Chapter 4

Batch Python Scripting

Python scripting can be leveraged in two ways within ParaView. First, Python scripts can automate the setup and execution of visualizations by performing the same actions as a user at the GUI. Second, Python scripts can be run inside pipeline objects, thereby performing parallel visualization algorithms. This chapter describes the first mode, batch scripting for automating the visualization.

Batch scripting is a good way to automate mundane or repetitive tasks, but it is also a critical component when using ParaView in situations where the GUI is undesired or unavailable. The automation of Python scripts allows you to leverage ParaView as a scalable parallel post-processing framework. We are also leveraging Python scripting to establish *in situ* computation within simulation code. (ParaView supports an *in situ* library called **Catalyst**, which is not documented in this tutorial. See http://catalyst.paraview.org/ for more information on Catalyst).

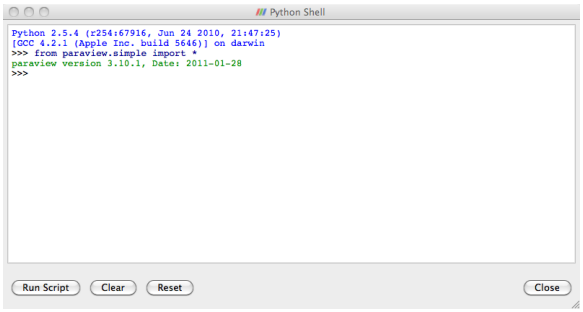
This tutorial gives only a brief introduction to Python scripting. The most recent and complete documentation is kept on the ParaView Wiki.

http://www.paraview.org/Wiki/ParaView/Python_Scripting

4.1 Starting the Python Interpreter

There are many ways to invoke the Python interpreter. The method you use depends on how you are using the scripting. The easiest way to get a python interpreter, and the method we use in this tutorial, is to select Tools → Python Shell from the menu. This will bring up a dialog box containing

controls for ParaView's Python shell. This is the Python interpreter, where you directly control ParaView via the interface described below.



If you are most interested in getting started on writing scripts, feel free to skip to the next section past the discussion of the other ways to invoke scripting.

ParaView comes with two command line programs that execute Python scripts: `pvpython` and `pvbatch`. They are similar to the `python` executable that comes with Python distributions in that they accept Python scripts either from the command line or from a file and they feed the scripts to the Python interpreter.

The difference between `pvpython` and `pvbatch` is subtle and has to do with the way they establish the visualization service. `pvpython` is roughly equivalent to the `paraview` client GUI with the GUI replaced with the Python interpreter. It is a serial application that connects to a ParaView server (which can be either builtin or remote). `pvbatch` is roughly equivalent to `pvserver` except that commands are taken from a Python script rather than from a socket connection to a ParaView client. It is a parallel application that can be launched with `mpirun` (assuming it was compiled with MPI), but it cannot connect to another server; it is its own server. In general, you should use `pvpython` if you will be using the interpreter interactively and `pvbatch` if you are running in parallel.

It is also possible to use the ParaView Python modules from programs outside of ParaView. This can be done by pointing the `PYTHONPATH` environment variable to the location of the ParaView libraries and Python modules and pointing the `LD_LIBRARY_PATH` (on Unix/Linux/Mac) or `PATH` (on Windows) environment variable to the ParaView libraries. Running the Python script this way allows you to take advantage of third-party applications such as IDLE. For more information on setting up your environment, consult the ParaView Wiki.

4.2 Tracing ParaView State

Before diving into the depths of the Python scripting features, let us take a moment to explore the automated facilities for creating Python scripts. The ParaView GUI's Python Trace feature allows one to very easily create Python scripts for many common tasks. To use Trace, one simply begins a trace recording via Start Trace, found in the Tools Menu, and ends a trace recording via Stop Trace, also found in the Tools Menu. This produces a Python script that reconstructs the actions taken in the GUI. That script contains the same set of operations that we are about to describe. As such, Trace recordings are a good resource when you are trying to figure out how to do some action via the Python interface, and conversely the following descriptions will help in understanding the contents of any given Trace script.

Exercise 4.1: Creating a Python Script Trace

If you have been following an exercise in a previous section, now is a good time to reset ParaView. The easiest way to do this is to press the  button.

1. Click the Start Trace in the Tool menu.
2. Build a simple pipeline in the main ParaView GUI. For example, create a sphere source and then clip it.
3. Click Stop Trace in the Tools meu.
4. An editing window will open populated with a Python script that replicates the operations you just made.

Even if you have not been exposed to ParaView's Python bindings, the commands being performed in the traced script should be familiar. Once saved to your hard drive, you can of course edit the script with your favorite editor. The final script can be interpreted by the `pvpython` or `pvbatch` program for totally automated visualization. It is also possible to run this script in the GUI. The Python Shell dialog has a Run Script button that invokes a saved script. ♦

It should be noted that there is also a way to capture the current ParaView state as a Python script without tracing actions. Simply select Save State... from the ParaView File menu and choose to save as a Python .py

state file (as opposed to a ParaView .pvsm state file). We will not have an exercise on state Python scripts, but suffice it to say they can be used in much the same way as traced Python scripts. You are welcome to experiment with this feature as you like.

4.3 Macros

A simple but powerful way to customize the behavior of ParaView is to add your Python script as a **macro**. A macro is simply an automated script that can be invoked through its button in a toolbar or its entry in the menu bar. Any Python script can be assigned to a macro.

Exercise 4.2: Adding a Macro

This exercise is a continuation of Exercise 4.1. You will need to finish that exercise before beginning this one. You should have the editing window containing the Python script created in Exercise 4.1 open.

1. In the menu bar (of the editing window), select File → Save As Macro....

2. Choose a descriptive name for the macro file and save it in the default directory provided by the browser. You should now see your macro on the Macro toolbar at the top of the ParaView GUI.

At this point, you should now see your macro added to the toolbars. By default, macro toolbar buttons are placed in the middle row all the way to the left. If you are short on space in your GUI, you may need to move toolbars around to see it. You will also see that your macro has been added to the Macros menu.

3. Close the Python editor window.

4. Delete the pipeline you have created by either selecting Edit → Delete All from the menu or pressing the  button.

5. Activate your macro by clicking on the toolbar button or selecting it in the Macros menu.

In this example our macro created something from scratch. This is helpful if you often load some data in the same way every time. You can also trace the creation of filters that are applied to existing data. A macro from a trace of this nature allows you to automate the same visualization on different data. ♦

4.4 Creating a Pipeline

As described in the previous two sections, the ParaView GUI's Python Trace feature provides a simple mechanism to create scripts. In this section we will begin to describe the basic bindings for ParaView scripting. This is important information in building Python scripts, but you can always fall back on producing traces with the GUI.

The first thing any ParaView Python script must do is load the `paraview.simple` module. This is done by invoking

```
from paraview.simple import *
```

In general, this command needs to be invoked at the beginning of any ParaView batch Python script. This command is automatically invoked for you when you bring up the scripting dialog in ParaView, but you must add it yourself when using the Python interpreter in other programs (including `pvpython` and `pvbatch`).

The `paraview.simple` module defines a function for every source, reader, filter, and writer defined in ParaView. The function will be the same name as shown in the GUI menus with spaces and special characters removed. For example, the `Sphere` function corresponds to Sources → Sphere in the GUI and the `PlotOverLine` function corresponds to Filters → Data Analysis → Plot Over Line. Each function creates a pipeline object, which will show up in the pipeline browser (with the exception of writers), and returns an object that is a **proxy** that can be used to query and manipulate the properties of that pipeline object.

There are also several other functions in the `paraview.simple` module that perform other manipulations. For example, the pair of functions `Show` and `Hide` turn on and off, respectively, the visibility of a pipeline object in a view. The `Render` function causes a view to be redrawn.

Exercise 4.3: Creating and Showing a Source

If you have been following an exercise in a previous section, now is a good time to reset ParaView. The easiest way to do this is to press the  button.

If you have not already done so, open the Python shell in the ParaView GUI by selecting Tools → Python Shell from the menu. You will notice that

```
from paraview.simple import *
```

has been added for you.

Create and show a Sphere source by typing the following in the Python shell.

```
sphere = Sphere()
Show()
Render()
```

The `Sphere` command creates a sphere pipeline object. Once it is executed you will see an item in the pipeline browser created. We save a proxy to the pipeline object in the variable `sphere`. We are not using this variable (yet), but it is good practice to save references to your pipeline objects.

The subsequent `Show` command turns on visibility of this object in the view, and the subsequent `Render` causes the results to be seen. At this point you can interact directly with the GUI again. Try changing the camera angle in the view with the mouse. ♦

Exercise 4.4: Creating and Showing a Filter

Creating filters is almost identical to creating sources. By default, the last created pipeline object will be set as the input to the newly created filter, much like when creating filters in the GUI.

This exercise is a continuation of Exercise 4.3. You will need to finish that exercise before beginning this one.

Type in the following script in the Python shell that hides the sphere and then adds the shrink filter to the sphere and shows that.

```
Hide()
shrink = Shrink()
Show()
Render()
```

The sphere should be replaced with the output of the `Shrink` filter, which makes all of the polygons smaller to give the mesh an exploded type of appearance. ♦

So far as we have built pipelines we have accepted the default parameters for the pipeline objects. As we have seen in the exercises of Chapter 2, it is common to have to modify the parameters of the objects using the properties panel.

In Python scripting, we use the **proxy** returned from the creation functions to manipulate the pipeline objects. These proxies are in fact Python objects with class attributes that correspond to the same properties you set in the properties panel. They have the same names as those in the properties panel with spaces and other illegal characters removed. You can set them by simply assigning them a value.

Exercise 4.5: Changing Pipeline Object Properties

This exercise is a continuation of Exercises 4.3 and 4.4. You will need to finish those exercises before beginning this one.

Recall that we have so far created two Python variables, `sphere` and `shrink`, that are proxies to the corresponding pipeline objects. First, enter the following command into the Python shell to get the current value of the Theta Resolution property of the sphere.

```
print sphere.ThetaResolution
```

The Python interpreter should respond with the result `8`. (Note that using the `print` keyword, which instructs Python to output the arguments to standard out, is superfluous here as the Python shell will output the result of any command anyway.) Let us double the number of polygons around the equator of the sphere by changing this property.

```
sphere.ThetaResolution = 16
Render()
```

The shrink filter has only one property, Shrink Factor. We can adjust this factor to make the size of the polygons larger or smaller. Let us change the factor to make the polygons smaller.

```
shrink.ShrinkFactor = 0.25
Render()
```

You may have noticed that as you type in Python commands to change the pipeline object properties, the GUI in the properties panel updates accordingly. ⧫

So far we have created only non-branching pipelines. This is a simple and common case and, like many other things in the `paraview.simple` module, is designed to minimize the amount of work for the simple and common case but also provide a clear path to the more complicated cases. As we have built the non-branching pipeline, ParaView has automatically connected the filter input to the previously created object so that the script reads like the sequence of operations it is. However, if the pipeline has branching, we need to be more specific about the filter inputs.

Exercise 4.6: Branching Pipelines

This exercise is a continuation of Exercises 4.3 through 4.5. You will need to finish Exercises 4.3 and 4.4 before beginning this one (Exercise 4.5 is optional).

Recall that we have so far created two Python variables, `sphere` and `shrink`, that are proxies to the corresponding pipeline objects. We will now add a second filter to the sphere source that will extract the wireframe from it. Enter the following in the Python shell.

```
wireframe = ExtractEdges(Input=sphere)
Show()
Render()
```

An Extract Edges filter is added to the sphere source. You should now see both the wireframe of the original sphere and the shrunken polygons.

Notice that we explicitly set the input for the Extract Edges filter by providing `Input=sphere` as an argument to the `ExtractEdges` function. What we are really doing is setting the `Input` property upon construction of the object. Although it would be possible to create the object with the default input, and then set the input later, it is not recommended. The problem is that not all filters accept all input. If you initially create a filter with the

wrong input, you could get error messages before you get a chance to change the `Input` property to the correct input.

The sphere source having two filters connected to its output is an example of **fan out** in the pipeline. It is always possible to have multiple filters attached to a single output. Some filters, but not all, also support having multiple filters connected to their input. Multiple filters are attached to an input is known as **fan in**. In ParaView's Python scripting, fan in is handled much like fan out, by explicitly defining a filter's inputs. When setting multiple inputs (on a single port), simply set the input to a list of pipeline objects rather than a single one. For example, let us group the results of the shrink and extract edges filters using the Group Datasets filter. Type the following line in the Python shell.

```
group = GroupDatasets(Input=[shrink,wireframe])
Show()
```

There is now no longer any reason for showing the shrink and extract edges filters, so let us hide them. By default, the `Show` and `Hide` functions operate on the last pipeline object created (much like the default input when creating a filter), but you can explicitly choose the object by giving it as an argument. To hide the shrink and extract edges filters, type the following in the Python shell.

```
Hide(shrink)
Hide(wireframe)
Render()
```

♦

In the previous exercise, we saw that we could set the `Input` property by placing `Input=`⟨*input object*⟩ in the arguments of the creator function. In general we can set any of the properties at object construction by specifying ⟨*property name*⟩=⟨*property value*⟩. For example, we can set both the Theta Resolution and Phi Resolution when we create a sphere with a line like this.

```
sphere = Sphere(ThetaResolution=360, PhiResolution=180)
```

4.5 Active Objects

If you have any experience with the ParaView GUI, then you should already be familiar with the concept of an active object. As you build and manipulate visualizations within the GUI, you first have to select an object in the pipeline browser. Other GUI panels such as the properties panel will change based on what the active object is. The active object is also used as the default object to use for some operations such as adding a filter.

The batch Python scripting also understands the concept of the active object. In fact, when running together, the GUI and the Python interpreter share the same active object. When you created filters in the previous section, the default input they were given was actually the active object. When you created a new pipeline object, that new object became the active one (just like when you create an object in the GUI).

You can get and set the active object with the `GetActiveSource` and `SetActiveSource` functions, respectively. You can also get a list of all pipeline objects with the `GetSources` function. When you click on a new object in the GUI pipeline browser, the active object in Python will change. Likewise, if you call `SetActiveSource` in python, you will see the corresponding entry become highlighted in the pipeline browser.

Exercise 4.7: Experiment with Active Pipeline Objects

This exercise is a continuation of the exercises in the previous section. However, if you prefer you can create any pipeline you want and follow along.

Play with active objects by trying the following.

- Get a list of objects by calling `GetSources()`. Find the sources and filters you created in that list.
- Get the active object by calling `GetActiveSource()`. Compare that to what is selected in the pipeline browser.
- Select something new in the pipeline browser and call `GetActiveSource()` again.
- Change the active object with the `SetActiveSource` function. Observe the change in the pipeline browser.

♦

In addition to maintaining an active pipeline object, ParaView Python scripting also maintains an active view. As a ParaView user, you should also already be familiar with multiple views and the active view. The active view is marked in the GUI with a blue border. The Python functions `GetActiveView` and `SetActiveView` allow you to query and change the active view. As with pipeline objects, the active view is synchronized between the GUI and the Python interpreter.

4.6 Online Help

This tutorial, as well as similar instructions in the ParaView book and Wiki, is designed to give the key concepts necessary to understand and create batch Python scripts. The detailed documentation including complete lists of functions, classes, and properties available is maintained by the ParaView build process and provided as online help from within the ParaView application. In this way we can ensure that the documentation is up to date for whatever version of ParaView you are using and that it is easily accessible.

The ParaView Python bindings make use of the `help` built-in function. This function takes as an argument any Python object and returns some documentation on it. For example, typing

```
help(paraview.simple)
```

returns a brief description of the module and then a list of all the functions included in the module with a brief synopsis of what each one does. For example

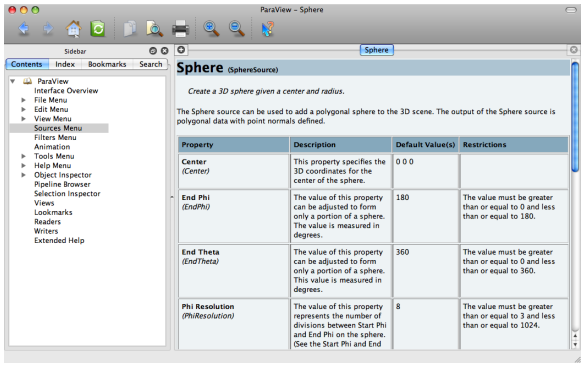
```
help(Sphere)
sphere = Sphere()
help(sphere)
```

will first give help on the `Sphere` function, then use it to create an object, and then give help on the object that was returned (including a list of all the properties for the proxy).

Most of the widgets displayed in the properties panel's Properties group are automatically generated from the same introspection that builds the Python classes. (There are a small number of exceptions where a custom

panel was created for better usability.) Thus, if you see a labeled widget in the properties panel, there is a good chance that there is a corresponding property in the Python object with the same name.

Regardless of whether the GUI contains a custom panel for a pipeline object, you can still get information about that object's properties from the GUI's online help. As always, bring up the help with the ❓ toolbar button. You can find documentation for all the available pipeline objects under the Sources Menu, Filters Menu, Readers, and Writers entries in the help Contents. Each entry gives a list of objects of that type. Clicking on any one of the objects gives a list of the properties you can set from within Python.



4.7 Reading from Files

The equivalent to opening a file in the ParaView GUI is to create a reader in Python scripting. Reader objects are created in much the same way as sources and filters; `paraview.simple` has a function for each reader type that creates the pipeline object and returns a proxy object. One can instantatiate any given reader directly as described below, or more simply call `reader = OpenDataFile(`⟨*filename*⟩`)`

All reader objects have at least one property (hidden in the GUI) that specifies the file name. This property is conventionally called either `FileName` or `FileNames`. You should always specify a valid file name when creating a reader by placing something like `FileName=`⟨*full path*⟩ in the arguments of the construction object. Readers often do not initialize correctly if not given a valid file name.

Exercise 4.8: Creating a Reader

We are going to start a fresh visualization, so if you have been following along with the exercises so far, now is a good time to reset ParaView. The easiest way to do this is to press the  button. You will also need the Python shell. If you have not already done so, open it with Tools → Python Shell from the menu.

In this exercise we are loading the disk_out_ref.ex2 file from the Python shell. Locate this file on your computer and be ready to type or copy it into the Python shell. We will reference it as ⟨*path*⟩`/disk_out_ref.ex2`.

Create the reader while specifying the file name by entering the following in the Python shell.

```
reader = OpenDataFile('⟨path⟩/disk_out_ref.ex2')
Show()
Render()
```

♦

4.8 Querying Field Attributes

In addition to having properties specific to the class, all proxies for pipeline objects share a set of common properties and methods. Two very important such properties are the `PointData` and `CellData` properties. These properties act like **dictionaries**, an associative array type in Python, that maps variable names (in strings) to `ArrayInformation` objects that hold some characteristics of the fields. Of particular note are the `ArrayInformation` methods `GetName`, which returns the name of the field, `GetNumberOfComponents`, which returns the size of each field value (1 for scalars, more for vectors), and `GetRange`, which returns the minimum and maximum values for a particular component.

Exercise 4.9: Getting Field Information

This exercise is a continuation of Exercise 4.8. You will need to finish that exercise before beginning this one.

To start with, get a handle to the point data and print out all of the point fields available.

```
pd = reader.PointData
print pd.keys()
```

Get some information about the "Pres" and "V" fields.

```
print pd['Pres'].GetNumberOfComponents()
print pd['Pres'].GetRange()
print pd['V'].GetNumberOfComponents()
```

Now let us get fancy. Use the Python `for` construct to iterate over all of the arrays and print the ranges for all the components.

```
for ai in pd.values():
    print ai.GetName(), ai.GetNumberOfComponents(),
    for i in xrange(ai.GetNumberOfComponents()):
        print ai.GetRange(i),
    print
```

♦

4.9 Representations

Representations are the "glue" between the data in a pipeline object and a view. The representation is responsible for managing how a data set is drawn in the view. The representation defines and manages the underlying rendering objects used to draw the data as well as other rendering properties such as coloring and lighting. Parameters made available in the Display group of the GUI are managed by representations. There is a separate representation object instance for every pipeline-object–view pair. This is so that each view can display the data differently.

Representations are created automatically. You can get the proxy to the representation objects with the `GetRepresentation` function. With no arguments, this function will return the representation for the active pipeline object and the active view. You can also specify a pipeline object or view or both.

Exercise 4.10: Coloring Data

This exercise is a continuation of Exercise 4.8 (and optionally Exercise 4.9). If you do not have the exodus file open, you will need to finish that exercise before beginning this one.

Let us change the color of the geometry to blue and give it a very pronounced specular highlight (that is, make it really shiny). Type in the following into the Python shell to get the representation and change the material properties.

```
readerRep = GetRepresentation()
readerRep.DiffuseColor = [0, 0, 1]
readerRep.SpecularColor = [1, 1, 1]
readerRep.SpecularPower = 128
readerRep.Specular = 1
Render()
```

Now rotate the camera with the mouse in the GUI to see the effect of the specular highlighting.

We can also use the representation to color by a field variable. Enter the following into the Python shell to color the mesh by the "Pres" field variable.

```
readerRep.ColorArrayName = 'Pres'
readerRep.LookupTable = \
  AssignLookupTable(reader.PointData['Pres'], 'Cool to Warm')
Render()
```