Traits 4 User Manual

Release 4.5.0-rc.1

Enthought, Inc.

April 03, 2014

CHAPTER 1

User Reference

1.1 Traits 4 User Manual

1.1.1 Traits 4 User Manual

Authors David C. Morrill, Janet M. Swisher

Version Document Version 4

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

1.1.2 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. 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*
- TraitsUI User Manual
- Traits Technical Notes

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 *explicitly 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 4.0 also supports defining and using abstract interfaces, as well as adapters between interfaces.
- **Deferral**: The value of a trait attribute can be contained either in the defining object or in another object that is *deferred 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 TraitsUI User Manual.

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 traits.api import Delegate, HasTraits, Instance,\
                               Int, Str
class Parent ( HasTraits ):

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


class Child ( HasTraits ):
```

[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>/traits/tutor/tutor.py <Traits dir>/docs/tutorials/doc_examples

```
    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 )

# 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()
```

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 4.0 package works with version 2.7 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.

1.1.3 Defining Traits: Initialization and Validation

Using the Traits package in a Python program involves the following steps:

1. Import the names you need from the Traits package 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 *Reusing Trait Definitions*).

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 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 <module>
  File "c:\svn\ets3\traits\enthought\traits\trait_handlers.py", line 175,
in error value )
traits.trait_errors.TraitError: The 'weight' trait of a Person
instance must be a float, but a value of 'average' <type 'str'> 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.

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, [2] and can accept a default value and possibly other arguments; all that are callable can also accept metadata as keyword arguments. (See *Other Predefined Traits* for information on trait signatures, and see *Trait Metadata* for information on metadata arguments.)

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 the following table.

[2] 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.

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
Str	CStr	String	''
Unicode	CUnicode	Unicode	u''

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

Type coercions permitted for coercing traits

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

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 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 <module>
  File "c:\svn\ets3\traits\enthought\traits\trait_handlers.py", line 175,
```

```
in error value )
traits.trait_errors.TraitError: The 'weight' trait of a Person
instance must be a float, but a value of '180' <type 'str'> was specified.
>>> bill.cweight = '180'  # OK, cast to float('180')
>>> print bill.cweight
180.0
>>>
```

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 the following table. Refer to the *Traits API Reference*, in the section for the module 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.

Predefined traits beyond simple types

Name	Callable Signature
Any	Any([*value* = None, ***metadata*])
Array	Array([*dtype* = None, *shape* = None,
Button	Button([*label* = '', *image* = None, *styl*
Callable	Callable([*value* = None, ***metadata*]
CArray	CArray([*dtype* = None, *shape* = None
Class	Class([*value*, ***metadata*])
Code	Code([*value* = '', *minlen* = 0, *maxlen*
Color	Color([**args*, ***metadata*])
CSet	CSet([*trait* = None, *value* = None, *ite*
n/a	Constant(*value*[, ****metadata*])
Dict, DictStrAny, DictStrBool, DictStrFloat, DictStrInt, DictStrList, DictStrLong, DictStrStr	Dict([*key_trait* = None, *value_trait* =
Directory	Directory([*value* = '', *auto_set* = Fals
Disallow	n/a
n/a	Either(*val1*[, *val2*, ..., *valN*, ***meta*
Enum	Enum(*values*[, ****metadata*])
Event	Event([*trait* = None, ***metadata*])
Expression	Expression([*value* = '0', ***metadata*]
false	n/a
File	File([*value* = '', *filter* = None, *auto_s*
Font	Font([**args*, ***metadata*])
Function	Fu
Generic	Ge
generic_trait	n/a
HTML	HT
Instance	Ins
List, ListBool, ListClass, ListComplex, ListFloat, ListFunction, ListInstance, ListInt, ListMethod, ListStr, ListThis, ListUnicode	Li
Method	M
Module	M
Password	Pa
Property	Pr
Python	Py
PythonValue	Py

Continued on next page

Table 1.1 – continued from previous page

Name	Callable Signature	
Range		Ra
ReadOnly		Re
Regex		Re
RGBColor		RG
self		n/a
Set		Se
String		St
This		n/a
ToolbarButton		To
true		n/a
Tuple		Tu
Type		Ty
undefined		n/a
UStr		US
UUID [3]		UU
WeakRef		We

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 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 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 on which it was defined. For example:

```
>>> from traits.api import HasTraits, This
>>> class Employee(HasTraits):
...     manager = This
...
>>> class Executive(Employee):
...  pass
...
>>> fred = Employee()
>>> mary = Executive()
```

[3] Available in Python 2.5.

```
>>> # The following is OK, because fred's manager can be an
>>> # instance of Employee or any subclass.
>>> fred.manager = mary
>>> # This is also OK, because mary's manager can be an Employee
>>> mary.manager = fred
```

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 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 <module>
  File "c:\svn\ets3\traits_3.0.3\enthought\traits\trait_handlers.py", line 175,
in error value )
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 <type
'int'> 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.

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.

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 *Deferring Trait Definitions* 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`

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 TraitsUI User Manual 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 that the trait value is not persisted when the object containing it is persisted. The default value for most predefined traits is False (the value will be persisted if its container is). You can set it to True for traits whose values you know you do not want to persist. Do not set it to True on traits where it is set internally to False, as doing so is likely to create unintended consequences. See *Persistence* for more information.

Other metadata attributes may be recognized by specific predefined traits.

Accessing Metadata Attributes

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

```
# keywords.py --- Example of trait keywords
from 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 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
        # (<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
```

1.1.4 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.

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:

on_trait_change(*handler*[, *name=None*, *remove=False*, *dispatch='same'*])

on_trait_event(*handler*[, *name=None*, *remove=False*, *dispatch='same'*])

In these signatures:

- *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 *The name Parameter*.

- *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.

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 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.

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.

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.

Semantics

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* trigger a notification.
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* trigger a notification, while changes to *item1* do not (i.e., the ':' indicates that changes to the link object are not 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.
item[]	A trait named *item* is a list. Changes to *item* or to its members triggers a notification.
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 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.

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**
`'foo[]'`	Matches a list trait on *object* named **foo**.
`'([left,right])*'`	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.
`'+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 be 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.

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 references 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 is a list of items that were added.
- *old*: The old value of the trait attribute that changed. For changes to List and Dict object, this is a list of items that were deleted. For event traits, this is Undefined.

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

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 *name* 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*+ and *item*[] syntaxes are both ways 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 traits.api import HasTraits, List

class Employee: pass

class Department( HasTraits ):
    employees = List(Employee)

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
```

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.

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 *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.

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 *The name Parameter*.

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

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 handlers 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.

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 *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 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.

Attribute-specific Handler Signatures

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

_name_changed()

_name_changed(*new*)

_name_changed(*old*, *new*)

_name_changed(*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. For List and Dict objects, this is a list of the items that were added.
- *old* is the old value assigned to the trait attribute. For List and Dict objects, this is a list of the items that were deleted.

- *name* is the name of the trait attribute. The extended trait name syntax is not supported. [4]

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 the following table.

Handler argument interpretation

Argument	*_name*_changed	@on_trait_change
arg1	*name*	*object*
arg2	*old*	*name*
arg3	*new*	*new*

General Static Handler Signatures

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

_anytrait_changed()

_anytrait_changed(*name*)

_anytrait_changed(*name*, *new*)

_anytrait_changed(*name*, *old*, *new*)

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

Trait Events

The Traits package defines a special type of trait called an event. Events are instances of (subclasses of) the Event class.

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 *Undefined Object*). 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:

[4] For List and Dict trait attributes, you can define a handler with the name *_name*_items_changed(), which receives notifications of changes to the contents of the list or dictionary. This feature exists for backward compatibility. The preferred approach is to use the @on_trait_change decorator with extended name syntax. For a static *_name*_items_changed() handler, the *new* parameter is a TraitListEvent or TraitDictEvent whose **index**, **added**, and **removed** attributes indicate the nature of the change, and the *old* parameter is Undefined.

```
# event.py --- Example of trait event
from 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(self):
        pass  # Not implemented for this example

    def _points_changed(self):
        self.updated = True

    def _updated_fired(self):
        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.

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.

1.1.5 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. [5]

DelegatesTo

class DelegatesTo (*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:

- 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 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 )
traits.trait_errors.TraitError:  The 'last_name' trait of a
Parent instance must be a string, but a value of <__main__.Parent object at
0x014D6D80> <class '__main__.Parent'> was specified.
"""
```

[5] 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.

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 DelegatesTo 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.

PrototypedFrom

class `PrototypedFrom`(*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.

Keyword Parameters

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

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 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.

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.

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 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 notif

1.1.6 Custom Traits

The predefined traits such as those described in *Predefined Traits* 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

Trait Subclassing

Starting with Traits version 3.0, most predefined traits are defined as subclasses of 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 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. [6]

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.

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`(*object*, *name*, *value*)

`post_setattr`(*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.

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.

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`(*object*, *name*)

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

set(*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.

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 TraitsUI 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.

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 *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:

traits.traits.**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 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'

Trait () Parameters

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

- *type*: See *Type*.
- *constant_value*: See *Constant Value*.
- *dictionary*: See *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 *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.

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.

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.

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 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.

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 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 traits.trait_handlers module.

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 traits.api import HasTraits, Trait
>>> from 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 <module>
  File "c:\svn\ets3\traits_3.0.3\enthought\traits\trait_handlers.py", line 1802,
 in validate self.error( object, name, value )
  File "c:\svn\ets3\traits_3.0.3\enthought\traits\trait_handlers.py", line 175,
in error value )
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'
 <type 'str'> was specified.
```

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 traits.api import Trait, TraitPrefixMap

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

Custom Trait Handlers

If you need a trait that cannot be defined using a predefined trait handler 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.

Example Custom Trait Handler

The following example defines the OddInt trait (also implemented as a trait type in *Defining a Trait Type*) using a TraitHandler subclass.

```
# custom_traithandler.py --- Example of a custom TraitHandler
import types
from 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 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 an AnOddClass instance
must be **a positive odd integer** or -10 <= an integer <= -1, but a value
of 0 <type 'int'> was specified.
```

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

1.1.7 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

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

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 a name based on the name of the trait attribute:

_name_default()

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, using a *_name*_default() method avoids the overhead of subclassing a trait.

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 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.

Reusing Trait Definitions

As mentioned in *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 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.

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.

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 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 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' <type 'str'> was specified.
"""
```

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

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 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).

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 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.

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.

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.

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.

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:

add_trait(*name*, *trait*)

For example:

```
# object_trait_attrs.py --- Example of per-object trait attributes
from 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 *HasStrictTraits*). 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.

Interfaces

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.

Defining an Interface

To define an interface, create a subclass of Interface:

```
from 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 as 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.

Implementing an Interface

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

`traits.has_traits.`**`provides`**(*interface*[, *interface2*, ..., *interfaceN*])

Interface names beyond the first one are optional. As for all class decorators, the call to provides must occur just before the class definition. For example:

```
from traits.api import HasTraits, Interface, provides, Str

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

@provides(IName)
class Person(HasTraits):

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

    # Implementation of the 'IName' interface:
    def get_name ( self ):
        ''' Returns the name of an object. '''
        name = '{first} {last}'
        return name.format(name=self.first_name, last=self.last_name)
```

You can specify whether the provides() decorator 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 traits.has_traits module:

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

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` or `Supports` traits:

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

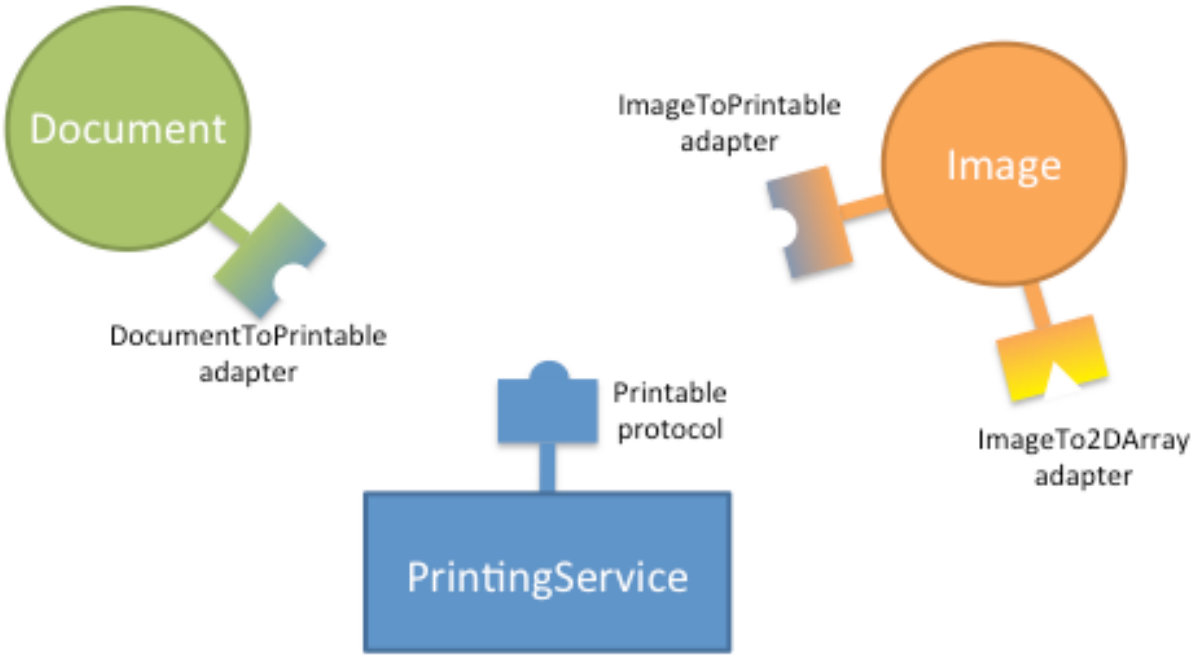
Using an interface class with an `Instance` trait definition declares that the trait accepts only values that implement the specified interface. Using the `Supports` traits, if the assigned object does not implement the interface, the Traits package may automatically substitute an adapter object that implements the specified interface. See *Adaptation* for more information.

Adaptation

The adaptation features of Traits have been rewritten in v. 4.4.0 . See the migration guide below for details regarding changes in API.

Adaptation is the process of transforming an object that does not implement a specific interface needed by a client into an object that does. In the adapter pattern, an object is wrapped in a second object, the *adapter*, that implements the target interface.

Adaptation enables a programming style in which each component or service in an application defines an interface through which it would like to receive information. Objects that need to communicate with the component declare an adapter for that interface, as illustrated in the figure below.



Adaptation allows decoupling the data model from the application components and services: introducing a new component in the application should not require modifying the data objects!

Traits provides a package to make this pattern easy and automatic: In the `traits.adaptation` package, adapters from a protocol (type or interface) to another can be registered with a manager object. HasTraits classes can either explicitly request to adapt an object to a protocol, or they can define special traits that automatically invoke the adaptation manager whenever it is necessary.

For example, if a `Supports` trait requires its values to implement interface `IPrintable`, and an object is assigned to it which is of class `Image`, which does not implement `IPrintable`, then Traits looks for an adapter from `Image` to `IPrintable`, and if one exists 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.

Main features

The main features of the `traits.adaptation` package are:

- Support for Python classes, ABCs, and traits `Interface`s

  Protocols can be specified using any of those.

- Chaining of adapters

  Adapters can be chained, i.e., an object can be adapted to a target protocol as long as there is a sequence of adapters that can be used to transform it.

- Conditional adaptation

  Adaptation of an object to a protocol can be conditional, i.e. it may succeed or fail depending on the state of the object.

- Lazy loading

  The classes for the adapter, the origin, and the target protocols can be specified as strings, and are only loaded if they are required.

Note on terminology

To avoid confusion, let's define two terms that we will use all the time:

- We say that a class *provides* a protocol if it is a subclass of the protocol, or if it implements the protocol (if it is an interface)
- We say that a class *supports* a protocol if it provides the protocol or an adapter object can be built that provides the protocol

Defining Adapters

The `Adapter` class The Traits package provides two classes for defining adapters, one for Traits adapters, `Adapter`, and one for for pure-Python adapters, `PurePythonAdapter`. These classes streamline the process of creating a new adapter class. They have 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 an `adaptee` attribute (a trait in the case of `Adapter`).

As an adapter writer, you need to take care of the following:

- Declare which interfaces the adapter class implements on behalf of the object it is adapting. For example, if we are working with Traits `Interface`s, the adapter would be decorated with the `provides()` decorator. In the case of Python ABCs, the class would be a subclass of the abstract base class, or be registered with it.
- Implement the methods defined in the interfaces declared in the previous step. Usually, these methods are implemented using appropriate members on the adaptee object.
- For Traits adapters, define a trait attribute named **adaptee** that declares what type of object it is an adapter for. Usually, this is an `Instance` trait.

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

```
from traits.api import Adapter, Instance, provides

# Declare what interfaces this adapter implements for its client
@provides(IName)
class PersonToIName(Adapter):
```

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

    # Implement the 'IName' interface on behalf of its client:
    def get_name ( self ):
        name = '{first} {last}'.format(first=self.adaptee.first_name,
                                       last=self.adaptee.last_name)
        return name
```

Registering adapters Once an adapter class has been defined, it has to be registered with the adaptation manager using the `register_factory()` function.

The signature of `register_factory()` is:

`traits.adaptation.api.`**`register_factory`**(*adapter_class*, *from_protocol*, *to_protocol*)

The `register_factory()` function takes as first argument the adapter class (or an *adapter factory*), followed by the protocol to be adapted (the one provided by the adaptee, `from_protocol`), and the protocol that it provides (`to_protocol`).

This is the example from the previous section, were the adapter is registered:

```
from traits.adaptation.api import Adapter, Instance, provides

# Declare what interfaces this adapter implements for its client
@provides(IName)
class PersonToIName(Adapter):

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

    # Implement the 'IName' interface on behalf of its client:
    def get_name ( self ):
        name = '{first} {last}'.format(first=self.adaptee.first_name,
                                       last=self.adaptee.last_name)
        return name

# ... somewhere else at application startup.
register_factory(PersonToIName, Person, IName)
```

Adapter factories, and conditional adaptation

The first argument to the `register_factory()` function needs not be an adapter *class*, it can be, more generally, an adapter *factory*.

An adapter factory can be any callable that accepts one positional argument, the adaptee object, and returns an adapter or None if the adaptation was not possible. Adapter factories allow flexibility in the adaptation process, as the result of adaptation may vary depending on the state of the adaptee object.

Conditional adaptation A common use of adapter factories is to allow adaptation only if the state of the adaptee object allows it. The factory returns an adapter object if adaptation is possible, or None if it is not.

In the following example, a `numpy.ndarray` object can be adapted to provide an `IImage` protocol only if the number of dimensions is 2. (For illustration, this example uses Python ABCs rather than Traits Interfaces.)

```
import abc
import numpy
from traits.api import Array, HasTraits
from traits.adaptation.api import adapt, Adapter, register_factory

class ImageABC(object):
    __metaclass__ = abc.ABCMeta

class NDArrayToImage(Adapter):
    adaptee = Array

# Declare that NDArrayToImage implements ImageABC.
ImageABC.register(NDArrayToImage)


def ndarray_to_image_abc(adaptee):
    """ An adapter factory from numpy arrays to the ImageABC protocol."""
    if adaptee.ndim == 2:
        return NDArrayToImage(adaptee=adaptee)
    return None

# ... somewhere else at application startup
register_factory(ndarray_to_image_abc, numpy.ndarray, ImageABC)

# Try to adapt numpy arrays to images. The 'adapt' function is
# introduced later in the docs, but you can probably guess what it does ;-)

# This adaptation fails, as the array is 1D
image = adapt(numpy.ndarray([1,2,3]), ImageABC, default=None)
assert image == None

# This succeeds.
image = adapt(numpy.array([[1,2],[3,4]]), ImageABC)
assert isinstance(image, NDArrayToImage)
```

Requesting an adapter

The `adapt` function Adapter classes are defined as described in the preceding sections, but you do not explicitly create instances of these classes.

Instead, the function `adapt()` is used, giving the object that needs to be adapted and the target protocol.

For instance, in the example in the *Conditional adaptation* section, a 2D numpy array is adapted to an ImageABC protocol with

```
image = adapt(numpy.array([[1,2],[3,4]]), ImageABC)
```

In some cases, no single adapter class is registered 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 returned.

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. Also, more specific adapters are preferred over less specific ones. For example, let's suppose we have a class `Document` and a subclass `HTMLDocument`. We register two adapters to an interface `IPrintable`, `DocumentToIPrintable` and `HTMLDocumentToIPrintable`. The call

```
html_doc = HTMLDocument()
printable = adapt(html_doc, IPrintable)
```

will return an instance of the `HTMLDocumentToIPrintable` adapter, as it is more specific than `DocumentToIPrintable`.

If no single adapter and no adapter chain can be constructed for the requested adaptation, an `AdaptationError` is raised. Alternatively, one can specify a default value to be returned in this case:

```
printable = adapt(unprintable_doc, IPrintable, default=EmptyPrintableDoc())
```

Using Traits interfaces An alternative syntax to create adapters when using Traits Interfaces is to use the interface class as an adapter factory, for example

```
printable = IPrintable(html_doc, None)
```

is equivalent to

```
printable = adapt(html_doc, IPrintable, default=None)
```

(the default argument, None, is optional).

Using the `Supports` and `AdaptsTo` traits Using the terminology introduced in this section, we can say that the `Instance` trait accepts values that *provide* the specified protocol.

Traits defines two additional traits that accept values that *support* a given protocol (they provide it or can be adapted to it) instead:

- The `Supports` trait accepts values that support the specified protocol. The value of the trait after assignment is the possibly adapted value (i.e., it is the original assigned value if that provides the protocol, or is an adapter otherwise).
- The `AdaptsTo` trait also accepts values that support the specified protocol. Unlike `Supports`, `AdaptsTo` stores the original, unadapted value.

If your application works with adaptation, it is natural to use the `Supports` trait in place of the `Instance` one in most cases. This will allow that application to be extended by adaptation in the future without changing the existing code, without having to invoke adaptation explicitly in your code.

For example, a Traits object can be written against the `IPrintable` interface and be open to extensions by adaptation as follows:

```
from traits.api import (Adapter, HasTraits, Interface, List, provides,
                        register_factory, Str, Supports)

class IPrintable(Interface):
    def get_formatted_text(self, n_cols):
        """ Return text formatted with the given number of columns. """

class PrintQueue(HasTraits):
    # This is the key part of the example: we declare a list of
    # items that provide or can be adapted to IPrintable
    queue = List(Supports(IPrintable))

    def is_empty(self):
        return len(self.queue) == 0

    def push(self, printable):
```

```
        self.queue.append(printable)

    def print_next(self):
        printable = self.queue.pop(0)

        # The elements from the list are guaranteed to provide
        # IPrintable, so we can call the interface without worrying
        # about adaptation.
        lines = printable.get_formatted_text(n_cols=20)

        print '-- Start document --'
        print '\n'.join(lines)
        print '-- End of document -\n'

class TextDocument(HasTraits):
    """ A text document. """
    text = Str

@provides(IPrintable)
class TextDocumentToIPrintable(Adapter):
    """ Adapt TextDocument and provide IPrintable. """

    def get_formatted_text(self, n_cols):
        import textwrap
        return textwrap.wrap(self.adaptee.text, n_cols)


# ---- Application starts here.

# Register the adapter.
register_factory(TextDocumentToIPrintable, TextDocument, IPrintable)

# Create two text documents.
doc1 = TextDocument(text='very very long text the will bore you for sure')
doc2 = TextDocument(text='once upon a time in a far away galaxy')

# The text documents can be pushed on the print queue; in the process,
# they are automatically adapted by Traits.
print_queue = PrintQueue()
print_queue.push(doc1)
print_queue.push(doc2)

while not print_queue.is_empty():
    print_queue.print_next()
```

This scripts produces this output:

```
-- Start document --
very very long text
the will bore you
for sure
-- End of document -

-- Start document --
once upon a time in
a far away galaxy
-- End of document -
```

Implementation details

The algorithm for finding a sequence of adapters adapting an object `adaptee` to a protocol `to_protocol` is based on a weighted graph.

Nodes on the graphs are protocols (types or interfaces). Edges are adaptation offers that connect a `offer.from_protocol` to a `offer.to_protocol`.

Edges connect protocol `A` to protocol `B` and are weighted by two numbers in this priority:

1. a unit weight (1) representing the fact that we use 1 adaptation offer to go from `A` to `B`
2. the number of steps up the type hierarchy that we need to take to go from `A` to `offer.from_protocol`, so that more specific adapters are always preferred

The algorithm finds the shortest weighted path between `adaptee` and `to_protocol`. Once a candidate path is found, it tries to create the chain of adapters using the factories in the adaptation offers that compose the path. If this fails because of conditional adaptation (i.e., an adapter factory returns None), the path is discarded and the algorithm looks for the next shortest path.

Cycles in adaptation are avoided by only considering path were every adaptation offer is used at most once.

Migration guide

The implementation of the adaptation mechanism changed in Traits 4.4.0 from one based on PyProtocols to a new, smaller, and more robust implementation.

Code written against `traits.protocols` will continue to work, although the *traits.protocols* API has been deprecated and its members will log a warning the first time they are accessed. The `traits.protocols` package will be removed in Traits 5.0 .

This is a list of replacements for the old API:

- `traits.protocols.api.AdaptationFailure`

  Use `traits.api.AdaptationError` instead.

- `traits.api.adapts()`

  Use the `traits.api.register_factory()` function.

- `implements()`

  Use the `traits.api.provides()` decorator instead.

- `traits.protocols.api.declareAdapter()`

  Use the function `traits.api.register_factory()`, or the function `traits.adaptation.api.register_offer()` instead. It is no longer necessary to distinguish between "types", "protocols", and "objects".

- `traits.protocols.api.declareImplementation()`

  This function was used occasionally to declare that an arbitrary type (e.g., `dict`) implements an interface. Users that use Python ABCs can use the `register` method for achieving the same result. Otherwise, use the function `traits.adaptation.api.register_provides()` that declares a "null" adapter to adapt the type to the interface.

- Testing if a class is an Interface

  `issubclass(klass, Interface)` is not reliable, use `traits.api.isinterface()` instead

Gotchas

1. The adaptation mechanism does not explicitly support old-style classes. Adaptation might work in particular cases but is not guaranteed to work correctly in situations involving old-style classes. When used with Traits, the classes involved in adaptation are typically subclasses of `HasTraits`, in which case this is not an issue.
2. The methods `register_factory()`, `adapt()`, etc. use a global adaptation manager, which is accessible through the function `get_global_adaptation_manager()`. The traits automatic adaptation features also use the global manager. Having a global adaptation manager can get you into trouble, for the usual reasons related to having a global state. If you want to have more control over adaptation, we recommend creating a new `AdaptationManager` instance, use it directly in your application, and set it as the global manager using `set_global_adaptation_manager()`. A common issue with the global manager arises in unittesting, where adapters registered in one test influence the outcome of other tests downstream. Tests relying on adaptation should make sure to reset the state of the global adapter using `reset_global_adaptation_manager()`.

Recommended readings about adaptation

This is a list of interesting readings about adaptation and the adapter pattern outside of Traits:

- PyProtocols, a precursor of `traits.adaptation`
- PEP 246 on object adaptation
- Article about adapters in Eclipse plugins

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 TraitUI view.

Property Factory Function

The Property() function has the following signature:

`traits.adaptation.api.`**`Property`**`(`[*fget=None*, *fset=None*, *fvalidate=None*, *force=False*, *handler=None*, *trait=None*, ***metadata*]`)`

All parameters are optional, including the *fget* "getter", *fvalidate* "validator" 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*(), _validate_*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.

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 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 *The name Parameter*. 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.

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).

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 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.

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 *Property Factory Function*)
- Shadow attributes for mapped traits (See *Mapped Traits*)
- Private attributes of HasPrivateTraits subclasses (See *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 *DelegatesTo*) You can mark a Delegate trait as transient if you do not want its value to ever be serialized.

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 key in state:
            del state[ key ]

    return state
```

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.

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.

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.

add_trait()

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

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.

set()

This method 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
```

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 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 traits.api import HasTraits, Trait

class Chicken(HasTraits):
    pass

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

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

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 defers its value through delegation or prototyping
- The complexity of a trait definition

If a trait attribute does not defer its value, 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 defer its value, the cases to be considered are:

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

In a typical application scenario, where attributes are read more often than they are written, and deferral 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 deferral is used heavily, either for delegation, or for prototyping to provide attributes with default values that are seldom changed. In this case, the cost of frequently following deferral chains may impose a measurable performance detriment on the application. Of course, this is offset by the convenience and flexibility provided by the deferral 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.

1.1.8 Testing Traits Classes

A mixin class is provided to facilitate writing tests for HasTraits classes. The following methods are available when `UnittestTools` is added as a mixin class in the developer's test cases.

`assertTraitChanges`	Assert an object trait changes a given number of times.
`assertTraitDoesNotChange`	Assert an object trait does not change.
`assertMultiTraitChanges`	Assert that traits on multiple objects do or do not change.
`assertTraitChangesAsync`	Assert an object trait eventually changes.
`assertEventuallyTrue`	Assert that the given condition is eventually true.

The above assert methods, except `assertEventuallyTrue()`, can be used as context managers, which at entry, hook a trait listeners on the class for the desired events and record the arguments passed to the change handler at every fired event. This way the developer can easily assert that specific events have been fired. Further analysis and checking can be performed by inspecting the list of recorded events. Both normal and extended trait names are supported. However, no check is performed regarding the validity of the trait name, thus care is required to safeguard against spelling mistakes in the names of the traits that we need to assert the behaviour.

The following example demonstrates the basic usage of the mixin class in a TestCase:

```
import unittest
from traits.api import HasTraits, Float, List, Bool, on_trait_change
from traits.testing.api import UnittestTools
```

```
class MyClass(HasTraits):

    number = Float(2.0)
    list_of_numbers = List(Float)
    flag = Bool

    @on_trait_change('number')
    def _add_number_to_list(self, value):
        """ Append the value to the list of numbers. """
        self.list_of_numbers.append(value)

    def add_to_number(self, value):
        """ Add the value to `number`. """
        self.number += value


class MyTestCase(unittest.TestCase, UnittestTools):

    def setUp(self):
        self.my_class = MyClass()

    def test_when_using_with(self):
        """ Check normal use cases as a context manager.
        """
        my_class = self.my_class

        # Checking for change events
        with self.assertTraitChanges(my_class, 'number') as result:
            my_class.number = 5.0

        # Inspecting the last recorded event
        expected = (my_class, 'number', 2.0, 5.0)
        self.assertSequenceEqual(result.events, [expected])

        # Checking for specific number of events
        with self.assertTraitChanges(my_class, 'number', count=3) as result:
            my_class.flag = True
            my_class.add_to_number(10.0)
            my_class.add_to_number(10.0)
            my_class.add_to_number(10.0)

        expected = [(my_class, 'number', 5.0, 15.0),
                    (my_class, 'number', 15.0, 25.0),
                    (my_class, 'number', 25.0, 35.0)]
        self.assertSequenceEqual(result.events, expected)

        # Check using extended names
        with self.assertTraitChanges(my_class, 'list_of_numbers[]'):
            my_class.number = -3.0

        # Check that event is not fired
        my_class.number = 2.0
        with self.assertTraitDoesNotChange(my_class, 'number') as result:
            my_class.flag = True
            my_class.number = 2.0  # The value is the same as the original
```

1.1.9 Using Mocks

Trying to mock a method in a `HasStrictTraits` instance will raise an error because the `HasStrictTraits` machinery does not allow any modification of the methods and attributes of a `HasStrictTraits` instance. To circumvent the `HasStrictTraits` machinery, and mock methods using the mock library, please follow the logic in the example below:

```
from traits.api import HasStrictTraits, Float
from mock import Mock

class MyClass(HasStrictTraits):

    number = Float(2.0)

    def add_to_number(self, value):
        """ Add the value to 'number'. """
        self.number += value

my_class = MyClass()

# Using my_class.add_to_number = Mock() will fail.
# But setting the mock on the instance '__dict__' works.
my_class.__dict__['add_to_number'] = Mock()

# We can now use the mock in our tests.
my_class.add_number(42)
print my_class.add_to_number.call_args_list
```

Note: The above method will not work for mocking `Property()` setters, getters and validators.

1.2 Indices and tables

- *genindex*
- *search*

CHAPTER 2

Developer Reference

2.1 API Reference

2.1.1 Traits core

`traits` Module

Defines the 'core' traits for the Traits package. A trait is a type definition that can be used for normal Python object attributes, giving the attributes some additional characteristics:

Initialization: Traits have predefined values that do not need to be explicitly initialized in the class constructor or elsewhere.

Validation: Trait attributes have flexible, type-checked values.

Delegation: Trait attributes' values can be delegated to other objects.

Notification: Trait attributes can automatically notify interested parties when their values change.

Visualization: Trait attributes can automatically construct (automatic or programmer-defined) user interfaces that allow their values to be edited or displayed)

Note: 'trait' is a synonym for 'property', but is used instead of the word 'property' to differentiate it from the Python language 'property' feature.

Classes

class `traits.traits.`**`CTrait`**

Extends the underlying C-based cTrait type.

`is_trait_type`(*trait_type*)

Returns whether or not this trait is of a specified trait type.

`get_editor`()

Returns the user interface editor associated with the trait.

`get_help`(*full=True*)

Returns the help text for a trait.

Parameters **full** (*bool*) – Indicates whether to return the value of the *help* attribute of the trait itself.

Description

If *full* is False or the trait does not have a **help** string, the returned string is constructed from the **desc** attribute on the trait and the **info** string on the trait's handler.

full_info(*object*, *name*, *value*)
Returns a description of the trait.

info()
Returns a description of the trait.

class traits.traits.**TraitFactory**(*maker_function=None*)

class traits.traits.**TraitImportError**(*message*)
Defines a factory class for deferring import problems until encountering code that actually tries to use the unimportable trait.

class traits.traits.**Default**(*func=None*, *args=()*, *kw=None*)
Generates a value the first time it is accessed.

A Default object can be used anywhere a default trait value would normally be specified, to generate a default value dynamically.

class traits.traits.**Trait**
Creates a trait definition.

Parameters

This function accepts a variety of forms of parameter lists:

Format	Example	Description
Trait(*default*)	Trait(150.0)	The type of the trait is inferred from the type of the default value, which must be in *ConstantTypes*.
Trait(*default*, *other1*, *other2*, ...)	Trait(None, 0, 1, 2, 'many')	The trait accepts any of the enumerated values, with the first value being the default value. The values must be of types in *ConstantTypes*, but they need not be of the same type. The *default* value is not valid for assignment unless it is repeated later in the list.
Trait([*default*, *other1*, *other2*, ...])	Trait([None, 0, 1, 2, 'many'])	Similar to the previous format, but takes an explicit list or a list variable.
Trait(*type*)	Trait(Int)	The *type* parameter must be a name of a Python type (see *PythonTypes*). Assigned values must be of exactly the specified type; no casting or coercion is performed. The default value is the appropriate form of zero, False, or emtpy string, set or sequence.

Trait(*class*)	`class MyClass:`
	`  pass`
	`foo = Trait(`
	`MyClass)`
		Values must be instances of *class* or of a subclass of *class*. The default value is None, but None cannot be assigned as a value.

Trait(None, *class*)	`class MyClass:`
	`  pass`
	`foo = Trait(`
	`None, MyClass)`
		Similar to the previous format, but None *can* be assigned as a value.

Trait(*instance*)	`class MyClass:`
	`  pass`
	`i = MyClass()`
	`foo =`
	`  Trait(i)`
		Values must be instances of the same class as *instance*, or of a subclass of that class. The specified instance is the default value.

Trait(*handler*)	Trait(TraitEnum)	Assignment to this trait is validated by an object derived from **traits.TraitHandler**.
Trait(*default*, { *type* | *constant* | *dict* | *class* | *function* | *handler* | *trait* }+)	Trait(0.0, 0.0 'stuff', TupleType)	This is the most general form of the function. The notation: `{...|...|...}+` means a list of one or more of any of the items listed between the braces. Thus, the most general form of the function consists of a default value, followed by one or more of several possible items. A trait defined by multiple items is called a "compound" trait.

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

The following predefined keywords are accepted:

Keywords

- **desc** (*str*) – Describes the intended meaning of the trait. It is used in exception messages and fly-over help in user interfaces.
- **label** (*str*) – Provides a human-readable name for the trait. It is used to label user interface editors for traits.
- **editor** (*traits.api.Editor*) – Instance of a subclass Editor object to use when creating a user interface editor for the trait. See the "Traits UI User Guide" for more information on trait editors.
- **comparison_mode** (*int*) – Indicates when trait change notifications should be generated based upon the result of comparing the old and new values of a trait assignment:
  - 0 (NO_COMPARE): The values are not compared and a trait change notification is generated on each assignment.
  - 1 (OBJECT_IDENTITY_COMPARE): A trait change notification is generated if the old and new values are not the same object.
  - 2 (RICH_COMPARE): A trait change notification is generated if the old and new values are not equal using Python's 'rich comparison' operator. This is the default.
- **rich_compare** (*bool*) – Indicates 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 whether the details of an object change, as long as the same object is used.

  Deprecated since version 3.0.3: Use `comparison_mode` instead

`traits.traits.`**`Property`**`()`

Returns a trait whose value is a Python property.

Parameters

- **fget** (*function*) – The "getter" function for the property.
- **fset** (*function*) – The "setter" function for the property.
- **fvalidate** (*function*) – The validation function for the property. The method should return the value to set or raise TraitError if the new value is not valid.
- **force** (*bool*) – Indicates whether to use only the function definitions specified by **fget** and **fset**, and not look elsewhere on the class.
- **handler** (*function*) – A trait handler function for the trait.
- **trait** (*Trait or value*) – A trait definition or a value that can be converted to a trait that constrains the values of the property trait.

Description

If no getter, setter or validate functions are specified (and **force** is not True), it is assumed that they are defined elsewhere on the class whose attribute this trait is assigned to. For example:

```
class Bar(HasTraits):

    # A float traits Property that should be always positive.
    foo = Property(Float)

    # Shadow trait attribute
    _foo = Float

    def _set_foo(self,x):
        self._foo = x

    def _validate_foo(self, x):
        if x <= 0:
            raise TraitError(
                'foo property should be a positive number')
        return x

    def _get_foo(self):
        return self._foo
```

You can use the **depends_on** metadata attribute to indicate that the property depends on the value of another trait. The value of **depends_on** is an extended name specifier for traits that the property depends on. The property will a trait change notification if any of the traits specified by **depends_on** change. For example:

```
class Wheel ( Part ):
    axle     = Instanced( Axle )
    position = Property( depends_on = 'axle.chassis.position' )
```

For details of the extended trait name syntax, refer to the on_trait_change() method of the HasTraits class.

class traits.traits.**ForwardProperty**(*metadata*, *validate=None*, *handler=None*)
Used to implement Property traits where accessor functions are defined implicitly on the class.

traits.traits.**Color**()
Returns a trait whose value must be a GUI toolkit-specific color.

Description

For wxPython, the returned trait accepts any of the following values:

- A wx.Colour instance
- A wx.ColourPtr instance
- an integer whose hexadecimal form is 0x*RRGGBB*, where *RR* is the red value, *GG* is the green value, and *BB* is the blue value

Default Value

For wxPython, 0x000000 (that is, white)

traits.traits.**RGBColor**()

Returns a trait whose value must be a GUI toolkit-specific RGB-based color.

Description

For wxPython, the returned trait accepts any of the following values:

- A tuple of the form (*r*, *g*, *b*), in which *r*, *g*, and *b* represent red, green, and blue values, respectively, and are floats in the range from 0.0 to 1.0
- An integer whose hexadecimal form is 0x*RRGGBB*, where *RR* is the red value, *GG* is the green value, and *BB* is the blue value

Default Value

For wxPython, (0.0, 0.0, 0.0) (that is, white)

traits.traits.**Font**()
Returns a trait whose value must be a GUI toolkit-specific font.

Description

For wxPython, the returned trait accepts any of the following:

- a wx.Font instance
- a wx.FontPtr instance
- a string describing the font, including one or more of the font family, size, weight, style, and typeface name.

Default Value

For wxPython, 'Arial 10'

Functions

traits.traits.**password_editor**(*auto_set=True*, *enter_set=False*)
Factory function that returns an editor for passwords.

traits.traits.**multi_line_text_editor**(*auto_set=True*, *enter_set=False*)
Factory function that returns a text editor for multi-line strings.

traits.traits.**code_editor**()
Factory function that returns an editor that treats a multi-line string as source code.

traits.traits.**shell_editor**()
Factory function that returns a Python shell for editing Python values.

traits.traits.**time_editor**()
Factory function that returns a Time editor for editing Time values.

traits.traits.**date_editor**()
Factory function that returns a Date editor for editing Date values.

traits.traits.**trait_factory**(*trait*)

traits.traits.**trait_cast**(*something*)
Casts a CTrait, TraitFactory or TraitType to a CTrait but returns None if it is none of those.

traits.traits.**try_trait_cast**(*something*)
Attempts to cast a value to a trait. Returns either a trait or the original value.

traits.traits.**trait_from**(*something*)
Returns a trait derived from its input.

Private Classes

class traits.traits.**_InstanceArgs**(*factory*, *args*, *kw*)

class traits.traits.**_TraitMaker**(**value_type*, ***metadata*)

adapter Module

An extension to PyProtocols to simplify the declaration of adapters.

Class

class traits.adapter.**Adapter**(**args*, ***kw*)

__init__(**args*, ***kw*)

Function

traits.adapter.**adapts**(**args*, ***kw*)
A class advisor for declaring adapters.

Parameters

- **from_** (*type or interface*) – What the adapter adapts *from*, or a list of such types or interfaces (the '_' suffix is used because 'from' is a Python keyword).
- **to** (*type or interface*) – What the adapter adapts *to*, or a list of such types or interfaces.
- **factory** (*callable*) – An (optional) factory for actually creating the adapters. This is any callable that takes a single argument which is the object to be adapted. The factory should return an adapter if it can perform the adaptation and **None** if it cannot.
- **cached** (*bool*) – Should the adapters be cached? If an adapter is cached, then the factory will produce at most one adapter per instance.
- **when** (*str*) – A Python expression that selects which instances of a particular type can be adapted by this factory. The expression is evaluated in a namespace that contains a single name *adaptee*, which is bound to the object to be adapted (e.g., 'adaptee.is_folder').

Note: The `cached` and `when` arguments are ignored if `factory` is specified.

category Module

Adds a "category" capability to Traits-based classes, similar to that provided by the Cocoa (Objective-C) environment for the Macintosh.

You can use categories to extend an existing HasTraits class, as an alternative to subclassing. An advantage of categories over subclassing is that you can access the added members on instances of the original class, without having to change them to instances of a subclass. Unlike subclassing, categories do not allow overriding trait attributes.

Classes

class traits.category.**MetaCategory**

class traits.category.**MetaCategoryObject**(*cls*, *class_name*, *bases*, *class_dict*, *is_category*)

class traits.category.**Category**
Used for defining "category" extensions to existing classes.

To define a class as a category, specify "Category," followed by the name of the base class name in the base class list.

The following example demonstrates defining a category:

```
from traits.api import HasTraits, Str, Category

class Base(HasTraits):
    x = Str("Base x")
    y = Str("Base y")

class BaseExtra(Category, Base):
    z = Str("BaseExtra z")
```

has_traits Module

Defines the HasTraits class, along with several useful subclasses and associated metaclasses.

Classes

class traits.has_traits.**ViewElement**

class traits.has_traits.**MetaHasTraits**

classmethod add_listener(*listener*, *class_name=''*)
Adds a class creation listener.

If the class name is the empty string then the listener will be called when *any* class is created.

classmethod remove_listener(*listener*, *class_name=''*)
Removes a class creation listener.

class traits.has_traits.**MetaInterface**
Meta class for interfaces.

Interfaces are simple ABCs with the following features:-

1.They cannot be instantiated (they are interfaces, not implementations!).

2.Calling them is equivalent to calling 'adapt'.

__init__()
x.__init__(...) initializes x; see help(type(x)) for signature

__call__(**args*, ***kw*)
Attempt to adapt the adaptee to this interface.

Note that this means that (intentionally ;^) that interfaces cannot be instantiated!

class traits.has_traits.**MetaHasTraitsObject**(*cls*, *class_name*, *bases*, *class_dict*, *is_category*)
Performs all of the meta-class processing needed to convert any subclass of HasTraits into a well-formed traits class.

__init__(*cls*, *class_name*, *bases*, *class_dict*, *is_category*)
Processes all of the traits related data in the class dictionary.

add_traits_meta_data(*bases*, *class_dict*, *base_traits*, *class_traits*, *instance_traits*, *prefix_traits*, *listeners*, *view_elements*)
Adds the Traits metadata to the class dictionary.

migrate_property(*name*, *property*, *property_info*, *class_dict*)
Migrates an existing property to the class being defined (allowing for method overrides).

class traits.has_traits.**HasTraits**
Enables any Python class derived from it to have trait attributes.

Most of the methods of HasTraits operated by default only on the trait attributes explicitly defined in the class definition. They do not operate on trait attributes defined by way of wildcards or by calling **add_trait()**. For example:

```
>>>class Person(HasTraits):
...     name = Str
...     age  = Int
...     temp_ = Any
>>>bob = Person()
>>>bob.temp_lunch = 'sandwich'
>>>bob.add_trait('favorite_sport', Str('football'))
>>>print bob.trait_names()
['trait_added', 'age', 'name']
```

In this example, the trait_names() method returns only the *age* and *name* attributes defined on the Person class. (The **trait_added** attribute is an explicit trait event defined on the HasTraits class.) The wildcard attribute *temp_lunch* and the dynamically-added trait attribute *favorite_sport* are not listed.

wrappers =

{'same': TraitChangeNotifyWrapper,
'extended': ExtendedTraitChangeNotifyWrapper,
'new': NewTraitChangeNotifyWrapper,
'fast_ui': FastUITraitChangeNotifyWrapper,
'ui': FastUITraitChangeNotifyWrapper}

Mapping from dispatch type to notification wrapper class type

trait_added = Event(basestring)
An event fired when a new trait is dynamically added to the object

trait_modified = Event
An event that can be fired to indicate that the state of the object has been modified

classmethod trait_monitor(*handler*, *remove=False*)
Adds or removes the specified *handler* from the list of active monitors.

Parameters

- **handler** (*function*) – The function to add or remove as a monitor.

- **remove** (*bool*) – Flag indicating whether to remove (True) or add the specified handler as a monitor for this class.

Description

If *remove* is omitted or False, the specified handler is added to the list of active monitors; if *remove* is True, the handler is removed from the active monitor list.

classmethod **add_class_trait**(*name*, **trait*)
Adds a named trait attribute to this class.

Parameters

- **name** (*str*) – Name of the attribute to add.
- ***trait** – A trait or a value that can be converted to a trait using Trait() Trait definition of the attribute. It can be a single value or a list equivalent to an argument list for the Trait() function.

classmethod **add_trait_category**(*category*)
Adds a trait category to a class.

classmethod **set_trait_dispatch_handler**(*name*, *klass*, *override=False*)
Sets a trait notification dispatch handler.

classmethod **trait_subclasses**(*all=False*)
Returns a list of the immediate (or all) subclasses of this class.

Parameters **all** (*bool*) – Indicates whether to return all subclasses of this class. If False, only immediate subclasses are returned.

has_traits_interface(**interfaces*)
Returns whether the object implements a specified traits interface.

Parameters ***interfaces** – One or more traits Interface (sub)classes.

Description

Tests whether the object implements one or more of the interfaces specified by *interfaces*. Return **True** if it does, and **False** otherwise.

trait_get(**names*, ***metadata*)
Shortcut for getting object trait attributes.

Parameters **names** (*list of strings*) – A list of trait attribute names whose values are requested.

Returns **result** (*dict*) – A dictionary whose keys are the names passed as arguments and whose values are the corresponding trait values.

Description

Looks up the value of each trait whose name is passed as an argument and returns a dictionary containing the resulting name/value pairs. If any name does not correspond to a defined trait, it is not included in the result.

If no names are specified, the result is a dictionary containing name/value pairs for *all* traits defined on the object.

get (**names*, ***metadata*)
Shortcut for getting object trait attributes.

Parameters **names** (*list of strings*) – A list of trait attribute names whose values are requested.

Returns **result** (*dict*) – A dictionary whose keys are the names passed as arguments and whose values are the corresponding trait values.

Description

Looks up the value of each trait whose name is passed as an argument and returns a dictionary containing the resulting name/value pairs. If any name does not correspond to a defined trait, it is not included in the result.

If no names are specified, the result is a dictionary containing name/value pairs for *all* traits defined on the object.

trait_set (*trait_change_notify=True*, ***traits*)
Shortcut for setting object trait attributes.

Parameters

- **trait_change_notify** (*bool*) – If **True** (the default), then each value assigned may generate a trait change notification. If **False**, then no trait change notifications will be generated. (see also: trait_setq)
- ****traits** – Key/value pairs, the trait attributes and their values to be set

Returns **self** – The method returns this object, after setting attributes.

Description

Treats each keyword argument to the method as the name of a trait attribute and sets the corresponding trait attribute to the value specified. This is a useful shorthand 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, you can write:

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

instead of:

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

set (*trait_change_notify=True*, ***traits*)
Shortcut for setting object trait attributes.

Parameters

- **trait_change_notify** (*bool*) – If **True** (the default), then each value assigned may generate a trait change notification. If **False**, then no trait change notifications will be generated. (see also: trait_setq)
- ****traits** – Key/value pairs, the trait attributes and their values to be set

Returns **self** – The method returns this object, after setting attributes.

Description

Treats each keyword argument to the method as the name of a trait attribute and sets the corresponding trait attribute to the value specified. This is a useful shorthand 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, you can write:

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

instead of:

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

trait_setq(***traits*)

Shortcut for setting object trait attributes.

Parameters ****traits** – Key/value pairs, the trait attributes and their values to be set. No trait change notifications will be generated for any values assigned (see also: trait_set).

Returns **self** – The method returns this object, after setting attributes.

Description

Treats each keyword argument to the method as the name of a trait attribute and sets the corresponding trait attribute to the value specified. This is a useful shorthand 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, you can write:

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

instead of:

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

reset_traits(*traits=None*, ***metadata*)

Resets some or all of an object's trait attributes to their default values.

Parameters **traits** (*list of strings*) – Names of trait attributes to reset.

Returns **unresetable** (*list of strings*) – A list of attributes that the method was unable to reset, which is empty if all the attributes were successfully reset.

Description

Resets each of the traits whose names are specified in the *traits* list to their default values. If *traits* is None or omitted, the method resets all explicitly-defined object trait attributes to their default values. Note that this does not affect wildcard trait attributes or trait attributes added via add_trait(), unless they are explicitly named in *traits*.

copyable_trait_names(***metadata*)

Returns the list of trait names to copy or clone by default.

all_trait_names()

Returns the list of all trait names, including implicitly defined traits.

copy_traits(*other*, *traits=None*, *memo=None*, *copy=None*, ***metadata*)

Copies another object's trait attributes into this one.

Parameters

- **other** (*object*) – The object whose trait attribute values should be copied.
- **traits** (*list of strings*) – A list of names of trait attributes to copy. If None or unspecified, the set of names returned by trait_names() is used. If 'all' or an empty list, the set of names returned by all_trait_names() is used.
- **memo** (*dict*) – A dictionary of objects that have already been copied.
- **copy** (*None | 'deep' | 'shallow'*) – The type of copy to perform on any trait that does not have explicit 'copy' metadata. A value of None means 'copy reference'.

Returns **unassignable** (*list of strings*) – A list of attributes that the method was unable to copy, which is empty if all the attributes were successfully copied.

clone_traits(*traits=None*, *memo=None*, *copy=None*, ***metadata*)

Clones a new object from this one, optionally copying only a specified set of traits.

Parameters

- **traits** (*list of strings*) – The list of names of the trait attributes to copy.
- **memo** (*dict*) – A dictionary of objects that have already been copied.
- **copy** (*str*) – The type of copy `deep` or `shallow` to perform on any trait that does not have explicit 'copy' metadata. A value of None means 'copy reference'.

Returns **new** – The newly cloned object.

Description

Creates a new object that is a clone of the current object. If *traits* is None (the default), then all explicit trait attributes defined for this object are cloned. If *traits* is 'all' or an empty list, the list of traits returned by all_trait_names() is used; otherwise, *traits* must be a list of the names of the trait attributes to be cloned.

edit_traits(*view=None*, *parent=None*, *kind=None*, *context=None*, *handler=None*, *id=''*, *scrollable=None*, ***args*)

Displays a user interface window for editing trait attribute values.

Parameters

- **view** (*View or string*) – A View object (or its name) that defines a user interface for editing trait attribute values of the current object. If the view is defined as an attribute on this class, use the name of the attribute. Otherwise, use a reference to the view object. If this attribute is not specified, the View object returned by trait_view() is used.
- **parent** (*toolkit control*) – The reference to a user interface component to use as the parent window for the object's UI window.
- **kind** (*str*) – The type of user interface window to create. See the **traitsui.view.kind_trait** trait for values and their meanings. If *kind* is unspecified or None, the **kind** attribute of the View object is used.
- **context** (*object or dictionary*) – A single object or a dictionary of string/object pairs, whose trait attributes are to be edited. If not specified, the current object is used.
- **handler** (*Handler*) – A handler object used for event handling in the dialog box. If None, the default handler for Traits UI is used.
- **id** (*str*) – A unique ID for persisting preferences about this user interface, such as size and position. If not specified, no user preferences are saved.

- **scrollable** (*bool*) – Indicates whether the dialog box should be scrollable. When set to True, scroll bars appear on the dialog box if it is not large enough to display all of the items in the view at one time.

trait_context()
Returns the default context to use for editing or configuring traits.

trait_view(*name=None*, *view_element=None*)
Gets or sets a ViewElement associated with an object's class.

Parameters

- **name** (*str*) – Name of a view element
- **view_element** (*ViewElement*) – View element to associate

Returns A view element.

Description

If both *name* and *view_element* are specified, the view element is associated with *name* for the current object's class. (That is, *view_element* is added to the ViewElements object associated with the current object's class, indexed by *name*.)

If only *name* is specified, the function returns the view element object associated with *name*, or None if *name* has no associated view element. View elements retrieved by this function are those that are bound to a class attribute in the class definition, or that are associated with a name by a previous call to this method.

If neither *name* nor *view_element* is specified, the method returns a View object, based on the following order of preference:

1. If there is a View object named `traits_view` associated with the current object, it is returned.
2. If there is exactly one View object associated the current object, it is returned.
3. Otherwise, it returns a View object containing items for all the non-event trait attributes on the current object.

default_traits_view()
Returns the name of the default traits view for the object's class.

classmethod class_default_traits_view()
Returns the name of the default traits view for the class.

trait_views(*klass=None*)
Returns a list of the names of all view elements associated with the current object's class.

Parameters **klass** (*class*) – A class, such that all returned names must correspond to instances of this class. Possible values include:

- Group
- Item
- View
- ViewElement
- ViewSubElement

Description

If *klass* is specified, the list of names is filtered such that only objects that are instances of the specified class are returned.

trait_view_elements()
Returns the ViewElements object associated with the object's class.

The returned object can be used to access all the view elements associated with the class.

classmethod class_trait_view_elements()
Returns the ViewElements object associated with the class.

The returned object can be used to access all the view elements associated with the class.

configure_traits(*filename=None*, *view=None*, *kind=None*, *edit=True*, *context=None*, *handler=None*, *id=''*, *scrollable=None*, ***args*)
Creates and displays a dialog box for editing values of trait attributes, as if it were a complete, self-contained GUI application.

Parameters

- **filename** (*str*) – The name (including path) of a file that contains a pickled representation of the current object. When this parameter is specified, the method reads the corresponding file (if it exists) to restore the saved values of the object's traits before displaying them. If the user confirms the dialog box (by clicking **OK**), the new values are written to the file. If this parameter is not specified, the values are loaded from the in-memory object, and are not persisted when the dialog box is closed.
- **view** (*View or str*) – A View object (or its name) that defines a user interface for editing trait attribute values of the current object. If the view is defined as an attribute on this class, use the name of the attribute. Otherwise, use a reference to the view object. If this attribute is not specified, the View object returned by trait_view() is used.
- **kind** (*str*) – The type of user interface window to create. See the **traitsui.view.kind_trait** trait for values and their meanings. If *kind* is unspecified or None, the **kind** attribute of the View object is used.
- **edit** (*bool*) – Indicates whether to display a user interface. If *filename* specifies an existing file, setting *edit* to False loads the saved values from that file into the object without requiring user interaction.
- **context** (*object or dictionary*) – A single object or a dictionary of string/object pairs, whose trait attributes are to be edited. If not specified, the current object is used
- **handler** (*Handler*) – A handler object used for event handling in the dialog box. If None, the default handler for Traits UI is used.
- **id** (*str*) – A unique ID for persisting preferences about this user interface, such as size and position. If not specified, no user preferences are saved.
- **scrollable** (*bool*) – Indicates whether the dialog box should be scrollable. When set to True, scroll bars appear on the dialog box if it is not large enough to display all of the items in the view at one time.

Description

This method is intended for use in applications that do not normally have a GUI. Control does not resume in the calling application until the user closes the dialog box.

The method attempts to open and unpickle the contents of *filename* before displaying the dialog box. When editing is complete, the method attempts to pickle the updated contents of the object back to *filename*. If the file referenced by *filename* does not exist, the object is not modified before displaying the dialog box. If *filename* is unspecified or None, no pickling or unpickling occurs.

If *edit* is True (the default), a dialog box for editing the current object is displayed. If *edit* is False or None, no dialog box is displayed. You can use `edit=False` if you want the object to be restored from the contents of *filename*, without being modified by the user.

editable_traits()
Returns an alphabetically sorted list of the names of non-event trait attributes associated with the current object.

classmethod class_editable_traits()
Returns an alphabetically sorted list of the names of non-event trait attributes associated with the current class.

print_traits(*show_help=False*, ***metadata*)
Prints the values of all explicitly-defined, non-event trait attributes on the current object, in an easily readable format.

Parameters **show_help** (*bool*) – Indicates whether to display additional descriptive information.

on_trait_change(*handler*, *name=None*, *remove=False*, *dispatch='same'*, *priority=False*, *deferred=False*, *target=None*)
Causes the object to invoke a handler whenever a trait attribute matching a specified pattern is modified, or removes the association.

Parameters

- **handler** (*function*) – A trait notification function for the *name* trait attribute, with one of the signatures described below.
- **name** (*str*) – The name of the trait attribute whose value changes trigger the notification. The *name* can specify complex patterns of trait changes using an extended *name* syntax, which is described below.
- **remove** (*bool*) – If True, removes the previously-set association between *handler* and *name*; if False (the default), creates the association.
- **dispatch** (*str*) – A string indicating the thread on which notifications must be run. Possible values are:

value	dispatch
`same`	Run notifications on the same thread as this one.
`ui`	Run notifications on the UI thread. If the current thread is the UI thread, the notifications are executed immediately; otherwise, they are placed on the UI event queue.
`fast_ui`	Alias for `ui`.
`new`	Run notifications in a new thread.

Description

Multiple handlers can be defined for the same object, or even for the same trait attribute on the same object. If *name* is not specified or is None, *handler* is invoked when any trait attribute on the object is changed.

The *name* parameter is a single *xname* or a list of *xname* names, where an *xname* is an extended name of the form:

```
xname2[('.'|':') xname2]*
```

An *xname2* is of the form:

```
( xname3 | '['xname3[','xname3]*']' ) ['*']
```

An *xname3* is of the form:

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

A *name* is any valid Python attribute name. The semantic meaning of this notation is as follows:

expression	meaning
`item1.item2`	means *item1* is a trait containing an object (or objects if *item1* is a list or dict) with a trait called *item2*. Changes to either *item1* or *item2* cause a notification to be generated.
`item1:item2`	means *item1* is a trait containing an object (or objects if *item1* is a list or dict) with a trait called *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 which 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 will be raised.
`prefix+`	Matches any trait on the object whose name begins with *prefix*.
`+metadata_name`	Matches any trait on the object having *metadata_name* metadata.
`-metadata_name`	Matches any trait on the object which does not have *metadata_name* metadata.
`prefix+metadata_name`	Matches any trait on the object whose name begins with *prefix* and which has *metadata_name* metadata.
`prefix-metadata_name`	Matches any trait on the object whose name begins with *prefix* and which does not have *metadata_name* metadata.
`+`	Matches all traits on the object.
`pattern*`	Matches object graphs where *pattern* occurs one or more times (useful for setting up listeners on recursive data structures like trees or linked lists).

Some examples of valid names and their meaning are as follows:

example	meaning
`foo,bar,baz`	Listen for trait changes to *object.foo*, *object.bar*, and *object.baz*.
`['foo','bar','baz']`	Equivalent to 'foo,bar,baz', but may be more useful in cases where the individual items are computed.
`foo.bar.baz`	Listen for trait changes to *object.foo.bar.baz* and report changes to *object.foo*, *object.foo.bar* or *object.foo.bar.baz*.
`foo:bar:baz`	Listen for changes to *object.foo.bar.baz*, and only report changes to *object.foo.bar.baz*.
`foo.[bar,baz]`	Listen for trait changes to *object.foo.bar* and *object.foo.baz*.
`[left,right]:name`	Listen for trait changes to the *name* trait of each node of a tree having *left* and *right* links to other tree nodes, and where *object* the method is applied to the root node of the tree.
`+dirty`	Listen for trait changes on any trait in the *object* which has the 'dirty' metadata set.
`foo.+dirty`	Listen for trait changes on any trait in *object.foo* which has the 'dirty' metadata set.
`foo.[bar,-dirty]`	Listen for trait changes on *object.foo.bar* or any trait on *object.foo* which does not have 'dirty' metadata set.

Note that any of the intermediate (i.e., non-final) links in a pattern can be traits of type Instance, List or

Dict. In the case of List and Dict traits, the subsequent portion of the pattern is applied to each item in the list, or value in the dictionary.

For example, if the self.children is a list, 'children.name' listens for trait changes to the *name* trait for each item in the self.children list.

Note that items added to or removed from a list or dictionary in the pattern will cause the *handler* routine to be invoked as well, since this is treated as an *implied* change to the item's trait being monitored.

The signature of the *handler* supplied also has an effect on how changes to intermediate traits are processed. The five valid handler signatures are:

1.handler()

2.handler(new)

3.handler(name,new)

4.handler(object,name,new)

5.handler(object,name,old,new)

For signatures 1, 4 and 5, any change to any element of a path being listened to invokes the handler with information about the particular element that was modified (e.g., if the item being monitored is 'foo.bar.baz', a change to 'bar' will call *handler* with the following information:

•object: object.foo

•name: bar

•old: old value for object.foo.bar

•new: new value for object.foo.bar

If one of the intermediate links is a List or Dict, the call to *handler* may report an *_items* changed event. If in the previous example, *bar* is a List, and a new item is added to *bar*, then the information passed to *handler* would be:

•object: object.foo

•name: bar_items

•old: Undefined

•**new: TraitListEvent whose *added* trait contains the new item** added to *bar*.

For signatures 2 and 3, the *handler* does not receive enough information to discern between a change to the final trait being listened to and a change to an intermediate link. In this case, the event dispatcher will attempt to map a change to an intermediate link to its effective change on the final trait. This only works if all of the intermediate links are single values (such as an Instance or Any trait) and not Lists or Dicts. If the modified intermediate trait or any subsequent intermediate trait preceding the final trait is a List or Dict, then a TraitError is raised, since the effective value for the final trait cannot in general be resolved unambiguously. To prevent TraitErrors in this case, use the ':' separator to suppress notifications for changes to any of the intermediate links.

Handler signature 1 also has the special characteristic that if a final trait is a List or Dict, it will automatically handle '_items' changed events for the final trait as well. This can be useful in cases where the *handler* only needs to know that some aspect of the final trait has been changed. For all other *handler* signatures, you must explicitly specify the 'xxx_items' trait if you want to be notified of changes to any of the items of the 'xxx' trait.

on_trait_event(*handler*, *name=None*, *remove=False*, *dispatch='same'*, *priority=False*, *deferred=False*, *target=None*)

Causes the object to invoke a handler whenever a trait attribute matching a specified pattern is modified, or removes the association.

Parameters

- **handler** (*function*) – A trait notification function for the *name* trait attribute, with one of the signatures described below.
- **name** (*str*) – The name of the trait attribute whose value changes trigger the notification. The *name* can specify complex patterns of trait changes using an extended *name* syntax, which is described below.
- **remove** (*bool*) – If True, removes the previously-set association between *handler* and *name*; if False (the default), creates the association.
- **dispatch** (*str*) – A string indicating the thread on which notifications must be run. Possible values are:

value	dispatch
`same`	Run notifications on the same thread as this one.
`ui`	Run notifications on the UI thread. If the current thread is the UI thread, the notifications are executed immediately; otherwise, they are placed on the UI event queue.
`fast_ui`	Alias for `ui`.
`new`	Run notifications in a new thread.

Description

Multiple handlers can be defined for the same object, or even for the same trait attribute on the same object. If *name* is not specified or is None, *handler* is invoked when any trait attribute on the object is changed.

The *name* parameter is a single *xname* or a list of *xname* names, where an *xname* is an extended name of the form:

```
xname2[('.'|':') xname2]*
```

An *xname2* is of the form:

```
( xname3 | '['xname3[','xname3]*']' ) ['*']
```

An *xname3* is of the form:

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

A *name* is any valid Python attribute name. The semantic meaning of this notation is as follows:

expression	meaning
`item1.item2`	means *item1* is a trait containing an object (or objects if *item1* is a list or dict) with a trait called *item2*. Changes to either *item1* or *item2* cause a notification to be generated.
`item1:item2`	means *item1* is a trait containing an object (or objects if *item1* is a list or dict) with a trait called *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 which 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 will be raised.
`prefix+`	Matches any trait on the object whose name begins with *prefix*.
`+metadata_name`	Matches any trait on the object having *metadata_name* metadata.
`-metadata_name`	Matches any trait on the object which does not have *metadata_name* metadata.
`prefix+metadata_name`	Matches any trait on the object whose name begins with *prefix* and which has *metadata_name* metadata.
`prefix-metadata_name`	Matches any trait on the object whose name begins with *prefix* and which does not have *metadata_name* metadata.
`+`	Matches all traits on the object.
`pattern*`	Matches object graphs where *pattern* occurs one or more times (useful for setting up listeners on recursive data structures like trees or linked lists).

Some examples of valid names and their meaning are as follows:

example	meaning
`foo,bar,baz`	Listen for trait changes to *object.foo*, *object.bar*, and *object.baz*.
`['foo','bar','baz']`	Equivalent to 'foo,bar,baz', but may be more useful in cases where the individual items are computed.
`foo.bar.baz`	Listen for trait changes to *object.foo.bar.baz* and report changes to *object.foo*, *object.foo.bar* or *object.foo.bar.baz*.
`foo:bar:baz`	Listen for changes to *object.foo.bar.baz*, and only report changes to *object.foo.bar.baz*.
`foo.[bar,baz]`	Listen for trait changes to *object.foo.bar* and *object.foo.baz*.
`[left,right]:name`	Listen for trait changes to the *name* trait of each node of a tree having *left* and *right* links to other tree nodes, and where *object* the method is applied to the root node of the tree.
`+dirty`	Listen for trait changes on any trait in the *object* which has the 'dirty' metadata set.
`foo.+dirty`	Listen for trait changes on any trait in *object.foo* which has the 'dirty' metadata set.
`foo.[bar,-dirty]`	Listen for trait changes on *object.foo.bar* or any trait on *object.foo* which does not have 'dirty' metadata set.

Note that any of the intermediate (i.e., non-final) links in a pattern can be traits of type Instance, List or Dict. In the case of List and Dict traits, the subsequent portion of the pattern is applied to each item in the list, or value in the dictionary.

For example, if the self.children is a list, 'children.name' listens for trait changes to the *name* trait for each item in the self.children list.

Note that items added to or removed from a list or dictionary in the pattern will cause the *handler* routine to be invoked as well, since this is treated as an *implied* change to the item's trait being monitored.

The signature of the *handler* supplied also has an effect on how changes to intermediate traits are processed. The five valid handler signatures are:

1.handler()

2.handler(new)

3.handler(name,new)

4.handler(object,name,new)

5.handler(object,name,old,new)

For signatures 1, 4 and 5, any change to any element of a path being listened to invokes the handler with information about the particular element that was modified (e.g., if the item being monitored is 'foo.bar.baz', a change to 'bar' will call *handler* with the following information:

•object: object.foo

•name: bar

•old: old value for object.foo.bar

•new: new value for object.foo.bar

If one of the intermediate links is a List or Dict, the call to *handler* may report an *_items* changed event. If in the previous example, *bar* is a List, and a new item is added to *bar*, then the information passed to *handler* would be:

•object: object.foo

•name: bar_items

•old: Undefined

•**new: TraitListEvent whose *added* trait contains the new item** added to *bar*.

For signatures 2 and 3, the *handler* does not receive enough information to discern between a change to the final trait being listened to and a change to an intermediate link. In this case, the event dispatcher will attempt to map a change to an intermediate link to its effective change on the final trait. This only works if all of the intermediate links are single values (such as an Instance or Any trait) and not Lists or Dicts. If the modified intermediate trait or any subsequent intermediate trait preceding the final trait is a List or Dict, then a TraitError is raised, since the effective value for the final trait cannot in general be resolved unambiguously. To prevent TraitErrors in this case, use the ':' separator to suppress notifications for changes to any of the intermediate links.

Handler signature 1 also has the special characteristic that if a final trait is a List or Dict, it will automatically handle '_items' changed events for the final trait as well. This can be useful in cases where the *handler* only needs to know that some aspect of the final trait has been changed. For all other *handler* signatures, you must explicitly specify the 'xxx_items' trait if you want to be notified of changes to any of the items of the 'xxx' trait.

sync_trait(*trait_name*, *object*, *alias=None*, *mutual=True*, *remove=False*)

Synchronizes the value of a trait attribute on this object with a trait attribute on another object.

Parameters

- **name** (*str*) – Name of the trait attribute on this object.
- **object** (*object*) – The object with which to synchronize.
- **alias** (*str*) – Name of the trait attribute on *other*; if None or omitted, same as *name*.
- **mutual** (*bool or int*) – Indicates whether synchronization is mutual (True or non-zero) or one-way (False or zero)
- **remove** (*bool or int*) – Indicates whether synchronization is being added (False or zero) or removed (True or non-zero)

Description

In mutual synchronization, any change to the value of the specified trait attribute of either object results in the same value being assigned to the corresponding trait attribute of the other object. In one-way synchronization, any change to the value of the attribute on this object causes the corresponding trait attribute of *object* to be updated, but not vice versa.

add_trait(*name*, **trait*)
Adds a trait attribute to this object.

Parameters

- **name** (*str*) – Name of the attribute to add.
- ***trait** – Trait or a value that can be converted to a trait by Trait(). Trait definition for *name*. If more than one value is specified, it is equivalent to passing the entire list of values to Trait().

remove_trait(*name*)
Removes a trait attribute from this object.

Parameters **name** (*str*) – Name of the attribute to remove.

Returns **result** (*bool*) – True if the trait was successfully removed.

trait(*name*, *force=False*, *copy=False*)
Returns the trait definition for the *name* trait attribute.

Parameters

- **name** (*str*) – Name of the attribute whose trait definition is to be returned.
- **force** (*bool*) – Indicates whether to return a trait definition if *name* is not explicitly defined.
- **copy** (*bool*) – Indicates whether to return the original trait definition or a copy.

Description

If *force* is False (the default) and *name* is the name of an implicitly defined trait attribute that has never been referenced explicitly (i.e., has not yet been defined), the result is None. In all other cases, the result is the trait definition object associated with *name*.

If *copy* is True, and a valid trait definition is found for *name*, a copy of the trait found is returned. In all other cases, the trait definition found is returned unmodified (the default).

base_trait(*name*)
Returns the base trait definition for a trait attribute.

Parameters **name** (*str*) – Name of the attribute whose trait definition is returned.

Description

This method is similar to the trait() method, and returns a different result only in the case where the trait attribute defined by *name* is a delegate. In this case, the base_trait() method follows the delegation chain until a non-delegated trait attribute is reached, and returns the definition of that attribute's trait as the result.

validate_trait(*name*, *value*)
Validates whether a value is legal for a trait.

Returns the validated value if it is valid.

traits(***metadata*)
Returns a dictionary containing the definitions of all of the trait attributes of this object that match the set of *metadata* criteria.

Parameters ****metadata** – Criteria for selecting trait attributes.

Description

The keys of the returned dictionary are the trait attribute names, and the values are their corresponding trait definition objects.

If no *metadata* information is specified, then all explicitly defined trait attributes defined for the object are returned.

Otherwise, the *metadata* keyword dictionary is assumed to define a set of search criteria for selecting trait attributes of interest. The *metadata* dictionary keys correspond to the names of trait metadata attributes to examine, and the values correspond to the values the metadata attribute must have in order to be included in the search results.

The *metadata* values either may be simple Python values like strings or integers, or may be lambda expressions or functions that return True if the trait attribute is to be included in the result. A lambda expression or function must receive a single argument, which is the value of the trait metadata attribute being tested. If more than one metadata keyword is specified, a trait attribute must match the metadata values of all keywords to be included in the result.

classmethod **class_traits**(***metadata*)
Returns a dictionary containing the definitions of all of the trait attributes of the class that match the set of *metadata* criteria.

Parameters ****metadata** – Criteria for selecting trait attributes.

Description

The keys of the returned dictionary are the trait attribute names, and the values are their corresponding trait definition objects.

If no *metadata* information is specified, then all explicitly defined trait attributes defined for the class are returned.

Otherwise, the *metadata* keyword dictionary is assumed to define a set of search criteria for selecting trait attributes of interest. The *metadata* dictionary keys correspond to the names of trait metadata attributes to examine, and the values correspond to the values the metadata attribute must have in order to be included in the search results.

The *metadata* values either may be simple Python values like strings or integers, or may be lambda expressions or functions that return **True** if the trait attribute is to be included in the result. A lambda expression or function must receive a single argument, which is the value of the trait metadata attribute being tested. If more than one metadata keyword is specified, a trait attribute must match the metadata values of all keywords to be included in the result.

trait_names(***metadata*)
Returns a list of the names of all trait attributes whose definitions match the set of *metadata* criteria specified.

Parameters ****metadata** – Criteria for selecting trait attributes.

Description

This method is similar to the traits() method, but returns only the names of the matching trait attributes, not the trait definitions.

classmethod **class_trait_names**(***metadata*)
Returns a list of the names of all trait attributes whose definitions match the set of *metadata* criteria specified.

Parameters ****metadata** – Criteria for selecting trait attributes.

Description

This method is similar to the traits() method, but returns only the names of the matching trait attributes, not the trait definitions.

class traits.has_traits.**HasStrictTraits**
This class guarantees that any object attribute that does not have an explicit or wildcard trait definition results in an exception.

This feature can be useful in cases where a more rigorous software engineering approach is being used 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.

class traits.has_traits.**HasPrivateTraits**
This class ensures that any public object attribute that does not have an explicit or wildcard trait definition results in an exception, but "private" attributes (whose names start with '_') have an initial value of **None**, and are not type-checked.

This feature is useful in cases where a class needs private attributes to keep track of its internal object state, which are not part of the class's public API. Such attributes do not need to be type-checked, because they are manipulated only by the (presumably correct) methods of the class itself.

class traits.has_traits.**SingletonHasTraits**
Singleton class that support trait attributes.

class traits.has_traits.**SingletonHasStrictTraits**
Singleton class that supports strict trait attributes.

Non-trait attributes generate an exception.

class traits.has_traits.**SingletonHasPrivateTraits**
Singleton class that supports trait attributes, with private attributes being unchecked.

class traits.has_traits.**Vetoable**
Defines a 'vetoable' request object and an associated event.

class traits.has_traits.**Interface**
The base class for all interfaces.

class traits.has_traits.**ISerializable**
A class that implemented ISerializable requires that all HasTraits objects saved as part of its state also implement ISerializable.

class traits.has_traits.**traits_super**

ABC classes

Note: These classes are only available when the abc module is present.

class traits.has_traits.**ABCMetaHasTraits**
A MetaHasTraits subclass which also inherits from abc.ABCMeta.

Note: The ABCMeta class is cooperative and behaves nicely with MetaHasTraits, provided it is inherited first.

class traits.has_traits.**ABCHasTraits**
A HasTraits subclass which enables the features of Abstract Base Classes (ABC). See the 'abc' module in the standard library for more information.

class traits.has_traits.**ABCHasStrictTraits**
A HasTraits subclass which behaves like HasStrictTraits but also enables the features of Abstract Base Classes (ABC). See the 'abc' module in the standard library for more information.

Functions

traits.has_traits.**get_delegate_pattern**(*name*, *trait*)
Returns the correct 'delegate' listener pattern for a specified name and delegate trait.

traits.has_traits.**weak_arg**(*arg*)
Create a weak reference to arg and wrap the function so that the dereferenced weakref is passed as the first argument. If arg has been deleted then the function is not called.

traits.has_traits.**property_depends_on**(*dependency*, *settable=False*, *flushable=False*)
Marks the following method definition as being a "cached property" that depends on the specified extended trait names. That is, it is a property getter which, for performance reasons, caches its most recently computed result in an attribute whose name is of the form: *_traits_cache_name*, where *name* is the name of the property. A method marked as being a cached property needs only to compute and return its result. The @property_depends_on decorator automatically wraps the decorated method in cache management code that will cache the most recently computed value and flush the cache when any of the specified dependencies are modified, thus eliminating the need to write boilerplate cache management code explicitly. For example:

```
file_name = File
file_contents = Property

@property_depends_on( 'file_name' )
def _get_file_contents(self):
    fh = open(self.file_name, 'rb')
    result = fh.read()
    fh.close()
    return result
```

In this example, accessing the *file_contents* trait calls the _get_file_contents() method only once each time after the **file_name** trait is modified. In all other cases, the cached value **_file_contents**, which is maintained by the @cached_property wrapper code, is returned.

traits.has_traits.**cached_property**(*function*)
Marks the following method definition as being a "cached property". That is, it is a property getter which, for performance reasons, caches its most recently computed result in an attribute whose name is of the form: *_traits_cache_name*, where *name* is the name of the property. A method marked as being a cached property needs only to compute and return its result. The @cached_property decorator automatically wraps the decorated method in cache management code, eliminating the need to write boilerplate cache management code explicitly. For example:

```
file_name = File
file_contents = Property( depends_on = 'file_name' )

@cached_property
def _get_file_contents(self):
    fh = open(self.file_name, 'rb')
    result = fh.read()
    fh.close()
    return result
```

In this example, accessing the *file_contents* trait calls the _get_file_contents() method only once each time after the **file_name** trait is modified. In all other cases, the cached value **_file_contents**, which maintained by the @cached_property wrapper code, is returned.

Note the use, in the example, of the **depends_on** metadata attribute to specify that the value of **file_contents** depends on **file_name**, so that _get_file_contents() is called only when **file_name** changes. For details, see the traits.traits.Property() function.

traits.has_traits.**on_trait_change**(*name*, *post_init=False*, **names*)
Marks the following method definition as being a handler for the extended trait change specified by *name(s)*.

Refer to the documentation for the on_trait_change() method of the **HasTraits** class for information on the correct syntax for the *name(s)* argument.

A handler defined using this decorator is normally effective immediately. However, if *post_init* is **True**, then the handler only become effective after all object constructor arguments have been processed. That is, trait values assigned as part of object construction will not cause the handler to be invoked.

traits.has_traits.**implements**(**interfaces*)
Declares the interfaces that a class implements.

Parameters ***interfaces** – A list of interface classes that the containing class implements.

Description

Registers each specified interface with the interface manager as an interface that the containing class implements. Each specified interface must be a subclass of **Interface**. This function should only be called from directly within a class body.

has_dynamic_views Module

Classes

interface_checker Module

An attempt at type-safe casting.

Classes

class traits.interface_checker.**InterfaceError**
The exception raised if a class does not really implement an interface.

class traits.interface_checker.**InterfaceChecker**
Checks that interfaces are actually implemented.

check_implements(*cls*, *interfaces*, *error_mode*)
Checks that the class implements the specified interfaces.

'interfaces' can be a single interface or a list of interfaces.

Function

traits.interface_checker.**check_implements**(*cls*, *interfaces*, *error_mode=0*)
Checks that the class implements the specified interfaces.

'interfaces' can be a single interface or a list of interfaces.

trait_base Module

Defines common, low-level capabilities needed by the Traits package.

Classes

traits.trait_base.**Uninitialized = <uninitialized>**
When the first reference to a trait is a 'get' reference, the default value of the trait is implicitly assigned and returned as the value of the trait. Because of this implicit assignment, a trait change notification is generated with the Uninitialized object as the 'old' value of the trait, and the default trait value as the 'new' value. This allows other parts of the traits package to recognize the assignment as the implicit default value assignment, and treat it specially.

traits.trait_base.**Undefined = <undefined>**
Singleton object that indicates that a trait attribute has not yet had a value set (i.e., its value is undefined). This object is used instead of None, because None often has other meanings, such as that a value is not used. When a trait attribute is first assigned a value, and its associated trait notification handlers are called, Undefined is passed as the *old* parameter, to indicate that the attribute previously had no value.

traits.trait_base.**Missing = <missing>**
Singleton object that indicates that a method argument is missing from a type-checked method signature.

traits.trait_base.**Self = <self>**
Singleton object that references the current 'object'.

Functions

traits.trait_base.**strx**(*arg*)
Wraps the built-in str() function to raise a TypeError if the argument is not of a type in StringTypes.

traits.trait_base.**class_of**(*object*)
Returns a string containing the class name of an object with the correct indefinite article ('a' or 'an') preceding it (e.g., 'an Image', 'a PlotValue').

traits.trait_base.**add_article**(*name*)
Returns a string containing the correct indefinite article ('a' or 'an') prefixed to the specified string.

traits.trait_base.**user_name_for**(*name*)
Returns a "user-friendly" version of a string, with the first letter capitalized and with underscore characters replaced by spaces. For example, `user_name_for('user_name_for')` returns `'User name for'`.

traits.trait_base.**traits_home**()
Gets the path to the Traits home directory.

traits.trait_base.**verify_path**(*path*)
Verify that a specified path exists, and try to create it if it does not exist.

traits.trait_base.**get_module_name**(*level=2*)
Returns the name of the module that the caller's caller is located in.

traits.trait_base.**get_resource_path**(*level=2*)
Returns a resource path calculated from the caller's stack.

traits.trait_base.**xgetattr**(*object*, *xname*, *default=<undefined>*)
Returns the value of an extended object attribute name of the form: name[.name2[.name3...]].

traits.trait_base.**xsetattr**(*object*, *xname*, *value*)
Sets the value of an extended object attribute name of the form: name[.name2[.name3...]].

traits.trait_base.**is_none**(*value*)

traits.trait_base.**not_none**(*value*)

traits.trait_base.**not_false**(*value*)

traits.trait_base.**not_event**(*value*)

traits.trait_base.**is_str**(*value*)

trait_errors Module

Defines the standard exceptions raised by the Traits package.

Functions

traits.trait_errors.**repr_type**(*obj*)
Return a string representation of a value and its type for readable error messages.

Classes

class traits.trait_errors.**TraitError**(*args=None*, *name=None*, *info=None*, *value=None*)

class traits.trait_errors.**TraitNotificationError**

class traits.trait_errors.**DelegationError**(*args*)

trait_handlers Module

Defines the BaseTraitHandler class and a standard set of BaseTraitHandler subclasses for use with the Traits package.

A trait handler mediates the assignment of values to object traits. It verifies (via its validate() method) that a specified value is consistent with the object trait, and generates a TraitError exception if it is not consistent.

Classes

class traits.trait_handlers.**BaseTraitHandler**
The task of this class and its subclasses is to verify the correctness of values assigned to object trait attributes.

This class is an alternative to trait validator functions. A trait handler has several advantages over a trait validator function, due to being an object:

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

error(*object*, *name*, *value*)
Raises a TraitError exception.

Parameters

- **object** (*object*) – The object whose attribute is being assigned.
- **name** (*str*) – The name of the attribute being assigned.
- **value** (*object*) – The proposed new value for the attribute.

Description

This method is called by the validate() method when an assigned value is not valid. Raising a TraitError exception either notifies the user of the problem, or, in the case of compound traits, provides a chance for another trait handler to handle to validate the value.

full_info(*object*, *name*, *value*)
Returns a string describing the type of value accepted by the trait handler.

Parameters

- **object** (*object*) – The object whose attribute is being assigned.
- **name** (*str*) – The name of the attribute being assigned.
- **value** – The proposed new value for the attribute.

Description

The string should be a phrase describing the type defined by the TraitHandler subclass, rather than a complete sentence. For example, use the phrase, "a square sprocket" instead of the sentence, "The value must be a square sprocket." The value returned by full_info() is combined with other information whenever an error occurs and therefore makes more sense to the user if the result is a phrase. The full_info() method is similar in purpose and use to the **info** attribute of a validator function.

Note that the result can include information specific to the particular trait handler instance. For example, TraitRange instances return a string indicating the range of values acceptable to the handler (e.g., "an integer in the range from 1 to 9"). If the full_info() method is not overridden, the default method returns the value of calling the info() method.

info()
Must return a string describing the type of value accepted by the trait handler.

The string should be a phrase describing the type defined by the TraitHandler subclass, rather than a complete sentence. For example, use the phrase, "a square sprocket" instead of the sentence, "The value must be a square sprocket." The value returned by info() is combined with other information whenever an error occurs and therefore makes more sense to the user if the result is a phrase. The info() method is similar in purpose and use to the **info** attribute of a validator function.

Note that the result can include information specific to the particular trait handler instance. For example, TraitRange instances return a string indicating the range of values acceptable to the handler (e.g., "an integer in the range from 1 to 9"). If the info() method is not overridden, the default method returns the value of the 'info_text' attribute.

repr(*value*)
Returns a printable representation of a value along with its type.

Deprecated since version 3.0.3: This functionality was only used to provide readable error messages. This functionality has been incorporated into TraitError itself.

Parameters **value** (*object*) – The value to be printed.

get_editor(*trait=None*)
Returns a trait editor that allows the user to modify the *trait* trait.

Parameters **trait** (*Trait*) – The trait to be edited.

Description

This method only needs to be specified if traits defined using this trait handler require a non-default trait editor in trait user interfaces. The default implementation of this method returns a trait editor that allows the user to type an arbitrary string as the value.

For more information on trait user interfaces, refer to the *Traits UI User Guide*.

create_editor()
Returns the default traits UI editor to use for a trait.

inner_traits()
Returns a tuple containing the *inner traits* for this trait. Most trait handlers do not have any inner traits, and so will return an empty tuple. The exceptions are **List** and **Dict** trait types, which have inner traits used to validate the values assigned to the trait. For example, in *List(Int)*, the *inner traits* for **List** are (**Int**,).

traits.trait_handlers.**NoDefaultSpecified**

class traits.trait_handlers.**TraitType**(*default_value=<traits.trait_handlers.NoDefaultSpecified object at 0x350e190>*, ***metadata*)
Base class for new trait types.

This class enables you to define new traits using a class-based approach, instead of by calling the Trait() factory function with an instance of a TraitHandler derived object.

When subclassing this class, you can implement one or more of the method signatures below. Note that these methods are defined only as comments, because the absence of method definitions in the subclass definition implicitly provides information about how the trait should operate.

The optional methods are as follows:

•get (self, object, name):

This is the getter method of a trait that behaves like a property.

Parameters **object** (*object*) – The object that the property applies to.

name (str) – The name of the property on *object* property.

Description

If neither this method nor the set() method is defined, the value of the trait is handled like a normal object attribute. If this method is not defined, but the set() method is defined, the trait behaves like a write-only property. This method should return the value of the *name* property for the *object* object.

•set (self, object, name, value)

This is the setter method of a trait that behaves like a property.

Parameters **object** (*object*) – The object that the property applies to.

name (str) – The name of the property on *object*.

value – The value being assigned as the value of the property.

Description

If neither this method nor the get() method is implemented, the trait behaves like a normal trait attribute. If this method is not defined, but the get() method is defined, the trait behaves like a read-only property. This method does not need to return a value, but it should raise a TraitError exception if the specified *value* is not valid and cannot be coerced or adapted to a valid value.

•validate (self, object, name, value)

This method validates, coerces, or adapts the specified *value* as the value of the *name* trait of the *object* object. This method is called when a value is assigned to an object trait that is based on this subclass of *TraitType* and the class does not contain a definition for either the get() or set() methods. This method must return the original *value* or any suitably coerced or adapted value that is a legal value for the trait. If *value* is not a legal value for the trait, and cannot be coerced or adapted to a legal value, the method should either raise a **TraitError** or call the **error** method to raise the **TraitError** on its behalf.

•is_valid_for (self, value)

As an alternative to implementing the **validate** method, you can instead implement the **is_valid_for** method, which receives only the *value* being assigned. It should return **True** if the value is valid, and **False** otherwise.

•value_for (self, value)

As another alternative to implementing the **validate** method, you can instead implement the **value_for** method, which receives only the *value* being assigned. It should return the validated form of *value* if it is valid, or raise a **TraitError** if the value is not valid.

•post_setattr (self, object, name, value)

This method allows the trait to do additional processing after *value* has been successfully assigned to the *name* trait of the *object* object. For most traits there is no additional processing that needs to be done, and this method need not be defined. It is normally used for creating "shadow" (i.e., "mapped" traits), but other uses may arise as well. This method does not need to return a value, and should normally not raise any exceptions.

init()

Allows the trait to perform any additional initialization needed.

get_default_value()

Returns a tuple of the form: (*default_value_type*, *default_value*) which describes the default value for this trait. The default implementation analyzes the value of the trait's **default_value** attribute and determines an appropriate *default_value_type* for *default_value*. If you need to override this method to provide a different result tuple, the following values are valid values for *default_value_type*:

•0, 1: The *default_value* item of the tuple is the default value.

•2: The object containing the trait is the default value.

•3: A new copy of the list specified by *default_value* is the default value.

•4: A new copy of the dictionary specified by *default_value* is the default value.

•5: A new instance of TraitListObject constructed using the *default_value* list is the default value.

•6: A new instance of TraitDictObject constructed using the *default_value* dictionary is the default value.

•7: *default_value* is a tuple of the form: (*callable*, *args*, *kw*), where *callable* is a callable, *args* is a tuple, and *kw* is either a dictionary or None. The default value is the result obtained by invoking callable(*args, **kw).

•8: *default_value* is a callable. The default value is the result obtained by invoking *default_value*(*object*), where *object* is the object containing the trait. If the trait has a validate() method, the validate() method is also called to validate the result.

•9: A new instance of TraitSetObject constructed using the *default_value* set is the default value.

clone(*default_value=<missing>*, ***metadata*)
Clones the contents of this object into a new instance of the same class, and then modifies the cloned copy using the specified *default_value* and *metadata*. Returns the cloned object as the result.

Note that subclasses can change the signature of this method if needed, but should always call the 'super' method if possible.

get_value(*object*, *name*, *trait=None*)
Returns the current value of a property-based trait.

set_value(*object*, *name*, *value*)
Sets the cached value of a property-based trait and fires the appropriate trait change event.

as_ctrait()
Returns a CTrait corresponding to the trait defined by this class.

class traits.trait_handlers.**TraitHandler**
The task of this class and its subclasses is to verify the correctness of values assigned to object trait attributes.

This class is an alternative to trait validator functions. A trait handler has several advantages over a trait validator function, due to being an object:

•Trait handlers have constructors and state. Therefore, you can use them to create *parametrized types*.

•Trait handlers can have multiple methods, whereas validator functions can have only one callable interface. This feature allows more flexibility in their implementation, and allows them to handle a wider range of cases, such as interactions with other components.

The only method of TraitHandler that *must* be implemented by subclasses is validate().

validate(*object*, *name*, *value*)
Verifies whether a new value assigned to a trait attribute is valid.

Parameters

- **object** (*object*) – The object whose attribute is being assigned.
- **name** (*str*) – The name of the attribute being assigned.
- **value** – The proposed new value for the attribute.

Returns If the new value is valid, this method must return either the original value passed to it, or an alternate value to be assigned in place of the original value. Whatever value this method returns is the actual value assigned to *object.name*.

Description

This method *must* be implemented by subclasses of TraitHandler. It is called whenever a new value is assigned to a trait attribute defined using this trait handler.

If the value received by validate() is not valid for the trait attribute, the method must called the predefined error() method to raise a TraitError exception

class traits.trait_handlers.**TraitRange**(*low=None*, *high=None*, *exclude_low=False*, *exclude_high=False*)

Ensures that a trait attribute lies within a specified numeric range.

TraitRange is the underlying handler for the predefined Range() trait factory.

Any value assigned to a trait containing a TraitRange handler must be of the correct type and in the numeric range defined by the TraitRange instance. No automatic coercion takes place. For example:

```
class Person(HasTraits):
    age = Trait(0, TraitRange(0, 150))
    weight = Trait(0.0, TraitRange(0.0, None))
```

This example defines a Person class, which has an **age** trait attribute, which must be an integer/long in the range from 0 to 150, and a **weight** trait attribute, which must be a non-negative float value.

class traits.trait_handlers.**TraitString**(*minlen=0*, *maxlen=9223372036854775807*, *regex=''*)

Ensures that a trait attribute value is a string that satisfied some additional, optional constraints.

The optional constraints include minimum and maximum lengths, and a regular expression that the string must match.

If the value assigned to the trait attribute is a Python numeric type, the TraitString handler first coerces the value to a string. Values of other non-string types result in a TraitError being raised. The handler then makes sure that the resulting string is within the specified length range and that it matches the regular expression.

Example

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

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

class traits.trait_handlers.**TraitCoerceType**(*aType*)

Ensures that a value assigned to a trait attribute is of a specified Python type, or can be coerced to the specified type.

TraitCoerceType is the underlying handler for the predefined traits and factories for Python simple types. The TraitCoerceType class is also an example of a parametrized type, because the single TraitCoerceType class allows creating instances that check for totally different sets of values. For example:

```
class Person(HasTraits):
    name = Trait('', TraitCoerceType(''))
    weight = Trait(0.0, TraitCoerceType(float))
```

In this example, the **name** attribute must be of type `str` (string), while the **weight** attribute must be of type `float`, although both are based on instances of the TraitCoerceType class. Note that this example is essentially the same as writing:

```
class Person(HasTraits):
    name = Trait('')
    weight = Trait(0.0)
```

This simpler form is automatically changed by the Trait() function into the first form, based on TraitCoerceType instances, when the trait attributes are defined.

For attributes based on TraitCoerceType instances, if a value that is assigned is not of the type defined for the trait, a TraitError exception is raised. However, in certain cases, if the value can be coerced to the required type,

then the coerced value is assigned to the attribute. Only *widening* coercions are allowed, to avoid any possible loss of precision. The following table lists the allowed coercions.

Trait Type	Coercible Types
complex	float, int
float	int
long	int
unicode	str

class `traits.trait_handlers.`**`TraitCastType`**(*aType*)

Ensures that a value assigned to a trait attribute is of a specified Python type, or can be cast to the specified type.

This class is similar to TraitCoerceType, but uses casting rather than coercion. Values are cast by calling the type with the value to be assigned as an argument. When casting is performed, the result of the cast is the value assigned to the trait attribute.

Any trait that uses a TraitCastType instance in its definition ensures that its value is of the type associated with the TraitCastType instance. For example:

```
class Person(HasTraits):
    name = Trait('', TraitCastType(''))
    weight = Trait(0.0, TraitCastType(float))
```

In this example, the **name** trait must be of type `str` (string), while the **weight** trait must be of type `float`. Note that this example is essentially the same as writing:

```
class Person(HasTraits):
    name = CStr
    weight = CFloat
```

To understand the difference between TraitCoerceType and TraitCastType (and also between Float and CFloat), consider the following example:

```
>>>class Person(HasTraits):
...     weight = Float
...     cweight = CFloat
>>>
>>>bill = Person()
>>>bill.weight = 180    # OK, coerced to 180.0
>>>bill.cweight = 180   # OK, cast to 180.0
>>>bill.weight = '180'  # Error, invalid coercion
>>>bill.cweight = '180' # OK, cast to float('180')
```

class `traits.trait_handlers.`**`ThisClass`**(*allow_none=False*)

Ensures that the trait attribute values belong to the same class (or a subclass) as the object containing the trait attribute.

ThisClass is the underlying handler for the predefined traits **This** and **self**, and the elements of ListThis.

class `traits.trait_handlers.`**`TraitInstance`**(*aClass*, *allow_none=True*, *adapt='no'*, *module=''*)

Ensures that trait attribute values belong to a specified Python class or type.

TraitInstance is the underlying handler for the predefined trait **Instance** and the elements of List(Instance).

Any trait that uses a TraitInstance handler ensures that its values belong to the specified type or class (or one of its subclasses). For example:

```
class Employee(HasTraits):
    manager = Trait(None, TraitInstance(Employee, True))
```

This example defines a class Employee, which has a **manager** trait attribute, which accepts either None or an instance of Employee as its value.

TraitInstance ensures that assigned values are exactly of the type specified (i.e., no coercion is performed).

class traits.trait_handlers.**TraitWeakRef**(*aClass*, *allow_none=True*, *adapt='no'*, *module=''*)

class traits.trait_handlers.**HandleWeakRef**(*object*, *name*, *value*)

class traits.trait_handlers.**TraitClass**(*aClass*)
Ensures that trait attribute values are subclasses of a specified class (or the class itself).

A value is valid if it is a subclass of the specified class (including the class itself), or it is a string that is equivalent to the name of a valid class.

class traits.trait_handlers.**TraitFunction**(*aFunc*)
Ensures that assigned trait attribute values are acceptable to a specified validator function.

TraitFunction is the underlying handler for the predefined trait **Function**, and for the use of function references as arguments to the Trait() function.

class traits.trait_handlers.**TraitEnum**(**values*)
Ensures that a value assigned to a trait attribute is a member of a specified list of values.

TraitEnum is the underlying handler for the forms of the Trait() function that take a list of possible values

class traits.trait_handlers.**TraitPrefixList**(**values*)
Ensures that a value assigned to a trait attribute is a member of a list of specified string values, or is a unique prefix of one of those values.

TraitPrefixList is a variation on TraitEnum. The values that can be assigned to a trait attribute defined using a TraitPrefixList handler is the set of all strings supplied to the TraitPrefixList constructor, as well as any unique prefix of those strings. That is, if the set of strings supplied to the constructor is described by [s_1, s_2, ..., s_n], then the string v is a valid value for the trait if $v == s_{i[:j]}$ for one and only one pair of values (i, j). If v is a valid value, then the actual value assigned to the trait attribute is the corresponding s_i value that v matched.

Example

```
class Person(HasTraits):
    married = Trait('no', TraitPrefixList('yes', 'no')
```

The Person class has a **married** trait that accepts any of the strings 'y', 'ye', 'yes', 'n', or 'no' as valid values. However, the actual values assigned as the value of the trait attribute are limited to either 'yes' or 'no'. That is, if the value 'y' is assigned to the **married** attribute, the actual value assigned will be 'yes'.

Note that the algorithm used by TraitPrefixList in determining whether a string is a valid value is fairly efficient in terms of both time and space, and is not based on a brute force set of comparisons.

class traits.trait_handlers.**TraitMap**(*map*)
Checks that the value assigned to a trait attribute is a key of a specified dictionary, and also assigns the dictionary value corresponding to that key to a *shadow* attribute.

A trait attribute that uses a TraitMap handler is called *mapped* trait attribute. In practice, this means that the resulting object actually contains two attributes: one whose value is a key of the TraitMap dictionary, and the other whose value is the corresponding value of the TraitMap dictionary. The name of the shadow attribute is simply the base attribute name with an underscore ('_') appended. Mapped trait attributes can be used to allow a variety of user-friendly input values to be mapped to a set of internal, program-friendly values.

Example

```
>>>class Person(HasTraits):
...     married = Trait('yes', TraitMap({'yes': 1, 'no': 0 })
>>>
>>>bob = Person()
>>>print bob.married
yes
>>>print bob.married_
1
```

In this example, the default value of the **married** attribute of the Person class is 'yes'. Because this attribute is defined using TraitPrefixList, instances of Person have another attribute, **married_**, whose default value is 1, the dictionary value corresponding to the key 'yes'.

class traits.trait_handlers.**TraitPrefixMap**(*map*)
A cross between the TraitPrefixList and TraitMap classes.

Like TraitMap, TraitPrefixMap is created using a dictionary, but in this case, the keys of the dictionary must be strings. Like TraitPrefixList, a string *v* is a valid value for the trait attribute if it is a prefix of one and only one key *k* in the dictionary. The actual values assigned to the trait attribute is *k*, and its corresponding mapped attribute is *map*[*k*].

Example

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

This example defines a Boolean trait that accepts any prefix of 'true', 'yes', 'false', or 'no', and maps them to 1 or 0.

class traits.trait_handlers.**TraitExpression**
Ensures that a value assigned to a trait attribute is a valid Python expression. The compiled form of a valid expression is stored as the mapped value of the trait.

class traits.trait_handlers.**TraitCompound**(**handlers*)
Provides a logical-OR combination of other trait handlers.

This class provides a means of creating complex trait definitions by combining several simpler trait definitions. TraitCompound is the underlying handler for the general forms of the Trait() function.

A value is a valid value for a trait attribute based on a TraitCompound instance if the value is valid for at least one of the TraitHandler or trait objects supplied to the constructor. In addition, if at least one of the TraitHandler or trait objects is mapped (e.g., based on a TraitMap or TraitPrefixMap instance), then the TraitCompound is also mapped. In this case, any non-mapped traits or trait handlers use identity mapping.

class traits.trait_handlers.**TraitTuple**(**args*)
Ensures that values assigned to a trait attribute are tuples of a specified length, with elements that are of specified types.

TraitTuple is the underlying handler for the predefined trait **Tuple**, and the trait factory Tuple().

Example

```
rank = Range(1, 13)
suit = Trait('Hearts', 'Diamonds', 'Spades', 'Clubs')
```

```
class Card(HasTraits):
    value = Trait(TraitTuple(rank, suit))
```

This example defines a Card class, which has a **value** trait attribute, which must be a tuple of two elments. The first element must be an integer in the range from 1 to 13, and the second element must be one of the four strings, 'Hearts', 'Diamonds', 'Spades', or 'Clubs'.

class traits.trait_handlers.**TraitCallable**
Ensures that the value of a trait attribute is a callable Python object (usually a function or method).

class traits.trait_handlers.**TraitListEvent**(*index=0*, *removed=None*, *added=None*)

class traits.trait_handlers.**TraitList**(*trait=None*, *minlen=0*, *maxlen=9223372036854775807*, *has_items=True*)
Ensures that a value assigned to a trait attribute is a list containing elements of a specified type, and that the length of the list is also within a specified range.

TraitList also makes sure that any changes made to the list after it is assigned to the trait attribute do not violate the list's type and length constraints. TraitList is the underlying handler for the predefined list-based traits.

Example

```
class Card(HasTraits):
    pass
class Hand(HasTraits):
    cards = Trait([], TraitList(Trait(Card), maxlen=52))
```

This example defines a Hand class, which has a **cards** trait attribute, which is a list of Card objects and can have from 0 to 52 items in the list.

class traits.trait_handlers.**TraitListObject**(*trait*, *object*, *name*, *value*)

class traits.trait_handlers.**TraitSetEvent**(*removed=None*, *added=None*)

class traits.trait_handlers.**TraitSetObject**(*trait*, *object*, *name*, *value*)

copy()
Return a true `set` object with a copy of the data.

class traits.trait_handlers.**TraitDictEvent**(*added=None*, *changed=None*, *removed=None*)

class traits.trait_handlers.**TraitDict**(*key_trait=None*, *value_trait=None*, *has_items=True*)
Ensures that values assigned to a trait attribute are dictionaries whose keys and values are of specified types.

TraitDict also makes sure that any changes to keys or values made that are made after the dictionary is assigned to the trait attribute satisfy the type constraints. TraitDict is the underlying handler for the dictionary-based predefined traits, and the Dict() trait factory.

Example

```
class WorkoutClass(HasTraits):
    member_weights = Trait({}, TraitDict(str, float))
```

This example defines a WorkoutClass class containing a *member_weights* trait attribute whose value must be a dictionary containing keys that are strings (i.e., the members' names) and whose associated values must be floats (i.e., their most recently recorded weight).

class traits.trait_handlers.**TraitDictObject**(*trait*, *object*, *name*, *value*)

Private Functions

traits.trait_handlers.**_arg_count**(*func*)
Returns the correct argument count for a specified function or method.

traits.trait_handlers.**_write_only**(*object*, *name*)

traits.trait_handlers.**_read_only**(*object*, *name*, *value*)

traits.trait_handlers.**_undefined_get**(*object*, *name*)

traits.trait_handlers.**_undefined_set**(*object*, *name*, *value*)

trait_numeric Module

Trait definitions related to the numpy library.

Classes

class traits.trait_numeric.**AbstractArray**(*dtype=None*, *shape=None*, *value=None*, *coerce=False*, *typecode=None*, ***metadata*)
Abstract base class for defining numpy-based arrays.

validate(*object*, *name*, *value*)
Validates that the value is a valid array.

info()
Returns descriptive information about the trait.

create_editor()
Returns the default UI editor for the trait.

get_default_value()
Returns the default value constructor for the type (called from the trait factory.

copy_default_value(*value*)
Returns a copy of the default value (called from the C code on first reference to a trait with no current value).

class traits.trait_numeric.**Array**(*dtype=None*, *shape=None*, *value=None*, *typecode=None*, ***metadata*)
Defines a trait whose value must be a numpy array.

class traits.trait_numeric.**CArray**(*dtype=None*, *shape=None*, *value=None*, *typecode=None*, ***metadata*)
Defines a trait whose value must be a numpy array, with casting allowed.

Function

traits.trait_numeric.**dtype2trait**(*dtype*)
Get the corresponding trait for a numpy dtype.

trait_types Module

Core Trait definitions.

Traits

class traits.trait_types.**Any**(*default_value=<traits.trait_handlers.NoDefaultSpecified object at 0x350e190>*, ***metadata*)

Defines a trait whose value can be anything.

default_value = None

The default value for the trait:

info_text = 'any value'

A description of the type of value this trait accepts:

class traits.trait_types.**Generic**(*default_value=<traits.trait_handlers.NoDefaultSpecified object at 0x350e190>*, ***metadata*)

Defines a trait whose value can be anything and whose definition can be redefined via assignment using a TraitValue object.

metadata = {'trait_value': True}

The standard metadata for the trait:

class traits.trait_types.**BaseInt**(*default_value=<traits.trait_handlers.NoDefaultSpecified object at 0x350e190>*, ***metadata*)

Defines a trait whose type must be an int or long.

evaluate

The function to use for evaluating strings to this type:

alias of int

default_value = 0

The default value for the trait:

info_text = 'an integer (int or long)'

A description of the type of value this trait accepts:

validate(*object*, *name*, *value*)

Validates that a specified value is valid for this trait.

create_editor()

Returns the default traits UI editor for this type of trait.

class traits.trait_types.**Int**(*default_value=<traits.trait_handlers.NoDefaultSpecified object at 0x350e190>*, ***metadata*)

Defines a trait whose type must be an int or long using a C-level fast validator.

fast_validate = (20,)

The C-level fast validator to use:

class traits.trait_types.**BaseLong**(*default_value=<traits.trait_handlers.NoDefaultSpecified object at 0x350e190>*, ***metadata*)

Defines a trait whose value must be a Python long.

evaluate

The function to use for evaluating strings to this type:

alias of long

default_value = 0L

The default value for the trait:

info_text = 'a long'

A description of the type of value this trait accepts:

validate(*object*, *name*, *value*)
Validates that a specified value is valid for this trait.

Note: The 'fast validator' version performs this check in C.

create_editor()
Returns the default traits UI editor for this type of trait.

class traits.trait_types.**Long**(*default_value=<traits.trait_handlers.NoDefaultSpecified object at 0x350e190>*, ***metadata*)
Defines a trait whose value must be a Python long using a C-level fast validator.

fast_validate = (11, <type 'long'>, None, <type 'int'>)
The C-level fast validator to use:

class traits.trait_types.**BaseFloat**(*default_value=<traits.trait_handlers.NoDefaultSpecified object at 0x350e190>*, ***metadata*)
Defines a trait whose value must be a Python float.

evaluate
The function to use for evaluating strings to this type:

alias of `float`

default_value = 0.0
The default value for the trait:

info_text = 'a float'
A description of the type of value this trait accepts:

validate(*object*, *name*, *value*)
Validates that a specified value is valid for this trait.

Note: The 'fast validator' version performs this check in C.

create_editor()
Returns the default traits UI editor for this type of trait.

class traits.trait_types.**Float**(*default_value=<traits.trait_handlers.NoDefaultSpecified object at 0x350e190>*, ***metadata*)
Defines a trait whose value must be a Python float using a C-level fast validator.

fast_validate = (11, <type 'float'>, None, <type 'int'>)
The C-level fast validator to use:

class traits.trait_types.**BaseComplex**(*default_value=<traits.trait_handlers.NoDefaultSpecified object at 0x350e190>*, ***metadata*)
Defines a trait whose value must be a Python complex.

evaluate
The function to use for evaluating strings to this type:

alias of `complex`

default_value = 0j
The default value for the trait:

info_text = 'a complex number'
A description of the type of value this trait accepts:

validate(*object*, *name*, *value*)
Validates that a specified value is valid for this trait.

Note: The 'fast validator' version performs this check in C.

create_editor()
Returns the default traits UI editor for this type of trait.

class traits.trait_types.**Complex**(*default_value=<traits.trait_handlers.NoDefaultSpecified object at 0x350e190>*, ***metadata*)
Defines a trait whose value must be a Python complex using a C-level fast validator.

fast_validate = (11, <type 'complex'>, None, <type 'float'>, <type 'int'>)
The C-level fast validator to use:

class traits.trait_types.**BaseStr**(*default_value=<traits.trait_handlers.NoDefaultSpecified object at 0x350e190>*, ***metadata*)
Defines a trait whose value must be a Python string.

default_value = ''
The default value for the trait:

info_text = 'a string'
A description of the type of value this trait accepts:

validate(*object*, *name*, *value*)
Validates that a specified value is valid for this trait.

Note: The 'fast validator' version performs this check in C.

create_editor()
Returns the default traits UI editor for this type of trait.

class traits.trait_types.**Str**(*default_value=<traits.trait_handlers.NoDefaultSpecified object at 0x350e190>*, ***metadata*)
Defines a trait whose value must be a Python string using a C-level fast validator.

fast_validate = (11, <type 'basestring'>)
The C-level fast validator to use:

class traits.trait_types.**Title**(*default_value=<traits.trait_handlers.NoDefaultSpecified object at 0x350e190>*, ***metadata*)
Defines a string type which by default uses the traits ui TitleEditor when used in a View.

create_editor()
Returns the default traits UI editor to use for a trait.

class traits.trait_types.**BaseUnicode**(*default_value=<traits.trait_handlers.NoDefaultSpecified object at 0x350e190>*, ***metadata*)
Defines a trait whose value must be a Python unicode string.

default_value = u''
The default value for the trait:

info_text = 'a unicode string'
A description of the type of value this trait accepts:

validate(*object*, *name*, *value*)
Validates that a specified value is valid for this trait.

Note: The 'fast validator' version performs this check in C.

create_editor()
Returns the default traits UI editor for this type of trait.

class traits.trait_types.**Unicode**(*default_value=<traits.trait_handlers.NoDefaultSpecified object at 0x350e190>*, ***metadata*)
Defines a trait whose value must be a Python unicode string using a C-level fast validator.

fast_validate = (11, <type 'unicode'>, None, <type 'str'>)
The C-level fast validator to use:

class traits.trait_types.**BaseBool**(*default_value=<traits.trait_handlers.NoDefaultSpecified object at 0x350e190>*, ***metadata*)
Defines a trait whose value must be a Python boolean.

evaluate
The function to use for evaluating strings to this type:

alias of bool

default_value = False
The default value for the trait:

info_text = 'a boolean'
A description of the type of value this trait accepts:

validate(*object*, *name*, *value*)
Validates that a specified value is valid for this trait.

Note: The 'fast validator' version performs this check in C.

create_editor()
Returns the default traits UI editor for this type of trait.

class traits.trait_types.**Bool**(*default_value=<traits.trait_handlers.NoDefaultSpecified object at 0x350e190>*, ***metadata*)
Defines a trait whose value must be a Python boolean using a C-level fast validator.

fast_validate = (11, <type 'bool'>)
The C-level fast validator to use:

class traits.trait_types.**BaseCInt**(*default_value=<traits.trait_handlers.NoDefaultSpecified object at 0x350e190>*, ***metadata*)
Defines a trait whose value must be a Python int and which supports coercions of non-int values to int.

evaluate
The function to use for evaluating strings to this type:

alias of int

validate(*object*, *name*, *value*)
Validates that a specified value is valid for this trait.

Note: The 'fast validator' version performs this check in C.

class traits.trait_types.**CInt**(*default_value=<traits.trait_handlers.NoDefaultSpecified object at 0x350e190>*, ***metadata*)
Defines a trait whose value must be a Python int and which supports coercions of non-int values to int using a C-level fast validator.

fast_validate = (12, <type 'int'>)
The C-level fast validator to use:

class traits.trait_types.**BaseCLong**(*default_value=<traits.trait_handlers.NoDefaultSpecified object at 0x350e190>*, ***metadata*)
Defines a trait whose value must be a Python long and which supports coercions of non-long values to long.

evaluate
The function to use for evaluating strings to this type:

alias of long

validate(*object*, *name*, *value*)
Validates that a specified value is valid for this trait.

Note: The 'fast validator' version performs this check in C.

class traits.trait_types.**CLong**(*default_value=<traits.trait_handlers.NoDefaultSpecified object at 0x350e190>*, ***metadata*)
Defines a trait whose value must be a Python long and which supports coercions of non-long values to long using a C-level fast validator.

fast_validate = (12, <type 'long'>)
The C-level fast validator to use:

class traits.trait_types.**BaseCFloat**(*default_value=<traits.trait_handlers.NoDefaultSpecified object at 0x350e190>*, ***metadata*)
Defines a trait whose value must be a Python float and which supports coercions of non-float values to float.

evaluate
The function to use for evaluating strings to this type:

alias of `float`

validate(*object*, *name*, *value*)
Validates that a specified value is valid for this trait.

Note: The 'fast validator' version performs this check in C.

class traits.trait_types.**CFloat**(*default_value=<traits.trait_handlers.NoDefaultSpecified object at 0x350e190>*, ***metadata*)
Defines a trait whose value must be a Python float and which supports coercions of non-float values to float using a C-level fast validator.

fast_validate = (12, <type 'float'>)
The C-level fast validator to use:

class traits.trait_types.**BaseCComplex**(*default_value=<traits.trait_handlers.NoDefaultSpecified object at 0x350e190>*, ***metadata*)
Defines a trait whose value must be a Python complex and which supports coercions of non-complex values to complex.

evaluate
The function to use for evaluating strings to this type:

alias of `complex`

validate(*object*, *name*, *value*)
Validates that a specified value is valid for this trait.

Note: The 'fast validator' version performs this check in C.

class traits.trait_types.**CComplex**(*default_value=<traits.trait_handlers.NoDefaultSpecified object at 0x350e190>*, ***metadata*)
Defines a trait whose value must be a Python complex and which supports coercions of non-complex values to complex using a C-level fast validator.

fast_validate = (12, <type 'complex'>)
The C-level fast validator to use:

class traits.trait_types.**BaseCStr**(*default_value=<traits.trait_handlers.NoDefaultSpecified object at 0x350e190>*, ***metadata*)
Defines a trait whose value must be a Python string and which supports coercions of non-string values to string.

validate(*object*, *name*, *value*)
Validates that a specified value is valid for this trait.

Note: The 'fast validator' version performs this check in C.

class traits.trait_types.**CStr**(*default_value=<traits.trait_handlers.NoDefaultSpecified object at 0x350e190>*, ***metadata*)

Defines a trait whose value must be a Python string and which supports coercions of non-string values to string using a C-level fast validator.

fast_validate = (7, ((12, <type 'str'>), (12, <type 'unicode'>)))

The C-level fast validator to use:

class traits.trait_types.**BaseCUnicode**(*default_value=<traits.trait_handlers.NoDefaultSpecified object at 0x350e190>*, ***metadata*)

Defines a trait whose value must be a Python unicode string and which supports coercions of non-unicode values to unicode.

validate(*object*, *name*, *value*)

Validates that a specified value is valid for this trait.

Note: The 'fast validator' version performs this check in C.

class traits.trait_types.**CUnicode**(*default_value=<traits.trait_handlers.NoDefaultSpecified object at 0x350e190>*, ***metadata*)

Defines a trait whose value must be a Python unicode string and which supports coercions of non-unicode values to unicode using a C-level fast validator.

fast_validate = (12, <type 'unicode'>)

The C-level fast validator to use:

class traits.trait_types.**BaseCBool**(*default_value=<traits.trait_handlers.NoDefaultSpecified object at 0x350e190>*, ***metadata*)

Defines a trait whose value must be a Python boolean and which supports coercions of non-boolean values to boolean.

evaluate

The function to use for evaluating strings to this type:

alias of bool

validate(*object*, *name*, *value*)

Validates that a specified value is valid for this trait.

Note: The 'fast validator' version performs this check in C.

class traits.trait_types.**CBool**(*default_value=<traits.trait_handlers.NoDefaultSpecified object at 0x350e190>*, ***metadata*)

Defines a trait whose value must be a Python boolean and which supports coercions of non-boolean values to boolean using a C-level fast validator.

fast_validate = (12, <type 'bool'>)

The C-level fast validator to use:

class traits.trait_types.**String**(*value=''*, *minlen=0*, *maxlen=9223372036854775807*, *regex=''*, ***metadata*)

Defines a trait whose value must be a Python string whose length is optionally in a specified range, and which optionally matches a specified regular expression.

validate(*object*, *name*, *value*)

Validates that the value is a valid string.

validate_all(*object*, *name*, *value*)

Validates that the value is a valid string in the specified length range which matches the specified regular expression.

validate_str(*object*, *name*, *value*)
Validates that the value is a valid string.

validate_len(*object*, *name*, *value*)
Validates that the value is a valid string in the specified length range.

validate_regex(*object*, *name*, *value*)
Validates that the value is a valid string which matches the specified regular expression.

info()
Returns a description of the trait.

create_editor()
Returns the default traits UI editor for this type of trait.

class traits.trait_types.**Regex**(*value=''*, *regex='.*'*, ***metadata*)
Defines a trait whose value is a Python string that matches a specified regular expression.

class traits.trait_types.**Code**(*value=''*, *minlen=0*, *maxlen=9223372036854775807*, *regex=''*, ***metadata*)
Defines a trait whose value is a Python string that represents source code in some language.

metadata = {'editor': <function code_editor at 0x3570398>}
The standard metadata for the trait:

class traits.trait_types.**HTML**(*value=''*, *minlen=0*, *maxlen=9223372036854775807*, *regex=''*, ***metadata*)
Defines a trait whose value must be a string that is interpreted as being HTML. By default the value is parsed and displayed as HTML in TraitsUI views. The validation of the value does not enforce HTML syntax.

metadata = {'editor': <function html_editor at 0x3570410>}
The standard metadata for the trait:

class traits.trait_types.**Password**(*value=''*, *minlen=0*, *maxlen=9223372036854775807*, *regex=''*, ***metadata*)
Defines a trait whose value must be a string, optionally of constrained length or matching a regular expression.

The trait is identical to a String trait except that by default it uses a PasswordEditor in TraitsUI views, which obscures text entered by the user.

metadata = {'editor': <function password_editor at 0x35702a8>}
The standard metadata for the trait:

class traits.trait_types.**Callable**(*default_value=<traits.trait_handlers.NoDefaultSpecified object at 0x350e190>*, ***metadata*)
Defines a trait whose value must be a Python callable.

metadata = {'copy': 'ref'}
The standard metadata for the trait:

default_value = None
The default value for the trait:

info_text = 'a callable value'
A description of the type of value this trait accepts:

validate(*object*, *name*, *value*)
Validates that the value is a Python callable.

class traits.trait_types.**BaseType**(*default_value=<traits.trait_handlers.NoDefaultSpecified object at 0x350e190>*, ***metadata*)
Defines a trait whose value must be an instance of a simple Python type.

validate(*object*, *name*, *value*)
Validates that the value is a Python callable.

class traits.trait_types.**This**(*value=None*, *allow_none=True*, ***metadata*)
Defines a trait whose value must be an instance of the defining class.

info_text = 'an instance of the same type as the receiver'
A description of the type of value this trait accepts:

fast_validate = (2,)
The C-level fast validator to use:

class traits.trait_types.**self**(*value=None*, *allow_none=True*, ***metadata*)
Defines a trait whose value must be an instance of the defining class and whose default value is the object containing the trait.

default_value_type = 2
The default value type to use (i.e. 'self'):

class traits.trait_types.**Function**(*default_value=<traits.trait_handlers.NoDefaultSpecified object at 0x350e190>*, ***metadata*)
Defines a trait whose value must be a Python function.

fast_validate = (11, <type 'function'>)
The C-level fast validator to use:

info_text = 'a function'
A description of the type of value this trait accepts:

class traits.trait_types.**Method**(*default_value=<traits.trait_handlers.NoDefaultSpecified object at 0x350e190>*, ***metadata*)
Defines a trait whose value must be a Python method.

fast_validate = (11, <type 'instancemethod'>)
The C-level fast validator to use:

info_text = 'a method'
A description of the type of value this trait accepts:

class traits.trait_types.**Class**(*default_value=<traits.trait_handlers.NoDefaultSpecified object at 0x350e190>*, ***metadata*)
Defines a trait whose value must be an old-style Python class.

fast_validate = (11, <type 'classobj'>)
The C-level fast validator to use:

info_text = 'an old-style class'
A description of the type of value this trait accepts:

class traits.trait_types.**Module**(*default_value=<traits.trait_handlers.NoDefaultSpecified object at 0x350e190>*, ***metadata*)
Defines a trait whose value must be a Python module.

fast_validate = (11, <type 'module'>)
The C-level fast validator to use:

info_text = 'a module'
A description of the type of value this trait accepts:

class traits.trait_types.**Python**(*default_value=<traits.trait_handlers.NoDefaultSpecified object at 0x350e190>*, ***metadata*)
Defines a trait that provides behavior identical to a standard Python attribute. That is, it allows any value to be assigned, and raises an ValueError if an attempt is made to get the value before one has been assigned. It has no

default value. This trait is most often used in conjunction with wildcard naming. See the *Traits User Manual* for details on wildcards.

metadata = {'type': 'python'}
The standard metadata for the trait:

default_value = <undefined>
The default value for the trait:

traits.trait_types.**ReadOnly**

traits.trait_types.**Disallow**

class traits.trait_types.**Constant**(*value*, ***metadata*)
Defines a trait whose value is a constant.

ctrait_type = 7
Defines the CTrait type to use for this trait:

metadata = {'transient': True, 'type': 'constant'}
The standard metadata for the trait:

class traits.trait_types.**Delegate**(*delegate*, *prefix=''*, *modify=False*, *listenable=True*, ***metadata*)
Defines a trait whose value is delegated to a trait on another object.

ctrait_type = 3
Defines the CTrait type to use for this trait:

metadata = {'transient': False, 'type': 'delegate'}
The standard metadata for the trait:

as_ctrait()
Returns a CTrait corresponding to the trait defined by this class.

class traits.trait_types.**DelegatesTo**(*delegate*, *prefix=''*, *listenable=True*, ***metadata*)
Defines a trait delegate that matches the standard 'delegate' design pattern.

class traits.trait_types.**PrototypedFrom**(*prototype*, *prefix=''*, *listenable=True*, ***metadata*)
Defines a trait delegate that matches the standard 'prototype' design pattern.

class traits.trait_types.**Expression**(*default_value=<traits.trait_handlers.NoDefaultSpecified object at 0x350e190>*, ***metadata*)
Defines a trait whose value must be a valid Python expression. The compiled form of a valid expression is stored as the mapped value of the trait.

default_value = '0'
The default value for the trait:

info_text = 'a valid Python expression'
A description of the type of value this trait accepts:

is_mapped = True
Indicate that this is a mapped trait:

validate(*object*, *name*, *value*)
Validates that a specified value is valid for this trait.

post_setattr(*object*, *name*, *value*)
Performs additional post-assignment processing.

mapped_value(*value*)
Returns the 'mapped' value for the specified **value**.

as_ctrait()
Returns a CTrait corresponding to the trait defined by this class.

class traits.trait_types.**PythonValue**(*default_value=<traits.trait_handlers.NoDefaultSpecified object at 0x350e190>*, ***metadata*)
Defines a trait whose value can be of any type, and whose default editor is a Python shell.

metadata = {'editor': <function shell_editor at 0x3570488>}
The standard metadata for the trait:

class traits.trait_types.**BaseFile**(*value=''*, *filter=None*, *auto_set=False*, *entries=0*, *exists=False*, ***metadata*)
Defines a trait whose value must be the name of a file.

info_text = 'a file name'
A description of the type of value this trait accepts:

validate(*object*, *name*, *value*)
Validates that a specified value is valid for this trait.

Note: The 'fast validator' version performs this check in C.

class traits.trait_types.**File**(*value=''*, *filter=None*, *auto_set=False*, *entries=0*, *exists=False*, ***metadata*)
Defines a trait whose value must be the name of a file using a C-level fast validator.

class traits.trait_types.**BaseDirectory**(*value=''*, *auto_set=False*, *entries=0*, *exists=False*, ***metadata*)
Defines a trait whose value must be the name of a directory.

info_text = 'a directory name'
A description of the type of value this trait accepts:

validate(*object*, *name*, *value*)
Validates that a specified value is valid for this trait.

Note: The 'fast validator' version performs this check in C.

class traits.trait_types.**Directory**(*value=''*, *auto_set=False*, *entries=0*, *exists=False*, ***metadata*)
Defines a trait whose value must be the name of a directory using a C-level fast validator.

class traits.trait_types.**BaseRange**(*low=None*, *high=None*, *value=None*, *exclude_low=False*, *exclude_high=False*, ***metadata*)
Defines a trait whose numeric value must be in a specified range.

init_fast_validator(**args*)
Does nothing for the BaseRange class. Used in the Range class to set up the fast validator.

validate(*object*, *name*, *value*)
Validate that the value is in the specified range.

float_validate(*object*, *name*, *value*)
Validate that the value is a float value in the specified range.

int_validate(*object*, *name*, *value*)
Validate that the value is an int value in the specified range.

long_validate(*object*, *name*, *value*)
Validate that the value is a long value in the specified range.

full_info(*object*, *name*, *value*)
Returns a description of the trait.

create_editor()
Returns the default UI editor for the trait.

class traits.trait_types.**Range**(*low=None*, *high=None*, *value=None*, *exclude_low=False*, *exclude_high=False*, ***metadata*)
Defines a trait whose numeric value must be in a specified range using a C-level fast validator.

init_fast_validator(**args*)
Set up the C-level fast validator.

class traits.trait_types.**BaseEnum**(**args*, ***metadata*)
Defines a trait whose value must be one of a specified set of values.

init_fast_validator(**args*)
Does nothing for the BaseEnum class. Used in the Enum class to set up the fast validator.

validate(*object*, *name*, *value*)
Validates that the value is one of the enumerated set of valid values.

full_info(*object*, *name*, *value*)
Returns a description of the trait.

create_editor()
Returns the default UI editor for the trait.

class traits.trait_types.**Enum**(**args*, ***metadata*)
Defines a trait whose value must be one of a specified set of values using a C-level fast validator.

init_fast_validator(**args*)
Set up the C-level fast validator.

class traits.trait_types.**BaseTuple**(**types*, ***metadata*)
Defines a trait whose value must be a tuple of specified trait types.

init_fast_validator(**args*)
Saves the validation parameters.

validate(*object*, *name*, *value*)
Validates that the value is a valid tuple.

full_info(*object*, *name*, *value*)
Returns a description of the trait.

create_editor()
Returns the default UI editor for the trait.

class traits.trait_types.**Tuple**(**types*, ***metadata*)
Defines a trait whose value must be a tuple of specified trait types using a C-level fast validator.

init_fast_validator(**args*)
Set up the C-level fast validator.

class traits.trait_types.**List**(*trait=None*, *value=None*, *minlen=0*, *maxlen=9223372036854775807*, *items=True*, ***metadata*)
Defines a trait whose value must be a list whose items are of the specified trait type.

validate(*object*, *name*, *value*)
Validates that the values is a valid list.

Note: *object* can be None when validating a default value (see e.g. `clone()`)

---

full_info(*object*, *name*, *value*)
Returns a description of the trait.

create_editor()
Returns the default UI editor for the trait.

inner_traits()
Returns the *inner trait* (or traits) for this trait.

class traits.trait_types.**CList**(*trait=None*, *value=None*, *minlen=0*, *maxlen=9223372036854775807*, *items=True*, ***metadata*)
Defines a trait whose values must be a list whose items are of the specified trait type or which can be coerced to a list whose values are of the specified trait type.

validate(*object*, *name*, *value*)
Validates that the values is a valid list.

full_info(*object*, *name*, *value*)
Returns a description of the trait.

class traits.trait_types.**Set**(*trait=None*, *value=None*, *items=True*, ***metadata*)
Defines a trait whose value must be a set whose items are of the specified trait type.

validate(*object*, *name*, *value*)
Validates that the values is a valid set.

Note: *object* can be None when validating a default value (see e.g. clone())

---

full_info(*object*, *name*, *value*)
Returns a description of the trait.

create_editor()
Returns the default UI editor for the trait.

inner_traits()
Returns the *inner trait* (or traits) for this trait.

class traits.trait_types.**CSet**(*trait=None*, *value=None*, *items=True*, ***metadata*)
Defines a trait whose values must be a set whose items are of the specified trait type or which can be coerced to a set whose values are of the specified trait type.

validate(*object*, *name*, *value*)
Validates that the values is a valid list.

full_info(*object*, *name*, *value*)
Returns a description of the trait.

class traits.trait_types.**Dict**(*key_trait=None*, *value_trait=None*, *value=None*, *items=True*, ***metadata*)
Defines a trait whose value must be a dictionary, optionally with specified types for keys and values.

validate(*object*, *name*, *value*)
Validates that the value is a valid dictionary.

Note: *object* can be None when validating a default value (see e.g. clone())

---

full_info(*object*, *name*, *value*)
Returns a description of the trait.

create_editor()
Returns the default UI editor for the trait.

inner_traits()
Returns the *inner trait* (or traits) for this trait.

class traits.trait_types.**BaseClass**(*default_value=<traits.trait_handlers.NoDefaultSpecified object at 0x350e190>*, ***metadata*)

Base class for types which have an associated class which can be determined dynamically by specifying a string name for the class (e.g. 'package1.package2.module.class'.

Any subclass must define instances with 'klass' and 'module' attributes that contain the string name of the class (or actual class object) and the module name that contained the original trait definition (used for resolving local class names (e.g. 'LocalClass')).

This is an abstract class that only provides helper methods used to resolve the class name into an actual class object.

class traits.trait_types.**BaseInstance**(*klass=None*, *factory=None*, *args=None*, *kw=None*, *allow_none=True*, *adapt=None*, *module=None*, ***metadata*)

Defines a trait whose value must be an instance of a specified class, or one of its subclasses.

validate(*object*, *name*, *value*)

Validates that the value is a valid object instance.

info()

Returns a description of the trait.

get_default_value()

Returns a tuple of the form: (default_value_type, default_value) which describes the default value for this trait.

create_editor()

Returns the default traits UI editor for this type of trait.

init_fast_validate()

Does nothing for the BaseInstance' class. Used by the 'Instance', 'AdaptedTo' and 'AdaptsTo' classes to set up the C-level fast validator.

class traits.trait_types.**Instance**(*klass=None*, *factory=None*, *args=None*, *kw=None*, *allow_none=True*, *adapt=None*, *module=None*, ***metadata*)

Defines a trait whose value must be an instance of a specified class, or one of its subclasses using a C-level fast validator.

init_fast_validate()

Sets up the C-level fast validator.

class traits.trait_types.**Supports**(*klass=None*, *factory=None*, *args=None*, *kw=None*, *allow_none=True*, *adapt=None*, *module=None*, ***metadata*)

A traits whose value must support a specified protocol.

In other words, the value of the trait directly provide, or can be adapted to, the given protocol (Interface or type).

The value of the trait after assignment is the possibly adapted value (i.e., it is the original assigned value if that provides the protocol, or is an adapter otherwise).

The original, unadapted value is stored in a "shadow" attribute with the same name followed by an underscore (e.g., 'foo' and '**foo_**').

post_setattr(*object*, *name*, *value*)

Performs additional post-assignment processing.

as_ctrait()

Returns a CTrait corresponding to the trait defined by this class.

class traits.trait_types.**AdaptsTo**(*klass=None*, *factory=None*, *args=None*, *kw=None*, *allow_none=True*, *adapt=None*, *module=None*, ***metadata*)

A traits whose value must support a specified protocol.

In other words, the value of the trait directly provide, or can be adapted to, the given protocol (Interface or type).

The value of the trait after assignment is the original, unadapted value.

A possibly adapted value is stored in a "shadow" attribute with the same name followed by an underscore (e.g., 'foo' and 'foo_').

class traits.trait_types.**Type**(*value=None*, *klass=None*, *allow_none=True*, ***metadata*)
Defines a trait whose value must be a subclass of a specified class.

validate(*object*, *name*, *value*)
Validates that the value is a valid object instance.

resolve(*object*, *name*, *value*)
Resolves a class originally specified as a string into an actual class, then resets the trait so that future calls will be handled by the normal validate method.

info()
Returns a description of the trait.

get_default_value()
Returns a tuple of the form: (default_value_type, default_value) which describes the default value for this trait.

resolve_default_value()
Resolves a class name into a class so that it can be used to return the class as the default value of the trait.

class traits.trait_types.**Event**(*trait=None*, ***metadata*)

full_info(*object*, *name*, *value*)
Returns a description of the trait.

class traits.trait_types.**Button**(*label=''*, *image=None*, *values_trait=None*, *style='button'*, *orientation='vertical'*, *width_padding=7*, *height_padding=5*, *view=None*, ***metadata*)
Defines a trait whose UI editor is a button.

class traits.trait_types.**ToolbarButton**(*label=''*, *image=None*, *style='toolbar'*, *orientation='vertical'*, *width_padding=2*, *height_padding=2*, ***metadata*)
Defines a trait whose UI editor is a button that can be used on a toolbar.

class traits.trait_types.**Either**(**traits*, ***metadata*)
Defines a trait whose value can be any of of a specified list of traits.

as_ctrait()
Returns a CTrait corresponding to the trait defined by this class.

class traits.trait_types.**Symbol**(*default_value=<traits.trait_handlers.NoDefaultSpecified object at 0x350e190>*, ***metadata*)

info_text = "an object or a string of the form '[package.package...package.]module[:symbol[([arg1,...,argn])]]' specifyi
A description of the type of value this trait accepts:

class traits.trait_types.**UUID**(***metadata*)
Defines a trait whose value is a globally unique UUID (type 4).

info_text = 'a read-only UUID'
A description of the type of value this trait accepts:

validate(*object*, *name*, *value*)
Raises an error, since no values can be assigned to the trait.

class traits.trait_types.**WeakRef**(*klass='traits.has_traits.HasTraits'*, *allow_none=False*, *adapt='yes'*, ***metadata*)

Returns a trait whose value must be an instance of the same type (or a subclass) of the specified *klass*, which can be a class or an instance. Note that the trait only maintains a weak reference to the assigned value.

traits.trait_types.**Date = <traits.trait_types.BaseInstance object at 0x35ddc10>**

Defines a trait whose value must be an instance of a specified class, or one of its subclasses.

traits.trait_types.**Time = <traits.trait_types.BaseInstance object at 0x35ddc90>**

Defines a trait whose value must be an instance of a specified class, or one of its subclasses.

traits.trait_types.**ListInt = <traits.trait_types.List object at 0x35ddd50>**

List of integer values; default value is [].

traits.trait_types.**ListFloat = <traits.trait_types.List object at 0x35dddd0>**

List of float values; default value is [].

traits.trait_types.**ListStr = <traits.trait_types.List object at 0x35ddf50>**

List of string values; default value is [].

traits.trait_types.**ListUnicode = <traits.trait_types.List object at 0x35ddfd0>**

List of Unicode string values; default value is [].

traits.trait_types.**ListComplex = <traits.trait_types.List object at 0x35e6090>**

List of complex values; default value is [].

traits.trait_types.**ListBool = <traits.trait_types.List object at 0x35e6110>**

List of Boolean values; default value is [].

traits.trait_types.**ListFunction = <traits.trait_types.List object at 0x35e6190>**

List of function values; default value is [].

traits.trait_types.**ListMethod = <traits.trait_types.List object at 0x35e6550>**

List of method values; default value is [].

traits.trait_types.**ListClass = <traits.trait_types.List object at 0x35e6510>**

List of class values; default value is [].

traits.trait_types.**ListInstance = <traits.trait_types.List object at 0x35e6450>**

List of instance values; default value is [].

traits.trait_types.**ListThis = <traits.trait_types.List object at 0x35e65d0>**

List of container type values; default value is [].

traits.trait_types.**DictStrAny = <traits.trait_types.Dict object at 0x35e6710>**

Only a dictionary of string:Any values can be assigned; only string keys can be inserted. The default value is {}.

traits.trait_types.**DictStrStr = <traits.trait_types.Dict object at 0x35e6810>**

Only a dictionary of string:string values can be assigned; only string keys with string values can be inserted. The default value is {}.

traits.trait_types.**DictStrInt = <traits.trait_types.Dict object at 0x35e6850>**

Only a dictionary of string:integer values can be assigned; only string keys with integer values can be inserted. The default value is {}.

traits.trait_types.**DictStrLong = <traits.trait_types.Dict object at 0x35e6910>**

Only a dictionary of string:long-integer values can be assigned; only string keys with long-integer values can be inserted. The default value is {}.

traits.trait_types.**DictStrFloat = <traits.trait_types.Dict object at 0x35e69d0>**

Only a dictionary of string:float values can be assigned; only string keys with float values can be inserted. The default value is {}.

traits.trait_types.**DictStrBool = <traits.trait_types.Dict object at 0x35e6a90>**
Only a dictionary of string:bool values can be assigned; only string keys with boolean values can be inserted. The default value is {}.

traits.trait_types.**DictStrList = <traits.trait_types.Dict object at 0x35e6b50>**
Only a dictionary of string:list values can be assigned; only string keys with list values can be assigned. The default value is {}.

Private Classes

class traits.trait_types.**HandleWeakRef**(*object*, *name*, *value*)

Functions

traits.trait_types.**default_text_editor**(*trait*, *type=None*)

trait_value Module

Defines the TraitValue class, used for creating special, dynamic trait values.

Classes

class traits.trait_value.**BaseTraitValue**

as_ctrait(*original_trait*)
Returns the low-level C-based trait for this TraitValue.

class traits.trait_value.**TraitValue**

default = Callable
The callable used to define a default value:

args = Tuple
The positional arguments to pass to the callable default value:

kw = Dict
The keyword arguments to pass to the callable default value:

type = Any
The trait to use as the new trait type:

delegate = Instance(HasTraits)
The object to delegate the new value to:

name = Str
The name of the trait on the delegate object to get the new value from:

Functions

traits.trait_value.**SyncValue**(*delegate*, *name*)

traits.trait_value.**TypeValue**(*type*)

traits.trait_value.**DefaultValue**(*default*, *args=()*, *kw={}*)

traits_listener Module

Defines classes used to implement and manage various trait listener patterns.

traits.traits_listener.**indent**(*text*, *first_line=True*, *n=1*, *width=4*)
Indent lines of text.

Parameters

- **text** (*str*) – The text to indent.
- **first_line** (*bool, optional*) – If False, then the first line will not be indented (default: True).
- **n** (*int, optional*) – The level of indentation (default: 1).
- **width** (*int, optional*) – The number of spaces in each level of indentation (default: 4).

Returns **indented** (*str*)

traits.traits_listener.**is_not_none**(*value*)

traits.traits_listener.**is_none**(*value*)

traits.traits_listener.**not_event**(*value*)

class traits.traits_listener.**ListenerBase**
Bases: traits.has_traits.HasPrivateTraits

register(*new*)
Registers new listeners.

unregister(*old*)
Unregisters any existing listeners.

handle(*object*, *name*, *old*, *new*)
Handles a trait change for a simple trait.

handle_list(*object*, *name*, *old*, *new*)
Handles a trait change for a list trait.

handle_list_items(*object*, *name*, *old*, *new*)
Handles a trait change for a list traits items.

handle_dict(*object*, *name*, *old*, *new*)
Handles a trait change for a dictionary trait.

handle_dict_items(*object*, *name*, *old*, *new*)
Handles a trait change for a dictionary traits items.

class traits.traits_listener.**ListenerItem**
Bases: traits.traits_listener.ListenerBase

name = Str
The name of the trait to listen to:

metadata_name = Str
The name of any metadata that must be present (or not present):

metadata_defined = Bool(True)
Does the specified metadata need to be defined (True) or not defined (False)?

handler = Any
The handler to be called when any listened-to trait is changed:

wrapped_handler_ref = Any
A weakref 'wrapped' version of 'handler':

dispatch = Str
The dispatch mechanism to use when invoking the handler:

priority = Bool(False)
Does the handler go at the beginning (True) or end (False) of the notification handlers list?

next = Instance(ListenerBase)
The next level (if any) of ListenerBase object to be called when any of this object's listened-to traits is changed:

type = Enum(ANY_LISTENER, SRC_LISTENER, DST_LISTENER)
The type of handler being used:

notify = Bool(True)
Should changes to this item generate a notification to the handler?

deferred = Bool(False)
Should registering listeners for items reachable from this listener item be deferred until the associated trait is first read or set?

is_any_trait = Bool(False)
Is this an 'any_trait' change listener, or does it create explicit listeners for each individual trait?

is_list_handler = Bool(False)
Is the associated handler a special list handler that handles both 'foo' and 'foo_items' events by receiving a list of 'deleted' and 'added' items as the 'old' and 'new' arguments?

active = Instance(WeakKeyDictionary, ())
A dictionary mapping objects to a list of all current active (*name*, *type*) listener pairs, where *type* defines the type of listener, one of: (SIMPLE_LISTENER, LIST_LISTENER, DICT_LISTENER).

register(*new*)
Registers new listeners.

unregister(*old*)
Unregisters any existing listeners.

handle_simple(*object*, *name*, *old*, *new*)
Handles a trait change for an intermediate link trait.

handle_dst(*object*, *name*, *old*, *new*)
Handles a trait change for an intermediate link trait when the notification is for the final destination trait.

handle_list(*object*, *name*, *old*, *new*)
Handles a trait change for a list (or set) trait.

handle_list_items(*object*, *name*, *old*, *new*)
Handles a trait change for items of a list (or set) trait.

handle_list_items_special(*object*, *name*, *old*, *new*)
Handles a trait change for items of a list (or set) trait with notification.

handle_dict(*object*, *name*, *old*, *new*)
Handles a trait change for a dictionary trait.

handle_dict_items(*object*, *name*, *old*, *new*)
Handles a trait change for items of a dictionary trait.

handle_error(*obj*, *name*, *old*, *new*)
Handles an invalid intermediate trait change to a handler that must be applied to the final destination object.trait.

class traits.traits_listener.**ListenerGroup**
Bases: traits.traits_listener.ListenerBase

handler = Property
The handler to be called when any listened-to trait is changed

wrapped_handler_ref = Property
A weakref 'wrapped' version of 'handler':

dispatch = Property
The dispatch mechanism to use when invoking the handler:

priority = ListProperty
Does the handler go at the beginning (True) or end (False) of the notification handlers list?

next = ListProperty
The next level (if any) of ListenerBase object to be called when any of this object's listened-to traits is changed

type = ListProperty
The type of handler being used:

notify = ListProperty
Should changes to this item generate a notification to the handler?

deferred = ListProperty
Should registering listeners for items reachable from this listener item be deferred until the associated trait is first read or set?

register(*new*)
Registers new listeners.

unregister(*old*)
Unregisters any existing listeners.

class traits.traits_listener.**ListenerParser**(*text=''*, ***traits*)
Bases: traits.has_traits.HasPrivateTraits

len_text = Int
The length of the string being parsed.

index = Int
The current parse index within the string

next = Property
The next character from the string being parsed

name = Property
The next Python attribute name within the string:

skip_ws = Property
The next non-whitespace character

backspace = Property
Backspaces to the last character processed

listener = Instance(ListenerBase)
The ListenerBase object resulting from parsing **text**

text = Str
The string being parsed

parse()
Parses the text and returns the appropriate collection of ListenerBase objects described by the text.

parse_group(*terminator=']'*)
Parses the contents of a group.

parse_item(*terminator*)
Parses a single, complete listener item or group string.

parse_metadata(*item*)
Parses the metadata portion of a listener item.

error(*msg*)
Raises a syntax error.

class traits.traits_listener.**ListenerNotifyWrapper**(*handler*, *owner*, *id*, *listener*, *target=None*)
Bases: traits.trait_notifiers.TraitChangeNotifyWrapper

listener_deleted(*ref*)

owner_deleted(*ref*)

class traits.traits_listener.**ListenerHandler**(*handler*)
Bases: object

listener_deleted(*ref*)

trait_notifiers Module

Classes that implement and support the Traits change notification mechanism

Classes

class traits.trait_notifiers.**NotificationExceptionHandlerState**(*handler*, *reraise_exceptions*, *locked*)

class traits.trait_notifiers.**NotificationExceptionHandler**

class traits.trait_notifiers.**StaticAnyTraitChangeNotifyWrapper**(*handler*)

class traits.trait_notifiers.**StaticTraitChangeNotifyWrapper**(*handler*)

class traits.trait_notifiers.**TraitChangeNotifyWrapper**(*handler*, *owner*, *target=None*)
Dynamic change notify wrapper.

This class is in charge to dispatch trait change events to dynamic listener, typically created using the *on_trait_change* method, or the decorator with the same name.

dispatch(*handler*, **args*)
Dispatch the event to the listener.

This method is normally the only one that needs to be overridden in a subclass to implement the subclass's dispatch mechanism.

class traits.trait_notifiers.**ExtendedTraitChangeNotifyWrapper**(*handler*, *owner*, *target=None*)
Change notify wrapper for "extended" trait change events..

The "extended notifiers" are set up internally when using extended traits, to add/remove traits listeners when one of the intermediate traits changes.

For example, in a listener for the extended trait *a.b*, we need to add/remove listeners to *a:b* when *a* changes.

class traits.trait_notifiers.**FastUITraitChangeNotifyWrapper**(*handler*, *owner*, *target=None*)

Dynamic change notify wrapper, dispatching on the UI thread.

This class is in charge to dispatch trait change events to dynamic listener, typically created using the *on_trait_change* method and the *dispatch* parameter set to 'ui' or 'fast_ui'.

class traits.trait_notifiers.**NewTraitChangeNotifyWrapper**(*handler*, *owner*, *target=None*)

Dynamic change notify wrapper, dispatching on a new thread.

This class is in charge to dispatch trait change events to dynamic listener, typically created using the *on_trait_change* method and the *dispatch* parameter set to 'new'.

Functions

traits.trait_notifiers.**set_ui_handler**(*handler*)

Sets up the user interface thread handler.

ustr_trait Module

Defines the UStr type and HasUniqueStrings mixin class for efficiently creating lists of objects containing traits whose string values must be unique within the list.

class traits.ustr_trait.**UStr**(*owner*, *list_name*, *str_name*, *default_value=<traits.trait_handlers.NoDefaultSpecified object at 0x350e190>*, ***metadata*)

Bases: traits.trait_handlers.TraitType

Trait type that ensures that a value assigned to a trait is unique within the list it belongs to.

str_type = <traits.trait_value.TraitValue object at 0x5183470>

The type value to assign to restore the original list item type when a list item is removed from the monitored list:

info_text = 'a unique string'

The informational text describing the trait:

validate(*object*, *name*, *value*)

Ensures that a value being assigned to a trait is a unique string.

class traits.ustr_trait.**HasUniqueStrings**

Bases: traits.has_traits.HasTraits

Mixin or base class for objects containing lists with items containing string valued traits that must be unique.

List traits within the class that contain items which have string traits which must be unique should indicate this by attaching metadata of the form:

```
unique_string = 'trait1, trait2, ..., traitn'
```

where each 'traiti' value is the name of a trait within each list item that must contain unique string data.

For example:

```
usa = List( State, unique_string = 'name, abbreviation' )
```

traits_init()
Adds any UStrMonitor objects to list traits with 'unique_string' metadata.

2.1.2 Subpackages

adaptation Package

adaptation Package

Adaptation package.

Copyright 2013 Enthought, Inc.

adaptation_error Module

Exception raised when a requested adaptation is not possible.

exception traits.adaptation.adaptation_error.**AdaptationError**
Bases: exceptions.TypeError

Exception raised when a requested adaptation is not possible.

adaptation_manager Module

Manages all registered adaptations.

traits.adaptation.adaptation_manager.**no_adapter_necessary**(*adaptee*)
An adapter factory used to register that a protocol provides another.

See 'register_provides' for details.

class traits.adaptation.adaptation_manager.**AdaptationManager**
Bases: traits.has_traits.HasTraits

Manages all registered adaptations.

static mro_distance_to_protocol(*from_type*, *to_protocol*)
Return the distance in the MRO from 'from_type' to 'to_protocol'.

If *from_type* provides *to_protocol*, returns the distance between *from_type* and the super-most class in the MRO hierarchy providing *to_protocol* (that's where the protocol was provided in the first place).

If *from_type* does not provide *to_protocol*, return None.

static provides_protocol(*type_*, *protocol*)
Does the given type provide (i.e implement) a given protocol?

'**type_**' is a Python 'type'. 'protocol' is either a regular Python class or a traits Interface.

Return True if the object provides the protocol, otherwise False.

adapt(*adaptee*, *to_protocol*, *default=<class 'traits.adaptation.adaptation_error.AdaptationError'>*)
Attempt to adapt an object to a given protocol.

adaptee is the object that we want to adapt. *to_protocol* is the protocol that the want to adapt the object to.

If *adaptee* already provides (i.e. implements) the given protocol then it is simply returned unchanged.

Otherwise, we try to build a chain of adapters that adapt *adaptee* to *to_protocol*.

If no such adaptation is possible then either an AdaptationError is raised (if default=Adaptation error), or *default* is returned (as in the default value passed to 'getattr' etc).

register_offer(*offer*)
Register an offer to adapt from one protocol to another.

register_factory(*factory*, *from_protocol*, *to_protocol*)
Register an adapter factory.

This is a simply a convenience method that creates and registers an 'AdaptationOffer' from the given arguments.

register_provides(*provider_protocol*, *protocol*)
Register that a protocol provides another.

supports_protocol(*obj*, *protocol*)
Does the object support a given protocol?

An object "supports" a protocol if either it "provides" it directly, or it can be adapted to it.

traits.adaptation.adaptation_manager.**set_global_adaptation_manager**(*new_adaptation_manager*)
Set the global adaptation manager to the given instance.

traits.adaptation.adaptation_manager.**reset_global_adaptation_manager**()
Set the global adaptation manager to a new AdaptationManager instance.

traits.adaptation.adaptation_manager.**get_global_adaptation_manager**()
Set a reference to the global adaptation manager.

traits.adaptation.adaptation_manager.**adapt**(*adaptee*, *to_protocol*, *default=<class 'traits.adaptation.adaptation_error.AdaptationError'>*)
Attempt to adapt an object to a given protocol.

traits.adaptation.adaptation_manager.**register_factory**(*factory*, *from_protocol*, *to_protocol*)
Register an adapter factory.

traits.adaptation.adaptation_manager.**register_offer**(*offer*)
Register an offer to adapt from one protocol to another.

traits.adaptation.adaptation_manager.**register_provides**(*provider_protocol*, *protocol*)
Register that a protocol provides another.

traits.adaptation.adaptation_manager.**supports_protocol**(*obj*, *protocol*)
Does the object support a given protocol?

traits.adaptation.adaptation_manager.**provides_protocol**(*type_*, *protocol*)
Does the given type provide (i.e implement) a given protocol?

adaptation_offer Module

An offer to provide adapters from one protocol to another.

class traits.adaptation.adaptation_offer.**AdaptationOffer**
Bases: traits.has_traits.HasTraits

An offer to provide adapters from one protocol to another.

An adaptation offer consists of a factory that can create adapters, and the protocols that define what the adapters adapt from and to.

factory = Property(Any)
A factory for creating adapters.

The factory must ba callable that takes exactly one argument which is the object to be adapted (known as the adaptee), and returns an adapter from the *from_protocol* to the *to_protocol*.

The factory can be specified as either a callable, or a string in the form 'foo.bar.baz' which is turned into an import statement 'from foo.bar import baz' and imported when the trait is first accessed.

from_protocol = Property(Any)
Adapters created by the factory adapt *from* this protocol.

The protocol can be specified as a protocol (class/Interface), or a string in the form 'foo.bar.baz' which is turned into an import statement 'from foo.bar import baz' and imported when the trait is accessed.

to_protocol = Property(Any)
Adapters created by the factory adapt *to* this protocol.

The protocol can be specified as a protocol (class/Interface), or a string in the form 'foo.bar.baz' which is turned into an import statement 'from foo.bar import baz' and imported when the trait is accessed.

adapter Module

Base classes for adapters.

Adapters do not have to inherit from these classes, as long as their constructor takes the object to be adapted as the first and only *positional* argument.

class traits.adaptation.adapter.**PurePythonAdapter**(*adaptee*)
Bases: object

Base class for pure Python adapters.

class traits.adaptation.adapter.**Adapter**(*adaptee*, ***traits*)
Bases: traits.has_traits.HasTraits

Base class for adapters with traits.

traits.adaptation.adapter.**adapts**(*from_*, *to*, *extra=None*, *factory=None*, *cached=False*, *when=''*)
A class advisor for declaring adapters.

Parameters

- **from_** (*type or interface*) – What the adapter adapts *from*, or a list of such types or interfaces (the '_' suffix is used because 'from' is a Python keyword).
- **to** (*type or interface*) – What the adapter adapts *to*, or a list of such types or interfaces.
- **factory** (*callable*) – An (optional) factory for actually creating the adapters. This is any callable that takes a single argument which is the object to be adapted. The factory should return an adapter if it can perform the adaptation and **None** if it cannot.
- **cached** (*bool*) – Should the adapters be cached? If an adapter is cached, then the factory will produce at most one adapter per instance.
- **when** (*str*) – A Python expression that selects which instances of a particular type can be adapted by this factory. The expression is evaluated in a namespace that contains a single name *adaptee*, which is bound to the object to be adapted (e.g., 'adaptee.is_folder').

Note: The cached and when arguments are ignored if factory is specified.

cached_adapter_factory Module

An adapter factory that caches adapters per instance.

class traits.adaptation.cached_adapter_factory.**CachedAdapterFactory**
Bases: traits.has_traits.HasTraits

An adapter factory that caches adapters per instance.

We provide this class to provide the caching functionality of the old traits 'adapts' implementation. However, note that the cache will not be cleared unless you take care of cleaning the 'adaptee' trait once your adapter are deleted.

This class will be removed when the 'adapts' function is removed.

factory = adapter
A callable that actually creates the adapters!

The factory must ba callable that takes exactly one argument which is the object to be adapted (known as the adaptee), and returns an adapter from the *from_protocol* to the *to_protocol*.

The factory can be specified as either a callable, or a string in the form 'foo.bar.baz' which is turned into an import statement 'from foo.bar import baz' and imported when the trait is first accessed.

is_empty = Property(Bool)
True if the cache is empty, otherwise False.

This method is mostly here to help testing - the framework does not rely on it for any other purpose.

etsconfig Package

etsconfig Package

Supports sharing settings across projects or programs on the same system. Part of the EnthoughtBase project.

etsconfig Module

Enthought Tool Suite configuration information.

traits.etsconfig.etsconfig.**ETSConfig = <traits.etsconfig.etsconfig.ETSConfig object at 0x352acd0>**
Enthought Tool Suite configuration information.

This class should not use ANY other package in the tool suite so that it will always work no matter which other packages are present.

protocols Package

Note: The traits.protocols package is deprecated. Use the traits.adaptation package instead in new code.

Trivial Interfaces and Adaptation from PyProtocols.

This package used to be a subset of the files from Phillip J. Eby's PyProtocols package. The package has been substituted by traits.adaptation as of Traits 4.4.0.

Currently, the package contains deprecated aliases for backward compatibility, and will be removed in Traits 5.0 .

testing Package

testing Package

Scripts and assert tools related to running unit tests.

These scripts also allow running test suites in separate processes and aggregating the results.

doctest_tools Module

Tools for having doctest and unittest work together more nicely.

Eclipse's PyDev plugin will run your unittest files for you very nicely. The doctest_for_module function allows you to easily run the doctest for a module along side your standard unit tests within Eclipse.

traits.testing.doctest_tools.**doctest_for_module**(*module*)
Create a TestCase from a module's doctests that will be run by the standard unittest.main().

Example tests/test_foo.py:

```
import unittest

import foo
from traits.testing.api import doctest_for_module

class FooTestCase(unittest.TestCase):
    ...

class FooDocTest(doctest_for_module(foo)):
    pass

if __name__ == "__main__":
    # This will run and report both FooTestCase and the doctests in
    # module foo.
    unittest.main()
```

Alternatively, you can say:

```
FooDocTest = doctest_for_module(foo)
```

instead of:

```
class FooDocTest(doctest_for_module(foo)):
    pass
```

nose_tools Module

Non-standard functions for the 'nose' testing framework.

traits.testing.nose_tools.**skip**(*f*)
Stub replacement for marking a unit test to be skipped in the absence of 'nose'.

traits.testing.nose_tools.**deprecated**(*f*)
Stub replacement for marking a unit test deprecated in the absence of 'nose'.

traits.testing.nose_tools.**performance**(*f*)
Decorator to add an attribute to the test to mark it as a performance-measuring test.

`trait_assert_tools` Module

Trait assert mixin class to simplify test implementation for Trait Classes.

`traits.testing.unittest_tools.`**`reverse_assertion`**(**args*, ***kwds*)

class `traits.testing.unittest_tools.`**`UnittestTools`**
Bases: `object`

Mixin class to augment the unittest.TestCase class with useful trait related assert methods.

`assertTraitChanges`(*obj*, *trait*, *count=None*, *callableObj=None*, **args*, ***kwargs*)
Assert an object trait changes a given number of times.

Assert that the class trait changes exactly *count* times during execution of the provided function.

Method can also be used in a with statement to assert that the a class trait has changed during the execution of the code inside the with statement (similar to the assertRaises method). Please note that in that case the context manager returns itself and the user can introspect the information of:

•The last event fired by accessing the `event` attribute of the returned object.

•All the fired events by accessing the `events` attribute of the return object.

Example:

```
class MyClass(HasTraits):
    number = Float(2.0)

my_class = MyClass()

with self.assertTraitChangesExactly(my_class, 'number', count=1):
    my_class.number = 3.0
```

Parameters

- **obj** (*HasTraits*) – The HasTraits class instance whose class trait will change.
- **trait** (*str*) – The extended trait name of trait changes to listen to.
- **count** (*int or None, optional*) – The expected number of times the event should be fired. When None (default value) there is no check for the number of times the change event was fired.
- **callableObj** (*callable, optional*) – A callable object that will trigger the expected trait change. When None (default value) a trigger is expected to be called under the context manger returned by this method.
- ***args** – List of positional arguments for `callableObj`
- ****kwargs** – Dict of keyword value pairs to be passed to the `callableObj`

Returns **context** (*context manager or None*) – If `callableObj` is None, an assertion context manager is returned, inside of which a trait-change trigger can be invoked. Otherwise, the context is used internally with `callableObj` as the trigger, in which case None is returned.

Note:

•Checking if the provided `trait` corresponds to valid traits in the class is not implemented yet.

•Using the functional version of the assert method requires the `count` argument to be given even if it is None.

---

assertTraitDoesNotChange(*obj*, *trait*, *callableObj=None*, **args*, ***kwargs*)
Assert an object trait does not change.

Assert that the class trait does not change during execution of the provided function.

Parameters

- **obj** (*HasTraits*) – The HasTraits class instance whose class trait will change.
- **trait** (*str*) – The extended trait name of trait changes to listen to.
- **callableObj** (*callable, optional*) – A callable object that should not trigger a change in the passed trait. When None (default value) a trigger is expected to be called under the context manger returned by this method.
- ***args** – List of positional arguments for `callableObj`
- ****kwargs** – Dict of keyword value pairs to be passed to the `callableObj`

Returns **context** (*context manager or None*) – If `callableObj` is None, an assertion context manager is returned, inside of which a trait-change trigger can be invoked. Otherwise, the context is used internally with `callableObj` as the trigger, in which case None is returned.

assertMultiTraitChanges(*objects*, *traits_modified*, *traits_not_modified*)
Assert that traits on multiple objects do or do not change.

This combines some of the functionality of *assertTraitChanges* and *assertTraitDoesNotChange*.

Parameters

- **objects** (*list of HasTraits*) – The HasTraits class instances whose traits will change.
- **traits_modified** (*list of str*) – The extended trait names of trait expected to change.
- **traits_not_modified** (*list of str*) – The extended trait names of traits not expected to change.

assertTraitChangesAsync(**args*, ***kwds*)
Assert an object trait eventually changes.

Context manager used to assert that the given trait changes at least *count* times within the given timeout, as a result of execution of the body of the corresponding with block.

The trait changes are permitted to occur asynchronously.

Example usage:

```
with self.assertTraitChangesAsync(my_object, 'SomeEvent', count=4):
    <do stuff that should cause my_object.SomeEvent to be
    fired at least 4 times within the next 5 seconds>
```

Parameters

- **obj** (*HasTraits*) – The HasTraits class instance whose class trait will change.
- **trait** (*str*) – The extended trait name of trait changes to listen to.
- **count** (*int, optional*) – The expected number of times the event should be fired.
- **timeout** (*float or None, optional*) – The amount of time in seconds to wait for the specified number of changes. None can be used to indicate no timeout.

assertEventuallyTrue(*obj*, *trait*, *condition*, *timeout=5.0*)
Assert that the given condition is eventually true.

Parameters

- **obj** (*HasTraits*) – The HasTraits class instance who's traits will change.
- **trait** (*str*) – The extended trait name of trait changes to listen to.
- **condition** (*callable*) – A function that will be called when the specified trait changes. This should accept `obj` and should return a Boolean indicating whether the condition is satisfied or not.
- **timeout** (*float or None, optional*) – The amount of time in seconds to wait for the condition to become true. None can be used to indicate no timeout.

`util` Package

`util` Package

Utility functions, part of the Traits project.

Copyright 2003-2013 Enthought, Inc.

`camel_case` Module

Defines utility functions for operating on camel case names.

class `traits.util.camel_case.`**`CamelCaseToPython`**
Simple functor class to convert names from camel case to idiomatic Python variable names.

For example::

```
>>> camel2python = CamelCaseToPython
>>> camel2python('XMLActor2DToSGML')
'xml_actor2d_to_sgml'
```

`traits.util.camel_case.`**`camel_case_to_words`**`(s)`
Convert a camel case string into words separated by spaces.

For example::

```
>>> camel_case_to_words('CamelCase')
'Camel Case'
```

`clean_strings` Module

Provides functions that mange strings to avoid characters that would be problematic in certain situations.

`traits.util.clean_strings.`**`clean_filename`**`(name)`
Munge a string to avoid characters that might be problematic as a filename in some filesystems.

`traits.util.clean_strings.`**`clean_timestamp`**`(dt=None, microseconds=False)`
Return a timestamp that has been cleansed of characters that might cause problems in filenames, namely colons. If no datetime object is provided, then uses the current time.

Description

The timestamp is in ISO-8601 format with the following exceptions:

•Colons ':' are replaced by underscores '_'.

•Microseconds are not displayed if the 'microseconds' parameter is False.

Parameters

- **dt** (*None or datetime.datetime*) – If None, then the current time is used.
- **microseconds** (*bool*) – Display microseconds or not.

Returns A string timestamp.

`traits.util.clean_strings.`**`python_name`**(*name*)
Attempt to make a valid Python identifier out of a name.

`deprecated` Module

A decorator for marking methods/functions as deprecated.

`traits.util.deprecated.`**`deprecated`**(*message*)
A factory for decorators for marking methods/functions as deprecated.

`home_directory` Module

`traits.util.home_directory.`**`get_home_directory`**()
Determine the user's home directory.

`resource` Module

Utility functions for managing and finding resources (ie. images/files etc).

get_path : Returns the absolute path of a class or instance

create_unique_name [Creates a name with a given prefix that is not in a] given list of existing names. The separator between the prefix and the rest of the name can also be specified (default is a '_')

find_resource: Given a setuptools project specification string ('MyProject>=2.1') and a partial path leading from the projects base directory to the desired resource, will return either an opened file object or, if specified, a full path to the resource.

`traits.util.resource.`**`get_path`**(*path*)
Returns an absolute path for the specified path.

'path' can be a string, class or instance.

`traits.util.resource.`**`create_unique_name`**(*prefix*, *names*, *separator='_'*)
Creates a name starting with 'prefix' that is not in 'names'.

`traits.util.resource.`**`find_resource`**(*project*, *resource_path*, *alt_path=None*, *return_path=False*)
Returns a file object or file path pointing to the desired resource.

Parameters

- **project** (*str*) – The name of the project to look for the resource in. Can be the name or a requirement string. Ex: 'MyProject', 'MyProject>1.0', 'MyProject==1.1'
- **resource_path** (*str*) – The path to the file from inside the package. If the file desired is MyProject/data/image.jpg, resource_path would be 'data/image.jpg'.

- **alt_path** (*str*) – The path to the resource relative to the location of the application's top-level script (the one with __main__). If this function is called in code/scripts/myscript.py and the resource is code/data/image.jpg, the alt_path would be '../data/image.jpg'. This path is only used if the resource cannot be found using setuptools.
- **return_path** (*bool*) – Determines whether the function should return a file object or a full path to the resource.

Returns **file** (*file object or file path*) – A file object containing the resource. If return_path is True, 'file' will be the full path to the resource. If the file is not found or cannot be opened, None is returned.

Description

This function will find a desired resource file and return an opened file object. The main method of finding the resource uses the pkg_resources resource_stream method, which searches your working set for the installed project specified and appends the resource_path given to the project path, leading it to the file. If setuptools is not installed or it cannot find/open the resource, find_resource will use the sys.path[0] to find the resource if alt_path is defined.

`traits.util.resource.`**`store_resource`**(*project*, *resource_path*, *filename*)
Store the content of a resource, given by the name of the project and the path (relative to the root of the project), into a newly created file.

The first two arguments (project and resource_path) are the same as for the function find_resource in this module. The third argument (filename) is the name of the file which will be created, or overwritten if it already exists. The return value in always None.

`import_symbol` Module

A function to import symbols.

`traits.util.import_symbol.`**`import_symbol`**(*symbol_path*)
Import the symbol defined by the specified symbol path.

Examples

import_symbol('tarfile:TarFile') -> TarFile import_symbol('tarfile:TarFile.open') -> TarFile.open

To allow compatibility with old-school traits symbol names we also allow all-dotted paths, but in this case you can only import top-level names from the module.

import_symbol('tarfile.TarFile') -> TarFile

`toposort` Module

A simple topological sort on a dictionary graph.

exception `traits.util.toposort.`**`CyclicGraph`**
Bases: `exceptions.Exception`

Exception for cyclic graphs.

`traits.util.toposort.`**`topological_sort`**(*graph*)
Returns the nodes in the graph in topological order.

`trait_documenter` Module

A Trait Documenter (Subclassed from the autodoc ClassLevelDocumenter)

copyright Copyright 2012 by Enthought, Inc

class `traits.util.trait_documenter.`**`TraitDocumenter`**(*directive*, *name*, *indent=u''*)
Bases: `sphinx.ext.autodoc.ClassLevelDocumenter`

Specialized Documenter subclass for trait attributes.

The class defines a new documenter that recovers the trait definition signature of module level and class level traits.

To use the documenter, append the module path in the extension attribute of the *conf.py*.

> **Warning:** Using the TraitDocumenter in conjunction with TraitsDoc is not advised.

`objtype` = 'traitattribute'

`directivetype` = 'attribute'

`member_order` = 60

`priority` = 12

classmethod `can_document_member`(*member*, *membername*, *isattr*, *parent*)
Check that the documented member is a trait instance.

`document_members`(*all_members=False*)
Trait attributes have no members

`add_content`(*more_content*, *no_docstring=False*)
Never try to get a docstring from the trait.

`import_object`()
Get the Trait object.

> **Note:** Code adapted from autodoc.Documenter.import_object.

`add_directive_header`(*sig*)
Add the directive header 'attribute' with the annotation option set to the trait definition.

`traits.util.trait_documenter.`**`setup`**(*app*)
Add the TraitDocumenter in the current sphinx autodoc instance.

2.2 Indices and tables

- *genindex*
- *search*
- *search*

t