# traity Documentation

***Release 0.0.1***

**Sean Ross-Ross**

August 16, 2016

Contents

Contents:

CHAPTER 1

# Traity's API

This is traity an alternative to enthought's traits package

Contents:

## 1.1 Events

### 1.1.1 Event Objects

**class** traity.events.**Event**(*snitch*, *target*, *dispatcher=None*, ***kwargs*)
An event should never need to be created directly. Use *events(obj).etrigger* to start a chain of events.

**Parameters**

- **event_type** – type of event
- **target** –

**dispatch**(*listener*)
call a listener object

**target**
The target this event will trigger action for.

### 1.1.2 Snitch Objects

**class** traity.events.**Snitch**(*instance*)
Snitch object handles events. and propegates them to the object's upstream nodes.

**etrigger**(*target*, ***metadata*)
trigger an event.

**Parameters** **target** – only listeners listening to this target will fire

**listen**(*target*, *listener*, *weak=None*)
Add a listener to listen to events with target 'target'.

**queue**(**args*, ***kwds*)
queue listened to events for this object.

**quiet**(**args*, ***kwds*)
Context manager to stop events being propegated.

**trigger**(*event*)

**Parameters** **event** – *Event* object to trigger

**unique**(**args*, ***kwds*)
remove duplicate events.

**unlisten**(*target*, *listener=None*)
Remove a listener.

### 1.1.3 Functions

traity.events.**connect**(*upstream*, *downstream*, *target*, *init=True*)
Connect two objects together so that events will bubble upstream.

traity.events.**disconnect**(*upstream*, *downstream*, *target*, *init=True*)
Discnnect two objects.

traity.events.**connected**(*upstream*, *downstream*)
Test if to objects are connected.

**See also:**

*connect()*

traity.events.**events**(*instance*, *init=False*)
Get the events class associated with this object. If init is true. call init_events if events are uninitialized, otherwize, raise an exception.

traity.events.**init_events**(**instances*)
Initialize the events for an object

traity.events.**init_events**(**instances*)
Initialize the events for an object

### 1.1.4 Global liteners

traity.events.**add_global_listener**(*listener*, *target=None*)
Add a listener to events on all objects.

traity.events.**remove_global_listener**(*listener*, *target=None*)
remove a add_global_listener

traity.events.**global_listener**(**args*, ***kwds*)

**Parameters**

- **listener** – callable object, signature must be *listener(event)*
- **target** – target tuple

Context manager for global listeners

```
def print_event(event):
    print event

with global_listener(print_event):
    ...
```

ok

### 1.1.5 Global dispatchers

traity.events.**add_global_dispatcher**(*dispatcher*)
Add a dispatcher to the stack.

**Parameters** **dispatcher** – callable object. signature must be *dispatcher(event, listener)*

traity.events.**pop_global_dispatcher**()
remove the last global disopatcher from the stack

traity.events.**remove_global_dispatcher**(*dispatcher*)
remove the last *dispatcher* from the stack

**Parameters** **dispatcher** – specific dispatcher to remove

traity.events.**global_dispatcher**(**args*, ***kwds*)

**Parameters** **dispatcher** – callable object. signature must be *dispatcher(event, listener)*

Add a global dispatcher in a context:

```
with global_dispatcher(myfunc):
    #Trigger events
```

ok

traity.events.**quiet**(**args*, ***kwds*)
Do not dispatch any events

traity.events.**queue**(**args*, ***kwds*)
Put all events into a queue. example:

```
with queue() as todo:
    ...

print "I have cought %i events" % len(todo)
```

this

traity.events.**unique**(**args*, ***kwds*)
Only process unique events eg:

```
num_calls = 0
with global_listener(inc_num_calls):
    with unique():
        triggers_event()
        triggers_event()

assert num_calls == 1
```

and how

## 1.2 Static Properties

Static properties allow users to define attributes to a class ad class creation.

Unlike python's native *property* descriptor. a vproperty's getter setter and deleter are already defined.

```
class Foo(object):
    x = vproperty()
```

Normal python descriptor symantics are already defined:

```
class Foo(object):

    @vproperty
    def x(self):
        'Getter for x'

    @x.setter
    def x(self):
        'Setter for x'
```

### 1.2.1 Validation

Type and instance arguments may be given to perform arbitrary valitation before the setter is called.

```
class Foo(object):
    x = vproperty(type=int)

class Bar(object):
    x = vproperty(isntance=int)
```

**The type argument is only required to be a function that acccepts a single input and returns a value::**

**class Foo(object):** x = vproperty(type=lambda item: item.lower())

or

```
class Foo(object):
    x = vproperty()

    @x.type
    def check_x_type(self, value):
        pass
```

The *instance* argument performs a strict *isinstance* check.

```
class Foo(object):
    x = vproperty(type=int)

class Bar(object):
    x = vproperty(isntance=int)

foo = Foo()
foo.x = '1' # x is set to one

bar = Bar()
bar.x = '1' # error x is not an int.
```

### 1.2.2 Delegation

Delegation may be one of two forms, strict: and can *only* be done when the instance argument is used or loose, where no error checking is done:

```
class Foo(object):
    x = vproperty(type=int)

class Bar(object):
```

```
    foo = vproperty(isntance=Foo)

    # Strong Delegation of bar.x to bar.foo.x
    x = foo.x

    # Weak delegation of bar.y to bar.foo.x
    y = delegate('foo', 'x')
    # Also can do
    y = delegate(foo, Foo.x)
```

### 1.2.3 vproperty objects

**class** traity.statics.**vproperty**(*fget=None*, *fset=None*, *fdel=None*, *type=None*, *ftype=None*, *instance=None*, *fdefault=None*)

**Parameters**

- **fget** – method to get attribute
- **fset** – method to set attribute
- **fdel** – method to delete attribute
- **ftype** – method to convert type of attribute on assignment
- **instance** – if given, all assignments to this property must pass an isinstance check.
- **fdefault** – method to get attribute if undefined
- **type** – shorthand for ftype, type may be a basic python type not a method. int, float etc.

Define a static attribute to a class. Behaves almost exactly the same as Python properties:

```
    class Parrot(object):
        def __init__(self):
            self._voltage = 100000

        @vproperty
        def voltage(self):
            """Get the current voltage."""
            return self._voltage
```

vproperty defines default fget, fset and fdel:

```
    class Parrot(object):
        def __init__(self):
            self.voltage = 100000

        voltage = vproperty()
```

vproperty adds ftype, instance and fdefault:

```
    class Parrot(object):

        voltage = vproperty(type=int)

        def __init__(self):
            self.voltage = 100000
            #OK
            self.voltage = '100000'
```

```
            #ValueError 'abc' is not convertable to int.
            self.voltage = 'abc'
```

or:

```
class Parrot(object):

    voltage = vproperty()

    @voltage.type
    def dynamic_range(self, value):
        return clamp(value, 0, self.max_volts)
```

defaults:

```
class Parrot(object):

    voltage = vproperty()

    @voltage.default
    def voltage_default(self):
        return 10000
```

**delegates_to**(*attr*)
Delegates to an inner attribute. instance= argument must be set in constructor.

Used by __getattr__ for convinience. eg *x.delegates_to('y')* can be written as *x.y*

**getter**(*value*)

**type**(*method*)
Set the type validation as a method of the containing class:

```
        @x.type
        def check(self, value):
            ...
```

.

### 1.2.4 delegate objects

**class** traity.statics.**delegate**(*outer*, *inner*, **chain*)
Delegate attribute lookup to an inner object

**delegates_to**(*attr*)
Delegate to an inner attribute. can also use getattr method/

**flat**
return the list of delegates as a flat list

## 1.3 Traits

### 1.3.1 trait class

**class** traity.traits.**trait**(*fget=None*, *fset=None*, *fdel=None*, *type=None*, *ftype=None*, *instance=None*, *fdefault=None*)

**Parameters**

- **fget** – Is a function to be used for getting an attribute value.
- **fset** – Is a function for setting
- **fdel** – a function for del'ing, an attribute.
- **ftype** – A type convert the argument to on *set*. *type* must be a function that accepts an argument and returns a single result. Like the builtin *int*
- **instance** – The argument must be of this instance.
- **fdefault** – Is a function to be called when the property is not set

### 1.3.2 delegate_trait class

**class** traity.traits.**delegate_trait**(*outer*, *inner*, **chain*)

### 1.3.3 Functions

traity.traits.**on_trait_change**(*traits*)
Statically Register a listener to a class:

```
@on_trait_change('attr')
def attr_changed(self, event):
    print 'hello!'
```

.

traity.traits.**on_change**(*instance*, *traits*, *function*, *weak=None*)
Register a listener to an object.

**Parameters**

- **traits** – either a *string*, *traity.events.listenable*, or a tuple of strings and listenables.
- **function** – function to call when event with target *target* is triggered.
- **instance** – The object to listen to events for.

## 1.4 Tools

### 1.4.1 Initializable Properties

Properties that have knowlede of the container class.

traity.tools.initializable_property.**init_properties**(*cls*)
Class decorator calles __init_property__ on all initializable objects defined in a class

**class** traity.tools.initializable_property.**persistent_property**
A persistent property.

**store_key**
This is set the the attribute name. for example:

x = persistent_property()

repr(x.store_key) '_x_'

### 1.4.2 Instance properties

Example

```
class MyObj(iobject): pass

def get_x(any):
    return 1

obj = MyObj()

set_iproperty(obj, 'x', get_x)

print obj.x
1
```

traity.tools.instance_properties.**set_iproperty**(*instance*, *attr*, *prop*)
set an instance property of an object

traity.tools.instance_properties.**get_iproperty**(*instance*, *attr*)
get an instance property of an object

**class** traity.tools.instance_properties.**iobject**
class must be a subclass of iobject to support instance properties

## 1.5 Compat

CHAPTER 2

# Comparison with traits

My ideal traits's like library.

traity.events + traity.statics == traity.traits.

### Objectives

#### Uphold the main principles and philosophy of traits

**Initialization:** A may optionally have a default value.

**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.

**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:** TBD - enaml?

#### Reduce codebase

- Traits is a huge project with **> 92,000** lines and **< %60** covered by unit-tests.
- Traity is a tiny project **< 2,000** lines and **> %95** covered by unit-tests.

Traity is designed to remain tiny kernel of stable code which add-ons and extension may be based on top of.

#### Extend the learning curve

It is my feeling that while traits may be a good learning tool for new scientists it's monolithic structure makes it very hard for an advanced programmer to do specialized things.

Traity is designed from the ground up to be tiny and transparent. Separating event notification from static typing from other more specialized helpers.

#### Python 3

Just mentioning it here. Traity is Python 3 compliant.

### Highlights

‘‘‘python

# Optional decorator does two things # 1. Makes all vproperty's including *trait* picklable. # 2. Allows static listeners to be defined on a class. @init_properties # Notice that the foo class does not have to inherit from any hastraits base class. class Foo(object):

#attr1 may emit events like changes attr1 = listenable()

#vproperty is a type checking propery. attr2 = vproperty(type=int)

#traits is a subclass of listenable and vproperty attr3 = trait(type=float)

**def __init__(self):** #Optional: Alows listenable and trait properties to emit events. init_events(self)

@attr3.changed def notify(self, event):

pass

‘‘‘

#### Building back up to traits

As I mentioned before, this is meant to be a tiny implementation and extensions can be built on top if traity.

Here is an example of how to re-create a simple HasTraits class from traity.

‘‘‘python class Int(trait):

**def __init__(default=0):** trait.__init__(self, type=int, fdefault=lambda self:default)

# Special method to allow trait to be called with or without parens ‘()’ # Method __init_property__ is invoked when init_properties is called @classmethod def __init_property__(cls, owner, key):

int_trait = cls() setattr(owner, key, int_trait) trait.__init_property__(int_trait, owner, key)

**class HasTraitsMeta(type):**

**def __new__(mcs, name, bases, dict):** cls = type.__new__(mcs, name, bases, dict) init_properties(cls) return cls

**class HasTraits(object):** __metaclass__ = HasTraitsMeta

**def __init__(**kwargs):** self.__dict__.update(kwargs) init_events(self)

**class MyObject(HasTraits):** i = Int

obj = MyObject(i=1) ‘‘‘

CHAPTER 3

# Indices and tables

- genindex
- modindex
- search

## t

## V