traitlets Documentation

Release 4.1.0

The IPython Development Team

September 06, 2016

Contents

1 Using Traitlets **3**
- 1.1 Dynamic default values . . . 3
- 1.2 Callbacks when trait attributes change . . . 3

2 Trait Types **5**
- 2.1 Numbers . . . 5
- 2.2 Strings . . . 5
- 2.3 Containers . . . 6
- 2.4 Classes and instances . . . 6
- 2.5 Miscellaneous . . . 6

3 Defining new trait types **7**

4 Configurable objects with traitlets.config **9**
- 4.1 The main concepts . . . 9
- 4.2 Configuration objects and files . . . 10
- 4.3 Configuration files inheritance . . . 11
- 4.4 Class based configuration inheritance . . . 12
- 4.5 Command-line arguments . . . 12
- 4.6 Design requirements . . . 14

5 Migration from Traitlets 4.0 to Traitlets 4.1 **15**
- 5.1 Separation of metadata and keyword arguments in `TraitType` contructors . . . 15
- 5.2 Deprecation of `on_trait_change` . . . 15
- 5.3 The new `@observe` decorator . . . 16
- 5.4 Deprecation of magic method for dynamic defaults generation . . . 17
- 5.5 Deprecation of magic method for cross-validation . . . 17
- 5.6 Backward-compatible upgrades . . . 18

6 Changes in Traitlets **21**
- 6.1 4.1 . . . 21
- 6.2 4.0 . . . 21

Python Module Index **23**

Release 4.1.0

Date September 06, 2016

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

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

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.

1.1 Dynamic default values

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()
```

1.2 Callbacks when trait attributes 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:

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`.

CHAPTER 2

Trait Types

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

2.1 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.**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)`.

2.2 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.

2.3 Containers

2.4 Classes and instances

2.5 Miscellaneous

class `traitlets.`**`CBool`**

Casting variant. When a value is assigned to the attribute, this will attempt to convert it by calling `value = bool(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], py3compat.string_types) and isinstance(value[1], int):
                    port = value[1]
                    if port >= 0 and port <= 65535:
                        return value
        self.error(obj, value)
```

CHAPTER 4

Configurable objects with traitlets.config

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

4.1 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.

4.2 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.

4.2.1 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.

4.2.2 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.

4.3 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.

4.4 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.

4.5 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 config file. Key/Value arguments *always* take a value, separated by '=' and no spaces.

4.5.1 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
```

4.5.2 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`.

4.6 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 5

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.

5.1 Separation of metadata and keyword arguments in `TraitType` contructors

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.

5.2 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(bar_changed, names='bar')
```

5.3 The new `@observe` decorator

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

In addition to the `names` argument, the `@observe` method decorator has a `type` keyword 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
```

5.4 Deprecation of magic method for dynamic defaults generation

The use of the magic methods `_{trait}_default` for dynamic default generation is deprecated, in favor a new `@default` method decorator.

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.
```

5.5 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.

5.6 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 signatu
    def _path_changed(self, change):
        self.prefix = os.path.dirname(change['value'])
```

CHAPTER 6

Changes in Traitlets

6.1 4.1

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.

6.2 4.0

4.0 on GitHub

First release of traitlets as a standalone package.

t

traitlets, 5
traitlets.config, 9

C

CBool (class in traitlets), 6
CBytes (class in traitlets), 5
CComplex (class in traitlets), 5
CFloat (class in traitlets), 5
CInt (class in traitlets), 5
CLong (class in traitlets), 5
CUnicode (class in traitlets), 5

D

default_value (traitlets.MyTrait attribute), 7

I

info_text (traitlets.MyTrait attribute), 7
Int (class in traitlets), 5
Integer (class in traitlets), 5

L

Long (class in traitlets), 5

M

MyTrait (class in traitlets), 7

T

traitlets (module), 5
traitlets.config (module), 9
TraitType (class in traitlets), 5

V

validate() (traitlets.MyTrait method), 7