Traits User Manual

David C. Morrill
Janet M. Swisher

Version 1.2

3-Jan-07

Enthought, Inc.
515 Congress Avenue
Suite 2100
Austin TX 78701
1.512.536.1057 (voice)
1.512.536.1059 (fax)
http://www.enthought.com
info@enthought.com

Table of Contents

Revision History

Version	*Date*	*Description*
1.0	12-May-05	Initial published version.
1.1	9-Feb-06	Converted source files from OpenOffice.org to Microsoft Word. Removed sections on Traits UI, as these are now covered in the *Traits UI User Guide*.
1.2	3-Jan-07	Converted to a template that is more compatible with Pydoh. Removed "Syntax and Class Reference", as this content is now covered in the *Traits API Reference*.

1 Introduction

The Traits package for the Python language allows Python programmers to use a special kind of type definition called a *trait*. This document introduces the concepts behind, and usage of, the Traits package.

For more information on the Traits package, refer to the Traits web page at http://code.enthought.com/traits . This page contains links to downloadable packages, the source code repository, and the Traits development website. Additional documentation for the Traits package is available from the Traits web page, including:

- *Traits API Reference*
- *Traits UI User Guide*
- Traits Technical Notes

1.1 What Are Traits?

A trait is a type definition that can be used for normal Python object attributes, giving the attributes some additional characteristics:

- **Initialization**—A trait has a *default value*, which is automatically set as the initial value of an attribute before its first use in a program.
- **Validation**—A trait attribute is *manifestly typed*. The type of a trait-based attribute is evident in the code, and only values that meet a programmer-specified set of criteria (i.e., the trait definition) can be assigned to that attribute. Note that the default value need not meet the criteria defined for assignment of values.
- **Delegation**—The value of a trait attribute can be contained either in the defining object or in another object that is *delegated* to by the trait.
- **Notification**—Setting the value of a trait attribute can *notify* other parts of the program that the value has changed.

- **Visualization**—User interfaces that allow a user to *interactively modify* the values of trait attributes can be automatically constructed using the traits' definitions. This feature requires that a supported GUI toolkit be installed. However, if this feature is not used, the Traits package does not otherwise require GUI support.

A class can freely mix trait-based attributes with normal Python attributes, or can opt to allow the use of only a fixed or open set of trait attributes within the class. Trait attributes defined by a class are automatically inherited by any subclass derived from the class.

The following example illustrates each of the features of the Traits package. These features are elaborated in the rest of this guide.

```
# all_traits_features.py --- Shows primary features of the Traits
#                            package
from enthought.traits.api import Delegate, HasTraits, Int, Trait

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

class Child(HasTraits):
    age = Int
    father = Trait(Parent) # VALIDATION: 'father' must be a Parent
                           # instance
    last_name = Delegate('father') # DELEGATION: 'last_name' is
                                   # delegated to father's
                                   # 'last_name'

    def _age_changed(self, old, new): # NOTIFICATION: This method
                                      # is called when 'age'
                                      # changes
        print 'Age changed from %s to %s ' % (old, new)

"""
>>> joe = Parent()
>>> joe.last_name = 'Johnson'
>>> moe = Child()
>>> moe.father = joe
>>> # DELEGATION in action
>>> print "Moe's last name is %s " % (moe.last_name)
Moe's last name is Johnson
>>> # NOTIFICATION in action
>>> moe.age = 10
Age changed from 0 to 10
>>> # VISUALIZATION: Displays a UI for editing moe's attributes
>>> # (if a supported GUI toolkit is installed)
```

```
>>> moe.configure_traits()

"""
```

In addition, traits can be used to define type-checked method signatures. The Traits package ensures that the arguments and return value of a method invocation match the traits defined for the parameters and return value in the method signature. This feature is described in Section 3.4.5, "Type-Checked Methods".

1.2 Background

Python does not require the data type of variables to be declared. As any experienced Python programmer knows, this flexibility has both good and bad points. The Traits package was developed to address some of the problems caused by not having declared variable types, in those cases where problems might arise. In particular, the motivation for Traits came as a direct result of work done on Chaco, an open source scientific plotting package.

Chaco provides a set of high-level plotting objects, each of which has a number of user-settable attributes, such as line color, text font, relative location, and so on. To make the objects easy for scientists and engineers to use, the attributes attempt to accept a wide variety and style of values. For example, a color-related attribute of a Chaco object might accept any of the following as legal values for the color red:

- 'red'
- 0xFF0000
- (1.0, 0.0, 0.0, 1.0)

Thus, the user might write:

```
plotitem.color = 'red'
```

In a predecessor to Chaco, providing such flexibility came at a cost:

- When the value of an attribute was used by an object internally (for example, setting the correct pen color when drawing a plot line), the object would often have to map the user-supplied

value to a suitable internal representation, a potentially expensive operation in some cases.
- If the user supplied a value outside the realm accepted by the object internally, it often caused disastrous or mysterious program behavior. This behavior was often difficult to track down because the cause and effect were usually widely separated in terms of the logic flow of the program.

So, one of the main goals of the Traits package is to provide a form of type checking that:

- Allows for flexibility in the set of values an attribute can have, such as allowing 'red', 0xFF0000 and (1.0, 0.0, 0.0, 1.0) as equivalent ways of expressing the color red.
- Catches illegal value assignments at the point of error, and provides a meaningful and useful explanation of the error and the set of allowable values.
- Eliminates the need for an object's implementation to map user-supplied attribute values into a separate internal representation.

In the process of meeting these design goals, the Traits package evolved into a useful component in its own right, satisfying all of the above requirements and introducing several additional, powerful features of its own. In projects where the Traits package has been used, it has proven valuable for enhancing programmers' ability to understand code, during both concurrent development and maintenance.

The Traits package works with version 2.2 and later of Python, and is similar in some ways to the Python *property* language feature. Standard Python properties provide the similar capabilities to the Traits package, but with more work on the part of the programmer.

2 Defining Traits: Initialization and Validation

Using the Traits package in a Python program requires the following operations:

1. Import the names you need from the Traits package (**enthought.traits.api**).

2. Define the traits you want to use.

3. Define classes derived from **HasTraits** (or a subclass of **HasTraits**), with attributes that use the traits you have defined.

In practice, steps 2 and 3 are often combined by defining traits in-line in an attribute definition. This strategy is used in many examples in this guide. However, you can also define traits independently, and reuse the trait definitions across multiple classes and attributes. Type-checked method signatures typically use independently defined traits.

In order to use trait attributes in a class, the class must inherit from the **HasTraits** class in the Traits package (or from a subclass of **HasTraits**). The following example defines a class called **Person** that has a single trait attribute **weight**, which is initialized to 150.0 and can only take floating point values.

```
 # minimal.py --- Minimal example of using traits.
from enthought.traits.api import HasTraits, Float

class Person(HasTraits):
    weight = Float(150.0)
```

In this example, the attribute named **weight** specifies that the class has a corresponding trait called **weight**. The value associated with the attribute **weight** (i.e., `Float(150.0)`) specifies a predefined trait provided with the Traits package, which requires that values assigned be of the standard Python type **float**. The value 150.0 specifies the default value of the trait.

The value associated with each class-level attribute determines the characteristics of the instance trait identified by the attribute name. For example:

```
>>> from minimal import Person
>>> joe = Person()
>>> print joe.weight
150.0
>>> joe.weight = 161.9     # OK to assign a float
>>> joe.weight = 162       # OK to assign an int
>>> joe.weight = 'average' # Error to assign a string
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "c:\wrk\src\lib\enthought\traits\trait_handlers.py", line
89, in error
    raise TraitError, ( object, name, self.info(),
value ) enthought.traits.trait_errors.TraitError: The 'weight'
trait of a Person instance must be a value of type 'float', but a
value of average was specified.
```

In this example, **joe** is an instance of the **Person** class defined in the previous example. The **joe** object has an instance attribute **weight**, whose initial value is the default value of the **Person.weight** trait (150.0), and whose assignment is governed by the **Person.weight** trait's validation rules. Assigning an integer to **weight** is acceptable because there is no loss of precision (but assigning a **float** to an **Int** trait would cause an error).

The Traits package allows creation of a wide variety of trait types, ranging from very simple to very sophisticated. The following section presents some of the simpler, more commonly used forms.

2.1 Simple Trait Definitions

Of the many methods for defining traits, some of the simplest methods include:

- Using predefined traits
- Defining by example
- Defining a list of all possible values

Using a predefined trait was illustrated at the beginning of this section by using the **Float()** function to create an attribute whose value must be a floating point value.

2.1.1 Predefined Traits

The Traits package includes a number of predefined traits for commonly used Python data types. Most of these predefined traits have factory functions that can take an argument, which becomes the default value for the trait being defined. For example:

```
account_balance = Float(10.0)
```

The predefined traits also have an appropriate built-in default value for the corresponding type. If you want to use the built-in default value for the type, you can use the predefined trait name without parentheses after it. The statements in the following example are equivalent:

```
account_balance = Float(0.0)
account_balance = Float
```

The statements are equivalent because 0.0 is the built-in default value for the predefined **Float** trait.

2.1.1.1 Predefined Traits for Simple Types

There are two categories of predefined traits corresponding to simple types: those that coerce values, and those that cast values. These categories vary in the way that they handle assigned values that do not match the type explicitly defined for the trait. However, they are similar in terms of the Python types they correspond to, and their built-in default values, as listed in Table 1.

Table 1 Predefined defaults for simple types

Coercing Trait	*Casting Trait*	*Python Type*	*Built-in Default Value*
Bool	**CBool**	Boolean	**False**
Complex	**CComplex**	Complex number	0+0j
Float	**CFloat**	Floating point number	0.0
Int	**CInt**	Plain integer	0

Coercing Trait	*Casting Trait*	*Python Type*	*Built-in Default Value*
Long	**CLong**	Long integer	0L
String	**CString**	String	''
Unicode	**CUnicode**	Unicode	u''

2.1.1.1.1 Trait Type Coercion

For trait attributes defined using the predefined "coercing" traits, if a value is assigned to a trait attribute that is not of the type defined for the trait, but it can be coerced to the required type, then the coerced value is assigned to the attribute. If the value cannot be coerced to the required type, a TraitError exception is raised. Only widening coercions are allowed, to avoid any possible loss of precision. Table 2 lists traits that coerce values, and the types that each coerces.

Table 2 Type coercions permitted for coercing traits

Trait	*Coercible Types*
Complex	Floating point number, plain integer
Float	Plain integer
Long	Plain integer
Unicode	String

2.1.1.1.2 Trait Type Casting

For trait attributes defined using the predefined "casting" traits, if a value is assigned to a trait attribute that is not of the type defined for the trait, but it can be cast to the required type, then the cast value is assigned to the attribute. If the value cannot be cast to the required type, a **TraitError** exception is raised. Internally, casting is done using the Python built-in functions for type conversion:

- **bool()**
- **complex()**
- **float()**
- **int()**
- **str()**
- **unicode()**

The following example illustrates the difference between coercing traits and casting traits.

```
>>> from enthought.traits.api import HasTraits, Float, CFloat
>>> class Person ( HasTraits ):
...    weight  = Float
...    cweight = CFloat
>>>
>>> bill = Person()
>>> bill.weight  = 180    # OK, coerced to 180.0
>>> bill.cweight = 180    # OK, cast to float(180)
>>> bill.weight  = '180'  # Error, invalid coercion
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "c:\wrk\src\lib\enthought\traits\trait_handlers.py",
line 89, in error
    raise TraitError, ( object, name, self.info(), value )
enthought.traits.trait_errors.TraitError: The 'weight' trait of a
Person instance must be a value of type 'float', but a value of
180 was specified.
>>>
>>> bill.cweight = '180'  # OK, cast to float('180')
>>> print bill.cweight
180.0
```

2.1.1.2 Other Predefined Traits

The Traits package provides a number of other predefined traits besides those for simple types, corresponding to other commonly used data types, which are listed in Table 3. Refer to the *Traits API Reference,* in the section for the module **enthought.traits.traits**, for details. Most can be used either as factory functions accepting a default value as an argument, or as simple names, which use their built-in default values.

Table 3 Predefined traits and trait factories beyond simple types

Predefined Traits	*Trait Factories*
Any	**Any**(*value* = **None**, ***metadata*)
Array	**Array**(*typecode* = **None**, *shape* = **None**, *value* = **None**, ***metadata*)

Predefined Traits	*Trait Factories*
Button	**Button**(*label* = '', *image* = **None**, *style* = 'button', *orientation* = 'vertical', *width_padding* = 7, *height_padding* = 5, ***metadata*)
Callable	
CArray	**CArray**(*typecode* = **None**, *shape* = **None**, *value* = **None**, ***metadata*)
Class	
Code	**Code**(*value* = '', *minlen* = 0, *maxlen* = **sys.maxint**, *regex* = '', ***metadata*)
Color	**Color**(**args*, ***metadata*)
Constant	**Constant**(*value*, ***metadata*)
Dict, **DictStrAny,** **DictStrBool,** **DictStrFloat,** **DictStrInt,** **DictStrList,** **DictStrLong,** **DictStrStr**	**Dict**(*key_trait* = **None**, *value_trait* = **None**, *value* = **None**, *items* = **True**, ***metadata*)
Directory	**Directory**(*value* = '', *auto_set* = **False**, ***metadata*)
Disallow	
Enum	**Enum**(**values*, ***metadata*)

Predefined Traits	*Trait Factories*
Event	`Event( *value_type, **metadata )`
false	
File	`File( value = '', filter = None, auto_set = False, **metadata )`
Font	`Font( *args, **metadata )`
Function	
generic_trait	
HTML	`HTML( value = '', **metadata )`
	`Instance( klass, args = None, kw = None, allow_none = True, **metadata )`
KivaFont	`KivaFont( *args, **metadata )`
List, ListBool, ListClass, ListComplex, ListFloat, ListFunction, ListInstance, ListInt, ListMethod, ListStr, ListThis, ListUnicode	`List( trait = None, value = None, minlen = 0, maxlen = sys.maxint, items = True, **metadata )`
Method	
missing	
Module	

Predefined Traits	*Trait Factories*
Password	**Password**(*value* = '', *minlen* = 0, *maxlen* = **sys.maxint**, *regex* = '', ***metadata*)
Property	**Property**(*fget* = **None**, *fset* = **None**, *fvalidate* = **None**, *force* = **False**, *handler* = **None**, *trait* = **None**, ***metadata*)
Python	
PythonValue	**PythonValue**(*value* = **None**, ***metadata*)
Range	**Range**(*low* = **None**, *high* = **None**, *value* = **None**, *exclude_low* = **False**, *exclude_high* = **False**, ***metadata*)
ReadOnly	
Regex	**Regex**(*value* = '', *regex* = '.*', ***metadata*)
RGBAColor	**RGBAColor**(**args*, ***metadata*)
RGBColor	**RGBColor**(**args*, ***metadata*)
self	
String	**String**(*value* = '', *minlen* = 0, *maxlen* = **sys.maxint**, *regex* = '', ***metadata*
This	

Predefined Traits	*Trait Factories*
ToolbarButton	**ToolbarButton**(*label* = '', *image* = **None**, *style* = 'toolbar', *orientation* = 'vertical', width_*padding* = 2, height_padding = 2, ***metadata*)
true	
Tuple	**Tuple**(**traits*, ***metadata*)
Type	
UIDebugger	**UIDebugger**(***metadata*)
undefined	
WeakRef	**WeakRef**(*klass* = 'enthought.traits.HasTraits', *allow_none* = **False**, ***metadata*)

2.1.1.3 This

One predefined trait that merits special explanation is **This**. Use **This** for attributes whose values must be of the same class (or a subclass) as the enclosing class. The default value is **None**.

The following is an example of using **This**:

```
# this.py --- Example of This predefined trait
from enthought.traits.api import HasTraits, This

class Employee(HasTraits):
    manager = This
```

This example defines an **Employee** class, which has a **manager** trait attribute, which accepts only other **Employee** instances as its value. It might be more intuitive to write the following:

```
# bad_self_ref.py --- Non-working example with self- referencing
#                     class definition
from enthought.traits.api import HasTraits, Trait

class Employee(HasTraits):
    manager = Trait(Employee)
```

However, the **Employee** class is not fully defined at the time that the **manager** attribute is defined. Handling this common design pattern is the main reason for providing the **This** trait.

Note that if a trait attribute is defined using **This** on one class and is referenced on an instance of a subclass, the **This** trait verifies values based on the class of the instance being referenced. For example:

```
>>> from enthought.traits.api import HasTraits, This
>>> class Employee(HasTraits):
...    manager = This
...
>>> class Executive(Employee):
...  pass
...
>>> fred = Employee()
>>> mary = Executive()
>>> fred.manager = mary
>>> # This is OK, because fred's manager can be an instance
>>> # of Employee or any subclass.
>>> mary.manager = fred
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "c:\wrk\src\lib\enthought\traits\trait_handlers.py", line
90, in error
    raise TraitError, ( object, name, self.info(), value )
enthought.traits.trait_errors.TraitError: The 'manager' trait of
an Executive instance must be an instance of the same type as the
receiver, but a value of <__main__.Employee object at 0x00994330>
was specified.
```

2.1.2 Defining By Example

In addition to predefined traits, another simple form of trait definition is defining a trait by example. To define a trait by example, call the **Trait()** function with the default value as the only argument. The **Trait()** function infers the type of the trait from the type of the specified default value. The following two statements are equivalent:

```
weight = Float(150.0)
weight = Trait(150.0)
```

A trait defined by example has the specified value as its default value, and allows only values of the same type as the default value to be assigned. The default value must be one of the simple Python data types (plain integer, long integer, floating point number, complex number, string, Unicode string, or Boolean).

2.1.3 List of Possible Values

In addition to a default value, a trait definition can specify an exhaustive set of all permitted values, as arguments to the **Trait()** function. The values must all be of simple Python data types, such as strings, integers, and floats, but they do not have to all be of the same type. This list of values can be a typical parameter list, an explicit (bracketed) list, or a variable whose type is list.

A trait defined in this fashion can accept only values that are contained in the list of permitted values. The default value is the first value specified; it is not considered a valid value for assignment unless it is repeated after the first position in the argument list.

```
>>> from enthought.traits.api import HasTraits, Str, Trait
>>> class InventoryItem(HasTraits):
...     name  = Str # String value, default is ''
...     stock = Trait(None, 0, 1, 2, 3, 'many')
...             # Enumerated list, default value is
...             #'None'
...
>>> hats = InventoryItem()
>>> hats.name = 'Stetson'

>>> print '%s: %s' % (hats.name, hats.stock)
Stetson: None

>>> hats.stock = 2      # OK
>>> hats.stock = 'many' # OK
>>> hats.stock = 4      # Error, value is not in \
>>>                     # permitted list
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "c:\wrk\src\lib\enthought\traits\trait_handlers.py", line
90, in error
    raise TraitError, ( object, name, self.info(), value )
```

```
enthought.traits.trait_errors.TraitError: The 'stock' trait of an
InventoryItem instance must be None or 0 or 1 or 2 or 3 or 'many',
but a value of 4 was specified.
```

This example defines an **InventoryItem** class, with two trait attributes, **name**, and **stock**. The **name** attribute is simply a string. The **stock** attribute has an initial value of **None**, and can be assigned the values 0, 1, 2, 3, and 'many'. The example then creates an instance of the **InventoryItem** class named **hats**, and assigns values to its attributes.

2.2 The Trait() Function

The predefined traits such as those listed in Table 3 are handy shortcuts for commonly used types. However, the Traits package provides a generic facility for defining complex or customized traits: the **Trait()** function.

The **Trait()** function has many forms, many of which are redundant with the predefined shortcut traits. For example, the simplest form **Trait**(*default_value*), is equivalent to the functions for simple types described in Section 2.1.1.1, "Predefined Traits for Simple Types".

The **Trait()** function's variety of forms allow a wide variety of flexible and powerful traits to be defined. Table 4 shows the complete set of forms understood by the **Trait()** function:

Table 4 Forms of the Trait() function

No.	*Form*	*Description*
1	`Trait(default_value)`	See Section 2.1.2, "Defining By Example", for an example.
2	`Trait(default_value, other_value2, other_value3, … )`	See Section 2.1.3, "List of Possible Values", for an example.
3	`Trait( [ default_value, other_value2, other_value3, … ] )`	Similar to Form 2, but takes an explicit list or a list variable.

No.	*Form*	*Description*
4	`Trait(`*`type`*`)`	The *type* parameter must be the name of a Python type. For simple types, the default values are the same as for the type-specific functions described in Section 2.1.1.1, "Predefined Traits for Simple Types"; for sequence types, the default value is an empty sequence. In contrast to predefined traits, this form does not cause assigned values to be coerced or cast. If the assigned value is not of the specified type, a **TraitError** is raised.
5	`Trait(`*`class`*`)`	Values must be instances of *class*, or a subclass of *class*. The default value is **None**, but **None** cannot be assigned as a value, as it is not an instance derived from *class*.
6	`Trait(None,` *`class`*`)`	Values must be instances of *class*, or a subclass of *class*. The default value is **None**, and **None** can also be assigned as a value.
7	`Trait(`*`instance`*`)`	Values must be instances of the same class as *instance*, or of a subclass of that class. The specified instance is the default value.
8	`Trait(`*`trait_handler`*`)`	The *trait_handler* parameter is class that verifies values for the trait. Refer to Section 2.5, "Trait Handlers", for details.

No.	Form	Description
9	**Trait**(*default_value*, { *type* | *constant_value*| *dictionary* | *class* | *function* | *trait_handler* | *trait* }+)	This is the most general of the forms of the **Trait()** function. Note that the notation { … | … | … }+ means a list of one or more of any of the items listed between the braces. Thus, the most general form of the function consists of a default value, followed by one or more of several possible items. A trait defined with multiple items is called a *compound* trait. When more than one item is specified, a trait value is considered valid if it meets the criteria of at least one of the items in the list.

The following is an example of a compound trait with multiple criteria.

```
multiple_criteria.py -- Example of multiple criteria in a trait
#                       definition
from enthought.traits.api import HasTraits, Trait
from types import TupleType

class Nonsense(HasTraits):
    rubbish = Trait(0.0, 0.0, 'stuff', TupleType)
```

The **Nonsense** class has a **rubbish** trait, which has a default value of 0.0, and can have any of the following values:

- The constant float value 0.0
- The constant string value 'stuff'
- Any Python tuple

Note that in this case, it is necessary to specify 0.0 twice: the first occurrence defines the default value, and the second occurrence specifies 0.0 as one of the values that can be assigned to the trait.

2.2.1 Compound Trait Parameters

The items listed as categories of values for a compound trait (in Table 4, Form 9) merit some further explanation.

- *type*—See Section 2.2.1.1, "Type".
- *constant_value*—See Section 2.2.1.2, "Constant Value".
- *dictionary*—See Section 2.3, "Mapped Traits"..
- *class*—Specifies that the trait value must be an instance of the specified class or one of its subclasses.
- *function*—See Section 2.4, "Validator Functions"..
- *trait_handler*—See Section 2.5, "Trait Handlers"..
- *trait*—Another trait object can be passed as a parameter; any value that is valid for the specified trait is also valid for the trait referencing it.

2.2.1.1 Type

A *type* parameter to the **Trait()** function can be any of the following standard Python types:

- **str** or **StringType**
- **unicode** or **UnicodeType**
- **int** or **IntType**
- **long** or **LongType**
- **float** or **FloatType**
- **complex** or **ComplexType**
- **bool** or **BooleanType**
- **list** or **ListType**
- **tuple** or **TupleType**
- **dict** or **DictType**
- **FunctionType**
- **MethodType**
- **ClassType**
- **InstanceType**
- **TypeType**
- **NoneType**

Specifying one of these types means that the value must be of the corresponding Python type.

2.2.1.2 Constant Value

A *constant_value* parameter to the **Trait()** function can be any constant belonging to one of the following standard Python types:

- **NoneType**
- **int**
- **long**
- **float**
- **complex**
- **bool**
- **str**
- **unicode**

Specifying a constant means that the trait can have the constant as a valid value. Note that Form 2 in Table 4 is a special case of the general form, in which the arguments consist of a default value followed by a series of constant values.

2.2.2 Keywords

All forms of the **Trait()** function accept both predefined and arbitrary keyword arguments. The value of each keyword argument becomes bound to the resulting trait object as the value of an attribute having the same name as the keyword. This feature lets you associate metadata with a trait.

The following predefined keywords are accepted by the **Trait()** function:

- **desc**: A string describing the intended meaning of the trait. It is used in exception messages and fly-over help in user interface trait editors.
- **label**: A string providing a human-readable name for the trait. It is used to label trait attribute values in user interface trait editors.
- **editor**: Specifies an instance of a subclass of **TraitEditor** to use when creating a user interface editor for the trait. Refer to the *Traits UI User Guide* for more information on trait editors.
- **rich_compare**: A Boolean indicating whether the basis for considering a trait attribute value to have changed is a "rich"

comparision (**True**, the default), or simple object identity (False). This attribute can be useful in cases where a detailed comparison of two objects is very expensive, or where you do not care if the details of an object change, as long as the same object is used.

For example:

```
# keywords.py --- Example of trait keywords
from enthought.traits.api import HasTraits, Trait

class Person(HasTraits):
    first_name = Trait('',
                       desc='first or personal name',
                       label='First Name')
    last_name =  Trait('',
                       desc='last or family name',
                       label='Last Name')
```

In this example, in a user interface editor for a **Person** object, the labels "First Name" and "Last Name" would be used for entry fields corresponding to the **first_name** and **last_name** trait attributes. If the user interface editor supports rollover tips, then the **first_name** field would display "first or personal name" when the user moves the mouse over it; the **last_name** field would display "last or family name" when moused over.

2.3 Mapped Traits

If the *Trait()* function is called with parameters that include one or more dictionaries, then the resulting trait is called a *mapped* trait. In practice, this means that the resulting object actually contains two attributes: one whose value is a key in the dictionary used to define the trait, and the other containing its corresponding value (i.e., the mapped or *shadow* value). The name of the shadow attribute is simply the base attribute name with an underscore appended.

Mapped traits can be used to allow a variety of user-friendly input values to be mapped to a set of internal, program-friendly values.

The following examples illustrates mapped traits that map color names to tuples representing red, green, blue, and transparency values:

```
# mapped.py --- Example of a mapped trait
from enthought.traits.api import HasTraits, Trait

standard_color = Trait ('black',
              {'black':       (0.0, 0.0, 0.0, 1.0),
               'blue':        (0.0, 0.0, 1.0, 1.0),
               'cyan':        (0.0, 1.0, 1.0, 1.0),
               'green':       (0.0, 1.0, 0.0, 1.0),
               'magenta':     (1.0, 0.0, 1.0, 1.0),
               'orange':      (0.8, 0.196, 0.196, 1.0),
               'purple':      (0.69, 0.0, 1.0, 1.0),
               'red':         (1.0, 0.0, 0.0, 1.0),
               'violet':      (0.31, 0.184, 0.31, 1.0),
               'yellow':      (1.0, 1.0, 0.0, 1.0),
               'white':       (1.0, 1.0, 1.0, 1.0),
               'transparent': (1.0, 1.0, 1.0, 0.0) } )

red_color = Trait ('red', standard_color)

class GraphicShape (HasTraits):
    line_color = standard_color
    fill_color = red_color
```

The **GraphicShape** class has two attributes: **line_color** and **fill_color**. These attributes are defined in terms of the **standard_color** trait, which uses a dictionary. The **standard_color** trait is a mapped trait, which means that each **GraphicShape** instance has two shadow attributes: **line_color_** and **fill_color_**. Any time a new value is assigned to either **line_color** or **fill_color**, the corresponding shadow attribute is updated with the value in the dictionary corresponding to the value assigned. For example:

```
>>> import mapped
>>> my_shape1 = mapped.GraphicShape()
>>> print my_shape1.line_color, my_shape1.fill_color
black red
>>> print my_shape1.line_color_, my_shape1.fill_color_
(0.0, 0.0, 0.0, 1.0) (1.0, 0.0, 0.0, 1.0)
>>> my_shape2 = mapped.GraphicShape()
>>> my_shape2.line_color = 'blue'
>>> my_shape2.fill_color = 'green'
>>> print my_shape2.line_color, my_shape2.fill_color
blue green
>>> print my_shape2.line_color_, my_shape2.fill_color_
(0.0, 0.0, 1.0, 1.0) (0.0, 1.0, 0.0, 1.0)
```

This example shows how a mapped trait can be used to create a user-friendly attribute (such as **line_color**) and a corresponding program-friendly shadow attribute (such as **line_color_**). The

shadow attribute is program-friendly because it is usually in a form that can be directly used by program logic.

There are a few other points to keep in mind when creating a mapped trait:

- If not all values passed to the **Trait()** function are dictionaries, the non-dictionary values are copied directly to the shadow attribute (i.e., the mapping used is the identity mapping).
- Assigning directly to a shadow attribute (the attribute with the trailing underscore in the name) is not allowed, and raises a **TraitError**.

2.4 Validator Functions

Rather than directly specifying legal values for a trait, you can instead specify a reference to a *validator* function as an argument to the **Trait()** function. A function reference is one of the items that permitted as an argument to **Trait()** in its most general form. The validator function determines at run time whether a value being assigned to the attribute is a legal value. Such a function must have the following prototype:

```
function( object, name, value )
```

In this prototype:

- *function* is the name of the function.
- *object* is the object whose attribute is being assigned to.
- *name* is the name of the attribute being assigned to.
- *value* is the value being assigned to the attribute.

The function is invoked whenever a value is assigned to the attribute. Normally the function does not need to know the object or attribute name being assigned to, but these parameters are provided in case the testing performed by the function is context-dependent.

The function indicates a value is valid by returning normally. The function can return the original value passed to it, or it can return

any other value, usually derived from the original value. The returned value is assigned to the attribute.

The function indicates that a value is not valid by throwing an exception. The type of exception thrown is immaterial because it is always caught by the trait mechanism and mapped into a **TraitError** exception. For example:

```
# validator.py --- Example of a validator function
from types import StringType

def bounded_string(object, name, value):
    if type(value) != StringType:
        raise TypeError
    if len(value) < 50:
        return value
    return '%s...%s' % (value[:24], value[-23:])
```

The **bounded_string()** function can be passed to the **Trait()** function to define a trait whose value must be a string, and whose value will never exceed 50 characters in length. Long strings are shortened to 50 characters by removing excess characters from the middle of the string.

In order to allow the exceptions generated by validator functions to be as descriptive as possible, you can attach a short string describing the values accepted by the function as the **info** attribute of the function.

For example, continuing the bounded_string example:

```
bounded_string.info =
    'a string no longer than 50 characters'
```

The string contained in the function's **info** attribute is merged with other information about the trait whenever an exception occurs while assigning a value to the trait attribute. If the **info** attribute is not defined, the string 'a legal value' is used in its place.

The following example shows how a validator function can be combined with other values passed to the **Trait()** function to create a compound trait, and how the validator function's **info** attribute is used when generating a TraitError exception.

```
>>> from enthought.traits.api import HasTraits, Trait
>>> from validator import bounded_string
>>>
>>> bounded_string.info = 'a string no longer than 50 characters'
>>> class DatabaseRecord(HasTraits):
...   part_desc = Trait(None, None, bounded_string)
...
>>> sprocket = DatabaseRecord()
>>> sprocket.part_desc = 0

Traceback (most recent call last):
  File "<pyshell#29>", line 1, in -toplevel-
    sprocket.part_desc = 0
  File "c:\wrk\src\lib\enthought\traits\trait_handlers.py", line
90, in error
    raise TraitError, ( object, name, self.info(), value )
TraitError: The 'part_desc' trait of a DatabaseRecord instance
must be a string no longer than 50 characters or None, but a value
of 0 was specified.
```

2.5 Trait Handlers

As an alternative to defining a trait validator function, you can use a predefined trait handler class or write your own. A trait handler is an instance of the **TraitHandler** class, or of a subclass, whose task is to verify the correctness of values assigned to object traits. When a value is assigned to an object trait that has a trait handler, the trait handler's **validate()** method checks the value, and assigns that value or a computed value, or raises a **TraitError** if the assigned value is not valid. Trait handlers have several advantages over trait validator functions, due to being classes:

- They can have constructors and state. Therefore, you can use them to create *parameterized types*.
- They can have multiple methods, whereas validator functions have only one callable interface. This feature allows more flexibility in their implementation, and allows them to handle a wider range of cases, such as interactions with other components.

The Traits package provides a number of predefined **TraitHandler** subclasses, which handle a wide variety of trait definition situations. (In fact, all of the trait definitions mentioned previously rely ultimately on one or more of the predefined **TraitHandler** subclasses.) A few of the predefined trait handler classes are

described in the following sections. For a complete list and descriptions of predefined **TraitHandler** subclasses, refer to the *Traits API Reference*, in the section on the **enthought.traits.trait_handlers** module.

You can also define your own trait handler class. For more information, refer to Section 3.1, "Custom Trait Handlers".

2.5.1 TraitString

An instance of the TraitString class ensures that a trait attribute value is a string that satisfies some additional, optional constraints. The constructor for TraitString has the following form:

```
TraitString(self, minlen=0, maxlen=sys.maxint,
    regex='')
```

In this constructor, *minlen* and *maxlen* are the minimum and maximum lengths allowed for the trait attribute's string value. The *regex* parameter is a string defining a Python regular expression that the string value must match.

The **TraitString** handler first coerces the value being assigned to a string, provided that the value is a Python **int**, **long**, **float**, **bool**, or **complex** value. Assigning values of other non-string types results in a **TraitError**. The handler then makes sure that the resulting string is within the specified length range and that it matches the specified regular expression. For example:

```
# traitstring.py --- Example of TraitString trait handler class
from enthought.traits.api import HasTraits, Trait, TraitString

class Person(HasTraits):
    name=Trait('', TraitString(maxlen=50, regex=r'^[A-Za-z]*$'))
```

This example defines a **Person** class with a **name** trait attribute, which must be a string of between 0 and 50 characters that consist only of upper and lower case letters.

2.5.2 TraitPrefixList

The **TraitPrefixList** handler accepts not only a specified set of strings as values, but also any unique prefix substring of those values. The value assigned to the trait attribute is the full string that the substring matches.

For example:

```
>>> from enthought.traits.api import HasTraits, Trait
>>> from enthought.traits.api import TraitPrefixList
>>> class Alien(HasTraits):
...   heads = Trait('one', TraitPrefixList(['one','two','three']))
...
>>> alf = Alien()
>>> alf.heads = 'o'
>>> print alf.heads
one
>>> alf.heads = 'tw'
>>> print alf.heads
two
>>> alf.heads = 't'  # Error, not a unique prefix
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "c:\wrk\src\lib\enthought\traits\trait_handlers.py", line
601, in validate
    self.error( object, name, self.repr( value ) )
  File "c:\wrk\src\lib\enthought\traits\trait_handlers.py", line
90, in error
    raise TraitError, ( object, name, self.info(), value )
enthought.traits.trait_errors.TraitError: The 'heads' trait of an
Alien instance must be 'one' or 'two' or 'three' (or any unique
prefix), but a value of 't' was specified.
```

2.5.3 TraitPrefixMap

The **TraitPrefixMap** handler combines the **TraitPrefixList** with mapped traits. Its constructor takes a parameter that is a dictionary whose keys are strings. A string is a valid value if it is a unique prefix for a key in the dictionary. The value assigned is the dictionary value corresponding to the matched key.

The following example uses **TraitPrefixMap** to define a Boolean trait that accepts any prefix of 'true', 'yes', 'false', or 'no', and maps them to 1 or 0.

```
# traitprefixmap.py --- Example of TraitPrefixMap handler
from enthought.traits.api import Trait, TraitPrefixMap

boolean_map = Trait('true', TraitPrefixMap( {
                                'true': 1,
                                'yes':  1,
                                'false': 0,
                                'no':   0 } ) )
```

3 Advanced Topics

The preceding sections provide enough information for you to use traits for manifestly-typed attributes, with initialization and validation. This section describes the advanced features of the Traits package, including custom trait handlers, notification, and delegation. It also delves into advanced aspects of initialization and validation.

3.1 Custom Trait Handlers

If you need a trait that cannot be defined using the standard set of trait handling classes, you can create your own subclass of **TraitHandler**. The constructor (i.e., **__init__** method) for your **TraitHandler** subclass can accept whatever additional information, if any, is needed to completely specify the trait. The constructor does not need to call the **TraitHandler** base class's constructor.

The only method that a custom trait handler *must* implement is **validate()**. Refer to the *Traits API Reference* for details about this function.

In many cases, it is also necessary to override one or more of the following methods in a **TraitHandler** subclass:

```
info(self)
post_setattr(self, object, name, value)
get_editor(self, trait)
```

3.1.1 Example Custom Trait Handler

To illustrate the process of creating a **TraitHandler** subclass, the following is a definition of a trait handler that only allows positive, odd integers as legal values:

```
# custom_traithandler.py --- Example of a custom TraitHandler
import types
from enthought.traits.api import TraitHandler
```

```
class TraitOddInteger(TraitHandler):


    def validate(self, object, name, value):
        if ((type(value) is types.IntType) and
            (value > 0) and ((value % 2) == 1)):
            return value
        self.error(object, name, value)

    def info(self):
        return 'a positive odd integer'
```

An application could use this new trait handler to define traits such as the following:

```
# use_custom_th.py --- Example of using a custom TraitHandler
from enthought.traits.api import HasTraits, Trait, TraitRange
from custom_traithandler import TraitOddInteger

class AnOddClass(HasTraits):
    oddball = Trait(1, TraitOddInteger())
    very_odd = Trait(-1, TraitOddInteger(),
                         TraitRange(-10, -1))
```

The following example demonstrates why the info() method returns a phrase rather than a complete sentence:

```
>>> from use_custom_th import AnOddClass
>>> odd_stuff = AnOddClass()
>>> odd_stuff.very_odd = 0
Traceback (most recent call last):
  File "test.py", line 25, in ?
    odd_stuff.very_odd = 0
  File "C:\cvsroot\traits\traits.py", line 1119, in validate
    raise TraitError, excp
traits.traits.TraitError: The 'very_odd' trait of a AnOddClass
instance must be a positive odd integer or an integer in the range
from -10 to -1, but a value of 0 was specified.
```

Note the emphasized result returned by the **info()** method, which is embedded in the exception generated by the invalid assignment.

3.2 Trait Notification

When the value of an attribute changes, other parts of the program might need to be notified that the change has occurred. The Traits

package makes this possible for trait attributes. This functionality lets you write programs using the same, powerful event-driven model that is used in writing user interfaces and for other problem domains.

Requesting trait attribute change notifications is done in one of two ways:

- Statically, by writing methods using a special naming convention in the class defining the trait attribute whose change notification is to be handled.
- Dynamically, by calling either **on_trait_change()** or **on_trait_event()** to establish (or remove) change notification handlers.

3.2.1 Static Notification

The static approach is the most convenient, but it is not always possible to use it. Writing a static change notification handler requires that, for a class whose trait attribute changes you are interested in, you write a method with a special name on that class (or a subclass).

There are two kinds of special method names that can be used for static trait attribute change notifications. One is attribute-specific, and the other applies to all trait attributes on a class.

To notify about changes to a trait attribute named *name*, define a method named *_name*_**changed()** or *_name*_**fired()**. The leading underscore indicates that attribute-specific notification handlers are normally part of a class's private API. Methods named *_name*_**fired()** are normally used with traits that are events, described in Section 3.2.3, "Trait Events".

To notify about changes to any trait attribute on a class, define a method named **_anytrait_changed()**.

Both of these types of static trait attribute notification methods are illustrated in the following example:

```
# static_notification.py --- Example of static attribute
#                            notification
from enthought.traits.api import HasTraits, Trait

class Person(HasTraits):
    weight_kg = Trait(0.0)
    height_m =  Trait(1.0)
    bmi = Trait(0.0)

    def _weight_kg_changed(self, old, new):
         print 'weight_kg changed from %s to %s ' % (old, new)
         if self.weight.kg != 0.0
             self.bmi = self.weight_kg / (self.height_m**2)

    def _anytrait_changed(self, name, old, new):
         print 'The %s trait changed from %s to %s ' \
                % (name, old, new)
"""
>>> bob = Person()
>>> bob.height_m = 1.75
The height_m trait changed from 1.0 to 1.75
>>> bob.weight_kg = 100.0
The weight_kg trait changed from 0.0 to 100.0
weight_kg changed from 0.0 to 100.0
The bmi trait changed from 0.0 to 32.6530612245
"""
```

In this example, the attribute-specific notification function is **_weight_kg_changed()**, which is called only when the **weight_kg** attribute changes. The class-specific notification handler is **_anytrait_changed()**, and is called when **weight_kg**, **height_m**, or bmi changes. Thus, both handlers are called when the **weight_kg** attribute changes. Also, the **_weight_kg_changed()** function modifies the **bmi** attribute, which causes **_anytrait_changed()** to be called for that attribute.

The arguments passed to the trait attribute change notification method depend on the method signature and on which type of static notification handler it is.

For an attribute specific notification handler, the method signatures supported are:

```
_name_changed(self)
_name_changed(self, new)
_name_changed(self, old, new)
_name_changed(self, name, old, new)
```

The method name can also be *_name*_**fired()**, with the same set of signatures.

In these signatures:

- *new* is the new value assigned to the trait attribute.
- *old* is the old value assigned to the trait attribute.
- *name* is the name of the trait attribute.

You can choose whatever method signature from this list is most convenient to use.

In the case of a non-attribute specific handler, the method signatures supported are:

```
_anytrait_changed(self)
_anytrait_changed(self, name)
_anytrait_changed(self, name, new)
_anytrait_changed(self, name, old, new)
```

The meanings for *name*, *new*, and *old* are the same as for attribute-specific notification functions.

3.2.2 Dynamic Notification

In cases where a notification handler cannot be defined on the class (or a subclass) whose trait attribute changes are to be monitored, you can use dynamic notification instead. In this case, you define a handler method, and then invoke the **on_trait_change()** or **on_trait_event()** method register that handler with the object being monitored. The handler registration methods have the following signatures:

```
obj.on_trait_change(handler, name=None,
    remove=False)
obj.on_trait_event(handler, name=None,
    remove=False)
```

The *handler* parameter specifies the function or bound method to be called whenever the *name* attribute of *obj* is modified. If *name* is **None** or omitted, handler is called whenever *any* trait attribute of *obj* is modified. If *remove* is **True** (or non-zero), then *handler* will no longer be called when the *name* (or any) trait attribute of *obj* is modified. In other words, it causes the handler to be "unhooked" from the event.

Setting up a dynamic trait attribute change notification handler is illustrated in the following example:

```
# dynamic_notification --- Example of dynamic notification
from enthought.traits.api import Float, HasTraits, Trait

class Part (HasTraits):
  cost = Trait(0.0)

class Widget (HasTraits):
  part1 = Trait(Part)
  part2 = Trait(Part)
  cost  = Float(0.0)

  def __init__(self):
    self.part1 = Part()
    self.part2 = Part()
    self.part1.on_trait_change(self.update_cost, 'cost')
    self.part2.on_trait_change(self.update_cost, 'cost')

  def update_cost(self):
    self.cost = self.part1.cost + self.part2.cost

"""
>>> w = Widget()
>>> w.part1.cost = 2.25
>>> w.part2.cost = 5.31
>>> print w.cost
7.56
"""
```

In this example, the **Widget** constructor sets up a dynamic trait attribute change notification so that its **update_cost()** method is called whenever the **cost** attribute of either its **part1** or **part2** attributes is modified.

The handler passed to **on_trait_change()** or **on_trait_event()** can have any one of the following signatures:

```
handler()
handler(new)
handler(name, new)
handler(object, name, new)
handler(object, name, old, new)
```

Unlike the static trait attribute change notification handlers, the signature of a dynamic handler does not depend upon whether the handler is attribute-specific.

3.2.3 Trait Events

The Traits package defines a special type of trait called an event. Events are created using the **Event()** function, which accepts all of the same arguments as the **Trait()** function.

There are two major differences between a normal trait and an event:

- All notification handlers associated with an event are called whenever *any* value is assigned to the event. A normal trait attribute only calls its associated notification handlers when the previous value of the attribute is different from the new value being assigned to it.
- An event does not use any storage, and in fact does not store the values assigned to it. Any value assigned to an event is reported as the *new* value to all associated notification handlers, and then immediately discarded. Because events do not retain a value, the *old* argument to a notification handler associated with an event is always the special **Undefined** object (see Section 3.2.4). Similarly, attempting to read the value of an event results in a **TraitError** exception, because an event has no value.

As an example of an event, consider:

```
# event.py --- Example of trait event
from enthought.traits.api import Event, HasTraits, List, Tuple

point_2d = Tuple(0, 0)
```

```
class Line2D(HasTraits):

    points = List(point_2d)
    line_color = RGBAColor('black')
    updated = Event

    def redraw():
        pass # Not implemented for this example

    def _points_changed():
        self.updated = True

    def _updated_fired():
        self.redraw()
```

In support of the use of events, the Traits package understands attribute-specific notification handlers with names of the form *_name*_**fired()**, with signatures identical to the *_name*_**changed()** functions. In fact, the Traits package does not check whether the trait attributes that *_name*_**fired()** handlers are applied to are actually events. The function names are simply synonyms for programmer convenience.

Similarly, a function named **on_trait_event()** can be used as a synonym for **on_trait_change()** for dynamic notification.

3.2.4 Undefined Object

Python defines a special, singleton object called **None**. The Traits package introduces an additional special, singleton object called **Undefined**.

The **Undefined** object is used to indicate that a trait attribute has not yet had a value set (i.e., its value is undefined). **Undefined** is used instead of **None**, because **None** is often used for other meanings, such as that the value is not used. In particular, when a trait attribute is first assigned a value and its associated trait notification handlers are called, **Undefined** is passed as the value of the *old* parameter to each handler, to indicate that the attribute previously had no value. Similarly, the value of a trait event is always **Undefined**.

3.3 Trait Delegation

One of the advanced capabilities of the Traits package is its ability to delegate the definition and default value of a trait attribute to another object than the one the attribute is defined on. This has many applications, especially in cases where objects are logically contained within other objects and may wish to inherit or derive some attributes from the object they are contained in or associated with. Delegation leverages the common "has-a" relationship between objects, rather than the "is-a" relationship that class inheritance provides.

Trait attributes based on delegation are defined using the **Delegate()** function, rather than the **Trait()** function.

3.3.1 Delegate() Function

The signature of the Delegate function is:

```
Delegate(delegate, prefix='', modify=False)
```

The *delegate* parameter is a string that specifies the name of the object attribute that refers to the trait's delegate. The current value of the trait attribute defined by *delegate* is used as the delegate whenever the delegate is needed. Therefore, if the attribute referenced by *delegate* changes its value, the delegation also changes to use the object referenced by the new value. The *prefix* and *modify* parameters to the **Delegate()** function specify additional information about how to do the delegation.

If *prefix* is the empty string or omitted, the delegation is to an attribute of the delegate object with the same name as the trait defined by the **Delegate()** function. Consider the following example:

```
# delegate.py --- Example of trait delegation
from enthought.traits.api import Delegate, HasTraits, Str, Trait

class Parent(HasTraits):
    first_name = Str
    last_name  = Str
```

```
class Child(HasTraits):

    first_name = Trait('')
    last_name  = Delegate('father')
    father     = Trait(Parent)
    mother     = Trait(Parent)
"""
>>> tony  = Parent(first_name='Anthony', last_name='Jones')
>>> alice = Parent(first_name='Alice', last_name='Smith')
>>> sally = Child( first_name='Sally', father=tony, mother=alice)
>>> print sally.last_name
Jones
>>> sally.last_name = 'Smith'
>>> sally.last_name = sally.mother # ERR: string expected
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "c:\wrk\src\lib\enthought\traits\trait_handlers.py", line
90, in error
    raise TraitError, ( object, name, self.info(), value )
enthought.traits.trait_errors.TraitError: The 'last_name' trait of
a Child instance must be a value of type 'str', but a value of
<__main__.Parent object at 0x009DD6F0> was specified.
"""
```

A **Child** object delegates its **last_name** attribute to its father object's **last_name** attribute. Because the *prefix* parameter was not specified in the **Delegate()** function used to define the **Child** class's **last_name** attribute, the attribute name of the delegate is the same as the original attribute name. Thus, by default, the **last_name** of a **Child** is the same as the **last_name** of its father.

Note, however, that once a value is explicitly assigned to the **last_name** attribute of a **Child**, it takes on the explicitly assigned value. This behavior is illustrated in the example when Sally's last name is set to be the same as her mother's last name. However, delegation still affects the type of values that can be assigned to the **last_name** attribute, as illustrated in the example by the attempt to assign Sally's mother (an object) as Sally's last name. The **last_name** attribute delegates the assignment to the **father** trait, whose **last_name** attribute specifies that the only legal values are strings.

3.3.1.1 Prefix Keyword

When the *prefix* parameter to the **Delegate()** function is a non-empty string, the rule for performing trait attribute look-up in the

delegated to object is modified, with the modification depending on the format of the prefix string:

- If *prefix* is a valid Python attribute name, then the original attribute name is replaced by *prefix* when performing the delegate object attribute look-up.
- If *prefix* ends with an asterisk ('*'), and is longer than one character, then *prefix*, minus the trailing asterisk, is added to the front of the original attribute name when performing the delegate object trait look-up.
- If *prefix* is equal to a single asterisk ('*'), the value of the object class's **__prefix__** attribute is added to the front of the original attribute name when performing the delegate object trait look-up.

Each of these three possibilities is illustrated in the following example:

```
# delegate_prefix.py --- Examples of Delegate() prefix parameter
from enthought.traits.api import \
    Delegate, Float, HasTraits, Str, Trait

class Parent (HasTraits):
    first_name = Str
    family_name = ''
    favorite_first_name = Str
    child_allowance = Float(1.00)

class Child (HasTraits):
    __prefix__ = 'child_'
    first_name = Delegate('mother', 'favorite_*')
    last_name  = Delegate('father', 'family_name')
    allowance  = Delegate('father', '*')
    father     = Trait(Parent)
    mother     = Trait(Parent)
```

In this example, instances of the **Child** class have three delegated trait attributes:

- **first_name**, which delegates to the **favorite_first_name** attribute of its **mother** attribute.
- **last_name**, which delegates to the **family_name** attribute of its **father** attribute.
- **allowance**, which delegates to the **child_allowance** attribute of its **father** attribute.

3.3.1.2 Modify Keyword

The final form of delegation occurs when the *modify* parameter to the **Delegate()** function is **True**. In this case, the attribute delegates to the attribute specified by the *delegate* and *prefix* parameters as before, but any changes to the attribute are made to the delegate object's attribute value, not to the object delegating the attribute. This form is useful for implementing a proxy design pattern, where the object using delegation is really a proxy for another object.

Note that when using delegation, the attribute being delegated to (such as **family_name** in the preceeding example) need not be defined by a trait. That is, the attribute that is delegated-to can be a standard Python attribute.

3.4 Initialization and Validation Revisited

The following sections present advanced topics related to the initialization and validation features of the Traits package.

- Dynamic initialization
- Overriding default values
- Reusing trait definitions
- Trait attribute definition strategies
- Type-checked methods

3.4.1 Dynamic Initialization

When you use the **Trait()** function or other trait factory functions to define traits, you specify their default values statically. You can also define a method that dynamically initializes a trait attribute the first time that the attribute value is accessed. To do this, you define a method with the following signature:

```
_name_default(self)
```

This method initializes the *name* trait attribute, returning its initial value. The method overrides any default value specified in the trait definition.

3.4.2 Overriding Default Values in a Subclass

Often, a subclass must override a trait attribute in a parent class by providing a different default value. You can specify a new default value without completely re-specifying the trait definition for the attribute. For example:

```
# override_default.py -- Example of overriding a default value for
#                        a trait attribute in a subclass
from enthought.traits.api import HasTraits, Range

class Employee(HasTraits):
    name = Str
    salary_grade = Range(value=1, low=1, high=10)

class Manager(Employee):
    salary_grade = 5
```

In this example, the **salary_grade** of the **Employee** class is a range from 1 to 10, with a default value of 1. In the **Manager** subclass, the default value of **salary_grade** is 5, but it is still a range as defined in the **Employee** class.

3.4.3 Reusing Trait Definitions

As mentioned in Section 2, "Defining Traits: Initialization and Validation", in most cases, traits are defined in-line in attribute definitions, but they can also be defined independently. A trait definition only describes the characteristics of a trait, and not the current value of a trait attribute, so it can be used in the definition of any number of attributes. For example:

```
# trait_reuse.py --- Example of reusing trait definitions
from enthought.traits.api import \
    HasTraits, Range, Trait, TraitRange
```

```
coefficient = Trait(0.0, TraitRange(-1.0, 1.0))


class quadratic(HasTraits):
    c2 = coefficient
    c1 = coefficient
    c0 = coefficient
    x  = Range(-100.0, 100.0, 0.0)
```

In this example, a trait named **coefficient** is defined externally to the class **quadratic**, which references **coefficient** in the definitions of its trait attributes **c2**, **c1**, and **c0**. Each of these attributes has a unique value, but they all use the same trait definition to determine whether a value assigned to them is valid.

3.4.4 Trait Attribute Definition Strategies

In the preceding examples in this guide, all trait attribute definitions have bound a single object attribute to a specified trait definition. This is known as explicit trait attribute definition. The Traits package supports other strategies for defining trait attributes. You can associate a category of attributes with a particular trait definition, using the trait attribute name wildcard. You can also dynamically create trait attributes that are specific to an instance, using the **add_trait()** method, rather than defined on a class. These strategies are described in the following sections.

3.4.4.1 Trait Attribute Name Wildcard

The Traits package enables you to define a category of trait attributes associated with a particular trait definition, by including an underscore ('_') as a wildcard at the end of a trait attribute name. For example:

```
# temp_wildcard.py --- Example of using a wildcard with a Trait
#                      attribute name
from enthought.traits.api import Any, HasTraits

class Person(HasTraits):
    temp_ = Any
```

This example defines a class **Person**, with a category of attributes that have names beginning with 'temp', and that are defined by the **Any** trait. Thus, any part of the program that uses a **Person** instance can reference attributes such as **tempCount**, **temp_name**, or **temp_whatever**, without having to explicitly declare these trait attributes. Each such attribute has **None** as the initial value and allows assignment of any value (because it is based on the **Any** trait).

You can even give all object attributes a default trait definition, by specifying only the wildcard character for the attribute name:

```
# all_wildcard.py --- Example of trait attribute wildcard rules
from enthought.traits.api import Any, HasTraits

class Person(HasTraits):
    _ = Any
```

In this case, all **Person** instance attributes can be created on the fly and are defined by the **Any** trait.

3.4.4.1.1 Wildcard Rules

When using wildcard characters in trait attribute names, the following rules are used to determine what trait definition governs an attribute:

1. If an attribute name exactly matches a name without a wildcard character, that definition applies.

2. Otherwise, if an attribute name matches one or more names with wildcard characters, the definition with the longest name applies.

Note that all possible attribute names are covered by one of these two rules. The base **HasTraits** class implicitly contains the attribute definition `_ = Python`. This rule guarantees that, by default, all attributes have standard Python language semantics.

These rules are demonstrated by the following example:

```
# wildcard_rules.py --- Example of trait attribute wildcard rules
from enthought.traits.api import Any, HasTraits, Int, Python
```

```
class Person(HasTraits):
    temp_count = Int(-1)
    temp_      = Any
    _          = Python
```

In this example, the **Person** class has a **temp_count** attribute, which must be an integer and which has an initial value of -1. Any other attribute with a name starting with 'temp' has an initial value of **None** and allows any value to be assigned. All other object attributes behave like normal Python attributes (i.e., they allow any value to be assigned, but they must have a value assigned to them before their first reference).

3.4.4.1.2 Disallow Object

The singleton object **Disallow** can be used with wildcards to disallow all attributes that are not explicitly defined. For example:

```
# disallow.py --- Example of using Disallow with wildcards
from enthought.traits.api import \
    Disallow, Float, HasTraits, Int, Str

class Person (HasTraits):
    name   = Str
    age    = Int
    weight = Float
    _      = Disallow
```

In this example, a Person instance has three trait attributes:

- **name**—Must be a string; its initial value is ''.
- **age**—Must be an integer; its initial value is 0.
- **weight**—Must be a float; its initial value is 0.0.

All other object attributes are explicitly disallowed. That is, any attempt to read or set any object attribute other than name, age, or weight causes an exception.

3.4.4.1.3 HasTraits Subclasses

Because the **HasTraits** class implicitly contains the attribute definition `_ = Python`, subclasses of **HasTraits** by default have

very standard Python attribute behavior for any attribute not explicitly defined as a trait attribute. However, the wildcard trait attribute definition rules make it easy to create subclasses of **HasTraits** with very non-standard attribute behavior. Two such subclasses are predefined in the Traits package: **HasStrictTraits** and **HasPrivateTraits.**

3.4.4.1.4 HasStrictTraits

This class guarantees that accessing any object attribute that does not have an explicit or wildcard trait definition results in an exception. This can be useful in cases where a more rigorous software engineering approach is employed than is typical for Python programs. It also helps prevent typos and spelling mistakes in attribute names from going unnoticed; a misspelled attribute name typically causes an exception. The definition of **HasStrictTraits** is the following:

```
class HasStrictTraits(HasTraits):
    _ = Disallow
```

HasStrictTraits can be used to create type-checked data structures, as in the following example:

```
class TreeNode(HasStrictTraits):
    left = This
    right = This
    value = Str
```

This example defines a **TreeNode** class that has three attributes: **left**, **right**, and **value**. The **left** and **right** attributes can only be references to other instances of **TreeNode** (or subclasses), while the **value** attribute must be a string. Attempting to set other types of values generates an exception, as does attempting to set an attribute that is not one of the three defined attributes. In essence, **TreeNode** behaves like a type-checked data structure.

3.4.4.1.5 HasPrivateTraits

This class is similar to **HasStrictTraits**, but allows attributes beginning with '_' to have an initial value of **None**, and to not be type-checked. This is useful in cases where a class needs private attributes, which are not part of the class's public API, to keep track

of internal object state. Such attributes do not need to be type-checked because they are only manipulated by the (presumably correct) methods of the class itself. The definition of **HasPrivateTraits** is the following:

```
class HasPrivateTraits(HasTraits):
    __ = Any
    _  = Disallow
```

These subclasses of **HasTraits** are provided as a convenience, and their use is completely optional. However, they do illustrate how easy it is to create subclasses with customized default attribute behavior if desired.

3.4.4.2 Per-Object Trait Attributes

The Traits package allows you to define dynamic trait attributes that are object-, rather than class-, specific. This is accomplished using the **add_trait()** method of the **HasTraits** class:

object.**add_trait**(*name*, *trait*)

For example:

```
# object_trait_attrs.py --- Example of per-object trait attributes
from enthought.traits.api import HasTraits, Range

class GUISlider (HasTraits):

    def __init__(self, eval=None, label='Value',
                 trait=None, min=0.0, max=1.0,
                 initial=None, **traits):
        HasTraits.__init__(self, **traits)
        if trait is None:
            if min > max:
                min, max = max, min
            if initial is None:
                initial = min
            elif not (min <= initial <= max):
                initial = [min, max][
                            abs(initial - min) >
                            abs(initial - max)]
            trait = Range(min, max, value = initial)
        self.add_trait(label, trait)
```

This example creates a **GUISlider** class, whose **__init__()** method can accept a string label and either a trait definition or minimum, maximum, and initial values. If no trait definition is specified, one is constructed based on the *max* and *min* values. A trait attribute whose name is the value of label is added to the object, using the trait definition (whether specified or constructed). Thus, the label trait attribute on the **GUISlider** object is determined by the calling code, and added in the **__init__()** method using **add_trait()**.

You can require that **add_trait()** must be used in order to add attributes to a class, by deriving the class from **HasStrictTraits** (see Section 3.4.4.1.4). When a class inherits from **HasStrictTraits**, the program cannot create a new attribute (either a trait attribute or a regular attribute) simply by assigning to it, as is normally the case in Python. In this case, **add_trait()** is the only way to create a new attribute for the class outside of the class definition.

3.4.5 Type-Checked Methods

In addition to providing type-checked attributes, the Traits package also provides the ability to create type-checked methods.

A type-checked method is created by writing a normal method definition within a class, preceded by a **method()** signature function call, as shown in the following example:

```
# type_checked_methods.py --- Example of traits-based method type
#                             checking
from enthought.traits.api import HasTraits, method, Tuple

Color = Tuple(int, int, int, int)

class Palette(HasTraits):

    method(Color, color1=Color, color2=Color)
    def blend (self, color1, color2):
        return ((color1[0] + color2[0]) / 2,
                (color1[1] + color2[1]) / 2,
                (color1[2] + color2[2]) / 2,
                (color1[3] + color2[3]) / 2 )
    method(Color, Color, Color)
    def max (self, color1, color2):
        return (max( color1[0], color2[0]),
                max( color1[1], color2[1]),
                max( color1[2], color2[2]),
                max( color1[3], color2[3]) )
```

In this example, **Color** is defined to be a trait that accepts tuples of four integer values. The **method()** signature function appearing before the definition of the **blend()** method ensures that the two arguments to **blend()** both match the **Color** trait definition, as does the result returned by **blend()**. The method signature appearing before the **max()** method does exactly the same thing, but uses positional rather than keyword arguments. When

Use of the **method()** signature function is optional. Methods not preceded by a **method()** function have standard Python behavior (i.e., no type-checking of arguments or results is performed). Also, the **method()** function can be used in classes that do not subclass from **HasTraits**, because the resulting method performs the type checking directly. And finally, when the **method()** function is used, it must directly precede the definition of the method whose type signature it defines. (However, white space is allowed.) If it does not, a **TraitError** is raised.

3.5 Useful Methods on HasTraits

The **HasTraits** class defines a number of methods, which are available to any class derived from it, i.e., any class that uses trait attributes. This section provides examples of a sample of these methods. Refer to the *Traits API Reference* for a complete list of **HasTraits** methods.

3.5.1 add_trait()

This method adds a trait attribute to an object dynamically, after the object has been created. For more information, see Section 3.4.4.2, “Per-Object Trait Attributes”.

3.5.2 clone_traits()

This method copies trait attributes from one object to another. It can copy specified attributes, all explicitly defined trait attributes, or all explicitly and implicitly defined trait attributes on the source object.

This method is useful if you want to allow a user to edit a clone of an object, so that changes are made permanent only when the user

commits them. In such a case, you might clone an object and its trait attributes; allow the user to modify the clone; and then re-clone only the trait attributes back to the original object when the user commits changes.

3.5.3 set()

This takes a list of keyword-value pairs, and sets the trait attribute corresponding to each keyword to the matching value. This shorthand is useful when a number of trait attributes need to be set on an object, or a trait attribute value needs to be set in a lambda function. For example:

```
person.set(name='Bill', age=27)
```

The statement above is equivalent to the following:

```
person.name = 'Bill'
person.age = 27
```

3.5.4 add_class_trait()

The **add_class_trait()** method is a class method, while the preceding **HasTraits** methods are instance methods. This method is very similar to the **add_trait()** instance method. The difference is that adding a trait attribute by using **add_class_trait()** is the same as having declared the trait as part of the class definition. That is, any trait attribute added using **add_class_trait()** is defined in every subsequently-created instance of the class, and in any subsequently-defined subclasses of the class. In contrast, the **add_trait()** method adds the specified trait attribute only to the object instance it is applied to.

In addition, if the name of the trait attribute ends with a '_', then a new (or replacement) prefix rule is added to the class definition, just as if the prefix rule had been specified statically in the class definition. It is not possible to define new prefix rules using the **add_trait()** method.

One of the main uses of the **add_class_trait()** method is to add trait attribute definitions that could not be defined statically as part of the body of the class definition. This occurs, for example, when two

classes with trait attributes are being defined and each class has a trait attribute that should contain a reference to the other. For the class that occurs first in lexical order, it is not possible to define the trait attribute that references the other class, since the class it needs to refer to has not yet been defined. This is illustrated in the following example:

```
# circular_definition.py --- Non-working example of mutually-
#                            referring classes
from enthought.traits.api import HasTraits, Trait

class Chicken(HasTraits):
    hatched_from = Trait(Egg)

class Egg(HasTraits):
    created_by = Trait(Chicken)
```

As it stands, this example will not run because the **hatched_from** attribute references the **Egg** class, which has not yet been defined. Reversing the definition order of the classes does not fix the problem, because then the **created_by** trait references the **Chicken** class, which has not yet been defined.

The problem can be solved using the add_class_trait() method, as shown in the following code:

```
# add_class_trait.py --- Example of mutually-referring classes
#                        using add_class_trait()
from enthought.traits.api import HasTraits, Trait

class Chicken(HasTraits):
    pass

class Egg(HasTraits):
    created_by = Trait(Chicken)

Chicken.add_class_trait('hatched_from', Egg)
```

3.6 Performance Considerations of Traits

Using traits can potentially impose a performance penalty on attribute access over and above that of normal Python attributes. For the most part, this penalty, if any, is small, because the core of

the Traits package is written in C, just like the Python interpreter. In fact, for some common cases, subclasses of **HasTraits** can actually have the same or better performance than old or new style Python classes.

However, there are a couple of performance-related factors to keep in mind when defining classes and attributes using traits:

- Whether a trait attribute delegates
- The complexity of a trait definition

If a trait attribute does not delegate, the performance penalty can be characterized as follows:

- Getting a value: No penalty (i.e., standard Python attribute access speed or faster)
- Setting a value: Depends upon the complexity of the validation tests performed by the trait definition. Many of the predefined trait handlers defined in the Traits package support fast C-level validation. For most of these, the cost of validation is usually negligible. For other trait handlers, with Python-level validation methods, the cost can be quite a bit higher.

If a trait attribute does delegate, the cases to be considered are:

- Getting the default value: Cost of following the delegation chain. The chain is resolved at the C level, and is quite fast, but its cost is linear with the number of delegation links that must be followed to find the default value for the trait.
- Getting an explicitly assigned value: No penalty (i.e., standard Python attribute access speed or faster)
- Setting: Cost of following the delegation chain plus the cost of performing the validation of the new value. The preceding discussions about delegation chain following and fast versus slow validation apply here as well.

Note that in the case where delegation modifies the delegate object, the cost of getting an attribute always includes the cost of following the delegation chain.

In a typical application scenario, where attributes are read more often than they are written, and delegation is not used, the impact

of using traits is often minimal, because the only cost occurs when attributes are assigned and validated.

The worst case scenario occurs when delegation is used heavily to provide attributes with default values that are seldom changed. In this case, the cost of frequently following delegation chains may impose a measurable performance detriment on the application. Of course, this is offset by the convenience and flexibility provided by the delegation model. As with any powerful tool, it is best to understand its strengths and weaknesses and apply that understanding in determining when use of the tool is justified and appropriate.