traitlets Documentation

Release 4.3.2

The IPython Development Team

Feb 23, 2017

Contents

Release 4.3.2

Date Feb 23, 2017

Traitlets is a framework that lets Python classes have attributes with type checking, dynamically calculated default values, and 'on change' callbacks.

The package also includes a mechanism to use traitlets for configuration, loading values from files or from command line arguments. This is a distinct layer on top of traitlets, so you can use traitlets in your code without using the configuration machinery.

CHAPTER 1

Using Traitlets

In short, traitlets let the user define classes that have

1. Attributes (traits) with type checking and dynamically computed default values
2. Traits emit change events when attributes are modified
3. Traitlets perform some validation and allow coercion of new trait values on assignment. They also allow the user to define custom validation logic for attributes based on the value of other attributes.

Default values, and checking type and value

At its most basic, traitlets provides type checking, and dynamic default value generation of attributes on :class:`traitlets.HasTraits` subclasses:

```
import getpass

class Identity(HasTraits):
    username = Unicode()

    @default('username')
    def _default_username(self):
        return getpass.getuser()
```

```
class Foo(HasTraits):
    bar = Int()

foo = Foo(bar='3')  # raises a TraitError
```

```
TraitError: The 'bar' trait of a Foo instance must be an int,
but a value of '3' <class 'str'> was specified
```

observe

Traitlets implement the observer pattern

```
class Foo(HasTraits):
    bar = Int()
    baz = Unicode()

foo = Foo()

def func(change):
    print(change['old'])
    print(change['new'])    # as of traitlets 4.3, one should be able to
                            # write print(change.new) instead

foo.observe(func, names=['bar'])
foo.bar = 1  # prints '0\n 1'
foo.baz = 'abc'  # prints nothing
```

When observers are methods of the class, a decorator syntax can be used.

```
class Foo(HasTraits):
    bar = Int()
    baz = Unicode()

    @observe('bar')
    def _observe_bar(self, change):
        print(change['old'])
        print(change['new'])
```

Validation

Custom validation logic on trait classes

```
from traitlets import HasTraits, TraitError, Int, Bool, validate

class Parity(HasTraits):
    value = Int()
    parity = Int()

    @validate('value')
    def _valid_value(self, proposal):
        if proposal['value'] % 2 != self.parity:
            raise TraitError('value and parity should be consistent')
        return proposal['value']

    @validate('parity')
    def _valid_parity(self, proposal):
        parity = proposal['value']
        if parity not in [0, 1]:
            raise TraitError('parity should be 0 or 1')
        if self.value % 2 != parity:
            raise TraitError('value and parity should be consistent')
        return proposal['value']
```

```
parity_check = Parity(value=2)

# Changing required parity and value together while holding cross validation
with parity_check.hold_trait_notifications():
    parity_check.value = 1
    parity_check.parity = 1
```

In the case where the a validation error occurs when `hold_trait_notifications` context manager is released, changes are rolled back to the initial state.

- Finally, trait type can have other events than trait changes. This capability was added so as to enable notifications on change of values in container classes. The items available in the dictionary passed to the observer registered with `observe` depends on the event type.

CHAPTER 2

Trait Types

class traitlets.**TraitType**
The base class for all trait types.

__init__(*default_value=traitlets.Undefined*, *allow_none=False*, *read_only=None*, *help=None*, *config=None*, ***kwargs*)
Declare a traitlet.

If *allow_none* is True, None is a valid value in addition to any values that are normally valid. The default is up to the subclass. For most trait types, the default value for `allow_none` is False.

Extra metadata can be associated with the traitlet using the .tag() convenience method or by using the traitlet instance's .metadata dictionary.

Numbers

class traitlets.**Integer**
An integer trait. On Python 2, this automatically uses the `int` or `long` types as necessary.

class traitlets.**Int**

class traitlets.**Long**
On Python 2, these are traitlets for values where the `int` and `long` types are not interchangeable. On Python 3, they are both aliases for *Integer*.

In almost all situations, you should use *Integer* instead of these.

class traitlets.**Float**(*default_value=traitlets.Undefined*, *allow_none=False*, ***kwargs*)
A float trait.

class traitlets.**Complex**(*default_value=traitlets.Undefined*, *allow_none=False*, *read_only=None*, *help=None*, *config=None*, ***kwargs*)
A trait for complex numbers.

class traitlets.**CInt**
class traitlets.**CLong**

class traitlets.**CFloat**
class traitlets.**CComplex**
Casting variants of the above. When a value is assigned to the attribute, these will attempt to convert it by calling e.g. `value = int(value)`.

Strings

class traitlets.**Unicode**(*default_value=traitlets.Undefined*, *allow_none=False*, *read_only=None*, *help=None*, *config=None*, ***kwargs*)
A trait for unicode strings.

class traitlets.**Bytes**(*default_value=traitlets.Undefined*, *allow_none=False*, *read_only=None*, *help=None*, *config=None*, ***kwargs*)
A trait for byte strings.

class traitlets.**CUnicode**
class traitlets.**CBytes**
Casting variants. When a value is assigned to the attribute, these will attempt to convert it to their type. They will not automatically encode/decode between unicode and bytes, however.

class traitlets.**ObjectName**(*default_value=traitlets.Undefined*, *allow_none=False*, *read_only=None*, *help=None*, *config=None*, ***kwargs*)
A string holding a valid object name in this version of Python.

This does not check that the name exists in any scope.

class traitlets.**DottedObjectName**(*default_value=traitlets.Undefined*, *allow_none=False*, *read_only=None*, *help=None*, *config=None*, ***kwargs*)
A string holding a valid dotted object name in Python, such as A.b3._c

Containers

class traitlets.**List**(*trait=None*, *default_value=None*, *minlen=0*, *maxlen=9223372036854775807*, ***kwargs*)
An instance of a Python list.

__init__(*trait=None*, *default_value=None*, *minlen=0*, *maxlen=9223372036854775807*, ***kwargs*)
Create a List trait type from a list, set, or tuple.

The default value is created by doing `list(default_value)`, which creates a copy of the `default_value`.

`trait` can be specified, which restricts the type of elements in the container to that TraitType.

If only one arg is given and it is not a Trait, it is taken as `default_value`:

```
c = List([1, 2, 3])
```

Parameters

- **trait** (*TraitType [optional]*) – the type for restricting the contents of the Container. If unspecified, types are not checked.
- **default_value** (*SequenceType [optional]*) – The default value for the Trait. Must be list/tuple/set, and will be cast to the container type.
- **minlen** (*Int [default 0]*) – The minimum length of the input list

- **maxlen** (*Int [default sys.maxsize]*) – The maximum length of the input list

class traitlets.**Set**(*trait=None*, *default_value=None*, *minlen=0*, *maxlen=9223372036854775807*, ***kwargs*)

An instance of a Python set.

__init__(*trait=None*, *default_value=None*, *minlen=0*, *maxlen=9223372036854775807*, ***kwargs*)

Create a Set trait type from a list, set, or tuple.

The default value is created by doing `set(default_value)`, which creates a copy of the `default_value`.

`trait` can be specified, which restricts the type of elements in the container to that TraitType.

If only one arg is given and it is not a Trait, it is taken as `default_value`:

```
c = Set({1, 2, 3})
```

Parameters

- **trait** (*TraitType [optional]*) – the type for restricting the contents of the Container. If unspecified, types are not checked.
- **default_value** (*SequenceType [optional]*) – The default value for the Trait. Must be list/tuple/set, and will be cast to the container type.
- **minlen** (*Int [default 0]*) – The minimum length of the input list
- **maxlen** (*Int [default sys.maxsize]*) – The maximum length of the input list

class traitlets.**Tuple**(**traits*, ***kwargs*)

An instance of a Python tuple.

__init__(**traits*, ***kwargs*)

Create a tuple from a list, set, or tuple.

Create a fixed-type tuple with Traits:

```
t = Tuple(Int(), Str(), CStr())
```

would be length 3, with Int,Str,CStr for each element.

If only one arg is given and it is not a Trait, it is taken as default_value:

```
t = Tuple((1, 2, 3))
```

Otherwise, `default_value` *must* be specified by keyword.

Parameters

- ***traits** (*TraitTypes [optional]*) – the types for restricting the contents of the Tuple. If unspecified, types are not checked. If specified, then each positional argument corresponds to an element of the tuple. Tuples defined with traits are of fixed length.
- **default_value** (*SequenceType [optional]*) – The default value for the Tuple. Must be list/tuple/set, and will be cast to a tuple. If `traits` are specified, `default_value` must conform to the shape and type they specify.

class traitlets.**Dict**(*trait=None*, *traits=None*, *default_value=traitlets.Undefined*, ***kwargs*)

An instance of a Python dict.

__init__(*trait=None*, *traits=None*, *default_value=traitlets.Undefined*, ***kwargs*)

Create a dict trait type from a Python dict.

The default value is created by doing `dict(default_value)`, which creates a copy of the `default_value`.

Parameters

- **trait** (*TraitType [optional]*) – The specified trait type to check and use to restrict contents of the Container. If unspecified, trait types are not checked.
- **traits** (*Dictionary of trait types [optional]*) – A Python dictionary containing the types that are valid for restricting the content of the Dict Container for certain keys.
- **default_value** (*SequenceType [optional]*) – The default value for the Dict. Must be dict, tuple, or None, and will be cast to a dict if not None. If *trait* is specified, the *default_value* must conform to the constraints it specifies.

Classes and instances

class traitlets.**Instance**(*klass=None*, *args=None*, *kw=None*, ***kwargs*)
A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

Subclasses can declare default classes by overriding the klass attribute

__init__(*klass=None*, *args=None*, *kw=None*, ***kwargs*)
Construct an Instance trait.

This trait allows values that are instances of a particular class or its subclasses. Our implementation is quite different from that of enthough.traits as we don't allow instances to be used for klass and we handle the `args` and `kw` arguments differently.

Parameters

- **klass** (*class, str*) – The class that forms the basis for the trait. Class names can also be specified as strings, like 'foo.bar.Bar'.
- **args** (*tuple*) – Positional arguments for generating the default value.
- **kw** (*dict*) – Keyword arguments for generating the default value.
- **allow_none** (*bool [default False]*) – Indicates whether None is allowed as a value.

Notes

If both `args` and `kw` are None, then the default value is None. If `args` is a tuple and `kw` is a dict, then the default is created as `klass(*args, **kw)`. If exactly one of `args` or `kw` is None, the None is replaced by `()` or `{}`, respectively.

class traitlets.**Type**(*default_value=traitlets.Undefined*, *klass=None*, ***kwargs*)
A trait whose value must be a subclass of a specified class.

__init__(*default_value=traitlets.Undefined*, *klass=None*, ***kwargs*)
Construct a Type trait

A Type trait specifies that its values must be subclasses of a particular class.

If only `default_value` is given, it is used for the `klass` as well. If neither are given, both default to `object`.

Parameters

- **default_value** (*class, str or None*) – The default value must be a subclass of klass. If an str, the str must be a fully specified class name, like 'foo.bar.Bah'. The string is resolved into real class, when the parent *HasTraits* class is instantiated.
- **klass** (*class, str [default object]*) – Values of this trait must be a subclass of klass. The klass may be specified in a string like: 'foo.bar.MyClass'. The string is resolved into real class, when the parent *HasTraits* class is instantiated.
- **allow_none** (*bool [default False]*) – Indicates whether None is allowed as an assignable value.

class traitlets.**This**(***kwargs*)
A trait for instances of the class containing this trait.

Because how how and when class bodies are executed, the `This` trait can only have a default value of None. This, and because we always validate default values, `allow_none` is *always* true.

class traitlets.**ForwardDeclaredInstance**(*klass=None*, *args=None*, *kw=None*, ***kwargs*)
Forward-declared version of Instance.

class traitlets.**ForwardDeclaredType**(*default_value=traitlets.Undefined*, *klass=None*, ***kwargs*)
Forward-declared version of Type.

Miscellaneous

class traitlets.**Bool**(*default_value=traitlets.Undefined*, *allow_none=False*, *read_only=None*, *help=None*, *config=None*, ***kwargs*)
A boolean (True, False) trait.

class traitlets.**CBool**
Casting variant. When a value is assigned to the attribute, this will attempt to convert it by calling `value = bool(value)`.

class traitlets.**Enum**(*values*, *default_value=traitlets.Undefined*, ***kwargs*)
An enum whose value must be in a given sequence.

class traitlets.**CaselessStrEnum**(*values*, *default_value=traitlets.Undefined*, ***kwargs*)
An enum of strings where the case should be ignored.

class traitlets.**UseEnum**(*enum_class*, *default_value=None*, ***kwargs*)
Use a Enum class as model for the data type description. Note that if no default-value is provided, the first enum-value is used as default-value.

```
# -- SINCE: Python 3.4 (or install backport: pip install enum34)
import enum
from traitlets import HasTraits, UseEnum

class Color(enum.Enum):
    red = 1         # -- IMPLICIT: default_value
    blue = 2
    green = 3

class MyEntity(HasTraits):
    color = UseEnum(Color, default_value=Color.blue)

entity = MyEntity(color=Color.red)
entity.color = Color.green    # USE: Enum-value (preferred)
```

```
entity.color = "green"         # USE: name (as string)
entity.color = "Color.green"   # USE: scoped-name (as string)
entity.color = 3               # USE: number (as int)
assert entity.color is Color.green
```

class traitlets.**TCPAddress**(*default_value=traitlets.Undefined*, *allow_none=False*, *read_only=None*, *help=None*, *config=None*, ***kwargs*)

A trait for an (ip, port) tuple.

This allows for both IPv4 IP addresses as well as hostnames.

class traitlets.**CRegExp**(*default_value=traitlets.Undefined*, *allow_none=False*, *read_only=None*, *help=None*, *config=None*, ***kwargs*)

A casting compiled regular expression trait.

Accepts both strings and compiled regular expressions. The resulting attribute will be a compiled regular expression.

class traitlets.**Union**(*trait_types*, ***kwargs*)

A trait type representing a Union type.

__init__(*trait_types*, ***kwargs*)

Construct a Union trait.

This trait allows values that are allowed by at least one of the specified trait types. A Union traitlet cannot have metadata on its own, besides the metadata of the listed types.

Parameters **trait_types** (*sequence*) – The list of trait types of length at least 1.

Notes

Union([Float(), Bool(), Int()]) attempts to validate the provided values with the validation function of Float, then Bool, and finally Int.

class traitlets.**Any**(*default_value=traitlets.Undefined*, *allow_none=False*, *read_only=None*, *help=None*, *config=None*, ***kwargs*)

A trait which allows any value.

CHAPTER 3

Defining new trait types

To define a new trait type, subclass from *TraitType*. You can define the following things:

class traitlets.**MyTrait**

info_text
A short string describing what this trait should hold.

default_value
A default value, if one makes sense for this trait type. If there is no obvious default, don't provide this.

validate(*obj*, *value*)
Check whether a given value is valid. If it is, it should return the value (coerced to the desired type, if necessary). If not, it should raise TraitError. TraitType.error() is a convenient way to raise an descriptive error saying that the given value is not of the required type.

obj is the object to which the trait belongs.

For instance, here's the definition of the *TCPAddress* trait:

```
class TCPAddress(TraitType):
    """A trait for an (ip, port) tuple.

    This allows for both IPv4 IP addresses as well as hostnames.
    """

    default_value = ('127.0.0.1', 0)
    info_text = 'an (ip, port) tuple'

    def validate(self, obj, value):
        if isinstance(value, tuple):
            if len(value) == 2:
                if isinstance(value[0], six.string_types) and isinstance(value[1],␣
↪int):
                    port = value[1]
                    if port >= 0 and port <= 65535:
```

```
                    return value
        self.error(obj, value)
```

CHAPTER 4

Traitlets API reference

Any class with trait attributes must inherit from *HasTraits*.

class traitlets.**HasTraits**(**args*, ***kwargs*)

has_trait(*name*)
Returns True if the object has a trait with the specified name.

trait_names(***metadata*)
Get a list of all the names of this class' traits.

classmethod class_trait_names(***metadata*)
Get a list of all the names of this class' traits.

This method is just like the *trait_names()* method, but is unbound.

traits(***metadata*)
Get a dict of all the traits of this class. The dictionary is keyed on the name and the values are the TraitType objects.

The TraitTypes returned don't know anything about the values that the various HasTrait's instances are holding.

The metadata kwargs allow functions to be passed in which filter traits based on metadata values. The functions should take a single value as an argument and return a boolean. If any function returns False, then the trait is not included in the output. If a metadata key doesn't exist, None will be passed to the function.

classmethod class_traits(***metadata*)
Get a dict of all the traits of this class. The dictionary is keyed on the name and the values are the TraitType objects.

This method is just like the *traits()* method, but is unbound.

The TraitTypes returned don't know anything about the values that the various HasTrait's instances are holding.

The metadata kwargs allow functions to be passed in which filter traits based on metadata values. The functions should take a single value as an argument and return a boolean. If any function returns False, then the trait is not included in the output. If a metadata key doesn't exist, None will be passed to the function.

trait_metadata(*traitname*, *key*, *default=None*)
Get metadata values for trait by key.

add_traits(***traits*)
Dynamically add trait attributes to the HasTraits instance.

You then declare the trait attributes on the class like this:

```
from traitlets import HasTraits, Int, Unicode

class Requester(HasTraits):
    url = Unicode()
    timeout = Int(30)   # 30 will be the default value
```

For the available trait types and the arguments you can give them, see *Trait Types*.

Dynamic default values

traitlets.**default**(*name*)
A decorator which assigns a dynamic default for a Trait on a HasTraits object.

Parameters **name** – The str name of the Trait on the object whose default should be generated.

Notes

Unlike observers and validators which are properties of the HasTraits instance, default value generators are class-level properties.

Besides, default generators are only invoked if they are registered in subclasses of *this_type*.

```
class A(HasTraits):
    bar = Int()

    @default('bar')
    def get_bar_default(self):
        return 11

class B(A):
    bar = Float()  # This trait ignores the default generator defined in
                   # the base class A


class C(B):

    @default('bar')
    def some_other_default(self):  # This default generator should not be
        return 3.0                 # ignored since it is defined in a
                                   # class derived from B.a.this_class.
```

To calculate a default value dynamically, decorate a method of your class with *@default({traitname})*. This method will be called on the instance, and should return the default value. For example:

```
import getpass

class Identity(HasTraits):
    username = Unicode()

    @default('username')
    def _username_default(self):
        return getpass.getuser()
```

Callbacks when trait attributes change

traitlets.**observe**(**names*, ***kwargs*)

A decorator which can be used to observe Traits on a class.

The handler passed to the decorator will be called with one `change` dict argument. The change dictionary at least holds a 'type' key and a 'name' key, corresponding respectively to the type of notification and the name of the attribute that triggered the notification.

Other keys may be passed depending on the value of 'type'. In the case where type is 'change', we also have the following keys: * `owner` : the HasTraits instance * `old` : the old value of the modified trait attribute * `new` : the new value of the modified trait attribute * `name` : the name of the modified trait attribute.

Parameters

- ***names** – The str names of the Traits to observe on the object.
- **type** (*str, kwarg-only*) – The type of event to observe (e.g. 'change')

To do something when a trait attribute is changed, decorate a method with *traitlets.observe()*. The method will be called with a single argument, a dictionary of the form:

```
{
  'owner': object, # The HasTraits instance
  'new': 6, # The new value
  'old': 5, # The old value
  'name': "foo", # The name of the changed trait
  'type': 'change', # The event type of the notification, usually 'change'
}
```

For example:

```
from traitlets import HasTraits, Integer, observe

class TraitletsExample(HasTraits):
    num = Integer(5, help="a number").tag(config=True)

    @observe('num')
    def _num_changed(self, change):
        print("{name} changed from {old} to {new}".format(**change))
```

Changed in version 4.1: The `_{trait}_changed` magic method-name approach is deprecated.

You can also add callbacks to a trait dynamically:

HasTraits.**observe**(*handler*, *names=traitlets.All*, *type='change'*)

Setup a handler to be called when a trait changes.

This is used to setup dynamic notifications of trait changes.

Parameters

- **handler** (*callable*) – A callable that is called when a trait changes. Its signature should be `handler(change)`, where `change` is a dictionary. The change dictionary at least holds a 'type' key. * `type`: the type of notification. Other keys may be passed depending on the value of 'type'. In the case where type is 'change', we also have the following keys: * `owner` : the HasTraits instance * `old` : the old value of the modified trait attribute * `new` : the new value of the modified trait attribute * `name` : the name of the modified trait attribute.
- **names** (*list, str, All*) – If names is All, the handler will apply to all traits. If a list of str, handler will apply to all names in the list. If a str, the handler will apply just to that name.
- **type** (*str, All (default: 'change')*) – The type of notification to filter by. If equal to All, then all notifications are passed to the observe handler.

Note: If a trait attribute with a dynamic default value has another value set before it is used, the default will not be calculated. Any callbacks on that trait will will fire, and *old_value* will be `None`.

Validating proposed changes

`traitlets.`**validate**(**names*)

A decorator to register cross validator of HasTraits object's state when a Trait is set.

The handler passed to the decorator must have one `proposal` dict argument. The proposal dictionary must hold the following keys: * `owner` : the HasTraits instance * `value` : the proposed value for the modified trait attribute * `trait` : the TraitType instance associated with the attribute

Parameters **names** – The str names of the Traits to validate.

Notes

Since the owner has access to the `HasTraits` instance via the 'owner' key, the registered cross validator could potentially make changes to attributes of the `HasTraits` instance. However, we recommend not to do so. The reason is that the cross-validation of attributes may run in arbitrary order when exiting the `hold_trait_notifications` context, and such changes may not commute.

Validator methods can be used to enforce certain aspects of a property. These are called on proposed changes, and can raise a TraitError if the change should be rejected, or coerce the value if it should be accepted with some modification. This can be useful for things such as ensuring a path string is always absolute, or check if it points to an existing directory.

For example:

```
from traitlets import HasTraits, Unicode, validate, TraitError

class TraitletsExample(HasTraits):
    path = Unicode('', help="a path")

    @validate('path')
    def _check_prime(self, proposal):
        path = proposal['value']
        if not path.endswith('/'):
```

```
            # ensure path always has trailing /
            path = path + '/'
        if not os.path.exists(path):
            raise TraitError("path %r does not exist" % path)
        return path
```

CHAPTER 5

Configurable objects with traitlets.config

This document describes *traitlets.config*, the traitlets-based configuration system used by IPython and Jupyter.

The main concepts

There are a number of abstractions that the IPython configuration system uses. Each of these abstractions is represented by a Python class.

Configuration object: `Config` A configuration object is a simple dictionary-like class that holds configuration attributes and sub-configuration objects. These classes support dotted attribute style access (`cfg.Foo.bar`) in addition to the regular dictionary style access (`cfg['Foo']['bar']`). The Config object is a wrapper around a simple dictionary with some convenience methods, such as merging and automatic section creation.

Application: `Application` An application is a process that does a specific job. The most obvious application is the **`ipython`** command line program. Each application reads *one or more* configuration files and a single set of command line options and then produces a master configuration object for the application. This configuration object is then passed to the configurable objects that the application creates. These configurable objects implement the actual logic of the application and know how to configure themselves given the configuration object.

Applications always have a *log* attribute that is a configured Logger. This allows centralized logging configuration per-application.

Configurable: `Configurable` A configurable is a regular Python class that serves as a base class for all main classes in an application. The `Configurable` base class is lightweight and only does one things.

This `Configurable` is a subclass of *HasTraits* that knows how to configure itself. Class level traits with the metadata `config=True` become values that can be configured from the command line and configuration files.

Developers create `Configurable` subclasses that implement all of the logic in the application. Each of these subclasses has its own configuration information that controls how instances are created.

Singletons: `SingletonConfigurable` Any object for which there is a single canonical instance. These are just like Configurables, except they have a class method `instance()`, that returns the current active instance (or creates one if it does not exist). `instance()`.

Note: Singletons are not strictly enforced - you can have many instances of a given singleton class, but the `instance()` method will always return the same one.

Having described these main concepts, we can now state the main idea in our configuration system: *"configuration" allows the default values of class attributes to be controlled on a class by class basis*. Thus all instances of a given class are configured in the same way. Furthermore, if two instances need to be configured differently, they need to be instances of two different classes. While this model may seem a bit restrictive, we have found that it expresses most things that need to be configured extremely well. However, it is possible to create two instances of the same class that have different trait values. This is done by overriding the configuration.

Now, we show what our configuration objects and files look like.

Configuration objects and files

A configuration object is little more than a wrapper around a dictionary. A configuration *file* is simply a mechanism for producing that object. The main IPython configuration file is a plain Python script, which can perform extensive logic to populate the config object. IPython 2.0 introduces a JSON configuration file, which is just a direct JSON serialization of the config dictionary, which is easily processed by external software.

When both Python and JSON configuration file are present, both will be loaded, with JSON configuration having higher priority.

Python configuration Files

A Python configuration file is a pure Python file that populates a configuration object. This configuration object is a `Config` instance. It is available inside the config file as `c`, and you simply set attributes on this. All you have to know is:

- The name of the class to configure.
- The name of the attribute.
- The type of each attribute.

The answers to these questions are provided by the various `Configurable` subclasses that an application uses. Let's look at how this would work for a simple configurable subclass

```
# Sample configurable:
from traitlets.config.configurable import Configurable
from traitlets import Int, Float, Unicode, Bool

class MyClass(Configurable):
    name = Unicode(u'defaultname'
        help="the name of the object"
    ).tag(config=True)
    ranking = Integer(0, help="the class's ranking").tag(config=True)
    value = Float(99.0)
    # The rest of the class implementation would go here..
```

In this example, we see that `MyClass` has three attributes, two of which (`name`, `ranking`) can be configured. All of the attributes are given types and default values. If a `MyClass` is instantiated, but not configured, these default values will be used. But let's see how to configure this class in a configuration file

```
# Sample config file
c.MyClass.name = 'coolname'
c.MyClass.ranking = 10
```

After this configuration file is loaded, the values set in it will override the class defaults anytime a `MyClass` is created. Furthermore, these attributes will be type checked and validated anytime they are set. This type checking is handled by the *`traitlets`* module, which provides the *`Unicode`*, *`Integer`* and *`Float`* types; see *Trait Types* for the full list.

It should be very clear at this point what the naming convention is for configuration attributes:

```
c.ClassName.attribute_name = attribute_value
```

Here, `ClassName` is the name of the class whose configuration attribute you want to set, `attribute_name` is the name of the attribute you want to set and `attribute_value` the the value you want it to have. The `ClassName` attribute of `c` is not the actual class, but instead is another `Config` instance.

Note: The careful reader may wonder how the `ClassName` (`MyClass` in the above example) attribute of the configuration object `c` gets created. These attributes are created on the fly by the `Config` instance, using a simple naming convention. Any attribute of a `Config` instance whose name begins with an uppercase character is assumed to be a sub-configuration and a new empty `Config` instance is dynamically created for that attribute. This allows deeply hierarchical information created easily (`c.Foo.Bar.value`) on the fly.

JSON configuration Files

A JSON configuration file is simply a file that contains a `Config` dictionary serialized to JSON. A JSON configuration file has the same base name as a Python configuration file, but with a .json extension.

Configuration described in previous section could be written as follows in a JSON configuration file:

```
{
  "version": "1.0",
  "MyClass": {
    "name": "coolname",
    "ranking": 10
  }
}
```

JSON configuration files can be more easily generated or processed by programs or other languages.

Configuration files inheritance

Note: This section only applies to Python configuration files.

Let's say you want to have different configuration files for various purposes. Our configuration system makes it easy for one configuration file to inherit the information in another configuration file. The `load_subconfig()`

command can be used in a configuration file for this purpose. Here is a simple example that loads all of the values from the file `base_config.py`:

```
# base_config.py
c = get_config()
c.MyClass.name = 'coolname'
c.MyClass.ranking = 100
```

into the configuration file `main_config.py`:

```
# main_config.py
c = get_config()

# Load everything from base_config.py
load_subconfig('base_config.py')

# Now override one of the values
c.MyClass.name = 'bettername'
```

In a situation like this the `load_subconfig()` makes sure that the search path for sub-configuration files is inherited from that of the parent. Thus, you can typically put the two in the same directory and everything will just work.

Class based configuration inheritance

There is another aspect of configuration where inheritance comes into play. Sometimes, your classes will have an inheritance hierarchy that you want to be reflected in the configuration system. Here is a simple example:

```
from traitlets.config.configurable import Configurable
from traitlets import Integer, Float, Unicode, Bool

class Foo(Configurable):
    name = Unicode(u'fooname', config=True)
    value = Float(100.0, config=True)

class Bar(Foo):
    name = Unicode(u'barname', config=True)
    othervalue = Int(0, config=True)
```

Now, we can create a configuration file to configure instances of `Foo` and `Bar`:

```
# config file
c = get_config()

c.Foo.name = u'bestname'
c.Bar.othervalue = 10
```

This class hierarchy and configuration file accomplishes the following:

- The default value for `Foo.name` and `Bar.name` will be 'bestname'. Because `Bar` is a `Foo` subclass it also picks up the configuration information for `Foo`.
- The default value for `Foo.value` and `Bar.value` will be `100.0`, which is the value specified as the class default.
- The default value for `Bar.othervalue` will be 10 as set in the configuration file. Because `Foo` is the parent of `Bar` it doesn't know anything about the `othervalue` attribute.

Command-line arguments

All configurable options can also be supplied at the command line when launching the application. Applications use a parser called `KeyValueLoader` to load values into a Config object.

By default, values are assigned in much the same way as in a config file:

```
$ ipython --InteractiveShell.use_readline=False --BaseIPythonApplication.profile=
↪'myprofile'
```

Is the same as adding:

```
c.InteractiveShell.use_readline=False
c.BaseIPythonApplication.profile='myprofile'
```

to your configuration file. Key/Value arguments *always* take a value, separated by '=' and no spaces.

Note: By default any error in configuration files with lead to this configuration file be ignored by default. Application subclasses may specify *raise_config_file_errors* = *True* to exit on failure to load config files, instead of the default of logging the failures.

New in version 4.3: The environement variable `TRAITLETS_APPLICATION_RAISE_CONFIG_FILE_ERROR` to `'1'` or `'true'` to change the defautl value of `raise_config_file_errors`.

Common Arguments

Since the strictness and verbosity of the KVLoader above are not ideal for everyday use, common arguments can be specified as *flags* or *aliases*.

Flags and Aliases are handled by `argparse` instead, allowing for more flexible parsing. In general, flags and aliases are prefixed by `--`, except for those that are single characters, in which case they can be specified with a single `-`, e.g.:

```
$ ipython -i -c "import numpy; x=numpy.linspace(0,1)" --profile testing --
↪colors=lightbg
```

Flags and aliases are declared by specifying `flags` and `aliases` attributes as dictionaries on subclasses of `Application`.

Aliases

For convenience, applications have a mapping of commonly used traits, so you don't have to specify the whole class name:

```
$ ipython --profile myprofile
# and
$ ipython --profile='myprofile'
# are equivalent to
$ ipython --BaseIPythonApplication.profile='myprofile'
```

Flags

Applications can also be passed **flags**. Flags are options that take no arguments. They are simply wrappers for setting one or more configurables with predefined values, often True/False.

For instance:

```
$ ipcontroller --debug
# is equivalent to
$ ipcontroller --Application.log_level=DEBUG
# and
$ ipython --matplotlib
# is equivalent to
$ ipython --matplotlib auto
# or
$ ipython --no-banner
# is equivalent to
$ ipython --TerminalIPythonApp.display_banner=False
```

Subcommands

Configurable applications can also have **subcommands**. Subcommands are modeled after **`git`**, and are called with the form **`command subcommand [...args]`**. For instance, the QtConsole is a subcommand of terminal IPython:

```
$ ipython qtconsole --profile myprofile
```

Subcommands are specified as a dictionary on `Application` instances, mapping subcommand names to 2-tuples containing:

1. The application class for the subcommand, or a string which can be imported to give this.
2. A short description of the subcommand for use in help output.

To see a list of the available aliases, flags, and subcommands for a configurable application, simply pass `-h` or `--help`. And to see the full list of configurable options (*very* long), pass `--help-all`.

Design requirements

Here are the main requirements we wanted our configuration system to have:

- Support for hierarchical configuration information.
- Full integration with command line option parsers. Often, you want to read a configuration file, but then override some of the values with command line options. Our configuration system automates this process and allows each command line option to be linked to a particular attribute in the configuration hierarchy that it will override.
- Configuration files that are themselves valid Python code. This accomplishes many things. First, it becomes possible to put logic in your configuration files that sets attributes based on your operating system, network setup, Python version, etc. Second, Python has a super simple syntax for accessing hierarchical data structures, namely regular attribute access (`Foo.Bar.Bam.name`). Third, using Python makes it easy for users to import configuration attributes from one configuration file to another. Fourth, even though Python is dynamically typed, it does have types that can be checked at runtime. Thus, a `1` in a config file is the integer '1', while a `'1'` is a string.
- A fully automated method for getting the configuration information to the classes that need it at runtime. Writing code that walks a configuration hierarchy to extract a particular attribute is painful. When you have complex configuration information with hundreds of attributes, this makes you want to cry.
- Type checking and validation that doesn't require the entire configuration hierarchy to be specified statically before runtime. Python is a very dynamic language and you don't always know everything that needs to be configured when a program starts.

CHAPTER 6

Utils

A simple utility to import something by its string name.

traitlets.**import_item**(*name*)

Import and return `bar` given the string `foo.bar`.

Calling `bar = import_item("foo.bar")` is the functional equivalent of executing the code `from foo import bar`.

Parameters **name** (*string*) – The fully qualified name of the module/package being imported.

Returns **mod** – The module that was imported.

Return type module object

Links

class traitlets.**link**(*source*, *target*)

Link traits from different objects together so they remain in sync.

Parameters

- **source**((*object / attribute name) pair*)–
- **target**((*object / attribute name) pair*)–

Examples

```
>>> c = link((src, 'value'), (tgt, 'value'))
>>> src.value = 5  # updates other objects as well
```

class traitlets.**directional_link**(*source*, *target*, *transform=None*)

Link the trait of a source object with traits of target objects.

Parameters

- **source** (*(object, attribute name) pair*) –
- **target** (*(object, attribute name) pair*) –
- **transform** (*callable (optional)*) – Data transformation between source and target.

Examples

```
>>> c = directional_link((src, 'value'), (tgt, 'value'))
>>> src.value = 5  # updates target objects
>>> tgt.value = 6  # does not update source object
```

CHAPTER 7

Migration from Traitlets 4.0 to Traitlets 4.1

Traitlets 4.1 introduces a totally new decorator-based API for configuring traitlets and a couple of other changes.

However, it is a backward-compatible release and the deprecated APIs will be supported for some time.

Separation of metadata and keyword arguments in `TraitType` constructors

In traitlets 4.0, trait types constructors used all unrecognized keyword arguments passed to the constructor (like `sync` or `config`) to populate the `metadata` dictionary.

In trailets 4.1, we deprecated this behavior. The preferred method to populate the metadata for a trait type instance is to use the new `tag` method.

```
x = Int(allow_none=True, sync=True)       # deprecated
x = Int(allow_none=True).tag(sync=True)   # ok
```

We also deprecated the `get_metadata` method. The metadata of a trait type instance can directly be accessed via the `metadata` attribute.

Deprecation of `on_trait_change`

The most important change in this release is the deprecation of the `on_trait_change` method.

Instead, we introduced two methods, `observe` and `unobserve` to register and unregister handlers (instead of passing `remove=True` to `on_trait_change` for the removal).

- The `observe` method takes one positional argument (the handler), and two keyword arguments, `names` and `type`, which are used to filter by notification type or by the names of the observed trait attribute. The special value `All` corresponds to listening to all the notification types or all notifications from the trait attributes. The `names` argument can be a list of string, a string, or `All` and `type` can be a string or `All`.

- The observe handler's signature is different from the signature of on_trait_change. It takes a single change dictionary argument, containing

```
{
    'type': The type of notification.
}
```

In the case where `type` is the string `'change'`, the following additional attributes are provided:

```
{
    'owner': the HasTraits instance,
    'old': the old trait attribute value,
    'new': the new trait attribute value,
    'name': the name of the changing attribute,
}
```

The `type` key in the change dictionary is meant to enable protocols for other notification types. By default, its value is equal to the `'change'` string which corresponds to the change of a trait value.

Example:

```
from traitlets import HasTraits, Int, Unicode

class Foo(HasTraits):

    bar = Int()
    baz = Unicode()

def handle_change(change):
    print("{name} changed from {old} to {new}".format(**change))

foo = Foo()
foo.observe(handle_change, names='bar')
```

The new `@observe` decorator

The use of the magic methods `_{trait}_changed` as change handlers is deprecated, in favor of a new `@observe` method decorator.

The `@observe` method decorator takes the names of traits to be observed as positional arguments and has a `type` keyword-only argument (defaulting to `'change'`) to filter by notification type.

Example:

```
class Foo(HasTraits):
    bar = Int()
    baz = EnventfulContainer()  # hypothetical trait type emitting
                                # other notifications types

    @observe('bar')  # 'change' notifications for `bar`
    def handler_bar(self, change):
        pass

    @observe('baz ', type='element_change')  # 'element_change' notifications for␣
↪`baz`
    def handler_baz(self, change):
        pass
```

```
    @observe('bar', 'baz', type=All)  # all notifications for `bar` and `baz`
    def handler_all(self, change):
        pass
```

dynamic defaults generation with decorators

The use of the magic methods `_{trait}_default` for dynamic default generation is not deprecated, but a new `@default` method decorator is added.

Example:

Default generators should only be called if they are registered in subclasses of `trait.this_type`.

```
from traitlets import HasTraits, Int, Float, default

class A(HasTraits):
    bar = Int()

    @default('bar')
    def get_bar_default(self):
        return 11

class B(A):
    bar = Float()  # This ignores the default generator
                   # defined in the base class A

class C(B):

    @default('bar')
    def some_other_default(self):  # This should not be ignored since
        return 3.0                 # it is defined in a class derived
                                   # from B.a.this_class.
```

Deprecation of magic method for cross-validation

`traitlets` enables custom cross validation between the different attributes of a `HasTraits` instance. For example, a slider value should remain bounded by the `min` and `max` attribute. This validation occurs before the trait notification fires.

The use of the magic methods `_{name}_validate` for custom cross-validation is deprecated, in favor of a new `@validate` method decorator.

The method decorated with the `@validate` decorator take a single `proposal` dictionary

```
{
    'trait': the trait type instance being validated
    'value': the proposed value,
    'owner': the underlying HasTraits instance,
}
```

Custom validators may raise `TraitError` exceptions in case of invalid proposal, and should return the value that will be eventually assigned.

Example:

```
from traitlets import HasTraits, TraitError, Int, Bool, validate

class Parity(HasTraits):
    value = Int()
    parity = Int()

    @validate('value')
    def _valid_value(self, proposal):
        if proposal['value'] % 2 != self.parity:
            raise TraitError('value and parity should be consistent')
        return proposal['value']

    @validate('parity')
    def _valid_parity(self, proposal):
        parity = proposal['value']
        if parity not in [0, 1]:
            raise TraitError('parity should be 0 or 1')
        if self.value % 2 != parity:
            raise TraitError('value and parity should be consistent')
        return proposal['value']

parity_check = Parity(value=2)

# Changing required parity and value together while holding cross validation
with parity_check.hold_trait_notifications():
    parity_check.value = 1
    parity_check.parity = 1
```

The presence of the `owner` key in the proposal dictionary enable the use of other attributes of the object in the cross validation logic. However, we recommend that the custom cross validator don't modify the other attributes of the object but only coerce the proposed value.

Backward-compatible upgrades

One challenge in adoption of a changing API is how to adopt the new API while maintaining backward compatibility for subclasses, as event listeners methods are *de facto* public APIs.

Take for instance the following class:

```
from traitlets import HasTraits, Unicode

class Parent(HasTraits):
    prefix = Unicode()
    path = Unicode()
    def _path_changed(self, name, old, new):
        self.prefix = os.path.dirname(new)
```

And you know another package has the subclass:

```
from parent import Parent
class Child(Parent):
    def _path_changed(self, name, old, new):
        super()._path_changed(name, old, new)
        if not os.path.exists(new):
            os.makedirs(new)
```

If the parent package wants to upgrade without breaking Child, it needs to preserve the signature of `_path_changed`. For this, we have provided an `@observe_compat` decorator, which automatically shims the deprecated signature into the new signature:

```
from traitlets import HasTraits, Unicode, observe, observe_compat

class Parent(HasTraits):
    prefix = Unicode()
    path = Unicode()

    @observe('path')
    @observe_compat # <- this allows super()._path_changed in subclasses to work with␣
↪the old signature.
    def _path_changed(self, change):
        self.prefix = os.path.dirname(change['value'])
```

CHAPTER 8

Changes in Traitlets

4.3

4.3.2

4.3.2 on GitHub

4.3.2 is a tiny release, relaxing some of the deprecations introduced in 4.1:

- using `_traitname_default()` without the `@default` decorator is no longer deprecated.
- Passing `config=True` in traitlets constructors is no longer deprecated.

4.3.1

4.3.1 on GitHub

- Compatibility fix for Python 3.6a1
- Fix bug in Application.classes getting extra entries when multiple Applications are instantiated in the same process.

4.3.0

4.3.0 on GitHub

- Improve the generated config file output.
- Allow TRAITLETS_APPLICATION_RAISE_CONFIG_FILE_ERROR env to override `Application.raise_config_file_errors`, so that config file errors can result in exiting immediately.
- Avoid using root logger. If no application logger is registered, the `'traitlets'` logger will be used instead of the root logger.
- Change/Validation arguments are now Bunch objects, allowing attribute-access, in addition to dictionary access.

- Reduce number of common deprecation messages in certain cases.
- Ensure command-line options always have higher priority than config files.
- Add bounds on numeric traits.
- Improves various error messages.

4.2

4.2.2 - 2016-07-01

4.2.2 on GitHub

Partially revert a change in 4.1 that prevented IPython's command-line options from taking priority over config files.

4.2.1 - 2016-03-14

4.2.1 on GitHub

Demotes warning about unused arguments in `HasTraits.__init__` introduced in 4.2.0 to DeprecationWarning.

4.2.0 - 2016-03-14

4.2 on GitHub

- `JSONFileConfigLoader` can be used as a context manager for updating configuration.
- If a value in config does not map onto a configurable trait, a message is displayed that the value will have no effect.
- Unused arguments are passed to `super()` in `HasTraits.__init__`, improving support for multiple inheritance.
- Various bugfixes and improvements in the new API introduced in 4.1.
- Application subclasses may specify `raise_config_file_errors = True` to exit on failure to load config files, instead of the default of logging the failures.

4.1 - 2016-01-15

4.1 on GitHub

Traitlets 4.1 introduces a totally new decorator-based API for configuring traitlets. Highlights:

- Decorators are used, rather than magic method names, for registering trait-related methods. See *Using Traitlets* and *Migration from Traitlets 4.0 to Traitlets 4.1* for more info.
- Deprecate `Trait(config=True)` in favor of `Trait().tag(config=True)`. In general, metadata is added via `tag` instead of the constructor.

Other changes:

- Trait attributes initialized with `read_only=True` can only be set with the `set_trait` method. Attempts to directly modify a read-only trait attribute raises a `TraitError`.
- The directional link now takes an optional *transform* attribute allowing the modification of the value.

- Various fixes and improvements to config-file generation (fixed ordering, Undefined showing up, etc.)
- Warn on unrecognized traits that aren't configurable, to avoid silently ignoring mistyped config.

4.0 - 2015-06-19

4.0 on GitHub

First release of traitlets as a standalone package.

Python Module Index

t

Index

Symbols

A

B

C

D

E

F

H

I

L

M

O

S

T

U

V