Traits Tutorial

Enthought Tool Suite

TRAITS

Initialization, Validation, Observation, and Visualization of Python class attributes

KIVA

2D primitives supporting path based rendering, affine transforms, alpha blending and more.

ENABLE

Object based 2D drawing canvas

CHACO

Plotting toolkit for building complex, interactive 2D plots

MAYAVI

3D Visualization of Scientific Data based on VTK

ENVISAGE

Application plugin framework for building scriptable and extensible applications

What are traits?

Traits provide additional characteristics for Python object attributes:

- Initialization
- Validation
- Delegation
- Notification
- Visualization

What are traits?

Traits provide additional characteristics for Python object attributes:

- Initialization
- Validation
- Delegation
- Notification
- Visualization
- Documentation

A Note About Examples

- The code in the scripts do not exactly mimic the slides. It has usually been simplified for brevity.
- Most non-ui examples are simple scripts that can be run from the command line or IPython:

```
In[1] run rect_1.py
```

- Some examples have a `main()` method that must be run to get their output. This is typically done for examples that throw tracebacks as part of the demo.
- Most UI examples must be run from IPython started using the ‘-wthread’ option, or from a wxPython based shell such as PyCrust.

Defining Simple Traits -- rect_1.py

```
from enthought.traits.api import HasTraits, Float

class Rectangle(HasTraits): # <----- Derive from HasTraits
    """ Simple rectangle class with two traits.
    """
    # Width of the rectangle
    width = Float    # <----- Declare Traits

    # Height of the rectangle
    height = Float  # <----- Declare Traits
```

Defining Simple Traits -- rect_1.py

```
from enthought.traits.api import HasTraits, Float

class Rectangle(HasTraits): # <----- Derive from HasTraits
    """ Simple rectangle class with two traits.
    """
    # Width of the rectangle
    width = Float    # <----- Declare Traits

    # Height of the rectangle
    height = Float  # <----- Declare Traits
```

Note: Run `main()` for this example to execute similar commands to those below.

```
In[1]: run rect_1.py
In[2]: main()
```

```
# Demo Code
>>> rect = Rectangle()
>>> rect.width
0.0

# Set rect width to 1.0
>>> rect.width = 1.0
1.0
```

```
# Float traits convert integers
>>> rect.width = 2
>>> rect.width
2.0

# THIS WILL THROW EXCEPTION
>>> rect.width = "1.0"
TraitError: The 'width' trait of a
Rectangle instance must be a value
of type 'float', but a value of 1.0
was specified.
```

Default Values -- rect_2.py

```
from enthought.traits.api import HasTraits, Float

class Rectangle(HasTraits):
    """ Simple rectangle class with two traits.
    """

    # Width of the rectangle
    width = Float(1.0)  # <----- Set default to 1.0

    # Height of the rectangle
    height = Float(2.0) # <----- Set default to 2.0
```

```
#Demo Code
>>> rect = Rectangle()
>>> rect.width
1.0
>>> rect.height
2.0
```

```
# Initialization via
# keyword arguments
>>> rect = Rectangle(width=2.0,
                     height=3.0)
>>> rect.width
2.0
>>> rect.height
3.0
```

Coercion and Casting -- rect_3.py

```
from enthought.traits.api import HasTraits, Float

class Rectangle(HasTraits):
   """ Simple rectangle class with two traits.
   """

   # Basic traits allow "widening" coersion (Int->Float).
   width = Float

   # CFloat traits apply float() to any assigned variable.
   height = CFloat  # <----- CFloat is the casting version
                    #        of the basic Float trait
```

```
# Demo Code
>>> rect = Rectangle()
>>> rect.height = "2.0" # <----- This Works!
>>> rect.width = "2.0"
TraitError: The 'width' trait of a Rectangle instance must be a value
of type 'float', but a value of 2.0 was specified.
```

Traits for Basic Python Types

Coercing Trait	Casting Trait	Python Type	Default Value
`Bool`	`CBool`	`bool`	`False`
`Complex`	`CComplex`	`complex`	`0+0j`
`Float`	`CFloat`	`float`	`0.0`
`Int`	`CInt`	`int`	`0`
`Long`	`CLong`	`long`	`0L`
`Str`	`CStr`	`str or unicode (whichever assigned)`	`''`
`Unicode`	`CUnicode`	`unicode`	`u''`

Traits Speed compared to Standard Python

Attribute Access Method	Get Attribute		Set Attribute	
	Time (µs)	Speed-up	Time (µs)	Speed-up
Global Module Variable	0.070	**2.29**	0.100	**2.50**
Old Style Instance Attribute	0.127	**1.26**	0.175	**1.43**
New Style Instance Attribute	0.160	**-**	0.250	**-**
Standard Python Property	0.960	0.17	1.180	0.21
“Any” Traits Attribute	0.100	**1.60**	0.240	**1.04**
“Int” Traits Attribute	0.090	**1.78**	0.260	0.96
“Range” Traits Attribute	0.110	**1.45**	0.280	0.89
Statically Observed Trait	0.090	**1.78**	2.590	0.10
Dynamically Observed Trait	0.100	**1.60**	3.780	0.07
Delegated Trait	0.250	0.64	0.400	0.63
Delegated Trait (2 levels)	0.450	0.36	0.530	0.47
Delegated Trait (3 levels)	0.550	0.29	0.650	0.38

Comparison of setting various types of traits to setting standard python class attributes and properties. (from enthought/traits/tests/test_timing.py on 2.13 GHz Pentium M laptop)

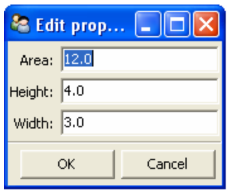
Properties -- rect_4.py

```
from enthought.traits.api import \
    HasTraits, Float, Property

class Rectangle(HasTraits):
    """ Rectangle class with
        read-only area property.
    """
    # Width of the rectangle
    width = Float(1.0)

    # Height of the rectangle
    height = Float(2.0)

    # The area of the rectangle
    # Defined as a property.
    area = Property

    # specially named method
    # automatically associated
    # with area.
    def _get_area(self):
        return width * height
```

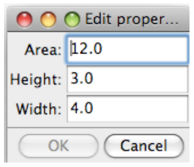
```
# Demo Code

>>> rect = Rectangle(width=2.0,
                     height=3.0)
>>> rect.area
6.0
>>> rect.width = 4.0
>>> rect.area
8.0
```

Traits UI – Default Views

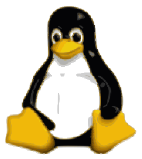
```
>>> rect = Rectangle(width=3.0, height = 4.0)
# Create a UI to edit the traits of the object.
>>> rect.edit_traits()
```

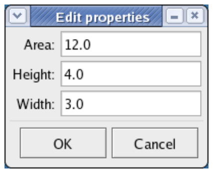


Area: 12.0
Height: 4.0
Width: 3.0



Area: 12.0
Height: 3.0
Width: 4.0



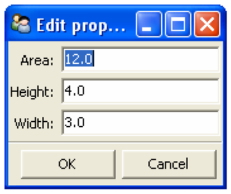


Area: 12.0
Height: 4.0
Width: 3.0

Properties Dependencies -- rect_5.py

```
from enthought.traits.api import HasTraits, Float, Property
from enthought.traits.ui.api import View, Item

class Rectangle(HasTraits):

    width = Float(1.0)
    height = Float(2.0)

    # Specify Dependencies with 'depends_on' meta-data.
    # This will update the area whenever width or height change.
    area = Property(depends_on=['width','height'], cached=True)

    @cached_property # will only recalculate area when it is "dirty"
    def _get_area(self):
        return width * height
```

```
# Demo Code
>>> rect = Rectangle(width=3.0,
                     height=4.0)
>>> rect.edit_traits()
```

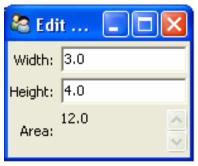


Area: 12.0
Height: 4.0
Width: 3.0

Default UI Views-- rect_6.py

```
from enthought.traits.api import HasTraits, Float, Property
from enthought.traits.ui.api import View, Item

class Rectangle(HasTraits):

    width = Float(1.0)
    height = Float(2.0)
    area = Property(depends_on=['width','height'])

    # Define a default view with the area as a readonly editor.
    view = View('width', 'height', Item('area', style='readonly'))

    def _get_area(self):
        return width * height
```

```
# Demo Code
>>> rect = Rectangle(width=3.0,
                     height=4.0)
>>> rect.edit_traits()
```

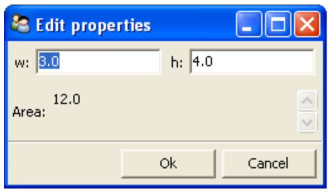


Width: 3.0
Height: 4.0
Area: 12.0

Simple UI Layout-- rect_6.py

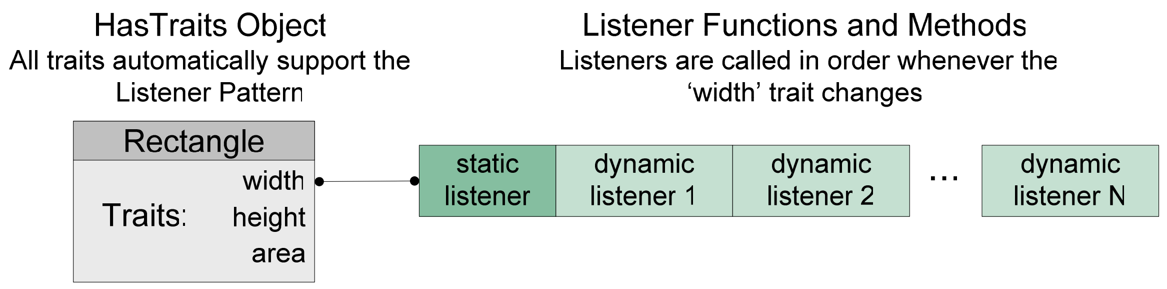
```
From enthought.traits.ui.api import View, HGroup, VGroup, Item

view1 = View(
             VGroup( # Create a Vertical layout group.
                    HGroup(# And a Horizontal group within that.
                           # Change the labels on each item.
                           Item('width',label='w'),
                           Item('height', label='h')
                    ),
                    Item('area', style='readonly'),
             ),
            # Add OK, Cancel buttons to the UI
            buttons=['Ok','Cancel']
        )
```

```
# Demo Code
>>> rect = Rectangle(width=3.0,
                     height=4.0)
>>> rect.edit_traits(view=view1)
```



w: 3.0 h: 4.0
Area: 12.0

Trait Listeners


HasTraits Object
All traits automatically support the Listener Pattern
Rectangle
Traits:
width
height
area
Listener Functions and Methods
Listeners are called in order whenever the 'width' trait changes
static listener
dynamic listener 1
dynamic listener 2
…
dynamic listener N

Static Trait Notification -- amplifier_1.py

```
class Amplifier(HasTraits):
    """ Guitar Amplifier Model
    """

    # Volume setting for the amplifier.
    volume = Range(0.0, 11.0, default=5.0)

    # Static observer method called whenever volume is set.
    def _volume_changed(self, old, new):
        if new == 11.0:
            print "This one goes to eleven"
```

```
# Demo Code
>>> spinal_tap = Amplifier()
>>> spinal_tap.volume = 11.0
This one goes to eleven
>>> spinal_tap.volume = 11.0 # nothing is printed because
                             # the value didn't change.
```

Valid Static Trait Notification Signatures

```
def _volume_changed(self):
       # no arguments...

def _volume_changed(self, new):
       # new -- the just-set value of volume

def _volume_changed(self, old, new):
       # old -- the previous value of volume
       # new -- the just-set value of volume

def _volume_changed(self, name, old, new):
       # name – the name of the trait (‘volume’)
       # old is the previous value of volume
       # new is the just-set value of volume

       # This signature is usually used for the
       # _anytrait_changed() observer that is called
       # whenever any trait is changed on the object.
```

Dynamic Trait Notification – amplifier_2.py

```
class Amplifier(HasTraits):
       """ Guitar Amplifier Model
       """

       # Volume setting for the amplifier.
       volume = Range(0.0, 11.0, default=5.0)


def printer(value):
       print “new value:”, value
```

```
# Demo Code
>>> spinal_tap = Amplifier()
# In the following, name can also be a list of trait names
>>> spinal_tap.on_trait_change(printer, name=‘volume’)
>>> spinal_tap.volume = 11.0
new value: 11.0
```

Valid Dynamic Trait Notification Signatures

```
def observer():
      # no arguments...

def observer(new):
      # new -- the new value of the changed trait

def observer(name, new):
      # name -- the name of the changed trait
      # new -- the new value of the changed trait

def observer(object, name, new):
      # object -- the object containing the changed trait
      # name -- the name of the changed trait
      # new – the new value of the changed trait

def observer(object, name, old, new):
      # object -- the object containing the changed trait
      # name -- the name of the changed trait
      # old – the previous value of the changed trait
      # new – the new value of the changed trait
```

Dynamic Trait Notification -- amplifier_3.py

```
class Amplifier(HasTraits):
        """ Guitar Amplifier Model
        """
        # Volume setting for the amplifier.
        volume = Range(0.0, 11.0, default=5.0)

class Listener:
        """ Class that will listen to the Amplifier volume
        """
        def printer(self, value):
                print "new value:", value
```

```
# Demo Code
>>> spinal_tap = Amplifier()
>>> listener = Listener()
>>> spinal_tap.on_trait_change(listener.printer, name='volume')
>>> spinal_tap.volume = 11.0
new value: 11.0

# since on_trait_change has a weak reference to the class method,
# when the class goes away, the method no longer fires.
>>> del listener
>>> spinal_tap.volume = 10.0
```

@on_trait_change decorator -- amplifier_4.py

```
from enthought.traits.api import HasTraits, Range, on_trait_change

class Amplifier(HasTraits):
    """ Guitar Amplifier Model
    """

    volume = Range(0.0, 11.0, value=5.0)
    reverb = Range(0, 10.0, value=5.0)

    # The on_trait_change decorator can listen to multiple traits
    # Note the "list" of traits is specified as a string.
    @on_trait_change('reverb, volume')
    def update(self, name, value):
        print 'trait %s updated to %s' % (name, value)

# Demo Code
>>> spinal_tap = Amplifier()
>>> spinal_tap.volume = 11.0
trait volume updated to 11.0
>>> spinal_tap.reverb = 2.0
trait reverb updated to 2.0
```

Listeners on a different thread -- amplifier_5.py

```
class Amplifier(HasTraits):
    volume = Range(0.0, 11.0, value=5.0)

    def __init__(self, *args, **kw):
        super(Amplifier, self).__init__(*args, **kw)

        # Use the "dispatch" keyword to run the listener on a
        # different thread.
        self.on_trait_change(self.update, 'volume', dispatch='new')

    def update(self, name, value):
        print 'thread %s sleeping for 1 second' % thread.get_ident()
        sleep(1.0)
        print 'trait %s updated to %s' % (name, value)


# Demo Code
>>> spinal_tap = Amplifier()
>>> spinal_tap.volume = 11.0
main thread: -1601355872
thread identity -1340948480 sleeping for 1 second
trait volume updated to 11.0
```

More advanced Trait Types

- Enum
- List
- Dict
- Array
- Instance
- This

Enum Trait – traffic_light_1.py

```
# enthought imports
from enthought.traits import HasTraits, Enum

class TrafficLight(HasTraits):

      color = Enum("green", "yellow", "red")
```

```
# Demo Code
>>> light = TrafficLight()
>>> light.color
"green"
>>> print "THIS WILL RAISE AN EXCEPTION"
>>> light.color = "blue"
TraitError: The 'color' trait of a TrafficLight instance must
be 'green' or 'yellow' or 'red', but a value of blue was
specified.
```

List Traits

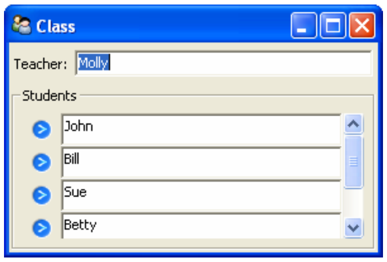
```
class Foo(HasTraits):

  # same as List(Any)
  a = List

  # List of Strings
  b = List(Str)

  # List of Person objects
  c = List(Instance(Person))

  # List of Ints with 3-5 elements. The default
  # value is [1,2,3]
  d = List([1,2,3], Int, minlen=3, maxlen=5)
```

List Trait – school_class_1.py

```
class SchoolClass(HasTraits):

    # List of the students in the class
    students = List(Str)                    # <-- List of strings

    def _students_changed(self, old, new):    # <-- called when list replaced
        print "The entire class has changed:", new

    def _students_items_changed(self, event): # <-- called when list items changed
        """ event.added -- A list of the items added to students
            event.removed -- A list of the items removed from students
            event.index -- Start index of the items that were added/removed
        """
        if event.added: print "added (index,name):", event.index, event.added
        else:  print "removed (index,name):", event.index, event.removed

# Demo Code
>>> school_class = SchoolClass()
>>> school_class.students = ["John", "Jane", "Jill"] # initial set of students.
The entire class has changed: ['John', 'Jane', 'Jill']
>>> school_class.students.append("Bill")            # add a student
students added (index,name): 3 ['Bill']
>>> del school_class.students[1:3]                  # remove some students
students removed (index,name): 1 ['Jane', 'Jill']
```

List Trait UI

```
# Create a customized view of the list.
>>> view = View('teacher',
                Group(
                      Item('students',
                           style='custom',
                           editor=ListEditor(rows=5)
                           show_label=False
                      ),
                     show_border=False,
                      label='Students'
                ),
                title = 'Class',
                width=300,
                height=200,
                resizable=True
            )

>>> school_class.edit_traits(view=view)
```



Teacher: Molly

John
Bill
Sue
Betty

Dict Traits

```
class Foo(HasTraits):

    # Signature: Dict(key_type, value_type)

    # Basic dictionary with unchecked key/value types
    # Same as Dict(Any, Any)
    a = Dict

    # Dictionary with checked Str key type
    # Same as Dict(Str, Any)
    b = Dict(Str)

    # Dictionary with string for keys and floats for values
    c = Dict(Str, Float)

    # Default value specified for the dictionary
    d = Dict(Str, Float, value={"hello": 1.0})
```

Array Objects

```
from numpy import float32, int32
from enthought.traits.api import Array, HasTraits

class TriangleMesh(HasTraits):

    # An Nx3 floating point array of points (vertices) within the mesh.
    points = Array(dtype=float32, shape = (None,3))

    # An Mx3 integer array of indices into the points array.
    # Each row defines a triangle in the mesh.
    triangles = Array(dtype=int32, shape=(None,3))

# Demo Code
points = numpy.array([[0,0,0], [1,0,0], [0,1,0], [0,0,1]], dtype=float32)
triangles = numpy.array([[0,1,3], [0,3,2], [1,2,3], [0,2,1]], dtype=int32)

# Demo Code
>>> tetra = TriangleMesh()
# Set the data points and connectivity
>>> tetra.points = points
>>> tetra.triangles = triangles
# THIS WILL RAISE AN EXCEPTION
>>> tetra.points = data[:,:2]
TraitError: The 'points' trait of a TriangleMesh instance must be an array
of 64bit float values with shape ('*', 3), but a value of array([[ 0., 0.],
```

Instance Objects – instance_1.py

```
class Person(HasTraits):
    first_name = Str("John")
    last_name = Str("Doe")

    def __repr__(self): return 'Person("%s %s")' % (self.first_name, self.last_name)

class Family(HasTraits):
    # Instantiate the default Person
    dad = Instance(Person,args=())

    # Instantiate a Person object with a different first name
    mom = Instance(Person, args=(), kw={'first_name':'Jane'})

    # Son is a Person object, but it defaults to 'None'
    son = Instance(Person)

    # In case you need "forward" declarations, you can use
    # the name as a string.  Default is None
    daughter = Instance('Person')
```

```
# Demo Code
>>> family = Family()
>>> family.dad
Person("John Doe")
>>> family.mom
Person("Jane Doe")
>>> family.son
None
```

```
>>> family.daughter
None
>>> family.son = Person(first_name="Bubba")
>>> family.daughter = Person(first_name='Sissy')
>>> family.son
Person("Bubba Doe")
>>> family.daughter
Person("Sissy Doe")
```

Delegation – instance_delegate.py

```
class Person(HasTraits):
    first_name = Str("John")
    last_name = Str("Doe")

    def __repr__(self):
        return 'Person("%s %s")' % (self.first_name, self.last_name)

class Child(Person):
    parent = Instance(Person, args=())

    # Define last_name to "delegate" to the parent's last name
    last_name = Delegate('parent', 'last_name')


# Demo Code
>>> dad = Person(first_name="Sam", last_name="Barns")
>>> child = Child(first_name="Jane", parent=dad)
>>> dad
Person("Sam Barns")
>>> child
Person("Jane Barns")
```

Instance List -- instance_observer_1.py

```
class Person(HasTraits):
    name = Str
    age = Int

class SchoolClass(HasTraits):
    teacher = Instance(Person)
    students = List(Person)

    def _age_changed_for_teacher(self, object, name, old, new):
        print 'The teacher is now', new, 'years old.'

    def _age_changed_for_students(self, object, name, old, new):
        print object.name, 'is now', new, 'years old.'
```

```
# Demo Code
the_class = SchoolClass()
teacher_ben = Person(name="Ben",
                     age=35)
the_class.teacher = teacher_ben
bob = Person(name="Bob", age=10)
the_class.students.append(bob)
jane = Person(name="Jane", age=11)
the_class.students.append(jane)
```

```
# This calls the SchoolClass observer
teacher_ben.age = 36
The teacher is now 36 years old.
# This calls the SchoolClass observer
>>> bob.age = 11
Bob is now 11 years old.
# Remove Bob from the Class and the
# observer is no longer called.
>>> the_class.students.remove(bob)
>>> bob.age = 12
```

Event Traits – event_1.py

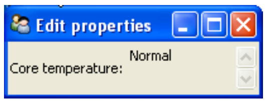
```
class Rectangle(HasTraits):

    # Width of the rectangle
    width = Float(1.0)

    # Height of the rectangle
    height = Float(2.0)

    # Set to notify others that you have changed to a Rectangle
    # Listen to this if you want to react to any changes to a Rectangle
    updated = Event

def rect_printer(rect, name, value):
    print 'rectangle (width, height): rect.width, rect.height'
```

```
# Demo Code
>>> rect = Rectangle()
# Hook up a dynamic listener to respond whenever rect is updated.
>>> rect.on_trait_change(rect_printer, name='updated')
# update multiple items
>>> rect.width = 10
>>> rect.height = 20
# now explicitly tell the item that it is updated
>>> rect.updated = True
rectangle (width, height): 10.0 20.0
```

Model View Pattern – model_view_1.py

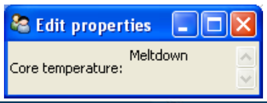
```
class Reactor(HasTraits):
   core_temperature = Range(-273.0, 100000.0)

class ReactorModelView(ModelView):

    # The "dummy" view of the reactor should be a waring string.
    core_temperature = Property(depends_on='model.core_temperature')

    def _get_core_temperature(self):
        temp = self.model.core_temperature
        if temp <= 500.0:
           return 'Normal'
        if temp < 2000.0:
           return 'Warning'
        return 'Meltdown'

my_view = View(Item('core_temperature', style = 'readonly'))
```

```
# Demo Code
>>> reactor = Reactor( core_temperature = 200.0 )
>>> view = ReactorModelView(model=reactor)
>>> view.edit_traits(view=my_view)
```



Core temperature: Normal

```
# Now change the temperature
>>> reactor.core_temperature = 5000.0
```



Core temperature: Meltdown

UI Demos

- Table Demo
- Polynomial Demo

Helpful resources

- Docs
  enthought/traits/docs/traits2_UM.doc
  enthought/traits/docs/traits UI Users Guide.doc

- Mailing List Help
  enthought-dev@enthought.com

- Wiki
  https://svn.enthought.com/enthought/wiki/Traits