Traits User Manual

David C. Morrill
Janet M. Swisher

Document Version 3

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*.
3.0	11-Apr-08	Revised and updated to reflect Traits 3.0. Incremented the version number to match the software version.

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. Traits 3.0 also supports defining and using abstract interfaces, as well as adapters between interfaces
- **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. For details on the visualization features of Traits, see the *Traits UI User Guide*.

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[1] 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, Instance,\
                                 Int, Str
import enthought.traits.ui

class Parent ( HasTraits ):

    # INITIALIZATION: last_name' is initialized to '':
    last_name = Str( '' )


class Child ( HasTraits ):

    age = Int

    # VALIDATION: 'father' must be a Parent instance:
    father = Instance( Parent )

    # DELEGATION: 'last_name' is delegated to father's
'last_name':
    last_name = Delegate( 'father' )

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

[1] All code examples in this guide that include a file name are also available as examples in the `tutorials/doc_examples/examples` subdirectory of the Traits `docs` directory. You can run them individually, or view them in a tutorial program by running:
`python <Traits dir>/enthought/traits/tutor/tutor.py <Traits dir>/docs/tutorials/doc_examples`

```
# Set up the example:
joe = Parent()
joe.last_name = 'Johnson'
moe = Child()
moe.father = joe

# DELEGATION in action:
print "Moe's last name is %s " % moe.last_name
# Result:
# Moe's last name is Johnson

# NOTIFICATION in action
moe.age = 10
# Result:
# 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 6.1.5, “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 3.0 package works with version 2.4 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 (see Section 6.1.3, "Reusing Trait Definitions"). 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 attribute identified by the attribute name. For example:

```
>>> from minimal import Person
>>> # instantiate the class
>>> joe = Person()
>>> # Show the default value
>>> joe.weight
150.0
>>> # Assign new values
>>> 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:\ src\traits\enthought\traits\trait_handlers.py", line
163, 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 Predefined Traits

The Traits package includes a large number of predefined traits for commonly used Python data types. In the simplest case, you can assign the trait name to an attribute of a class derived from HasTraits; any instances of the class will have that attribute initialized to the built-in default value for the trait. For example:

```
account_balance = Float
```

This statement defines an attribute whose value must be a floating point number, and whose initial value is 0.0 (the built-in default value for Floats).

If you want to use an initial value other than the built-in default, you can pass it as an argument to the trait:

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

Most predefined traits are callable,* and can accept a default value and possibly other arguments; all that are callable can also accept metadata as keyword arguments. (See Section 2.1.2 for information on trait signatures, and see Section for information on metadata arguments.)

2.1.1 Predefined Traits for Simple Types

There are two categories of predefined traits corresponding to Python 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
Long	CLong	Long integer	0L
String	CString	String	''
Unicode	CUnicode	Unicode	u''

2.1.1.1 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

* Most callable predefined traits are classes, but a few are functions. The distinction does not make a difference unless you are trying to extend an existing predefined trait. See the Traits API Reference for details on particular traits, and see Chapter 5 for details on extending existing traits.

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

2.1.1.2 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:\src\traits\enthought\traits\trait_handlers.py",
```

```
line 163, 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
>>>
```

2.1.2 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; these predefined traits 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 simple names, which use their built-in default values, or as callables, which can take additional arguments. If the trait cannot be used as a simple name, it is omitted from the Name column of the table.

Table 3 Predefined traits beyond simple types

Name	***Callable Signature***
Any	`Any( value = None, **metadata )`

Array	`Array( typecode = None,`
	`shape = None,`
	`value = None,`
	`**metadata )`

Button	`Button( label = '',`
	`image = None,`
	`style = 'button',`
	`orientation = 'vertical',`
	`width_padding = 7,`
	`height_padding = 5,`
	`**metadata )`

Callable	n/a

CArray	`CArray( typecode = None,`
	`shape = None,`
	`value = None,`
	`**metadata )`

Class	n/a

Name	*Callable Signature*
Code	`Code( value = '', minlen = 0, maxlen = sys.maxint, regex = '', **metadata )`
Color	`Color( *args, **metadata )`
n/a	`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	n/a
n/a	`Either ( val1, val2, …, valN )`
Enum	`Enum( *values, **metadata )`
Event	`Event( *value_type, **metadata )`
Expression	`Expression(value='0', **metadata)`
false	n/a
File	`File( value = '', filter = None, auto_set = False, **metadata )`
Font	`Font( *args, **metadata )`
Function	n/a
Generic	`Generic( value = None, **metadata )`
generic_trait	n/a
HTML	`HTML( value = '', **metadata )`

Name	*Callable Signature*

Instance	`Instance( klass = None,`
	`  factory = None,`
	`  args = None,`
	`  kw = None,`
	`  allow_none = True,`
	`  adapt = 'yes',`
	`  module = None,`
	`  **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	n/a
missing	n/a
Module	n/a

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 )`
	See Section 6.4, “Property Traits”, for details.

Python	`Python ( **metadata )`
PythonValue	`PythonValue( value = None,` `  **metadata )`

Name	*Callable Signature*
Range	`Range( low = None, high = None, value = None, exclude_low = False, exclude_high = False, **metadata )`
ReadOnly	n/a
Regex	`Regex( value = '', regex = '.*', **metadata )`
RGBColor	`RGBColor( *args, **metadata )`
self	n/a
String	`String( value = '', minlen = 0, maxlen = sys.maxint, regex = '', **metadata`
This	n/a
ToolbarButton	`ToolbarButton( label = '', image = None, style = 'toolbar', orientation = 'vertical', width_padding = 2, height_padding = 2, **metadata )`
true	n/a
Tuple	`Tuple( *traits, **metadata )`
Type	n/a
UIDebugger	`UIDebugger( **metadata )`
undefined	n/a
WeakRef	`WeakRef( klass = 'enthought.traits.HasTraits', allow_none = False, **metadata )`

2.1.2.1 This and self

A couple of predefined traits that merit special explanation are This and **self**. They are intended for attributes whose values must be of the same class (or a subclass) as the enclosing class. The default value of This is None; the default value of **self** is the object containing the attribute.

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, Instance
class Employee(HasTraits):
    manager = Instance(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()
>>> # The following is OK, because fred's manager can be an
>>> # instance of Employee or any subclass.
>>> fred.manager = mary
>>> mary.manager = fred
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "c:\src\trait\enthought\traits\trait_handlers.py", line
```

```
163, 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.
```

2.1.2.2 List of Possible Values

You can define a trait whose possible values include disparate types. To do this, use the predefined Enum trait, and pass it a list of all possible values. The values must all be of simple Python data types, such as strings, integers, and floats, but they do not have to be all 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. The first item in the list is used as the default value.

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 also a valid value for assignment.

```
>>> from enthought.traits.api import Enum, HasTraits, Str
>>> class InventoryItem(HasTraits):
...    name  = Str # String value, default is ''
...    stock = Enum(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:\src\traits\enthought\traits\trait_handlers.py", line
163, 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 None, 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 Trait Metadata

Trait objects can contain metadata attributes, which fall into three categories:

- Internal attributes, which you can query but not set.
- Recognized attributes, which you can set to determine the behavior of the trait.
- Arbitrary attributes, which you can use for your own purposes.

You can specify values for recognized or arbitrary metadata attributes by passing them as keyword arguments to callable traits. 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.

2.2.1 Internal Metadata Attributes

The following metadata attributes are used internally by the Traits package, and can be queried:

- **array**: Indicates whether the trait is an array.
- **default**: Returns the default value for the trait, if known; otherwise it returns Undefined.
- **default_kind**: Returns a string describing the type of value returned by the **default** attribute for the trait. The possible values are:
  - `value`: The **default** attribute returns the actual default value.
  - `list`: A copy of the list default value.
  - `dict`: A copy of the dictionary default value.
  - `self`: The default value is the object the trait is bound to; the **default** attribute returns Undefined.
  - `factory`: The default value is created by calling a factory; the **default** attribute returns Undefined.

    - `method`: The default value is created by calling a method on the object the trait is bound to; the **default** attribute returns Undefined.
- **delegate**: The name of the attribute on this object that references the object that this object delegates to.
- **inner_traits**: Returns a tuple containing the "inner" traits for the trait. For most traits, this is empty, but for List and Dict traits, it contains the traits that define the items in the list or the keys and values in the dictionary.
- **parent**: The trait from which this one is derived.
- **prefix**: A prefix or substitution applied to the delegate attribute. See Section 4, "Deferring Trait" for details.
- **trait_type**: Returns the type of the trait, which is typically a handler derived from TraitType.
- **type**: One of the following, depending on the nature of the trait:
    - `constant`
    - `delegate`
    - `event`
    - `property`
    - `trait`

2.2.2 Recognized Metadata Attributes

The following metadata attributes are not predefined, but are recognized by HasTraits objects:

- **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.
- **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.
- **label**: A string providing a human-readable name for the trait. It is used to label trait attribute values in user interface trait editors.
- **rich_compare**: A Boolean indicating whether the basis for considering a trait attribute value to have changed is a "rich" comparison (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.

- **trait_value**: A Boolean indicating whether the trait attribute accepts values that are instances of TraitValue. The default is False. The TraitValue class provides a mechanism for dynamically modifying trait definitions. See the *Traits API Reference* for details on TraitValue. If **trait_value** is True, then setting the trait attribute to TraitValue(), with no arguments, resets the attribute to it original default value.
- **transient**: A Boolean indicating whether the trait value is persisted when the object containing it is persisted. The default value for most predefined traits is True. You can set it to False for traits whose values you know you do not want to persist. Do not set it to False on traits where it is set internally to True, as doing so is likely to create unintended consequences. See Section 6.5, "Persistence" for more information.

Other metadata attributes may be recognized by specific predefined traits.

2.2.3 Accessing Metadata Attributes

Here is an example of setting trait metadata using keyword arguments:

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

class Person(HasTraits):
    first_name = Str('',
                     desc='first or personal name',
                     label='First Name')
    last_name =  Str('',
                     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.

To get the value of a trait metadata attribute, you can use the trait() method on a HasTraits object to get a reference to a specific trait, and then access the metadata attribute:

```
# metadata.py --- Example of accessing trait metadata attributes
from enthought.traits.api import HasTraits, Int, List, Float, \
                                 Instance, Any, TraitType

class Foo( HasTraits ): pass

class Test( HasTraits ):
    i = Int(99)
    lf = List(Float)
    foo = Instance( Foo, () )
    any = Any( [1, 2, 3 ] )

t = Test()

print t.trait( 'i' ).default                      # 99
print t.trait( 'i' ).default_kind                 # value
print t.trait( 'i' ).inner_traits                 # ()
print t.trait( 'i' ).is_trait_type( Int )         # True
print t.trait( 'i' ).is_trait_type( Float )       # False

print t.trait( 'lf' ).default                     # []
print t.trait( 'lf' ).default_kind                # list
print t.trait( 'lf' ).inner_traits
         # (<enthought.traits.traits.CTrait object at
0x01B24138>,)
print t.trait( 'lf' ).is_trait_type( List )       # True
print t.trait( 'lf' ).is_trait_type( TraitType )  # True
print t.trait( 'lf' ).is_trait_type( Float )      # False
print t.trait( 'lf' ).inner_traits[0].is_trait_type( Float ) #
True

print t.trait( 'foo' ).default                    # <undefined>
print t.trait( 'foo' ).default_kind               # factory
print t.trait( 'foo' ).inner_traits               # ()
print t.trait( 'foo' ).is_trait_type( Instance )  # True
print t.trait( 'foo' ).is_trait_type( List  )     # False

print t.trait( 'any' ).default                    # [1, 2, 3]
print t.trait( 'any' ).default_kind               # list
print t.trait( 'any' ).inner_traits               # ()
print t.trait( 'any' ).is_trait_type( Any )       # True
print t.trait( 'any' ).is_trait_type( List )      # False
```

3 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 can be done in several ways:

- Dynamically, by calling on_trait_change() or on_trait_event() to establish (or remove) change notification handlers.
- Statically, by decorating methods on the class with the @on_trait_change decorator to indicate that they handle notification for specified attributes.
- Statically, by using a special naming convention for methods on the class to indicate that they handle notifications for specific trait attributes.

3.1 Dynamic Notification

Dynamic notification is useful in cases where a notification handler cannot be defined on the class (or a subclass) whose trait attribute changes are to be monitored, or if you want to monitor changes on certain instances of a class, but not all of them. To use dynamic notification, you define a handler method or function, and then invoke the on_trait_change() or on_trait_event() method to register that handler with the object being monitored. Multiple handlers can be defined for the same object, or even for the same trait attribute on the same object.

The handler registration methods have the following signatures:

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

- *handler*—Specifies the function or bound method to be called whenever the trait attributes specified by the *name* parameter are modified.

- *name*—Specifies trait attributes whose changes trigger the handler being called. If this parameter is omitted or is None, the handler is called whenever *any* trait attribute of the object is modified. The syntax supported by this parameter is discussed in Section 3.1.2.
- *remove*—If True (or non-zero), then *handler* will no longer be called when the specified trait attributes are modified. In other words, it causes the handler to be "unhooked".
- *dispatch*—String indicating the thread on which notifications must be run. In most cases, it can be omitted. See the *Traits API Reference* for details on non-default values.

3.1.1 Example of a Dynamic Notification Handler

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

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

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

class Widget (HasTraits):
  part1 = Instance(Part)
  part2 = Instance(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

# Example:
w = Widget()
w.part1.cost = 2.25
w.part2.cost = 5.31
print w.cost
# Result: 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** attribute is modified. This method then updates the **cost** attribute of the widget object.

3.1.2 The *name* Parameter

The *name* parameter of on_trait_change() and on_trait_event() provides significant flexibility in specifying the name or names of one or more trait attributes that the handler applies to. It supports syntax for specifying names of trait attributes not just directly on the current object, but also on sub-objects referenced by the current object.

The *name* parameter can take any of the following values:

- Omitted, None, or `'anytrait'`: The handler applies to any trait attribute on the object.
- A name or list of names: The handler applies to each trait attribute on the object with the specified names.
- An extended name or list of extended names: The handler applies to each trait attribute that matches the specified extended names.

3.1.2.1 Syntax

Extended names use the following syntax:

```
xname  ::= xname2['.'xname2]*

xname2 ::=
          ( xname3 | '['xname3[','xname3]*']' ) ['*']

xname3 ::= xname | ['+'|'-'][name] |
             name['?' | ('+'|'-')[name]]
```

A *name* is any valid Python attribute name.

3.1.2.2 Semantics

Table 4 Semantics of extended name notation

Pattern	*Meaning*
`item1.item2`	A trait named *item1* contains an object (or objects, if *item1* is a list or dictionary), with a trait named *item2*. Changes to either *item1* or *item2* cause a notification to be generated.
`item1:item2`	A trait named *item1* contains an object (or objects, if *item1* is a list or dictionary), with a trait named *item2*. Changes to *item2* cause a notification to be generated, while changes to *item1* do not (i.e., the ':' indicates that changes to the *link* object should not be reported.
`[item1, item2, …, itemN]`	A list that matches any of the specified items. Note that at the topmost level, the surrounding square brackets are optional.
`name?`	If the current object does not have an attribute called *name*, the reference can be ignored. If the '?' character is omitted, the current object must have a trait called *name*; otherwise, an exception is raised.
`prefix+`	Matches any trait attribute on the object whose name begins with *prefix*.
`+metadata_name`	Matches any trait on the object that has a metadata attribute called *metadata_name*.
`-metadata_name`	Matches any trait on the current object that does *not* have a metadata attribute called *metadata_name*.
`prefix+metadata_name`	Matches any trait on the object whose name begins with *prefix* and that has

Pattern	*Meaning*
	a metadata attribute called *metadata_name*.
prefix-metadata_name	Matches any trait on the object whose name begins with *prefix* and that does *not* have a metadata attribute called *metadata_name*.
+	Matches all traits on the object.
*pattern**	Matches object graphs where *pattern* occurs one or more times. This option is useful for setting up listeners on recursive data structures like trees or linked lists.

3.1.2.3 Examples

Table 5 Examples of extended name notation

Example	*Meaning*
'foo, bar, baz'	Matches *object*.**foo**, *object*.**bar**, and *object*.**baz**.
['foo', 'bar', 'baz']	Equivalent to 'foo, bar, baz', but may be useful in cases where the individual items are computed.
'foo.bar.baz'	Matches *object*.**foo**.**bar**.**baz**
'foo.[bar,baz]'	Matches *object*.**foo**.**bar** and *object*.**foo**.**baz**
'([left,right]).name*'	Matches the **name** trait of each tree node object that is linked from the **left** or **right** traits of a parent node, starting with the current object as the root node. This pattern also matches the **name** trait of the current object, as the **left** and **right** modifiers are optional.

Example	*Meaning*
`'+dirty'`	Matches any trait on the current object that has a metadata attribute named **dirty** set.
`'foo.+dirty'`	Matches any trait on *object*.**foo** that has a metadata attribute named **dirty** set.
`'foo.[bar,-dirty]'`	Matches *object*.**foo**.**bar** or any trait on *object*.**foo** that does not have a metadata attribute named **dirty** set.

For a pattern that references multiple objects, any of the intermediate (non-final) links can traits of type Instance, List, or Dict. In the case of List or Dict traits, the subsequent portion of the pattern is applied to each item in the list or value in the dictionary. For example, if **self.children** is a list, a handler set for `'children.name'` listens for changes to the **name** trait for each item in the **self.children** list.

The handler routine is also invoked when items are added or removed from a list or dictionary, because this is treated as an implied change to the item's trait being monitored.

3.1.3 Notification Handler Signatures

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)
```

These signatures use the following parameters:

- *object*—The object whose trait attribute changed.
- *name*—The attribute that changed. If one of the objects in a sequence is a List or Dict, and its membership changes, then this is the name of the trait that reference it, with `_items` appended. For example, if the handler is monitoring `'foo.bar.baz'`,

where **bar** is a List, and an item is added to **bar**, then the value of the *name* parameter is `bar_items`.
- *new*—The new value of the trait attribute that changed. For changes to List and Dict objects, this value is a TraitListEvent or TraitDictEvent whose **index**, **added**, and **removed** attributes indicate the nature of the change.
- *old*—The old value of the trait attribute that changed. For event traits, this is Undefined.

If the handler is a bound method, it also implicitly has *self* as a first argument.

3.1.4 Dynamic Handler Special Cases

In the one- and two-parameter signatures, the handler does not receive enough information to distinguish between a change to the final trait attribute being monitored, and a change to an intermediate object. In this case, the notification dispatcher attempts to map a change to an intermediate object to its effective change on the final trait attribute. This mapping is only possible if all the intermediate objects are single values (such as Instance or Any traits), and not List or Dict traits. If the change involves a List or Dict, then the notification dispatcher raises a TraitError when attempting to call a one- or two-parameter handler function, because it cannot unambiguously resolve the effective value for the final trait attribute.

Zero-parameter signature handlers receive special treatment if the final trait attribute is a List or Dict, and if the string used for the *names* parameter is not just a simple trait name. In this case, the handler is automatically called when the membership of a final List or Dict trait is changed. This behavior can be useful in cases where the handler needs to know only that some aspect of the final trait has changed. For all other signatures, the handler function must be explicitly set for the *name*`_items` trait in order to called when the membership of the *name* trait changes. (Note that the `prefix+` syntax is one way to specify both a trait name and its `_items` variant.)

This behavior for zero-parameter handlers is *not* triggered for simple trait names, to preserve compatibility with code written for versions of Traits prior to 3.0. Earlier versions of Traits required handlers to be separately set for a trait and its items, which would

result in redundant notifications under the Traits 3.0 behavior. Earlier versions also did not support the extended trait name syntax, accepting only simple trait names. Therefore, to use the "new style" behavior of zero-parameter handlers, be sure to include some aspect of the extended trait name syntax in the name specifier.

```
# list_notifier.py --- Example of zero-parameter handlers for
#                      an object containing a list
from enthought.traits.api import HasTraits, List

class Employee: pass

def a_handler(): print "A handler"
def b_handler(): print "B handler"
def c_handler(): print "C handler"

fred = Employee()
mary = Employee()
donna = Employee()

dept = Department(employees=[fred, mary])

# "Old style" name syntax
# a_handler is called only if the list is replaced:
dept.on_trait_change( a_handler, 'employees' )
# b_handler is called if the membership of the list changes:
dept.on_trait_change( b_handler, 'employees_items')

# "New style" name syntax
# c_handler is called if 'employees' or its membership change:
dept.on_trait_change( c_handler, '[employees]' )

print "Changing list items"
dept.employees[1] = donna     # Calls B and C
print "Replacing list"
dept.employees = [donna]      # Calls A and C
```

3.2 Static Notification

The static approach is the most convenient option, but it is not always possible. Writing a static change notification handler requires that, for a class whose trait attribute changes you are interested in, you write a method on that class (or a subclass). Therefore, you must know in advance what classes and attributes you want notification for, and you must be the author of those

classes. Static notification also entails that every instance of the class has the same notification handlers.

To indicate that a particular method is a static notification handler for a particular trait, you have two options:

- Apply the @on_trait_change decorator to the method.
- Give the method a special name based on the name of the trait attribute it "listens" to.

3.2.1 Handler Decorator

The most flexible method of statically specifying that a method is a notification handler for a trait is to use the @on_trait_change() decorator. The @on_trait_change() decorator is more flexible than specially-named method handlers, because it supports the very powerful extended trait name syntax (see Section 3.1.2, "The name Parameter"). You can use the decorator to set handlers on multiple attributes at once, on trait attributes of linked objects, and on attributes that are selected based on trait metadata.

3.2.1.1 Decorator Syntax

The syntax for the decorator is:

```
@on_trait_change( 'extended_trait_name' )
def any_method_name( self, … ):
    …
```

In this case, *extended_trait_name* is a specifier for one or more trait attributes, using the syntax described in Section 3.1.2, "The name Parameter".

The signatures that are recognized for "decorated" handlers are the same as those for dynamic notification handlers, as described in Section 3.1. That is, they can have an *object* parameter, because they can handle notifications for trait attributes that do not belong to the same object.

3.2.1.2 Decorator Semantics

The functionality provided by the @on_trait_change() decorator is identical to that of specially-named handlers, in that both result in a call to on_trait_change() to register the method as a notification

handler. However, the two approaches differ in *when* the call is made. Specially-named handlers are registered at class construction time; decorated handers are registered at instance creation time, prior to setting any object state.

A consequence of this difference is that the @on_trait_change() decorator causes any default initializers for the traits it references to be executed at instance construction time. In the case of specially-named handlers, any default initializers are executed lazily.

3.2.2 Specially-named Notification Handlers

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 single 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 3.3, "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, Float

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

    def _weight_kg_changed(self, old, new):
         print 'weight_kg changed from %s to %s ' % (old, new)
         if self.height_m != 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 that are passed to the trait attribute change notification method depend on the method signature and on which type of static notification handler it is.

3.2.3 Attribute-specific Handler Signatures

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. The extended trait name syntax is not supported.

Note that these signatures follow a different pattern for argument interpretation from dynamic handlers and decorated static handlers. Both of the following methods define a handler for an object's **name** trait:

```
def _name_changed( self, arg1, arg2, arg3):
    pass

@on_trait_change('name')
def some_method( self, arg1, arg2, arg3):
    pass
```

However, the interpretation of arguments to these methods differs, as shown in Table 6. Be

Table 6 Handler argument interpretation

Argument	*_name_changed*	*@on_trait_change*
arg1	*name*	*object*
arg2	*old*	*name*
arg3	*new*	*new*

3.2.4 General Static Handler Signatures

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.

3.3 Trait Events

The Traits package defines a special type of trait called an event. Events are created using the Event() 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 3.4). 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.

3.4 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.

4 Deferring Trait Definitions

One of the advanced capabilities of the Traits package is its support for trait attributes to defer their definition and value 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. Deferring leverages the common "has-a" relationship between objects, rather than the "is-a" relationship that class inheritance provides.

There are two ways that a trait attribute can defer to another object's attribute: *delegation* and *prototyping*. In delegation, the deferring attribute is a complete reflection of the delegate attribute. Both the value and validation of the delegate attribute are used for the deferring attribute; changes to either one are reflected in both. In prototyping, the deferring attribute gets its value and validation from the prototype attribute, *until the deferring attribute is explicitly changed*. At that point, while the deferring attribute still uses the prototype's validation, the link between the values is broken, and the two attributes can change independently. This is essentially a "copy on write" scheme.

The concepts of delegation and prototyping are implemented in the Traits package by two classes derived from TraitType: DelegatesTo and PrototypedFrom.[2]

4.1 DelegatesTo

The signature of the DelegatesTo initializer is:

```
def __init__(self, delegate, prefix='',
    listenable=True, **metadata)
```

The *delegate* parameter is a string that specifies the name of an attribute on the same object, which refers to the object whose attribute is deferred to; it is usually an Instance trait. The value of the delegating attribute changes whenever:

[2] Both of these class es inherit from the Delegate class. Explicit use of Delegate is deprecated, as its name and default behavior (prototyping) are incongruous.

- The value of the appropriate attribute on the delegate object changes.
- The object referenced by the trait named in the *delegate* parameter changes.
- The delegating attribute is explicitly changed.

Changes to the delegating attribute are propagated to the delegate object's attribute.

The *prefix* and *listenable* parameters to the initializer 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 DelegatesTo object. Consider the following example:

```
# delegate.py --- Example of trait delegation
from enthought.traits.api \
    import DelegatesTo, HasTraits, Instance, Str

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

class Child(HasTraits):
    first_name = Str
    last_name  = DelegatesTo('father')
    father     = Instance(Parent)
    mother     = Instance(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 = 'Cooper' # Updates delegatee
>>> print tony.last_name
Cooper
>>> sally.last_name = sally.mother # ERR: string expected
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "c:\src\trunk\enthought\traits\trait_handlers.py", line
163, 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 value to its **father** object's **last_name** attribute. Because the *prefix* parameter was not

specified in the Delegate initializer, the attribute name on the delegatee is the same as the original attribute name. Thus, the **last_name** of a Child is the same as the **last_name** of its father. When either the **last_name** of the Child or the **last_name** of the father is changed, both attributes reflect the new value.

4.2 PrototypedFrom

The signature of the PrototypedFrom initializer function is:

```
def __init__(self, prototype, prefix='',
    listenable=True, **metadata)
```

The *prototype* parameter is a string that specifies the name of an attribute on the same object, which refers to the object whose attribute is prototyped; it is usually an Instance trait. The prototyped attribute behaves similarly to a delegated attribute, until it is explicitly changed; from that point forward, the prototyped attribute changes independently from its prototype.

The *prefix* and *listenable* parameters to the initializer function specify additional information about how to do the prototyping.

4.3 Keyword Parameters

The *prefix* and *listenable* parameters of the DelegatesTo and PrototypedFrom initializer functions behave similarly for both classes.

4.3.1 Prefix Keyword

When the *prefix* parameter is a non-empty string, the rule for performing trait attribute look-up in the deferred-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 looking up the deferred-to attribute.

- 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 looking up the object attribute.
- 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 looking up the object attribute.

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

```
# prototype_prefix.py --- Examples of PrototypedFrom()
#                         prefix parameter
from enthought.traits.api import \
    PrototypedFrom, Float, HasTraits, Instance, Str

class Parent (HasTraits):
    first_name = Str
    family_name = ''
    favorite_first_name = Str
    child_allowance = Float(1.00)
class Child (HasTraits):
    __prefix__ = 'child_'
    first_name = PrototypedFrom('mother', 'favorite_*')
    last_name  = PrototypedFrom('father', 'family_name')
    allowance  = PrototypedFrom('father', '*')
    father     = Instance(Parent)
    mother     = Instance(Parent)

"""
>>> fred = Parent( first_name = 'Fred', family_name = 'Lopez', \
... favorite_first_name = 'Diego', child_allowance = 5.0 )
>>> maria = Parent(first_name = 'Maria', family_name =
'Gonzalez',\
... favorite_first_name = 'Tomas', child_allowance = 10.0 )
>>> nino = Child( father=fred, mother=maria )
>>> print '%s %s gets $%.2f for allowance' %
(nino.first_name, \ ... nino.last_name, nino.allowance)
Tomas Lopez gets $5.00 for allowance
"""
```

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

- **first_name**, which prototypes from the **favorite_first_name** attribute of its **mother** object.
- **last_name**, which prototyped from the **family_name** attribute of its **father** object.
- **allowance**, which prototypes from the **child_allowance** attribute of its **father** object.

4.3.2 Listenable Keyword

By default, you can attach listeners to deferred trait attributes, just as you can attach listeners to most other trait attributes, as described in the following section. However, implementing the notifications correctly requires hooking up complicated listeners under the covers. Hooking up these listeners can be rather more expensive than hooking up other listeners. Since a common use case of deferring is to have a large number of deferred attributes for static object hierarchies, this feature can be turned off by setting *listenable*=`False` in order to speed up instantiation.

4.4 Notification with Deferring

While two trait attributes are linked by a deferring relationship (either delegation, or prototyping before the link is broken), notifications for changes to those attributes are linked as well. When the value of a deferred-to attribute changes, notification is sent to any handlers on the deferring object, as well as on the deferred-to object. This behavior is new in Traits version 3.0. In previous versions, only handlers for the deferred-to object (the object directly changed) were notified. This behavior is shown in the following example:

```
# deferring_notification.py -- Example of notification with
#                              deferring
from enthought.traits.api \
    import HasTraits, Instance, PrototypedFrom, Str

class Parent ( HasTraits ):

    first_name = Str
    last_name  = Str

    def _last_name_changed(self, new):
        print "Parent's last name changed to %s." % new

class Child ( HasTraits ):

    father = Instance( Parent )
    first_name = Str
    last_name  = PrototypedFrom( 'father' )

    def _last_name_changed(self, new):
        print "Child's last name changed to %s." % new
```

```
"""
>>> dad = Parent( first_name='William', last_name='Chase' )
Parent's last name changed to Chase.
>>> son = Child( first_name='John', father=dad )
Child's last name changed to Chase.
>>> dad.last_name='Jones'
Parent's last name changed to Jones.
Child's last name changed to Jones.
>>> son.last_name='Thomas'
Child's last name changed to Thomas.
>>> dad.last_name='Riley'
Parent's last name changed to Riley.
>>> del son.last_name
Child's last name changed to Riley.
>>> dad.last_name='Simmons'
Parent's last name changed to Simmons.
Child's last name changed to Simmons.
"""
```

Initially, changing the last name of the father triggers notification on both the father and the son. Explicitly setting the son's last name breaks the deferring link to the father; therefore changing the father's last name does not notify the son. When the son reverts to using the father's last name (by deleting the explicit value), changes to the father's last name again affect and notify the son.

5 Custom Traits

The predefined traits such as those described in Section 2.1 are handy shortcuts for commonly used types. However, the Traits package also provides facilities for defining complex or customized traits:

- Subclassing of traits
- The Trait() factory function
- Predefined or custom trait handlers

5.1 Trait Subclassing

Starting with Traits version 3.0, most predefined traits are defined as subclasses of enthought.traits.trait_handlers.TraitType. As a result, you can subclass one of these traits, or TraitType, to derive new traits. Refer to the *Traits API Reference* to see whether a particular predefined trait derives from TraitType.

Here's an example of subclassing a predefined trait class:

```
# trait_subclass.py -- Example of subclassing a trait class
from enthought.traits.api import BaseInt

class OddInt ( BaseInt ):

    # Define the default value
    default_value = 1

    # Describe the trait type
    info_text = 'an odd integer'

    def validate ( self, object, name, value ):
        value = super(OddInt, self).validate(object, name, value)
        if (value % 2) == 1:
            return value

        self.error( object, name, value )
```

The OddInt class defines a trait that must be an odd integer. It derives from BaseInt, rather than Int, as you might initially expect. BaseInt and Int are exactly the same, except that Int has a **fast_validate** attribute, which causes it to quickly check types at the

C level, not go through the expense of executing the general validate() method.[3]

As a subclass of BaseInt, OddInt can reuse and change any part of the BaseInt class behavior that it needs to. In this case, it reuses the BaseInt class's validate() method, via the call to super() in the OddInt validate() method. Further, OddInt is related to BaseInt, which can be useful as documentation, and in programming.

You can use the subclassing strategy to define either a trait *type* or a trait *property*, depending on the specific methods and class constants that you define. A trait type uses a validate() method, while a trait property uses get() and set() methods.

5.1.1 Defining a Trait Type

The members that are specific to a trait type subclass are:

- validate() method
- post_setattr() method
- **default_value** attribute or get_default_value() method

Of these, only the validate() method *must* be overridden in trait type subclasses.

A trait type uses a validate() method to determine the validity of values assigned to the trait. Optionally, it can define a post_setattr() method, which performs additional processing after a value has been validated and assigned.

The signatures of these methods are:

```
validate ( self, object, name, value )
post_setattr ( self, object, name, value )
```

The parameters of these methods are:

- *object*—The object whose trait attribute whose value is being assigned.
- *name*—The name of the trait attribute whose value is being assigned.
- *value*—The value being assigned.

[3] All of the basic predefined traits (such as Float and Str) have a Base*Type* version that does not have the **fast_validate** attribute.

The validate() method returns either the original value or any suitably coerced or adapted value that is legal for the trait. If the value is not legal, and cannot be coerced or adapted to be legal, the method must either raise a TraitError, or calls the error() method to raise a TraitError on its behalf.

The subclass can define a default value either as a constant or as a computed value. To use a constant, set the class-level **default_value** attribute. To compute the default value, override the TraitType class's get_default_value() method.

5.1.2 Defining a Trait Property

A trait property uses get() and set() methods to interact with the value of the trait. If a TraitType subclass contains a get() method or a set() method, any definition it might have for validate() is ignored.

The signatures of these methods are:

```
get( self, object, name)
set( self, object, name, value)
```

In these signatures, the parameters are:

- *object*—The object that the property applies to.
- *name*—The name of the trait property attribute on the object.
- *value*—The value being assigned to the property.

If only a get() method is defined, the property behaves as read-only. If only a set() method is defined, the property behaves as write-only.

The get() method returns the value of the *name* property for the specified *object*. The set() method does not return a value, but will raise a TraitError if the specified *value* is not valid, and cannot be coerced or adapted to a valid value.

5.1.3 Other TraitType Members

The following members can be specified for either a trait type or a trait property:

- **info_text** attribute or info() method

- init() method
- create_editor() method

A trait must have an information string that describes the values accepted by the trait type (for example 'an odd integer'). Similarly to the default value, the subclass's information string can be either a constant string or a computed string. To use a constant, set the class-level **info_text** attribute. To compute the info string, override the TraitType class's info() method, which takes no parameters.

If there is type-specific initialization that must be performed when the trait type is created, you can override the init() method. This method is automatically called from the __init__() method of the TraitType class.

If you want to specify a default Traits UI editor for the new trait type, you can override the create_editor() method. This method has no parameters, and returns the default trait editor to use for any instances of the type.

For complete details on the members that can be overridden, refer to the Traits API Reference sections on the TraitType and BaseTraitHandler classes.

5.2 The Trait() Factory Function

The Trait() function is a generic factory for trait definitions. It 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, "Predefined Traits for Simple Types". For the full variety of forms of the Trait() function, refer to the *Traits API Reference*.

The most general form of the Trait() function is:

```
Trait(default_value, {type | constant_value    |
                      dictionary | class       |
                      function | trait_handler |
                      trait }+ )
```

The notation { | | }+ means a list of one or more of any of the items listed between the braces. Thus, this 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.

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

class Die ( HasTraits ):

    # Define a compound trait definition:
    value = Trait( 1, Range( 1, 6 ),
                   'one', 'two', 'three', 'four', 'five', 'six' )
```

The Die class has a **value** trait, which has a default value of 1, and can have any of the following values:

- An integer in the range of 1 to 6
- One of the following strings: 'one', 'two', 'three', 'four', 'five', 'six'

5.2.1 Trait () Parameters

The items listed as possible arguments to the Trait() function merit some further explanation.

- *type*—See Section 5.2.1.1, “Type”.
- *constant_value*—See Section 5.2.1.2, “Constant Value”.
- *dictionary*—See Section 5.2.2, “Mapped Traits”.
- *class*—Specifies that the trait value must be an instance of the specified class or one of its subclasses.
- *function*— A “validator” function that determines whether a value being assigned to the attribute is a legal value. Traits version 3.0 provides a more flexible approach, which is to subclass an existing trait (or TraitType) and override the validate() method.
- *trait_handler*—See Section 5.3, “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.

5.2.1.1 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 trait value must be of the corresponding Python type.

5.2.1.2 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. Passing a list of constants to the Trait() function is equivalent to using the Enum predefined trait.

5.2.2 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:

- An attribute whose value is a key in the dictionary used to define the trait.
- An attribute 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.

The concept of a mapped trait extends beyond traits defined via a dictionary. Any trait that has a shadow value is a mapped trait. For example, for the Expression trait, the assigned value must be a valid Python expression, and the shadow value is the compiled form of the expression.

5.3 Trait Handlers

In some cases, you may want to define a customized trait that is unrelated to any predefined trait behavior, or that is related to a predefined trait that happens to not be derived from TraitType. The option for such cases is to use a trait handler, either a predefined one or a custom one that you write.

A trait handler is an instance of the enthought.traits.trait_handlers. 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. Both TraitHandler and TraitType derive from BaseTraitHandler; TraitHandler has a more limited interface.

The Traits package provides a number of predefined TraitHandler subclasses A few of the predefined trait handler classes are described in the following sections. These sections also demonstrate how to define a trait using a trait handler and the Trait() factory function. 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.

5.3.1 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.
```

5.3.2 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 using the TraitPrefixMap
handler
from enthought.traits.api import Trait, TraitPrefixMap

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

5.4 Custom Trait Handlers

If you need a trait that cannot be defined using a predefined trait hander class, 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.

5.4.1 Example Custom Trait Handler

The following example defines the OddInt trait (also implemented as a trait type in Section 5.1.1) using a TraitHandler subclass.

```
# 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:\wrk\src\lib\enthought\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.

6 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

6.1 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

6.1.1 Dynamic Initialization

When you define trait attributes using predefined traits, the Trait() factory function or trait handlers, you typically 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 on the same class as the trait attribute, 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.

It is also possible to define a dynamic method for the default value in a trait type subclass (get_default_value()). However, however, using a _*name*_default() method avoids the overhead of subclassing a trait.

6.1.2 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, Str

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.

6.1.3 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

coefficient = Range(-1.0, 1.0, 0.0))

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.

6.1.4 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.

6.1.4.1 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, Int, Str

class Person ( HasTraits ):

    # Normal, explicitly defined trait:
    name = Str

    # By default, let all traits have any value:
    _ = Any

    # Except for this one, which must be an Int:
    age = Int

"""
>>> bill = Person()
>>> # These assignments should all work:
>>> bill.name      = 'William'
>>> bill.address  = '121 Drury Lane'
>>> bill.zip_code = 55212
>>> bill.age      = 49
>>> # This should generate an error (must be an Int):
>>> bill.age = 'middle age'
Traceback (most recent call last):
  File "all_wildcard.py", line 33, in <module>
    bill.age = 'middle age'
  File "c:\wrk\src\lib\enthought\traits\\trait_handlers.py", line
163, in error
    raise TraitError, ( object, name, self.info(), value )
TraitError: The 'age' trait of a Person instance must be an
integer, but a value of middle age was specified.
```

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

6.1.4.1.1 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).

6.1.4.1.2 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.

6.1.4.1.3 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.

6.1.4.1.4 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.

6.1.4.1.5 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.

6.1.4.2 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 6.1.4.1.4). 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.

6.1.5 Type-Checked Methods

In addition type-checked attributes, the Traits package 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.

6.2 Interfaces

Starting in version 3.0, the Traits package supports declaring and implementing *interfaces*. An interface is an abstract data type that defines a set of attributes and methods that an object must have to work in a given situation. The interface says nothing about what the attributes or methods do, or how they do it; it just says that they have to be there. Interfaces in Traits are similar to those in Java. They can be used to declare a relationship among classes which have similar behavior but do not have an inheritance relationship. Like Traits in general, Traits interfaces don't make anything possible that is not already possible in Python, but they can make relationships more explicit and enforced. Python programmers routinely use implicit, informal interfaces (what's known as "duck typing"). Traits allows programmers to define explicit and formal interfaces, so that programmers reading the code can more easily understand what kinds of objects are actually *intended* to be used in a given situation.

6.2.1 Defining an Interface

To define an interface, create a subclass of Interface:

```
# interface_definition.py -- Example of defining an interface
from enthought.traits.api import Interface

class IName(Interface):

    def get_name(self):
        """ Returns a string which is the name of an object. """
```

Interface classes serve primarily has documentation of the methods and attributes that the interface defines. In this case, a class that implements the IName interface must have a method named get_name(), which takes no arguments and returns a string. Do not include any implementation code in an interface declaration. However, the Traits package does not actually check to ensure that interfaces do not contain implementations.

By convention, interface names have a capital 'I' at the beginning of the name.

6.2.2 Implementing an Interface

A class declares that it implements one or more interfaces using the implements() function, which has the signature:

```
implements( interface, interface2 , … ,
    interfaceN )
```

Interface names beyond the first one are optional. The call to implements() must occur at class scope within the class definition. For example:

```
# interface_implementation.py -- Example of implementing an
#                                interface
from enthought.traits.api import HasTraits, implements, Str
from interface_definition import IName

class Person(HasTraits):
    implements(IName)

    first_name = Str( 'John' )
    last_name  = Str( 'Doe' )
```

```
    # Implementation of the 'IName' interface:
    def get_name ( self ):
        """ Returns the name of an object. """
        return ('%s %s' % ( self.first_name, self.last_name ))
```

A class can contain at most one call to implements().

In version 3.0, you can specify whether the implements() function verifies that the class calling it actually implements the interface that it says it does. This is determined by the CHECK_INTERFACES variable, which can take one of three values:

- 0 (default): Does not check whether classes implement their declared interfaces.
- 1: Verifies that classes implement the interfaces they say they do, and logs a warning if they don't.
- 2: Verifies that classes implement the interfaces they say they do, and raises an InterfaceError if they don't.

The CHECK_INTERFACES variable must be imported directly from the enthought.traits.has_traits module:

```
import enthought.traits.has_traits
enthought.traits.has_traits.CHECK_INTERFACES = 1
```

6.2.3 Using Interfaces

You can use an interface at any place where you would normally use a class name. The most common way to use interfaces is with the Instance trait:

```
>>> from enthought.traits.api import HasTraits, Instance
>>> from interface_definition import IName
>>> class Apartment(HasTraits):
...     renter = Instance(IName)
>>> from interface_implementation import Person
>>> william = Person(first_name='William', last_name='Adams')
>>> apt1 = Apartment( renter=william )
>>> print 'Renter is: ', apt1.renter.get_name()
Renter is: William Adams
```

Using an interface class with an Instance trait definition declares that the trait accepts only values that implement the specified interface. (If the assigned object does not implement the interface, the Traits package may automatically substitute an adapter object that implements the specified interface. See the following section for information on adaptation.)

6.3 Adaptation

Adaptation is the process of transforming an object that does not implement a specific interface (or set of interfaces) into an object that does. In Traits, this process is accomplished with *adapters*, which are special classes whose purpose is to adapt objects from one set of interfaces to another. Once adapter classes are defined, they are implicitly instantiated whenever they are needed to fulfill interface requirements. That is, if an Instance trait requires its values to implement interface IFoo, and an object is assigned to it which is of class Bar, which does not implement IFoo, then an adapter from Bar to IFoo is instantiated (if such an adapter class exists), and the adapter object is assigned to the trait. If necessary, a "chain" of adapter objects might be created, in order to perform the required adaptation.

6.3.1 Defining Adapters

The Traits package provides several mechanisms for defining adapter classes:

- Subclassing Adapter
- Defining an adapter class without subclassing Adapter
- Declaring a class to be an adapter externally to the class

6.3.1.1 Subclassing Adapter

The Traits package provides an Adapter class as convenience. This class streamlines the process of creating a new adapter class. It has a standard constructor that does not normally need to be overridden by subclasses. This constructor accepts one parameter, which is the object to be adapted, and assigns that object to the **adaptee** trait attribute.

As an adapter writer, the only members you need to add to a subclass of Adapter are:

- A call to implements() declaring which interfaces the adapter class implements on behalf of the object it is adapting.
- A trait attribute named **adaptee** that declares what type of object it is an adapter for. Usually, this is an Instance trait.

- Implementations of the interfaces declared in the implements() call. Usually, these methods are implemented using appropriate members on the **adaptee** object.

The following code example shows a definition of a simple adapter class:

```
# simple_adapter.py -- Example of adaptation using Adapter
from enthought.traits.api import Adapter, Instance, implements
from interface_definition import IName
from interface_implementation import Person

class PersonINameAdapter( Adapter ):

    # Declare what interfaces this adapter implements for its
    # client:
    implements( IName )

    # Declare the type of client it supports:
    adaptee = Instance( Person )

    # Implement the 'IName' interface on behalf of its client:
    def get_name ( self ):
        return ('%s %s' % ( self.adaptee.first_name,
                            self.adaptee.last_name ))
```

6.3.1.2 Creating an Adapter from Scratch

You can create an adapter class without subclassing Adapter. If so, you must provide the same information and setup that are implicitly provided by Adapter.

In particular, you must use the adapts() function instead of the implements() function, and you must define a constructor that corresponds to the constructor of Adapter. The adapts() function defines the class that contains it as an adapter class, and declares the set of interfaces that the class implements.

The signature of the adapts() function is:

```
adapts( adaptee_class, interface, interface2, … ,
     interfacen )
```

This signature is very similar to that of implements(), but adds the class being adapted as the first parameter. Interface names beyond the first one are optional.

The constructor for the adapter class must accept one parameter, which is the object being adapted, and it must save this reference in an attribute that can be used by implementation code.

The following code shows an example of implementing an adapter without subclassing Adapter.

```
# scratch_adapter.py - Example of writing an adapter from scratch
from enthought.traits.api import HasTraits, Instance, adapts
from interface_definition import IName
from interface_implementation import Person

class PersonINameAdapter ( HasTraits ):
    # Declare what interfaces this adapter implements,
    # and for what class:
    adapts( Person, IName )
    # Declare the type of client it supports:
    client = Instance( Person )

    # Implement the adapter's constructor:
    def __init__ ( self, client ):
        self.client = client

    # Implement the 'IName' interface on behalf of its client:
    def get_name ( self ):
        return ('%s %s' % ( self.client.first_name,
                            self.client.last_name ))
```

6.3.1.3 Declaring a Class as an Adapter Externally

You can declare a class to be an adapter by calling the adapts() function externally to the class definition. The class must provide the same information and setup as the Adapter class, just as in the case where adapts() is called within the class definition. That is, it must provide a constructor that accepts the object being adapted as a parameter, and it must implement the interfaces specified in the call to adapts().

In this case, signature of the adapts() function is:

```
adapts( adapter_class, adaptee_class, interface,
    interface2, … , interfacen )
```

As with implements() and the other form of adapts(), interface names beyond the first one are optional.

The following code shows this use of the adapts() function.

```
# external_adapter.py -- Example of declaring a class as an
#                        adapter externally to the class
```

```
from enthought.traits.api import adapts
from interface_definition import IName
from interface_implementation import Person

class AnotherPersonAdapter ( object ):

    # Implement the adapter's constructor:
    def __init__ ( self, person ):
        self.person = person

    # Implement the 'IName' interface on behalf of its client:
    def get_name ( self ):
        return ('%s %s' % ( self.person.first_name,
                            self.person.last_name ))

adapts( AnotherPersonAdapter, Person, IName )
```

6.3.2 Using Adapters

You define adapter classes as described in the preceding sections, but you do not explicitly create instances of these classes. The Traits package automatically creates them whenever an object is assigned to an interface Instance trait, and the object being assigned does not implement the required interface. If an adapter class exists that can adapt the specified object to the required interface, an instance of the adapter class is created for the object, and is assigned as the actual value of the Instance trait.

In some cases, no single adapter class exists that adapts the object to the required interface, but a series of adapter classes exist that together perform the required adaptation. In such cases, the necessary set of adapter objects are created, and the "last" link in the chain, the one that actually implements the required interface, is assigned as the trait value. When a situation like this arises, the adapted object assigned to the trait always contains the smallest set of adapter objects needed to adapt the original object.

6.3.3 Controlling Adaptation

Adaptation normally happens automatically when needed, and when appropriate adapter classes are available. However, the Instance trait lets you control how adaptation is performed, through its **adapt** metadata attribute. The adapt metadata attribute can have one of the following values:

- `no`—Adaptation is not allowed for this trait attribute.
- `yes`—Adaptation is allowed. If adaptation fails, an exception is raised.
- `default`—Adaptation is allowed. If adaptation fails, the default value for the trait is assigned instead.

The default value for the **adapt** metadata attribute is `yes`.

The following code is an example of an interface Instance trait attribute that uses adapt metadata.

```
# adapt_metadata.py - Example of using 'adapt' metadata
from enthought.traits.api import HasTraits, Instance
from interface_definition import IName

class Apartment( HasTraits ):
    renter = Instance( IName, adapt='no' )
```

Using this definition, any value assigned to **renter** *must* implement the IName interface. Otherwise, an exception is raised.

6.4 Property Traits

The predefined Property() trait factory function defines a Traits-based version of a Python property, with "getter" and "setter" methods. This type of trait provides a powerful technique for defining trait attributes whose values depend on the state of other object attributes. In particular, this can be very useful for creating synthetic trait attributes which are editable or displayable in a Trait UI view.

6.4.1 Property Factory Function

The Property() function has the following signature:

```
Property( fget=None, fset=None, fvalidate=None,
          force=False, handler=None, trait=None,
          **metadata )
```

All parameters are optional, including the *fget* "getter" and *fset* "setter" methods. If no parameters are specified, then the trait looks for and uses methods on the same class as the attribute that the trait is assigned to, with names of the form _get_*name* and _set_*name*, where *name* is the name of the trait attribute.

If you specify a trait as either the *fget* parameter or the *trait* parameter, that trait's handler supersedes the *handler* argument, if any. Because the *fget* parameter accepts either a method or a trait, you can define a Property trait by simply passing another trait. For example:

```
source = Property( Code )
```

This line defines a trait whose value is validated by the Code trait, and whose getter and setter methods are defined elsewhere on the same class.

If a Property trait has only a getter function, it acts as read-only; if it has only a setter function, it acts as write-only. It can lack a function due to two situations:

- A function with the appropriate name is not defined on the class.
- The *force* option is True, (which requires the Property() factory function to ignore functions on the class) and one of the access functions was not specified in the arguments.

6.4.2 Caching a Property Value

In some cases, the cost of computing the value of a property trait attribute may be very high. In such cases, it is a good idea to cache the most recently computed value, and to return it as the property value without recomputing it. When a change occurs in one of the attributes on which the cached value depends, the cache should be cleared, and the property value should be recomputed the next time its value is requested.

One strategy to accomplish caching would be to use a private attribute for the cached value, and notification listener methods on the attributes that are depended on. However, to simplify the situation, Property traits support a @cached_property decorator and **depends_on** metadata. Use @cached_property to indicate that a getter method's return value should be cached. Use **depends_on** to indicate the other attributes that the property depends on. For example:

```
# cached_prop.py - Example of @cached_property decorator
from enthought.traits.api import HasPrivateTraits, List, Int,\
                                 Property, cached_property
```

```
class TestScores ( HasPrivateTraits ):

    scores  = List( Int )
    average = Property( depends_on = 'scores' )

    @cached_property
    def _get_average ( self ):
        s = self.scores
        return (float( reduce( lambda n1, n2: n1 + n2, s, 0 ) )
                / len( s ))
```

The @cached_property decorator takes no arguments. Place it on the line preceding the property's getter method.

The **depends_on** metadata attribute accepts extended trait references, using the same syntax as the on_trait_change() method's *name* parameter, described in Section 3.1.2. As a result, it can take values that specify attributes on referenced objects, multiple attributes, or attributes that are selected based on their metadata attributes.

6.5 Persistence

In version 3.0, the Traits package provides __getstate__() and __setstate__() methods on HasTraits, to implement traits-aware policies for serialization and deserialization (i.e., pickling and unpickling).

6.5.1 Pickling HasTraits Objects

Often, you may wish to control for a HasTraits subclass which parts of an instance's state are saved, and which are discarded. A typical approach is to define a __getstate__() method that copies the object's __dict__ attribute, and deletes those items that should not be saved. This approach works, but can have drawbacks, especially related to inheritance.

The HasTraits __getstate__() method uses a more generic approach, which developers can customize through the use of traits metadata attributes, often without needing to override or define a __getstate__() method in their application classes. In particular, the HasTraits __getstate__() method discards the values of all trait attributes that have the **transient** metadata attribute set to True,

and saves all other trait attributes. So, to mark which trait values should *not* be saved, you set **transient** to True in the metadata for those trait attributes. The benefits of this approach are that you do not need to override __getstate__(), and that the metadata helps document the pickling behavior of the class.

For example:

```
# transient_metadata.py - Example of using 'transient' metadata
from enthought.traits.api import HasTraits, File, Any

class DataBase ( HasTraits ):
    # The name of the data base file:
    file_name = File

    # The open file handle used to access the data base:
    file = Any( transient = True )
```

In this example, the DataBase class's **file** trait is marked as transient because it normally contains an open file handle used to access a data base. Since file handles typically cannot be pickled and restored, the file handle should not be saved as part of the object's persistent state. Normally, the file handle would be re-opened by application code after the object has been restored from its persisted state.

6.5.2 Predefined Transient Traits

A number of the predefined traits in the Traits package are defined with **transient** set to True, so you do not need to explicitly mark them. The automatically transient traits are:

- Constant
- Event
- Read-only and write-only Property traits (See Section 6.4.1, "Property Factory Function")
- Shadow attributes for mapped traits (See Section 5.2.2, "Mapped Traits")
- Private attributes of HasPrivateTraits subclasses (See Section 6.1.4.1.5, "HasPrivateTraits")
- Delegate traits that do not have a local value overriding the delegation. Delegate traits with a local value are non-transient, i.e., they *are* serialized. (See Section 4.1, "DelegatesTo") You can

mark a Delegate trait as transient if you do not want its value to ever be serialized.

6.5.3 Overriding __getstate__()

In general, try to avoid overriding __getstate__() in subclasses of HasTraits. Instead, mark traits that should not be pickled with `transient = True` metadata.

However, in cases where this strategy is insufficient, use the following pattern to override __getstate__() to remove items that should not be persisted:

```
def __getstate__ ( self ):
    state = super( XXX, self ).__getstate__()

    for key in [ 'foo', 'bar' ]:
        if state.has_key( key ):
            del state[ key ]

    return state
```

6.5.4 Unpickling HasTraits Objects

The __setstate__() method of HasTraits differs from the default Python behavior in one important respect: it explicitly sets the value of each attribute using the values from the state dictionary, rather than simply storing or copying the entire state dictionary to its __dict__ attribute. While slower, this strategy has the advantage of generating trait change notifications for each attribute. These notifications are important for classes that rely on them to ensure that their internal object state remains consistent and up to date.

6.5.5 Overriding __setstate__()

You may wish to override the HasTraits __setstate__() method, for example for classes that do not need to receive trait change notifications, and where the overhead of explicitly setting each attribute is undesirable. You can override __setstate__() to update the object's __dict__ directly. However, in such cases, it is important ensure that trait notifications are properly set up so that later change notifications are handled. You can do this in two ways:

- Call the __setstate__() super method (for example, with an empty state dictionary).
- Call the HasTraits class's private _init_trait_listeners() method; this method has no parameters and does not return a result.

6.6 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 sampling of these methods. Refer to the *Traits API Reference* for a complete list of HasTraits methods.

6.6.1 add_trait()

This method adds a trait attribute to an object dynamically, after the object has been created. For more information, see Section 6.1.4.2, "Per-Object Trait Attributes".

6.6.2 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.

6.6.3 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
```

6.6.4 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)
```

6.7 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.