellDataInformation() and
# GetFieldDataInformation() for cell and field data.
>>> pdi = di.GetPointDataInformation()
# We are now directly accessing the wrapper for a VTK class
>>> pdi.GetClassName()
'vtkPVDataSetAttributesInformation'
>>> pdi.GetNumberOfArrays()
1
# Get information for a point array
>>> ai = pdi.GetArrayInformation(0)
>>> ai.GetClassName()
'vtkPVArrayInformation'
>>> ai.GetName()
'Elevation'
>>> ai.GetNumberOfComponents()
1
>>> ai.GetComponentRange(0)
(0.0, 0.5)
```

When meta-data is not enough and you need access to the raw data, you can use `Fetch()` to bring it to the client side[8]. `Fetch()` has three modes:

- Append all of the data together and bring it to the client (only available for polygonal and unstructured datasets). **Note:** Do not do this if data is large otherwise the client will run out of memory.
- Bring data from a given process to the client.
- Use a reduction algorithm and bring its output to the client. For example, find the minimum value of an attribute.

---

[7] http://www.paraview.org/doc/nightly/html/classvtkPVDataInformation.html

[8] When running in built-in mode, there is a way of getting access to the output of an algorithm directly but it is too advanced to cover here.

Here is a demonstration.

```
>>> from paraview import servermanager
>>> servermanager.Connect("kamino")
Connection (kamino:11111)
>>> s = servermanager.sources.SphereSource()
>>> s.UpdatePipeline()
# Get the whole sphere. DO NOT DO THIS IF THE DATA IS LARGE otherwise
# the client will run out of memory.
>>> allsphere = servermanager.Fetch(s)
getting appended
use append poly data filter
>>> allsphere.GetNumberOfPolys()
96
# Get the piece of the sphere on process 0.
>>> onesphere = servermanager.Fetch(s, 0)
getting node 0
>>> onesphere.GetNumberOfPolys()
48
# Apply the elevation filter so that we have a useful scalar array.
>>> elev = servermanager.filters.ElevationFilter(Input=s)
>>> elev.UpdatePipeline()
# We will use the MinMax algorithm to compute the minimum value of
# elevation. MinMax will be first applied on each processor. The results
# will then be gathered to the first node. MinMax will be then applied
# to the gathered results.
>>> mm = servermanager.filters.MinMax()
# How do I make it compute minimum? I think the property is called
# Operation
>>> d = mm.Operation.NewDomainIterator().GetDomain()
>>> d.GetNumberOfStrings()
3
>>> for i in range(3):
      print d.GetString(i)

MIN
MAX
SUM
>>> mm.Operation = "MIN"
# Get the minimum
>>> mindata = servermanager.Fetch(elev, mm, mm)
applying operation
# The result is a vtkPolyData with one point
>>> mindata.GetPointData().GetNumberOfArrays()
2
>>> a0 = mindata.GetPointData().GetArray(1)
>>> a0.GetName()
'Elevation'
>>> a1.GetTuple1(0)
0.0
```

## Representations and Views

Once a pipeline is created, it can be rendered using representations and views. A view is essentially a "window" in which multiple representations can be displayed. When the view is a VTK view (such as RenderView), this corresponds to a collection of objects including vtkRenderers and a vtkRenderWindow. However, there is no requirement for a view to be a VTK view or to render anything. A representation is a collection of objects, usually a pipeline, that takes a data object, converts it to something that can be rendered and renders it. When the view is a VTK view, this corresponds to a collection of objects including geometry filters, level-of-detail algorithms, vtkMappers and vtkActors.

Using the servermanager module, it is easy to create default representations and views.

```
>>> view = servermanager.CreateRenderView()
>>> rep = servermanager.CreateRepresentation(glyph, view)
```

`CreateRenderView()` is a special method that creates the render view appropriate for the `ActiveConnection` (or for another connection specified as an argument). It returns a sub-class of Proxy. Like the constructor of Proxy, it can take an arbitrary number of keyword arguments to set initial values for properties.

Once you have a render view, you can use `CreateRepresentation()` to create a default representation appropriate for that view. Under the cover, this function uses `CreateDefaultRepresentation()` defined by vtkSMViewProxy.

Although it is possible to create representations and views directly by instantiating their class objects, we recommend not doing it. Not all views work with all connections types and not all representations work with all view types.

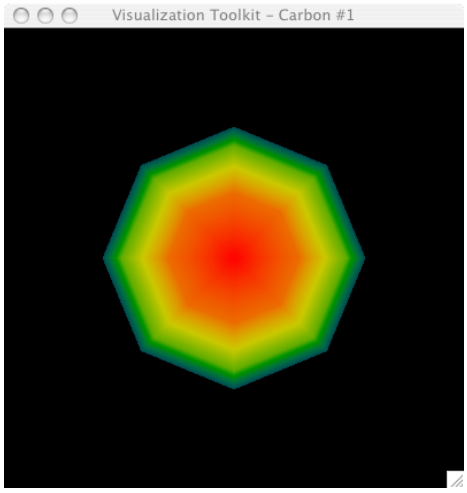
Representations and views have a large number of unfortunately poorly documented methods. We will cover some them here.

```
>>> from paraview import servermanager
# Create a simple pipeline
>>> sph = servermanager.sources.SphereSource()
>>> elev = servermanager.filters.ElevationFilter(Input=sph)
# Create representation, view
>>> view = servermanager.CreateRenderView()
>>> rep = servermanager.CreateRepresentation(elev, view)
>>> view.StillRender()
>>> view.ResetCamera()
>>> view.StillRender()
# Look at the domain for the Representation property to get a
# list of possible representation values.
>>> domain = rep.Representation.NewDomainIterator().GetDomain()
>>> for i in range(domain.GetNumberOfEntries()):
      print domain.GetEntryValue(i), ":", domain.GetEntryText(i)

0 : Points
1 : Wireframe
2 : Surface
3 : Outline
```

```
# Change the representation to points
>>> rep.Representation = 0
>>> view.StillRender()
# Change the representation to wireframe
>>> rep.Representation = 1
>>> view.StillRender()
# Change the representation to surface
>>> rep.Representation = 2
>>> view.StillRender()
# Let's get some information about the output of the elevation
# filter. We want to color the representation by one of it's
# arrays.
>>> di = elev.GetDataInformation()
# First array = Normals. Boring.
>>> ai = di.GetPointDataInformation().GetArrayInformation(0)
>>> ai.GetName()
'Normals'
# Second array = Elevation. Interesting. Let's use this one.
>>> ai = di.GetPointDataInformation().GetArrayInformation(1)
>>> ai.GetName()
'Elevation'
# What is its range?
>>> ai.GetComponentRange(0)
(0.0, 0.5)
# To color the representation by an array, we need to first create
# a lookup table.
>>> lt = servermanager.rendering.PVLookupTable()
>>> rep.LookupTable = lt
>>> rep.ColorAttributeType = 0 # point data
>>> rep.ColorArrayName = 'Elevation' # color by Elevation
# The lookup table is used to map scalar values to colors. The
# values in the lookup table have 4 numbers: 1 scalar value, 3 color
# values (rgb). Thisparticular table has 2 values, 0 = (0, 0, 1) (blue)
# and 0.5 = (1, 0, 0) (red). The colors in between are interpolated.
>>> lt.RGBPoints = [0, 0, 0, 1, 0.5, 1, 0, 0]
# Use the HSV color space to interpolate
>>> lt.ColorSpace = 1 # HSV
>>> view.StillRender()
```

Here is the result:



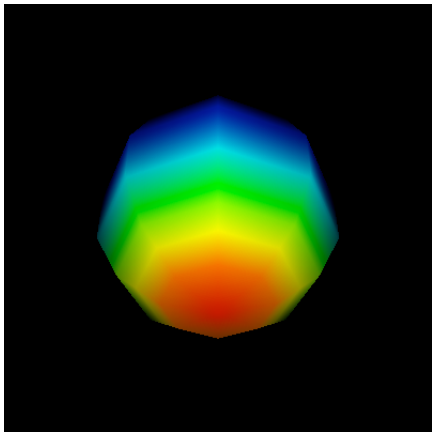

Once you create a scene, you will probably want to interact with the camera and `ResetCamera()` is likely to be insufficient. In this case, you can directly get the camera from the view and manipulate it. `GetActiveCamera()` returns a VTK object (not a proxy) with which you can interact.

```
>>> camera = view.GetActiveCamera()
>>> camera
<libvtkCommonPython.vtkCamera vtkobject at 0xe290>
>>> camera.Elevation(45)
>>> view.StillRender()
```

Another common thing to do is to save the view as an image. For this purpose, you can use the `WriteImage()` method provided by the view:

```
>> view.WriteImage("/Users/berk/image.png", "vtkPNGWriter", 1)
0
```

The resulting image.png looks like this.



The last, integer argument to `WriteImage()` controls the magnification factor. The dimensions of the saved image are magnification factor times the dimensions of the render view. This argument is optional.

# *Proxy Manager and Interaction with GUI*

So far, we only talked about Python scripts in isolation and did not clarify how you can interact with the user interface (i.e. notify the application that you created a proxy or access proxies created using the user interface). For this, we need to talk about the Proxy Manager. The Proxy Manager is a helper class used to create and manage proxies. We have already seen how it can be used to create proxies.

```
>>> pxm = servermanager.ProxyManager()
>>> sphereSourceP = pxm.SMProxyManager.NewProxy("sources", "SphereSource")
>>> type(sphereSourceP)
<type 'vtkobject'>
>>> sphereSourceP.GetClassName()
'vtkSMSourceProxy'
```

Under the cover, Proxy's `__init__()` calls `NewProxy()` to create the VTK proxy. Another important role of the Proxy Manager is to manage instances of proxies in different groups to allow easy access to them. Like Proxy and Property classes, servermanager defines a Python-friend wrapper around the Server Manager Proxy Manager (vtkSMProxyManager). This class is called ProxyManager. Although you can create as many ProxyManager instances you want, there is only one instance of vtkSMProxyManager underneath[9].

## Registering Proxies

You can register proxies under any arbitrary group using `RegisterProxy()`.

```
>>> sph=servermanager.sources.SphereSource()
>>> print sph
<paraview.servermanager.SphereSource object at 0x4d2fad30>
>>> pm=servermanager.ProxyManager()
>>> pm.RegisterProxy("foo", "bar", sph)
```

In this example, we registered the sphere source proxy under the name "bar" in the group "foo". We can verify that the proxy is registered.

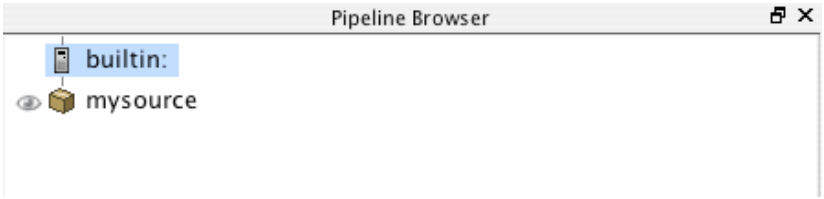
```
>>> print pm.GetProxy("foo", "bar")
<paraview.servermanager.SphereSource object at 0x4d2fad30>
```

If you are typing this script in the Python shell of the paraview application, you should see no change in the pipeline browser yet. The paraview application keeps the proxies registered under several predetermined groups and will not respond to the creation of arbitrary groups. Now try this:

```
>>> pm.RegisterProxy("sources", "mysource", sph)
```

You should now see something like this.

[9] Singleton design pattern




Note that the visibility of the new source is off. This is because there is no representation associated with it. You can simply click on the eye and have paraview create one or you can create on yourself. However, this requires a view. How do we get a proxy to the active view? We will learn in the next section. Let's create and register an elevation filter next:

```
>>> elev=servermanager.filters.ElevationFilter()
>>> elev.Input=sph
>>> pm.RegisterProxy("sources", "elev", elev)
```

Note: All algorithms (even filters) are registered in the "sources" group.

## Finding Registered Proxies

If you know the group and registered name of the proxy, it is easy to get it:

```
>>> print pm.GetProxy("sources", "elev")
<paraview.servermanager.ElevationFilter object at 0x4f4373b0>
```

This is the easiest way of getting access to an algorithm proxy created using the user interface since it is registered under the group "sources" using the name shown in the pipeline browser.

If you only know the group the proxy is registered in, you can get all proxies registered in a group by using introspection.

```
>>> print pm.GetProxiesInGroup("sources")
{'mysource': <paraview.servermanager.SphereSource object at 0x4d2fad30>,
'elev': <paraview.servermanager.ElevationFilter object at 0x4f4373b0>}
```

Note that `GetProxiesInGroup()` returns a dictionary that uses the registration name as the key. We can use this method to get our view.

```
>>> print pm.GetProxiesInGroup("views")
{'RenderView1': <paraview.servermanager.RenderView object at 0x4f4376b0>}
```

Here, I made use of the fact that views are registered under "views". Now we can create a representation for our source.

```
>>> view = pm.GetProxy("views", "RenderView1")
>>> print view
<paraview.servermanager.RenderView object at 0x4f437610>
>>> rep = servermanager.CreateRepresentation(sph, view)
>>> print rep
<paraview.servermanager.GeometryRepresentation object at 0x4f4376b0>
>>> pm.RegisterProxy("representations", "myrep1", rep)
```

To make sure that the user interface is notified of this representation, we registered it under the group "representations".

If you do not know the group or the name of the proxy, you can iterate over all proxies and search for the one you are interested in.

```
>>> for proxy in pm:
...   print proxy
```

The methods covered here are enough to accomplish most tasks. For information on other methods, see the ProxyManager, ProxyIterator and vtkSMProxyIterator[10] documentation.

# *Advanced Concepts*

## Loading State and Manipulating It

Let's say you created a complicated visualization using the paraview application and now you want to make slight changes to it and run it in a loop as a batch script. What do you do? The best way of dealing with this is to save your visualization state and then load it from Python. Let's say you have a state file saved as myteststate.pvsm:

```
>>> from paraview import servermanager
# Connect to a server running on the localhost
>>> servermanager.Connect("localhost")
Connection (localhost:11111)
# Load the state
>>> servermanager.LoadState("/Users/berk/myteststate.pvsm")
# Use the utility method to get the first render view. We could have
# looked in the "views" group instead.
>>> view = servermanager.GetRenderView()
# Render the scene
>>> view.StillRender()
```

Once the state is loaded, you can access the new proxies using the Proxy Manager and the introspection methods described in the previous section.

You can also save state.

```
>>> from paraview import servermanager
>>> servermanager.Connect()
Connection (builtin:5)
>>> sph = servermanager.sources.SphereSource()
>>> view = servermanager.CreateRenderView()
>>> rep = servermanager.CreateRepresentation(sph, view)
>>> view.StillRender()
>>> view.ResetCamera()
>>> view.StillRender()
# Make sure proxies are registered; otherwise they won't show
# up in the state.
```

[10] http://www.paraview.org/ParaView3/Doc/Nightly/html/classvtkSMProxyIterator.html

```
>>> pm = servermanager.ProxyManager()
>>> pm.RegisterProxy("sources", "sph", sph)
>>> pm.RegisterProxy("representations", "d1", rep)
>>> pm.RegisterProxy("views", "v1", view)
>>> servermanager.SaveState("/Users/berk/pythonstate.pvsm")
```

## Dealing with Time

If a reader or a filter supports time, it is easy to request a certain time step from Python. All time requests are set on views, which then propagate them to the representations which then propagate them to the visualization pipeline. Here is an example demonstrating how a time request can be made.

```
>>> reader = servermanager.sources.ExodusIIReader(FileName=".../can.ex2")
>>> view = servermanager.CreateRenderView()
>>> rep = servermanager.CreateRepresentation(reader, view)
>>> view.StillRender()
>>> view.ResetCamera()
>>> view.StillRender()
# Get a nice view angle
>>> cam=view.GetActiveCamera()
>>> cam.Elevation(45)
>>> view.StillRender()
# Check the current view time
>>> view.ViewTime
Property name= ViewTime value = 0.0
# Update the information properties. We want to know which
# time values the reader has
>>> reader.UpdatePropertyInformation()
>>> reader.TimestepValues
Property name= TimestepValues value = [0.0, 0.00010007373930420727,
0.00019990510190837085, 0.00029996439116075635, 0.00040008654468692839,
...]
>>> tsteps = reader.TimestepValues
# Let's be fancy and use a time annotation filter. This will show the
# current time value of the reader as text in the corner of the view.
>>> annTime = servermanager.filters.TimeToTextConvertor(Input=reader)
>>> timeRep =
servermanager.rendering.TextSourceRepresentation(Input=annTime)
>>> view.Representations.append(timeRep)
# Make the text a bit bigger
>>> timeRep.Position2 = [0.2, 0.2]
# Look at a few time steps. Note that the time value is requested not
# the time step index.
>>> view.ViewTime = tsteps[2]
>>> view.StillRender()
>>> view.ViewTime = tsteps[4]
>>> view.StillRender()
```

## Animating

Server Manager has a complicated animation engine based on keyframes and scenes. This section will introduce a few simple ways of animating your visualization.

If you have a time-aware reader, you can animate it with `AnimateReader()`.

```
reader = sources.ExodusIIReader("../can.ex2")
view = CreateRenderView()
repr = CreateRepresentation(reader, view)
view.StillRender();
view.ResetCamera();
view.StillRender();
c = view.GetActiveCamera()
c.Elevation(95)
# Animate over all time steps. Note that we are not passing the optional
# 3rd argument here. If you pass a filename as the 3rd argument,
# AnimateReader will create a movie.
AnimateReader(reader, view)
```

`AnimateReader()` is a utility function that creates scene and cue proxies. Its implementation looks like this.

```
# Create an animation scene.
scene = animation.AnimationScene();
# Add one view.
scene.ViewModules = [view];
# Update the reader to get the time information
reader.UpdatePipelineInformation()
scene.TimeSteps = reader.TimestepValues.GetData()
# Animate from 1st time step to last
scene.StartTime = reader.TimestepValues.GetData()[0]
scene.EndTime = reader.TimestepValues.GetData()[-1];

# Each frame will correspond to a time step
scene.PlayMode = 2; #Snap To Timesteps

# Create a special animation cue for time.
cue = animation.TimeAnimationCue();
cue.AnimatedProxy = view;
cue.AnimatedPropertyName = "ViewTime";
scene.Cues = [cue];

if filename:
    writer = vtkSMAnimationSceneImageWriter();
    writer.SetFileName(filename);
    writer.SetFrameRate(1);
    writer.SetAnimationScene(scene.SMProxy);

    # Now save the animation.
    if not writer.Save():
        raise exceptions.RuntimeError, "Saving of animation failed!"
```

```
else:
    scene.Play()
```

To animate properties other than time, you can use regular keyframes. From `demo5()` in servermanager.

```
# Create an animation scene
scene = animation.AnimationScene();
# Add one view
scene.ViewModules = [view];

# Create a cue to animate the StartTheta property
cue = animation.KeyFrameAnimationCue();
cue.AnimatedProxy = sphere;
cue.AnimatedPropertyName = "StartTheta";
# Add it to the scene's cues
scene.Cues = [cue];

# Create 2 keyframes for the StartTheta track
keyf0 = animation.CompositeKeyFrame();
keyf0.Type = 2 ;# Set keyframe interpolation type to Ramp.
# At time = 0, value = 0
keyf0.KeyTime = 0;
keyf0.KeyValues= [0];

keyf1 = animation.CompositeKeyFrame();
# At time = 1.0, value = 200
keyf1.KeyTime = 1.0;
keyf1.KeyValues= [200];

# Add keyframes.
cue.KeyFrames = [keyf0, keyf1];

scene.Play()
```

You can change the settings of the animation.

```
# PlayMode controls the mode in which the animation is played
>>> d = scene.PlayMode.NewDomainIterator().GetDomain()
>>> d
<libvtkPVServerManagerPython.vtkSMEnumerationDomain vtkobject at 0xe6b0>
>>> for i in range(d.GetNumberOfEntries()):
      print d.GetEntryValue(i), d.GetEntryText(i)

0 Sequence  # Play a given number of frames sequentially, by linearly
            # changing time between StartTime and EndTime
1 Real Time # Given a duration in seconds, play the animation.
            # The time is scaled to vary between StartTime and EndTime.
2 Snap To TimeSteps # Play mode in which each frame corresponds to a time
                    # step. Time may vary non-linearly.
>>> scene.PlayMode
Property name= PlayMode value = 0
```

```
>>> scene.NumberOfFrames
Property name= NumberOfFrames value = 10
# Change the number of frames to 100 and play
>>> scene.NumberOfFrames = 100
>>> scene.Play()
# Switch to real time mode and play a 20-second animation
>>> scene.PlayMode = 1
>>> scene.Duration
Property name= Duration value = 10
>>> scene.Duration = 20
>>> scene.Play()
```

## Loading Plugins

Plugins can be loaded using `LoadPlugin()`. This method will load shared libraries as necessary and import xml configurations if any. Proxies defined under sources, filters, look-up_tables, representations and views group are added to appropriate sub-modes (sources, filters and rendering). For more information on how to create plugins, see the The ParaView Guide or http://paraview.org/Wiki/Plugin_HowTo[11].

```
# Load the example plugin provided with ParaView
>>> servermanager.LoadPlugin(".../libSMMyElevation.dylib")
# MyElevationFilter is defined by the plugin
>>> myelev = servermanager.filters.MyElevationFilter()
```

[11] GUI plugins are not supported by the servermanager module.