Python scripting in ParaView

Standard Python scripting in ParaView

- Why use scripting?
  - automate mundane or repetitive tasks, e.g., making frames for a movie
  - document and store your workflow, perhaps share it with colleagues
  - use ParaView on clusters from the command line and/or via batch jobs

1. In the GUI: View | Python Shell opens a Python interpreter
   - write or paste your script there
   - use the button to run an external script from a file
2. */usr/local/bin/ /Applications/Paraview*.app/Contents/bin/ C:\Program Files\ParaView*\bin* pvpython will give you a Python shell connected to a ParaView server (local or remote) without the GUI
3. */usr/local/bin/ /Applications/Paraview*.app/Contents/bin/ C:\Program Files\ParaView*\bin* pvbatch --force-offscreen-rendering script.py is a serial (on some machines parallel) application using a local ParaView server ✎ make sure to save your visualization
4. */usr/local/bin/ /Applications/Paraview*.app/Contents/bin/ C:\Program Files\ParaView*\bin* paraview --script=codes/script.py to start ParaView GUI and auto-run the script

Less obvious ways to use Python scripting in ParaView

5. Python Calculator Filter (vs. regular Calculator Filter) to create new data arrays
   - treats data as NumPy arrays
   - can use short syntax `0.5*rho*mag(V)**2` or long syntax `0.5*inputs[0].PointData['rho']*inputs[0].PointData['V']**2`
   - can access data from multiple time steps, multiple inputs, point vs. cell data
   - can control array precision / type, but not the underlying discretization
   - I don't use it myself; I typically use the regular Calculator or the Programmable Filter
6. Programmable Source / Filter to create new geometry
   - to create new dicretizations / VTK objects
   - useful for creating custom objects from scratch, e.g. projecting a volume onto a plane
   - an alternative way to build your own custom file reader (if format not supported out of the box)
7. Python state files: more robust than `*.pvsm` files for complex workflows

First script

1. Bring up View | Python Shell
2. "Run Script" `codes/displaySphere.py`

displaySphere.py

```
from paraview.simple import *

sphere = Sphere()                # create a sphere pipeline object
print(sphere.ThetaResolution)    # print one of the attributes of the sphere
sphere.ThetaResolution = 16

Show()                           # make it visible in the view
Render()
```

3. Can always get help from the command line

```
help(paraview.simple)    # will display a help page on paraview.simple module
help(Sphere)
help(Show)
help(sphere)             # to see this object's attributes
dir(paraview.simple)     # to see everything inside this module's namespace
```

Using filters

1. Reset ParaView
2. "Run Script" `codes/displayWireframe.py`

displayWireframe.py

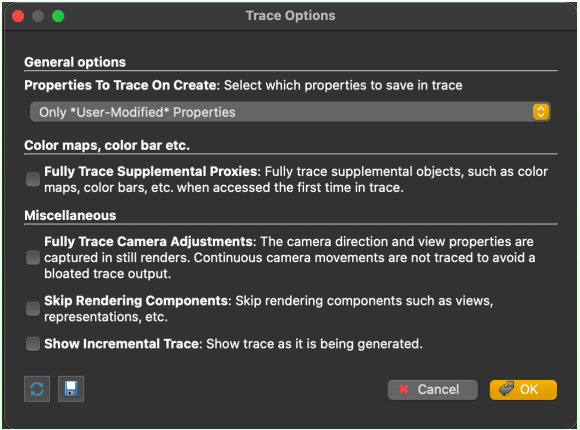
```
from paraview.simple import *

sphere = Sphere(ThetaResolution=36, PhiResolution=18)

wireframe = ExtractEdges(Input=sphere)    # apply Extract Edges to sphere

Show()                                    # make the last object visible in the view
Render()
```

3. Try replacing Show() with Show(sphere)
4. Also try replacing `Render()` with `SaveScreenshot('/path/to/wireframe.png')` and running via `pvbatch`

Trace tool

Generate Python code from GUI operations

- Tools | Start / Stop Trace



Exercise: write and run a complete off-screen script

1. Mac/Linux/Windows: create a script with standalone ParaView GUI
   - use Start/Stop Trace
   - load `data/sineEnvelope.nc` and draw an isosurface at $\rho = 0.15$
   - save the image as PNG
2. Test-run your script with `pvbatch` on your laptop

   ```
   $ pvbatch --force-offscreen-rendering script.py
   ```

   - Linux: `pvbatch` should be in one of your system's `bin` directories
   - Mac: `pvbatch` should be in `/Applications/ParaView*.app/Contents/bin`
   - Windows: `pvbatch` does not exist (or so I am told), but you can use `pvpython`
     ➠ you will need to locate it yourself
3. Those of you with an Alliance account can run this script on one of our HPC clusters with

   ```
   $ module load paraview
   $ pvbatch --force-offscreen-rendering script.py
   ```

   3.1 copy the script and the data file to the cluster
   3.2 edit the paths inside the script

Passing information down the pipeline

... and other useful high-level workflow functions

- `GetSources()` gets a list of pipeline objects
- `GetActiveSource()` gets the active object
- `SetActiveSource()` sets the active object
- `GetRepresentation()` returns the *view representation* for the active pipeline object and the active view
- `GetActiveCamera()` returns the active camera for the active view
- `GetActiveView()` returns the active view
- `CreateRenderView()` creates standard 3D render view
- `ResetCamera()` resets the camera to include the entire scene but preserve orientation (or does nothing ☺)

There is quite a bit of overlap between these two:

```
help(GetActiveCamera())
help(GetActiveView())
```

Camera animation with scripting

1. Let's load `data/sineEnvelope.nc` and draw an isosurface at $\rho = 0.15$
2. Ensure the focal point is at the dataset center to keep the object in view
   - in a 3D scene, the focal point is the specific coordinate in space that the camera is looking at and rotating around
   - if not $\Rightarrow$ `ResetCamera()`

```
v1 = GetActiveView()
print(v1.CameraFocalPoint)
```

3. Look up azimuthal rotation

```
camera = GetActiveCamera()
dir(camera)
help(camera.Azimuth)
```

4. Rotate by 10° around the view-up vector

```
camera.Azimuth(10)
Render()
```

Camera animation: full rotation

✎ Can paste longer commands from `clipboard.txt`

5. Do full rotation and save to disk

```
nframes = 360
v1 = GetActiveView()
camera = GetActiveCamera()
for i in range(nframes):
    print(v1.CameraPosition)
    camera.Azimuth(360./nframes)    # rotate by 1 degree
    SaveScreenshot('/path/to/frame%04d'%(i)+'.png')
```

6. Merge all frames into a movie at 30 fps

```
ffmpeg -r 30 -i frame%04d.png -c:v libx264 -pix_fmt yuv420p \
       -vf "scale=trunc(iw/2)*2:trunc(ih/2)*2" spin.mp4
```

Camera animation: flying towards the focal point

1. Optionally reset the view manually or with `ResetCamera()`
2. Now let's fly 2/3 of the way towards the focal point

```
initialCameraPosition = v1.CameraPosition[:]   # force a real copy
nframes = 100
for i in range(nframes):
    coef = float(i+0.5)/float(1.5*nframes)  # runs from 0 to 2/3
    print(coef, v1.CameraPosition)
    v1.CameraPosition = [((1.-coef)*a + coef*b) \
      for a, b in zip(initialCameraPosition,v1.CameraFocalPoint)]
    SaveScreenshot('/path/to/out%04d'%(i)+'.png')
```

3. Create a movie

```
ffmpeg -r 30 -i out%04d.png -c:v libx264 -pix_fmt yuv420p \
       -vf "scale=trunc(iw/2)*2:trunc(ih/2)*2" approach.mp4
```

Exercise: add animation to your off-screen script

Modify your script from the previous exercise to create some animation

- try a camera animation from the previous slides, all from the terminal
- try animating the contour's isovalue

Extracting data from VTK objects

Do this from *View | Python Shell* or from *pvpython* (either shell will work)

```
# codes/extractValues.py
from paraview.simple import *

dir = '/Users/razoumov/training/paraviewWorkshop/data/'   # edit the path
data = NetCDFReader(FileName=[dir+'sineEnvelope.nc'])
print(type(data))                  # class 'paraview.servermanager.NetCDFReader'

local = servermanager.Fetch(data) # get this data from the server
print(local.GetNumberOfPoints())
print(local.GetDimensions())

# the following are populated with data only after servermanager.Fetch(data) is called
pointInfo = data.GetDataInformation().GetPointDataInformation()
narrays = pointInfo.GetNumberOfArrays()
print(narrays)

for i in range(narrays):
    print(pointInfo.GetArrayInformation(i).GetName())

for i in range(10):
    print(local.GetPoint(i))       # coordinates of the first 10 points
```

Feeding data into arrays for further post-processing

```
vtkArray = local.GetPointData().GetArray('density')
print(vtkArray.GetDataSize(), vtkArray.GetRange())

for i in range(10):
    print(vtkArray.GetValue(i))        # values at the first 10 points -- awkward to use

from vtk.numpy_interface import dataset_adapter as dsa

wrapped = dsa.WrapDataObject(local)    # wrap it in a NumPy-friendly object
flatArray = wrapped.PointData['density']
print(flatArray.shape)
print(flatArray)

import numpy as np

dims = local.GetDimensions()
array3d = np.reshape(flatArray, (dims[2], dims[1], dims[0]))
print(array3d.shape)
print(array3d)
```

Using 3rd-party libraries from ParaView's Python

- `pvpython` includes few common 3rd-party libraries such as `numpy`, `scipy`, `pandas`
- What if you want to use other libraries that were not bundled with ParaView?

Using 3rd-party libraries from ParaView's Python

- `pvpython` includes few common 3rd-party libraries such as `numpy`, `scipy`, `pandas`
- What if you want to use other libraries that were not bundled with ParaView?

1. Let's assume you work on an Alliance cluster; check your ParaView's Python version

```
module load paraview
pvpython   # let's assume it says Python 3.13.2
```

2. Load the closest Python module, create a virtual env. and install your library there

```
module avail python                        # python/3.13.2 is one of them
module load python/3.13.2
python -m venv ~/env-astro                 # this will install a new virtual environment into ~/astro
source ~/env-astro/bin/activate
python -m pip install --upgrade pip --no-index
python -m pip install --no-index xarray   # install an external package into this new environment
```

3. Next time you log in to the cluster, start `pvpython`:

```
module load paraview
pvpython
```

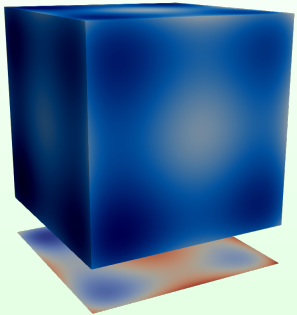
4. Load your new virtual environment directly from Python:

```
venv_path = '/home/user01/env-astro/lib/python3.13/site-packages'
import sys
sys.path.insert(0, venv_path)    # activate your new env from the shell
from paraview.simple import *
import xarray                    # this xarray comes from your new virtual environment
```

5. For batch workflows, replace `pvpython` with `pvbatch`

Creating/modifying VTK objects

Let's say we want to plot a projection of a cubic dataset along one of its principal axes, or do some other transformation for which there is no filter



- Calculator / Python Calculator filter cannot modify the geometry ...

Programmable filter

Watch our webinar `https://bit.ly/programmablefilter`

1. Apply Programmable Filter with OutputDataSetType = vtkUnstructuredGrid
2. Paste the following code `codes/projectionUnstructured.py` into the filter (this code was last tested in ParaView 6.0.1)

```
numPoints = inputs[0].GetNumberOfPoints()
side = int(round(numPoints**(1./3.)))     # round() in this Python returns float type
layer = side*side
rho = inputs[0].PointData['density']      # 1D flat array
points = vtk.vtkPoints()       # create vtkPoints instance, to contain 100^2 points in the projection
proj = vtk.vtkDoubleArray(); proj.SetName('projection')    # create the projection array
for i in range(layer):         # loop through 100x100 points
    x, y, z, column = inputs[0].GetPoint(i)[0:2], -20., 0.
    for j in range(side):
        column += rho.GetValue(i+layer*j)
    points.InsertNextPoint(x,y,z)          # also points.InsertPoint(i,x,y,z)
    proj.InsertNextValue(column)           # add value to this point

output.SetPoints(points)                   # add points to vtkUnstructuredGrid
output.GetPointData().SetScalars(proj)     # add projection array to these points

quad = vtk.vtkQuad()                       # create a cell
output.Allocate(side, side)                # allocate space for side^2 'cells'
for i in range(side-1):
    for j in range(side-1):
        quad.GetPointIds().SetId(0,i+j*side)
        quad.GetPointIds().SetId(1,(i+1)+j*side)
        quad.GetPointIds().SetId(2,(i+1)+(j+1)*side)
        quad.GetPointIds().SetId(3,i+(j+1)*side)
        output.InsertNextCell(vtk.VTK_QUAD, quad.GetPointIds())
```