Building interactive applications with ETS

Prabhu Ramachandran and Pankaj Pandey

PyCon India, Bangalore
September 28, 2012

About the Tutorial

Intended Audience

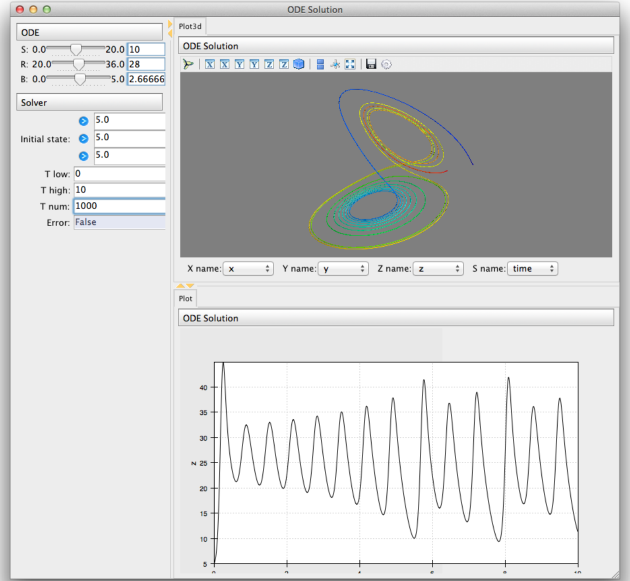
- Use Python to build interactive desktop applications

Goal: Successful participants will be able to

- Start using Enthought Tool Suite (ETS) to build non-trivial applications

Outline

1. Introduction
   - ODE 101
2. Traits
3. TraitsUI
4. Chaco
5. Mayavi
6. Putting it all together
7. Note on Envisage
8. Summary


ODE Solution
ODE
S: 0.0 20.0 10
R: 20.0 36.0 28
B: 0.0 5.0 2.66666
Solver
Initial state: 5.0 5.0 5.0
T low: 0
T high: 10
T num: 1000
Error: False
Plot3d
ODE Solution
X name: x
Y name: y
Z name: z
S name: time
Plot
ODE Solution
z
40
35
30
25
20
15
10
5

Approach

- A graphical explorer for ODEs from the ground up
- Support arbitrary 2 and 3 dimensional systems
- Using ETS

Why?

- Something interesting and concrete
- Same ideas extend to other situations

Outline

1. Introduction
   - ODE 101
2. Traits
3. TraitsUI
4. Chaco
5. Mayavi
6. Putting it all together
7. Note on Envisage
8. Summary

ODE 101

- Used to model many systems
  - Physics, astronomy
  - Geology (weather modeling)
  - Chemistry (reactions)
  - Biology
  - Ecology/population modeling
  - Economics (stock trends, interest rates etc.)
- Rich behavior
- Numerical solution: `scipy`

Simple equation

$$\frac{dx}{dt} = f(x, t) \tag{1}$$

x can be a vector with many components.

Solving ODEs using SciPy

- Consider the spread of an epidemic in a population
- $\frac{dy}{dt} = ky(L - y)$ gives the spread of the disease
- L is the total population.
- Use $L = 2.5E5, k = 3E-5, y(0) = 250$
- Define a function as below

```
In []: from scipy.integrate import odeint
In []: def epid(y, t):
  ....      k = 3.0e-5
  ....      L = 2.5e5
  ....      return k*y*(L-y)
  ....
```

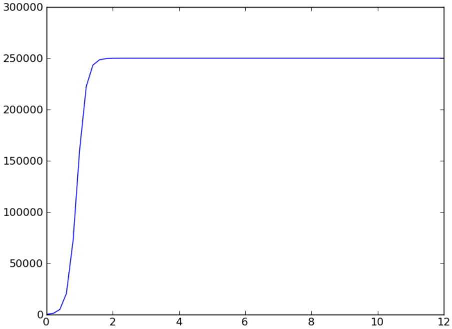
Solving ODEs using SciPy . . .

```
In []: t = linspace(0, 12, 61)

In []: y = odeint(epid, 250, t)

In []: plot(t, y)
```

Result



Lorenz equation example

$$\begin{aligned}
\frac{dx}{dt} &= s(y - x) \\
\frac{dy}{dt} &= rx - y - xz \\
\frac{dz}{dt} &= xy - bz
\end{aligned}$$

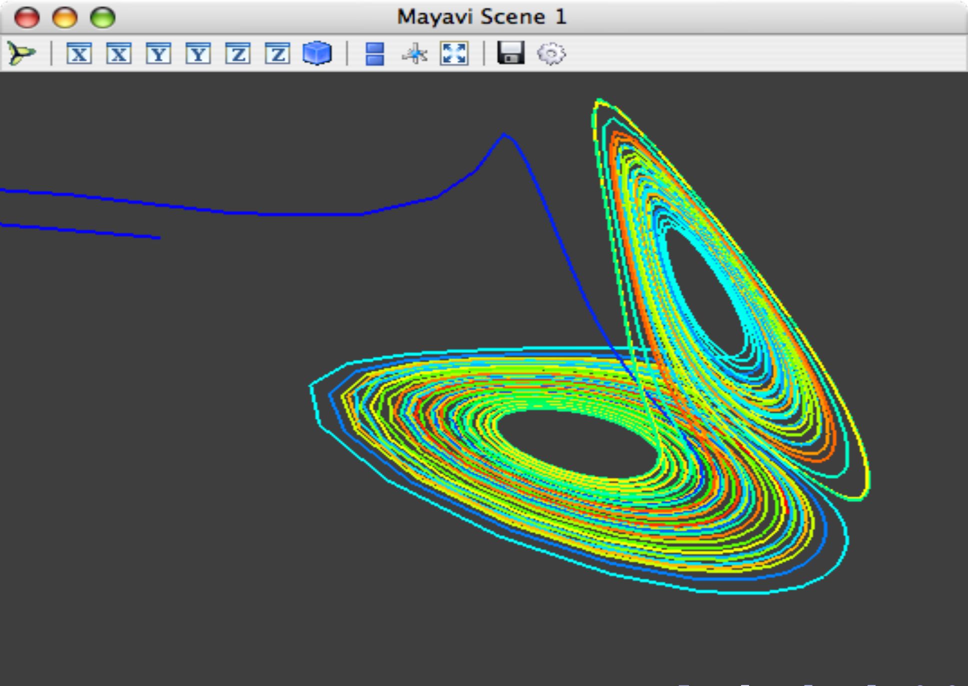
- Specifies the evolution of the system
- Think: Velocity of a particle in 3D
- Lets trace its path

Solution

```
import numpy as np
from scipy.integrate import odeint
def lorenz(r, t s=10.,r=28., b=8./3.):
    x, y, z = r
    u = s*(y-x)
    v = r*x -y - x*z
    w = x*y - b*z
    return np.array([u, v, w])

start = (10., 50., 50.)
t = np.linspace(0., 50., 2000)
r = odeint(lorenz, start, t)
x, y, z = r[:,0], r[:,1], r[:,2]

mlab.plot3d(x, y, z, t,
from mayavi import mlab
                    tube_radius=None)
```


Mayavi Scene 1

Now what?

- An application to explore these
- Use cases
  - Interactive exploration
  - Change the equations on UI
  - See output immediately
  - Standard equations (fully setup)

ETS: Enthought Tool Suite

- Traits: Object Models
- TraitsUI: Views for Objects having Traits
- Chaco: 2D Visualizations
- Mayavi: 3D Visualizations
- Envisage: Application Framework

- Miscellaneous libraries

Outline

1 Introduction
- ODE 101

2 Traits

3 TraitsUI

4 Chaco

5 Mayavi

6 Putting it all together

7 Note on Envisage

8 Summary

Introduction to Traits

- trait: Python object attribute with additional characteristics

- `http://code.enthought.com/projects/traits`
- `http://github.enthought.com/traits/tutorials`

Trait features

- Initialization: default value
- Validation: strongly typed
- Delegation: value delegation
- Notification: events
- Visualization: MVC, automatic GUI!

Traits Example

```
from traits.api import (Delegate, HasTraits,
    Instance, Int, Str)

class Parent(HasTraits):
    # INITIALIZATION: 'last_name' initialized to ''
    last_name = Str('')

class Child(HasTraits):
    age = Int
    # VALIDATION: 'father' must be Parent instance
    father = Instance(Parent)
    # DELEGATION: 'last_name' delegated to father's
    last_name = Delegate('father')
    # NOTIFICATION: Method called when 'age' changes
    def _age_changed(self, old, new):
        print 'Age changed from %s to %s ' % (old, new)
```

Traits Example

```python
from traits.api import (Delegate, HasTraits,
    Instance, Int, Str)

class Parent(HasTraits):
    # INITIALIZATION: 'last_name' initialized to ''
    last_name = Str('')

class Child(HasTraits):
    age = Int
    # VALIDATION: 'father' must be Parent instance
    father = Instance(Parent)
    # DELEGATION: 'last_name' delegated to father's
    last_name = Delegate('father')
    # NOTIFICATION: Method called when 'age' changes
    def _age_changed(self, old, new):
        print 'Age changed from %s to %s ' % (old, new)
```

Traits Example

```
In []: joe = Parent()
In []: joe.last_name = 'Johnson'
In []: moe = Child()
In []: moe.father = joe

In []: moe.last_name # Delegation
Out[]: "Johnson"

In []: moe.age = 10 # Notification
Age changed from 0 to 10

In []: moe.configure_traits() # Visualization
```

10 m

Predefined Trait Types

- Standard:
  `Bool, Int, Float, Str, Tuple, List, Dict`
- Constrained:
  `Range, Regex, Expression, ReadOnly`
- Special:
  `Either, Enum, Array, File, Color, Font`
- Generic: **`Instance, Any, Callable`**
- ...
- Custom traits: 2D/3D plots etc.

Trait Change Notifications

- Static: `def _<trait_name>_changed()`
- Decorator:
  `@on_trait_change('extended.trait[].name')`
- Dynamic:

```
obj.on_trait_change(handler,
                    ['extended.trait[].name'])
```

Notification Example

```
class Parent(HasTraits):
    last_name = Str('')
class Child(HasTraits):
    age = Int
    father = Instance(Parent)

    def _age_changed(self, old, new):
        print 'Age changed from %s to %s ' % (old, new)

    @on_trait_change('father.last_name')
    def _dad_name_updated(self):
        print self.father.last_name

def handler(obj, name, old, new):
    print obj, name, old, new

c = Child(father=Parent(last_name='Ram'))
c.on_trait_change(handler, ['father', 'age'])
```

Notification Example

```
class Parent(HasTraits):
    last_name = Str('')
class Child(HasTraits):
    age = Int
    father = Instance(Parent)

    def _age_changed(self, old, new):
        print 'Age changed from %s to %s ' % (old, new)

    @on_trait_change('father.last_name')
    def _dad_name_updated(self):
        print self.father.last_name

def handler(obj, name, old, new):
    print obj, name, old, new

c = Child(father=Parent(last_name='Ram'))
c.on_trait_change(handler, ['father', 'age'])
```

Designing the ODE explorer app

Think!

Focus on the object model

Designing the ODE explorer app

Think!

Focus on the object model

Object model

- A class to represent the equation and parameters
- A class for the ODE solution
- Make sure it works – TDD

Lorenz Equation

```
import numpy as np
from traits.api import HasTraits, Float

class LorenzEquation(HasTraits):
    s = Float(10)
    r = Float(28)
    b = Float(8./3)

    def eval(self, X, t):
        x, y, z = X[0], X[1], X[2]
        u = self.s*(y-x)
        v = self.r*x - y - x*z
        w = x*y - self.b*z
        return np.array([u, v, w])
```

Generalizing the ODE Equation model

```python
from traits.api import (Either, HasTraits, List,
    Str)

class ODE(HasTraits):
    """ An ODE of the form dX/dt = f(X) """
    name = Str
    vars = Either(List(Str), Str,
                  desc='The names of variables')
    def eval(self, X, t):
        """ Evaluate the derivative, f(X).
        """
        raise NotImplementedError
```

Lorenz Equation as an ODE Subclass

```
class LorenzEquation(ODE):
    name = 'Lorenz Equation'
    vars = ['x', 'y', 'z']
    s = Float(10)
    r = Float(28)
    b = Float(8./3)

    def eval(self, X, t):
        x, y, z = X[0], X[1], X[2]
        u = self.s*(y-x)
        v = self.r*x - y - x*z
        w = x*y - self.b*z
        return np.array([u, v, w])
```

Or . . .

```
from traits.api import HasTraits, Range, # ...

class LorenzEquation(HasTraits):
    # ...
    s = Range(0.0, 20.0, 10.0,
                   desc='the parameter s')
    r = Range(0.0, 50.0, 28.0)
    b = Range(0.0, 10.0, 8./3)
    # ...
```

Solving the ODE: ODESolver

```
class ODESolver(HasTraits):
    ode = Instance(ODE)
    initial_state = Either(Float, Array)
    t = Array

    solution = Property(Array,
            depends_on='initial_state, t, ode')

    @cached_property
    def _get_solution(self):
        return self.solve()

    def solve(self):
        """ Solve the ODE and return the values
        of the solution vector at specified times t.
        """
        from scipy.integrate import odeint
        return odeint(self.ode.eval, self.initial_state,
                      self.t)
```

Solving the ODE: ODESolver

```
class ODESolver(HasTraits):
    ode = Instance(ODE)
    initial_state = Either(Float, Array)
    t = Array

    solution = Property(Array,
            depends_on='initial_state, t, ode')

    @cached_property
    def _get_solution(self):
        return self.solve()

    def solve(self):
        """ Solve the ODE and return the values
        of the solution vector at specified times t.
        """
        from scipy.integrate import odeint
        return odeint(self.ode.eval, self.initial_state,
                      self.t)
```

Solving the ODE: ODESolver

```python
class ODESolver(HasTraits):
    ode = Instance(ODE)
    initial_state = Either(Float, Array)
    t = Array

    solution = Property(Array,
            depends_on='initial_state, t, ode')

    @cached_property
    def _get_solution(self):
        return self.solve()

    def solve(self):
        """ Solve the ODE and return the values
        of the solution vector at specified times t.
        """
        from scipy.integrate import odeint
        return odeint(self.ode.eval, self.initial_state,
                      self.t)
```

Testing

```
class TestLorenzEquation(unittest.TestCase):
    def setUp(self):
        self.ode = LorenzEquation()
        self.solver = ODESolver(ode=self.ode)
        self.solver.initial_state = [10.,50.,50.]
        self.solver.t = numpy.linspace(0, 10, 1001)

    def test_eval(self):
        dX = self.ode.eval(self.solver.initial_state, 0.0)
        self.assertAlmostEqual(dX[0], 400)
        self.assertAlmostEqual(dX[1], -270)
        self.assertAlmostEqual(dX[2], 1100/3.)

    def test_solve(self):
        soln = self.solver.solution[1,:]
        self.assertAlmostEqual(soln[0], 13.65484958)
        self.assertAlmostEqual(soln[1], 46.64090341)
        self.assertAlmostEqual(soln[2], 54.35797299)
```

Exercise

Solve an ODE

Use the given skeleton code of the ODE equation (**`solve_ode.py`**) and the solver. Put them together to get a solution. Print the final solution.

Outline

1. Introduction
   - ODE 101
2. Traits
3. TraitsUI
4. Chaco
5. Mayavi
6. Putting it all together
7. Note on Envisage
8. Summary

TraitsUI

- Implement MVC design pattern
- Create default views for models
- Keep multiple views synced with model
- Create UI with minimal toolkit knowledge

Default Traits View

```
father = Parent(last_name='Joe')
child = Child(age=2, father=father)
child.configure_traits()
```

- Declarative
- Automatic UI creation with `configure_traits()`
- Sync with model
- Sync between different views

Default Traits View

```
father = Parent(last_name='Joe')
child = Child(age=2, father=father)
child.configure_traits()
```

- Declarative
- Automatic UI creation with `configure_traits()`
- Sync with model
- Sync between different views

Simplest view

If you aren't happy with the defaults …

```
from traitsui.api import View, Item
class LorenzEquation(ODE):
    # ...
    view = View(Item('s'),
                Item('r'),
                Item('b'),
                title='Lorenz equation')
```

MVC?

```
from traitsui.api import View, Item
class LorenzEquation(ODE):
    # ...

view = View(Item('s'),
            Item('r'),
            Item('b'))
eq = LorenzEquation()
eq.configure_traits(view=view)
```

Views: common parameters

- **`title`**: Title of the view
- **`kind`**: `'modal', 'live', 'livemodal'`
- **`resizable`**
- **`width, height`**
- **`buttons`**: OK, Cancel and other buttons
- **`id`**: persists view
- …and a lot more

Items: common parameters

- `name`: name of trait being edited
- `label`: optional label to use
- `style`: `'simple'`/`'custom'`/`'readonly'`/...
- `show_label`: `True/False`
- `help`: Help text for item
- `editor`: Specific editor to use
- `defined_when`: expression
- `visible_when`: expression
- `enabled_when`: expression
- ...

Groups

- **`Group, VGroup, HGroup`**: group of items
- Parameters:
  - **`label`**
  - **`layout`**: **`'normal', 'split', 'tabbed', ...`**
  - **`show_labels`**: **`True/False`**
  - **`defined_when, ...`**
  - **`width, height`**

Lorenz Equation View

Configure the parameters of the Lorenz equation *s*, *r* and *b*

```
class LorenzEquation(ODE):
    ...
    view = View(
        Item('s',
             editor=RangeEditor(low=0.0, high=20.0)),
        Item('r',
            editor=RangeEditor(low=20.0, high=36.0)),
        Item('b',
            editor=RangeEditor(low=0.0, high=5.0))
        )
```

Exercise

Customize the view for Lorenz equation

Use the existing `solve_ode.py` file and setup the views for one or more of the classes (as many as you are able).

Outline

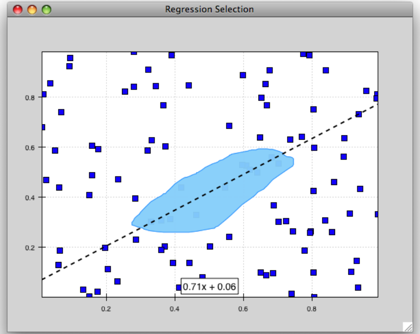
1. Introduction
   - ODE 101
2. Traits
3. TraitsUI
4. Chaco
5. Mayavi
6. Putting it all together
7. Note on Envisage
8. Summary

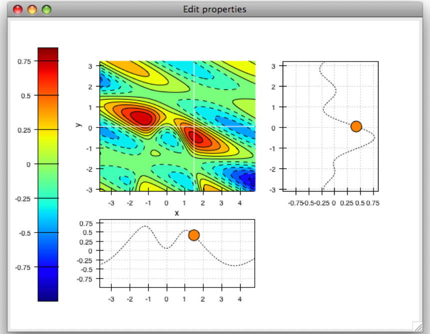
Chaco

- 2D plotting library
- Embeddable in any wx/Qt application
- Fast and interactive visualizations
- Integrates well with Traits and TraitsUI
- Easily extensible to create new types of plots and interactions

Chaco Interactive Plotting

`http://docs.enthought.com/chaco/user_manual/annotated_examples.html`





Core Ideas

- Plots are compositions of visual components
- Separation between data and screen space
- Modular design and extensible classes

Chaco Architecture Overview

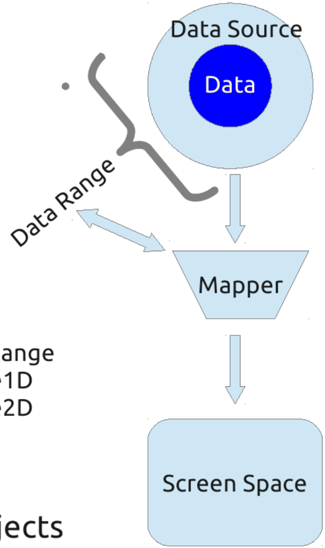
- Data Handling: wrap input data, transform co-ordinates between data and screen space (eg., **`ArrayDataSource, LinearMapper`**)
- Visual components: render to the screen (eg. **`LinePlot, ScatterPlot, Legend, PlotAx`**
- Tools: handle keyboard or mouse events and modify other components (eg. **`PanTool, ZoomTool, ScatterInspector`**)

Simple Plotting with Chaco

```
class LinePlot(HasTraits):
    plot = Instance(Plot)
    traits_view = View(
        Item('plot',editor=ComponentEditor(),
             show_label=False),
             width=500, height=500,
        resizable=True,
        title="Chaco Plot")

    def _plot_default (self):
        x = linspace(-14, 14, 100)
        y = sin(x) * x**3
        plotdata = ArrayPlotData(x = x, y = y)
        plot = Plot(plotdata)
        plot.plot(("x", "y"), type="line", color="blue")
        plot.title = "sin(x) * x^3"
        return plot
```

Chaco Data Model



- ArrayDataSource
- DataContextDataSource
- GridDataSource
- ImageData
- MultiArrayDataSource
- PointDataSource
  - data_changed
  - bounds_changed
  - metadata_changed

- BaseDataRange
- DataRange1D
- DataRange2D

- Base1DMapper
- LinearMapper
- LogMapper
- GridMapper
- PolarMapper

Data Objects

Commonly used Classes

Containers

- Handle layout
- Similar to layout grids in GUI toolkits
- Efficient way for event dispatch, since screen space is partitioned logically
    - OverlayPlotContainer
    - HplotContainer
    - VplotContainer
    - GridPlotContainer

Tools

- Take events from a component and perform actions based on that
    - PanTool
    - ZoomTool
    - ...

Renderers

- Actually draw a type of plot
    - BarPlot
    - Base2DPlot
    - ContourLinePlot
    - ContourPolyPlot
    - ImagePlot
    - CMapImagePlot
    - LinePlot
    - ErrorBarPlot
    - PolygonPlot
    - FilledLinePlot
    - ScatterPlot
    - ColormappedScatterPlot
    - ColorBar
    - PolarLineRenderer

Overlays

Plotting the ODE Solution

The Plot view, plot and the ODE solution. ODE can be obtained from ODESolver, so it can be a property.

```
class ODEPlot(HasTraits):
    """ A 2D plot of ode solution variables. """
    plot = Instance(Component)
    # We need to set data when solution changes
    pd = Instance(ArrayPlotData, args=())

    ode = Property(Instance(ODE), depends_on='solver')
    solver = Instance(ODESolver)

    def _get_ode(self):
        return self.solver and self.solver.ode
    ...
```

Plotting the ODE Solution

We can create a plot using the first solution array.

```
class ODEPlot(HasTraits):
    ...
    traits_view = View(
            Item('plot', editor=ComponentEditor(),
                 show_label=False),
            resizable=True, title="ODE Solution")

    def _plot_default(self):
        self.pd.set_data('index', self.solver.t)
        # Set the first array as value array for plot.
        self.pd.set_data('value',
                         self.solver.solution[:,0])
        plot = Plot(self.pd)
        plot.plot(('index', 'value'))
        return plot
```

Adding some interactivity and labels

Use Chaco Tools

```
...
def _plot_default(self):
    ...
    # Add some interactivity to the plots.
    plot.tools.append(ZoomTool(component=plot))
    plot.tools.append(PanTool(component=plot))
    plot.tools.append(TraitsTool(component=plot))
    plot.x_axis.title = 'time'
    plot.y_axis.title = 'x'
    plot.title = self.ode.name
    return plot
```

Plotting the ODE Solution

We can make the plot react to changes to the solution (changing initial condition etc.)

```
class ODEPlot(HasTraits):
    ...
    @on_trait_change('solver.solution')
    def _on_soln_changed(self):
        self.pd.set_data('index',
                self.solver.t)
        self.pd.set_data('value',
                self.solver.solution[:, 0])
```

Outline

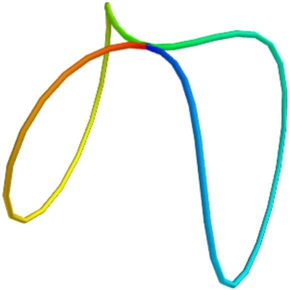
1. Introduction
   - ODE 101
2. Traits
3. TraitsUI
4. Chaco
5. Mayavi
6. Putting it all together
7. Note on Envisage
8. Summary

Mayavi

- `from mayavi import mlab`
- Easy to use
- Uses Traits/TraitsUI heavily
- Can embed 3D plots

A simple plot

1D data



```
>>> from numpy import *
>>> t = linspace(0, 2*pi, 50)
>>> u = cos(t)*pi
>>> x, y, z = sin(u), cos(u), sin(t)
```

```
>>> mlab.plot3d(x, y, z, t)
```

Changing how things look

Clearing the view

```
>>> mlab.clf()
```

IPython is your friend!

```
>>> mlab.points3d?
```

- Extra argument: Scalars
- Keyword arguments
- UI

Changing how things look

Clearing the view

```
>>> mlab.clf()
```

IPython is your friend!

```
>>> mlab.points3d?
```

- Extra argument: Scalars
- Keyword arguments
- UI

Embedding a 3D plot

```
from mayavi.core.ui.api import MayaviScene, \
    MlabSceneModel, SceneEditor

class Plot3D(HasTraits):
    scene = Instance(MlabSceneModel, args=())
    view = View(Item(name='scene',
                     editor=SceneEditor(
                         scene_class=MayaviScene),
                     show_label=False, resizable=True,
                     height=500, width=500),
                resizable=True)

    @on_trait_change('scene.activated')
    def generate_data(self):
        # Create some data
        X, Y = mgrid[-2:2:100j, -2:2:100j]
        R = 10*sqrt(X**2 + Y**2)
        Z = sin(R)/R
        self.scene.mlab.surf(X,Y,Z,colormap='gist_earth')
```

Embedding a 3D plot

```
from mayavi.core.ui.api import MayaviScene, \
    MlabSceneModel, SceneEditor

class Plot3D(HasTraits):
    scene = Instance(MlabSceneModel, args=())
    view = View(Item(name='scene',
                     editor=SceneEditor(
                         scene_class=MayaviScene),
                     show_label=False, resizable=True,
                     height=500, width=500),
                resizable=True)

   @on_trait_change('scene.activated')
   def generate_data(self):
        # Create some data
        X, Y = mgrid[-2:2:100j, -2:2:100j]
        R = 10*sqrt(X**2 + Y**2)
        Z = sin(R)/R
        self.scene.mlab.surf(X,Y,Z,colormap='gist_earth')
```

Exercise

Add a slider to the UI

Add a Range slider to the above example to change the `R = 10*sqrt(X**2 + Y**2)` to `R = self.factor*sqrt(X**2 + Y**2)` such that the `factor` can be adjusted.

Solution

```python
from traits.api import Range # <--
class Plot3D(HasTraits):
    scene = Instance(MlabSceneModel, args=())
    factor = Range(0.0, 20.0, 10.0) # <--
    view = View(Item(name='scene',
                     # ...
                     ),
                Item(name='factor'), # <--
                resizable=True)

   @on_trait_change('scene.activated, factor') # <--
   def generate_data(self):
        # ...
```

Outline

1 Introduction
- ODE 101

2 Traits

3 TraitsUI

4 Chaco

5 Mayavi

6 Putting it all together

7 Note on Envisage

8 Summary

The application

- Easy to put this together
- Exercise for you!

Hint: Compose various views using InstanceEditor with '*custom*' style!

Exercise

ODE application

Given the existing code, in `solve_ode.py` and `embed_3d_ex.py` create a UI for the Lorenz equation along with the solver and a 3D or 2D plot.

So what?

- Focus on the object model
- Solve the actual problem
- Model separation
- Easy to “wire-up”
- UI is mostly declarative

What next?

Outline

1. Introduction
   - ODE 101
2. Traits
3. TraitsUI
4. Chaco
5. Mayavi
6. Putting it all together
7. Note on Envisage
8. Summary

Envisage

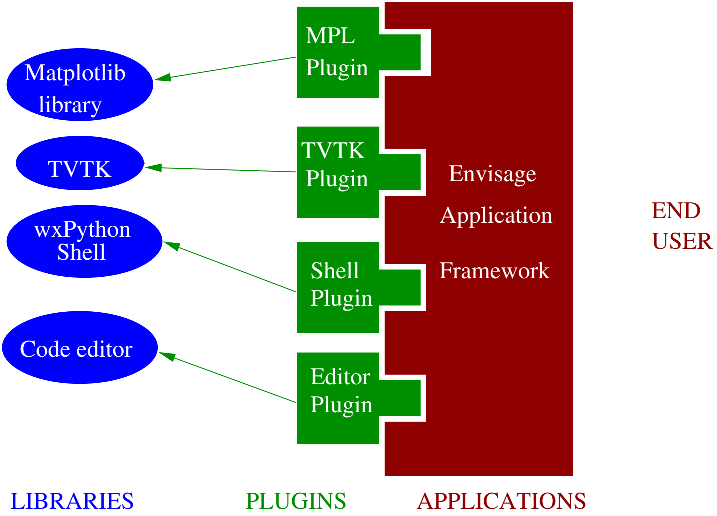
- Extensible framework
- `Plugins`
- `ExtensionPoints`
- `Extensions`
- Similar to Eclipse

Application frameworks . . .

Central idea

- Developer focuses on making a clean library
- Framework API specifies how different pieces of the app interact
- Plugin writer exposes the library or objects involved into the framework
- Application writer uses plugins to create new application easily

Application frameworks . . .



Big picture

Application and plugins

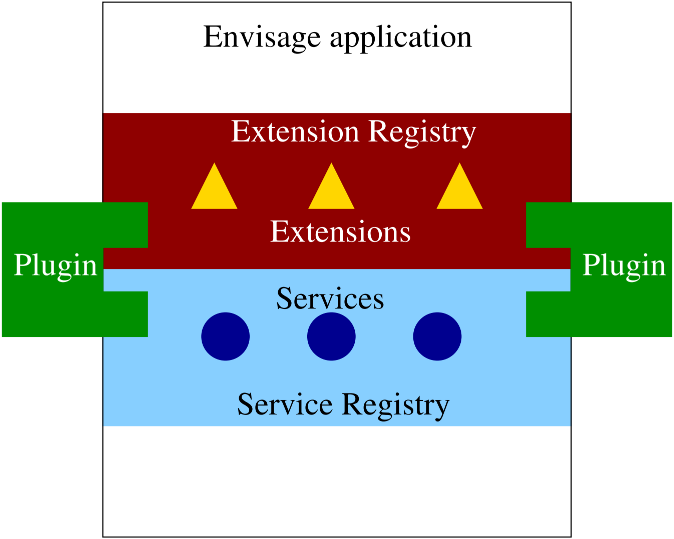
- Envisage application: given a list of plugins
- Plugins setup their contributions
- Application starts: all plugins are started
- Plugins add their capabilties to the app
- Application stops: all plugins are stopped

Big picture

Interaction/communication between plugins

- Plugins define extension points
- Extensions $\longrightarrow$ extension points
- Services
  - Well known/shared objects
  - Can be registered and looked-up with the application

Envisage application



Outline

1. Introduction
   - ODE 101
2. Traits
3. TraitsUI
4. Chaco
5. Mayavi
6. Putting it all together
7. Note on Envisage
8. Summary

Summary

- Traits
- TraitsUI
- Chaco
- Mayavi
- Envisage