properties Documentation

Release 0.5.4

Seequent

Oct 02, 2018

Contents

1 Overview Video **3**

2 Why **5**

3 Scope **7**

4 Goals **9**

5 Documentation **11**

6 Alternatives **13**

7 Connections **15**

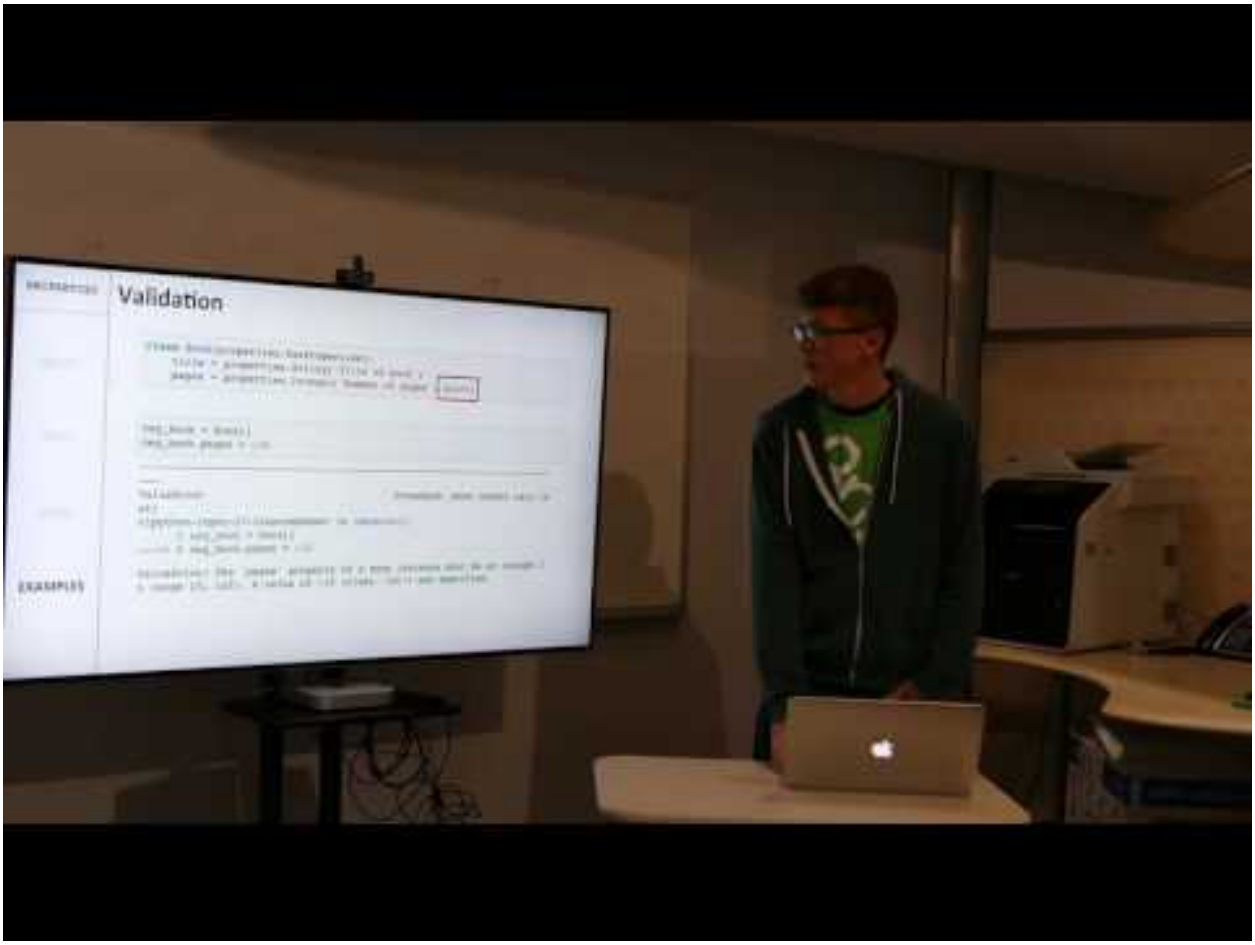
8 Installation **17**

8.1 Examples 17

9 Indices and tables **43**

CHAPTER 1

Overview Video



An overview of Properties, November 2016.

CHAPTER 2

Why

Properties provides structure to aid development in an interactive programming environment while allowing for an easy transition to production code. It emphasizes usability and reproducibility for developers and users at every stage of the code life cycle.

CHAPTER 3

Scope

The `properties` package enables the creation of **strongly typed** objects in a consistent, declarative way. This allows **validation** of developer expectations and hooks into **notifications** and other libraries. It provides **documentation** with no extra work, and **serialization** for portability and reproducibility.

CHAPTER 4

Goals

- Keep a clean namespace for easy interactive programming
- Prioritize documentation
- Provide built-in serialization/deserialization
- Connect to other libraries for GUIs and visualizations

CHAPTER 5

Documentation

API Documentation is available at ReadTheDocs.

CHAPTER 6

Alternatives

- attrs - "Python Classes Without Boilerplate" - This is a popular, actively developed library that aims to simplify class creation, especially around object protocols (i.e. dunder methods), with concise, declarative code.

  Similarities to Properties include type-checking, defaults, validation, and coercion. There are a number of differences:

  1. attrs acts somewhat like a *namedtuple*, whereas properties acts more like a *dict* or mutable object.
     - as a result, attrs is able to tackle hashing, comparison methods, string representation, etc.
     - attrs does not suffer runtime performance penalties as much as properties
     - on the other hand, properties focuses on interactivity, with notifications, serialization/deserialization, and mutable, possibly invalid states.
  2. properties has many built-in types with existing, complex validation already in place. This includes recursive validation of container and instance properties. attrs only allows attribute type to be specified.
  3. properties is more prescriptive and detailed around auto-generated class documentation, for better or worse.

- traitlets (Jupyter project) and traits (Enthought) - These libraries are driven by GUI development (much of the Jupyter environment is built on traitlets; traits has automatic GUI generation) which leads to many similar features as properties such as strong typing, validation, and notifications. Also, some Properties features and aspects of the API take heavy inspiration from traitlets.

  However, There are a few key areas where properties differs:

  1. properties has a clean namespace - this (in addition to *?* docstrings) allows for very easy discovery in an interactive programming environment.
  2. properties prioritizes documentation - this is not explicitly implemented yet in traits or traitlets, but works out-of-the-box in properties.
  3. properties prioritizes serialization - this is present in traits with pickling (but difficult to customize) and in traitlets with configuration files (which require extra work beyond standard class definition); in properties, serialization works out of the box but is also highly customizable.

4. properties allows invalid object states - the GUI focus of traits/traitlets means an invalid object state at any time is never ok; without that constraint, properties allows interactive object building and experimentation. Validation then occurs when the user is ready and calls `validate`

Significant advantages of traitlets and traits over properties are GUI interaction and larger suites of existing property types. Besides numerous types built-in to these libraries, some other examples are trait types that support unit conversion and NumPy/SciPy trait types (note: properties has a NumPy array property type).

Note: properties provides a `link` object which inter-operates with traitlets and follows the same API as traitlets links

---

- param - This library also provides type-checking, validation, and notification. It has a few unique features and parameter types (possibly of note is the ability to provide dynamic values for parameters at any time, not just as the default). This was first introduced before built-in Python properties, and current development is very slow.
- built-in Python dataclass decorator - provides "mutable named tuples with defaults" - this provides similar functionality to the attrs by adding several object protocol dunder methods to a class. Data Classes are clean, lightweight and included with Python 3.7. However, they don't provide as much builtin functionality or customization as the above libraries.
- built-in Python property - properties/traits-like behavior can be mostly recreated using `@property`. This requires significantly more work by the programmer, and results in not-declarative, difficult-to-read code.
- mypy, PEP 484, and PEP 526 - This provides static typing for Python without coersion, notifications, etc. It has a very different scope and implementation than traits-like libraries.

CHAPTER 7

Connections

- casingSimulations - Research repository for electromagnetic simulations in the presence of well casing
- OMF - Open Mining Format API and file serialization
- SimPEG - Simulation and Parameter Estimation in Geophysics
- Steno3D - Python client for building and uploading 3D models

CHAPTER 8

Installation

To install the repository, ensure that you have pip installed and run:

```
pip install properties
```

For the development version:

```
git clone https://github.com/seequent/properties.git
cd properties
pip install -e .
```

8.1 Examples

Lets start by making a class to organize your coffee habits.

```
import properties
class CoffeeProfile(properties.HasProperties):
    name = properties.String('What should I call you?')
    count = properties.Integer(
        'How many coffees have you had today?',
        default=0
    )
    had_enough_coffee = properties.Bool(
        'Have you had enough coffee today?',
        default=False
    )
    caffeine_choice = properties.StringChoice(
        'How do you take your caffeine?' ,
        choices=['coffee', 'tea', 'latte', 'cappuccino', 'something fancy'],
        required=False
    )
```

The `CoffeeProfile` class has 4 properties, all of which are documented! These can be set on class instantiation:

```
profile = CoffeeProfile(name='Bob')
print(profile.name)

Out [1]: Bob
```

Since a default value was provided for `had_enough_coffee`, the response is (naturally)

```
print(profile.had_enough_coffee)

Out [2]: False
```

We can set Bob's `caffeine_choice` to one of the available choices; he likes coffee

```
profile.caffeine_choice = 'coffee'
```

Also, Bob is half way through his fourth cup of coffee today:

```
profile.count = 3.5

Out [3]: ValueError: The 'count' property of a CoffeeProfile instance must
         be an integer.
```

Ok, Bob, chug that coffee:

```
profile.count = 4
```

Now that Bob's `CoffeeProfile` is established, `properties` can check that it is valid:

```
profile.validate()

Out [4]: True
```

Property Classes are auto-documented in Sphinx-style reStructuredText! When you ask for the doc string of `CoffeeProfile`, you get

```
**Required Properties:**

* **count** (:class:`Integer <properties.basic.Integer>`): How many coffees have you␣
↪had today?, an integer, Default: 0
* **had_enough_coffee** (:class:`Bool <properties.basic.Bool>`): Have you had enough␣
↪coffee today?, a boolean, Default: False
* **name** (:class:`String <properties.basic.String>`): What should I call you?, a␣
↪unicode string

**Optional Properties:**

* **caffeine_choice** (:class:`StringChoice <properties.basic.StringChoice>`): How do␣
↪you take your caffeine?, any of "coffee", "tea", "latte", "cappuccino", "something␣
↪fancy"
```

Contents:

8.1.1 HasProperties

class properties.**HasProperties**(***kwargs*)

Base class to enable *Property* behavior

Classes that inherit **HasProperties** need simply to declare the Properties they need. **HasProperties** will save these Properties as `_props` on the class. Property values will be saved to `_backend` on the instance.

HasProperties classes also store a registry of all **HasProperties** classes in as `_REGISTRY`. If a subclass redeclares `_REGISTRY`, the subsequent subclasses will be saved to this new registry.

The *PropertyMetaclass* contains more information about what goes into **HasProperties** class construction and validation.

classmethod deserialize(*value*, *trusted=False*, *strict=False*, *assert_valid=False*, ***kwargs*)
Creates **HasProperties** instance from serialized dictionary

This uses the Property deserializers to deserialize all JSON-compatible dictionary values into their corresponding Property values on a new instance of a **HasProperties** class. Extra keys in the dictionary that do not correspond to Properties will be ignored.

Parameters:

- **value** - Dictionary to deserialize new instance from.
- **trusted** - If True (and if the input dictionary has `'__class__'` keyword and this class is in the registry), the new **HasProperties** class will come from the dictionary. If False (the default), only the **HasProperties** class this method is called on will be constructed.
- **strict** - Requires `'__class__'`, if present on the input dictionary, to match the deserialized instance's class. Also disallows unused properties in the input dictionary. Default is False.
- **assert_valid** - Require deserialized instance to be valid. Default is False.
- Any other keyword arguments will be passed through to the Property deserializers.

serialize(*include_class=True*, *save_dynamic=False*, ***kwargs*)
Serializes a **HasProperties** instance to dictionary

This uses the Property serializers to serialize all Property values to a JSON-compatible dictionary. Properties that are undefined are not included. If the **HasProperties** instance contains a reference to itself, a `properties.SelfReferenceError` will be raised.

Parameters:

- **include_class** - If True (the default), the name of the class will also be saved to the serialized dictionary under key `'__class__'`
- **save_dynamic** - If True, dynamic properties are written to the serialized dict (default: False).
- Any other keyword arguments will be passed through to the Property serializers.

validate()
Call all registered class validator methods

These are all methods decorated with `@properties.validator`. Validator methods are expected to raise a ValidationError if they fail.

class properties.base.**PropertyMetaclass**
Metaclass to establish behavior of **HasProperties** classes

On class construction:

- Build Property dictionary from the class dictionary and the base classes' Properties.
- Build listener dictionaries from class dictionary and the base classes' listeners.
- Check Property names are not private.
- Ensure the Property names referred to by *Renamed Properties* and handlers are valid.

- Build class docstring.
- Construct default value dictionary, and check that any provided defaults are valid.
- Add the class to the **HasProperties** `_REGISTRY` or the closest parent class with a new registry defined

On class instantiation:

- Initialize private backend dictionary where Property values are stored.
- Initialize private listener dictionary and set the listeners on the class instance.
- Set all the default values on the class without firing change notifications.

Functions that act on HasProperties instances:

`properties.`**`copy`**(*value*, ***kwargs*)

Return a copy of a **HasProperties** instance

A copy is produced by serializing the HasProperties instance then deserializing it to a new instance. Therefore, if any properties cannot be serialized/deserialized, `copy` will fail. Any keyword arguments will be passed through to both `serialize` and `deserialize`.

`properties.`**`equal`**(*value_a*, *value_b*)

Determine if two **HasProperties** instances are equivalent

Equivalence is determined by checking if (1) the two instances are the same class and (2) all Property values on two instances are equal, using `Property.equal`. If the two values are the same HasProperties instance (eg. `value_a is value_b`) this method returns True. Finally, if either value is not a HasProperties instance, equality is simply checked with ==.

Note: HasProperties objects with recursive self-references will not evaluate to equal, even if their property values and structure are equivalent.

HasProperties features:

- *Documentation* - Classes are auto-documented with a sphinx-style docstring.
- *Validation* - Instances ensure property values remain correct, compatible, and complete.
- *Notifications* - Classes allow callbacks to be registered for property changes.
- *Serialization* - Instances may be serialized to and deserialized from JSON.
- *Defaults* - Default property values can be set at the HasProperties or Property level.
- *Registry* - All HasProperties classes are saved to a class registry.

Documentation

HasProperties class docstrings are written in the metaclass. These docstrings include any docstring that is provided in the class definition as well as information about all the Properties on the class, including their name, description, default value, and if they are required

---

Note: Properties are documented in three groups: Required, Optional, and Other. Within these groups, they are in alphabetical order by default. This can be overridden by defining `_doc_order`, a list of Property names in the desired order, in a HasProperties class. However, this alternative order only applies within the Required/Optional/Other groupings; it does not supersede these groups.

By default, docstrings are formatted in Sphinx-style reStructuredText. This simplifies creation of easy-to-read, linked html documentation. Format is slightly modified for readability in an IPython; however, this only applies to the auto-generated portion of docstrings. Explicit Sphinx tags and formatting present in the source code will not be rewritten.

Note: Intersphinx linking requires some care to be taken when constructing docs:

- Linked classes (for example, Instance Property classes or custom Property subclasses) must be present somewhere in the docs with their *full* module path, even if they are exported to a different namespace.
- If external classes are used, the outside library must be referenced with `intersphinx_mapping` in the `conf.py` Sphinx configuration file.
- To customize Sphinx linking the `sphinx_class` method on Property subclasses must be overridden

---

Validation

Validation is used for type-checking, value coercion, and checking HasProperties instances are composed correctly. Invalid values raises a `ValueError`. There are three components of validation:

1. **Property validation** - This occurs when the *`Property.validate`* method is called. It contains Property-specific type checking and coersion. On a HasProperties class, every time a Property value is set, the corresponding validate method is called and the output of the validate function is used for the new Property value. If the value is not valid, a `ValueError` is raised.
2. **HasProperties property validators** - These are callback methods registered to fire on specific HasProperties-class properties. They are called when the property is set after Property validation but before the property is saved (unlike *observers* which fire after the value is saved). These validators may perform further type-checking or coercion that is related to the HasProperties class. See *`properties.validator`* (Mode 1) for more details on using these validators. The *`properties.validators_disabled`* and *`properties.listeners_disabled`* context managers may be used to disable these validators.
3. **HasProperties class validators** - These are callback methods registered to fire only when *`HasProperties.validate`* is called. They are used to cross-validate Properties and ensure that a HasProperties instance is correctly constructed. See *`properties.validator`* (Mode 2) for more details on using these validators.

`properties.`**`validator`**(*names_or_instance*, *names=None*, *func=None*)

Specify a callback function to fire on class validation OR property set

This function has two modes of operation:

1. Registering callback functions that validate Property values when they are set, before the change is saved to the HasProperties instance. This mode is very similar to the `observer` function.
2. Registering callback functions that fire only when the HasProperties `validate` method is called. This allows for cross-validation of Properties that should only fire when all required Properties are set.

Mode 1:

Validator functions on a HasProperties class fire on set but before the observed Property or Properties have been changed (unlike observer functions that fire after the value has been changed).

You can use this method as a decorator inside a HasProperties class

```
@properties.validator('variable_name')
def callback_function(self, change):
    print(change)
```

or you can use it to register a function to a single HasProperties instance

```
properties.validator(my_has_props, 'variable_name', callback_function)
```

The variable name must refer to a Property name on the HasProperties class. A list of Property names may also be used; the same callback function will fire when any of these Properties change. Also, *properties.everything* may be specified instead of the variable name. In that case, the callback function will fire when any Property changes.

The callback function must take two arguments. The first is the HasProperties instance; the second is the change notification dictionary. This dictionary contains:

- 'name' - the name of the changed Property
- 'previous' - the value of the Property prior to change (this will be `properties.undefined` if the value was not previously set)
- 'value' - the new value of the Property (this will be `properties.undefined` if the value is deleted)
- 'mode' - the mode of the change; for validators, this is 'validate'

Mode 2:

When used as a decorator without arguments (i.e. called directly on a HasProperties method), the decorated method is registered as a class validator. These methods execute only when `validate()` is called on the HasProperties instance.

```
@properties.validator
def validation_method(self):
    print('validating instance of {}'.format(self.__class__))
```

The decorated function must only take one argument, the HasProperties instance.

class properties.**validators_disabled**

Context manager for disabling all property change validators

This context manager behaves like *properties.listeners_disabled*, but only affects HasProperties methods decorated with `@validator`

Notifications

properties.**observer**(*names_or_instance*, *names=None*, *func=None*, *change_only=False*)

Specify a callback function that will fire on Property value change

Observer functions on a HasProperties class fire after the observed Property or Properties have been changed (unlike validator functions that fire on set before the value is changed).

You can use this method as a decorator inside a HasProperties class

```
@properties.observer('variable_name')
def callback_function(self, change):
    print(change)
```

or you can use it to register a function to a single HasProperties instance

```
properties.observer(my_has_props, 'variable_name', callback_function)
```

The variable name must refer to a Property name on the HasProperties class. A list of Property names may also be used; the same callback function will fire when any of these Properties change. Also, *properties.everything* may be specified instead of the variable name. In that case, the callback function will fire when any Property changes.

The callback function must take two arguments. The first is the HasProperties instance; the second is the change notification dictionary. This dictionary contains:

- 'name' - the name of the changed Property
- 'previous' - the value of the Property prior to change (this will be `properties.undefined` if the value was not previously set)
- 'value' - the new value of the Property (this will be `properties.undefined` if the value is deleted)
- 'mode' - the mode of the change; for observers, this is either 'observe_set' or 'observe_change'

Finally, the keyword argument **change_only** may be specified as a boolean. If False (the default), the callback function will fire any time the Property is set. If True, the callback function will only fire if the new value is different than the previous value, determined by the `Property.equal` method.

class properties.**listeners_disabled**(*disable_type=None*)

Context manager for disabling all HasProperties listeners

Code that runs inside this context manager will not fire HasProperties methods decorated with `@validator` or `@observer`. This context manager has no effect on Property validation.

```
with properties.listeners_disabled():
    self.quietly_update()
```

class properties.**observers_disabled**

Context manager for disabling all property change observers

This context manager behaves like *properties.listeners_disabled*, but only affects HasProperties methods decorated with `@observer`

Linking across Properties/Traitlets

Properties has `link` functions similar to those from traitlets. This allows easy connection to IPython widgets and other projects that build on traitlets.

class properties.**directional_link**(*source*, *target*, *update_now=False*, *change_only=True*, *transform=None*)

Link two properties so updating the source updates the target

source and **target** must each be tuples of HasProperties (or traitlets.HasTraits, if available) instances and property (or trait) name.

If **update_now** is True, the target value will be updated to the source value on link. If False, it will not update until the source value is set. The default is False to prevent conflicts with how properties and traitlets deal with uninitialized values.

The **change_only** keyword argument determines if target updates when the source value is set but unchanged. If True, the target only updates when the source value changes; this is the default to mirror behavior from traitlets. It should only be set to False when the source instance is HasProperties.

If a **transform** function is provided, the target will be updated with the transformed source value.

relink()
Re-enable an unlinked directional link

unlink()
Disable a directional link

Note: This does not delete the observer callbacks; it simply makes them non-functional.

---

class properties.**link**(**items*, ***kwargs*)
Link property values to keep them in sync

`link` takes two or more **items** to link. Each item must be a tuple of HasProperties (or traitlets.HasTraits, if available) instances and property (or trait) name. This creates a series of directional links to connect all items.

Available keyword arguments are **update_now** and **change_only**. These are passed through to the *directional links*.

Note: If an error is encountered when updating multiple linked items, some linked properties may not get updated. The order in which properties are updated depends on the order of **items**. There is no validation to ensure linked items are compatible Property types.

Warning: Linking `n` items sets up `n*(n-1)` directional links, all of which may fire on one change. Some care should be taken when creating links among a large number of items.

relink()
Re-enable all unlinked directional links used by link

unlink()
Disable all directional links used by link

Serialization

HasProperties come with relatively naive JSON serialization built-in. To use this, simply call `serialize()` on a HasProperties instance.

However, built-in serialization is somewhat limited.

- Some property types are not JSON-serializable out of the box, for example, *File*. Other properties may have unwanted results when serializing to JSON (for example, *Arrays* will become a list).
- HasProperties instances are serialized as nested dictionaries, so self references will prevent serialization.

To overcome this a *Property* instance may have a serializer and/or deserializer registered. These are functions that take a Property value into and out of any arbitrary serialized state; this state could be anything from an alternative JSON form to a saved file to a web request.

Validation vs. Serialization/Deserialization

For some Property types, validation and serialization/deserialization look very similar; they both convert between an invalid-but-understood value and a valid Property value. However, they remain separate because they serve different purposes:

Validation and coercion happen on input of Property values and on `validate()`. This is taking "human-accessible" user input and ensuring it is the "valid" type.

Serialization takes the *valid* `HasProperties` class and converts it to something that can be saved to a file. Deserialization is the reverse of that process, and should be used only on serialization's output.

With simple properties like strings, validation and serialization almost identical. User input, valid value, and saveable-to-file value are all just the same string. However, the differences are apparent with more complicated properties like Array - in that case, user input may be a list or a numpy array, valid type is a numpy array, and serialized value may be a binary file or something. Validate needs to deal with the user input whereas deserialize needs to deal with the binary file.

Defaults

When a HasProperties class is instantiated, default Property values may come from three places. These include, in order of precedence:

1. `_defaults` dictionary on a HasProperties class. This dictionary has Property name/value pairs.

   Note: Property values specified in `_defaults` are inherited by subclasses unless they are explicitly overwritten in a subclass's `_defaults` dictionary.

2. `default` value specified as a keyword argument on the *Property* instance.
3. `_class_default` defined on the Property class.

Note: Regardless of where the default value is defined, there are several points to note:

- Default values may be callables. In this case `value()` will be used as the default rather than `value`. For example, if you want a *properties.List* to default to an empty list, you set the default to `list` rather than `list()` or `[]`, so a new list is created every time.
- To eliminate any default value, the default can be set to *properties.undefined*. This is also the fallback *_class_default* for all Properties if no other default is specified.
- Default values are validated in the *HasProperties metaclass*

Registry

Whenever a new HasProperties class is created, it is added to the class `_REGISTRY` defined on *HasProperties*. This allows classes to be easily referenced and accessed by name. For example, when serializing an instance, its `__class__` may be saved. Then on deserialization, the instance can be reconstructed based on the corresponding entry in the registry.

`_REGISTRY` can also be overridden in HasProperties subclasses. This creates a separate registry branch where all subclasses on the branch are saved to the new registry. Overriding `_REGISTRY` may be necessary to prevent namespace conflicts when importing multiple modules with HasProperties classes.

8.1.2 Property

class properties.**Property**(*doc*, ***kwargs*)
Property class provides documentation, validation, and serialization

When defined within a HasProperties class, each Property contributes to class documentation, validation, and serialization while behaving for the user just like `@property` values on the class. For examples, see the *HasProperties* documentation and documentation for specific *Property types*.

Available keywords:

- **doc** - Docstring for the Property. Must be provided on instantiation.
- **default** - Default value for the Property. This may be a callable that takes no arguments. Upon HasProperties instantiation, default value is assigned to the Property. If no default is given, the Property value will be undefined.
- **required** - If True, Property must be given a value for the containing HasProperties instance to pass `validate()`. If false, the Property may remain undefined. By default, required is True.
- **serializer** - Function that will serialize the Property value when the containing HasProperties instance is serialized. The serializer must be a callable that takes the value to be serialized and possibly keyword arguments passed to `serialize`. By default, the serializer writes to JSON.
- **deserializer** - Function that will deserialize an input value to a valid Property value when a HasProperties instance is deserialized. The deserializer must be a callable that takes the value to be deserialized and possibly keyword arguments passed to `deserialize`. By default, the deserializer writes to JSON.
- **name** - Name of the Property. This is overwritten in the HasProperties metaclass to correspond to the Property's assigned name.

assert_valid(*instance*, *value=None*)
Returns True if the Property is valid on a HasProperties instance

Raises a ValueError if the value required and not set, not valid, not correctly coerced, etc.

Note: Unlike `validate`, this method requires instance to be a HasProperties instance; it cannot be None.

---

deserialize(*value*, ***kwargs*)
Deserialize input value to valid Property value

This method uses the Property `deserializer` if available. Otherwise, it uses `from_json`. Any keyword arguments are passed through to these methods.

equal(*value_a*, *value_b*)
Check if two valid Property values are equal

Note: This method assumes that `None` and `properties.undefined` are never passed in as values

---

error(*instance*, *value*, *error_class=None*, *extra=''*)
Generate a `ValueError` for invalid value assignment

The instance is the containing HasProperties instance, but it may be None if the error is raised outside a HasProperties class.

static from_json(*value*, ***kwargs*)
Statically load a Property value from JSON value

meta
Get the tagged metadata of a Property instance

serialize(*value*, ***kwargs*)
Serialize a valid Property value

This method uses the Property `serializer` if available. Otherwise, it uses `to_json`. Any keyword arguments are passed through to these methods.

tag(**tag*, ***kwtags*)
Tag a Property instance with metadata dictionary

static to_json(*value*, ***kwargs*)
Statically convert a valid Property value to JSON value

validate(*instance*, *value*)
Check if the value is valid for the Property

If valid, return the value, possibly coerced from the input value. If invalid, a ValueError is raised.

> **Warning:** Calling `validate` again on a coerced value must not modify the value further.

Note: This function should be able to handle `instance=None` since valid Property values are independent of containing HasProperties class. However, the instance is passed to `error` for a more verbose error message, and it may be used for additional optional validation.

Defining custom Property types

Custom Property types can be created by subclassing *`Property`* and customizing a few attributes and methods. These include:

`class_info`/`info`

This are used when documenting the Property. `class_info` is a general, descriptive string attribute of the new Property class. `info` is an `@property` method that gives an instance-specific description of the Property, if necessary. If `info` is not defined, it defaults to `class_info`. This string is used in HasProperties class docstrings and error messages.

`validate(self, instance, value)`

This method defines what values the Property will accept. It must return the validated value. This value may be coerced from the input **value**; however, validating on the coerced value must not modify the value further.

The input **instance** is the containing HasProperties instance or None if the Property is not part of a HasProperties instance, so validate must account for either of these scenarios. Usually, Property validation should be instance-independent.

If **value** is invalid, a ValueError should be raised by calling `self.error(instance, value)`

`to_json(value, **kwargs)`/`from_json(value, **kwargs)`

These static methods should allow converting between a validated Property value and a JSON-dumpable version of the Property value. Both these methods assume the value is valid.

The `serialize` and `deserialize` should not need to be customized in new Properties; they simply call upon these methods.

`equal(self, value_a, value_b)`

This method defines how valid property values should be compared for equality if the default *value_a == value_b* is insufficient.

`_class_default`

This should be set to the default value of the new property class. It may also be a callable that returns the default value. Almost always this should be left untouched; in that case, the default will be `properties.undefined`. However, in some specific cases, it may make sense to override.

8.1.3 Built-in Property types

In addition to setting up the base *HasProperties* and *Property* behavior, the `properties` library defines many built-in Property types.

Basic Property types

- *Primitive Properties* - Properties for primitive data types (e.g. integers, strings, etc.)
- *Math Properties* - Math Properties that rely on numpy
- *Image Properties* - Image Properties that rely on external image libraries
- *Other Property Types* - Other basic Properties with no extra dependencies

Advanced Property types

- *Instance Property* - Property for HasProperties (or other class) instances
- *Container Properties* - Tuple, list, and set properties
- *Union Property* - Properties that may be multiple types

Special Property types

- *Gettable Property* - Immutable Property set when Property is defined
- *Dynamic Property* - Property that is calculated dynamically and never saved
- *Renamed Property* - Used to maintain backwards compatibility when renaming Properties

Primitive Properties

Boolean

class `properties.`**Boolean**(*doc*, ***kwargs*)

Property for True or False values

Available keywords (in addition to those inherited from *Property*):

- **cast** - convert input value to boolean based on its truth value. By default, cast is False.

Integer

class `properties.`**Integer**(*doc*, ***kwargs*)

Property for integer values

Available keywords (in addition to those inherited from *Property*):

- **min** - Minimum valid value, inclusive. If None (the default), there is no minimum limit.

- **max** - Maximum valid value, inclusive. If None (the default), there is no maximum limit.
- **cast** - Attempt to convert input value to integer. By default, cast is False.

Float

class properties.**Float**(*doc*, ***kwargs*)
Property for float values

Available keywords (in addition to those inherited from *Property*):

- **min** - Minimum valid value, inclusive. If None (the default), there is no minimum limit.
- **max** - Maximum valid value, inclusive. If None (the default), there is no maximum limit.
- **cast** - Attempt to convert input value to integer. By default, cast is False.

Complex

class properties.**Complex**(*doc*, ***kwargs*)
Property for complex numbers

Available keywords (in addition to those inherited from *Property*):

- **cast** - Attempt to convert input value to integer. By default, cast is False.

String

class properties.**String**(*doc*, ***kwargs*)
Property for string values

Available keywords (in addition to those inherited from *Property*):

- **strip** - Substring to strip off input. By default, nothing is stripped.
- **change_case** - If 'lower', coerces input to lowercase; if 'upper', coerce input to uppercase. If None (the default), case is left unchanged.
- **unicode** - If True, coerce strings to unicode. Default is True to ensure consistent behavior across Python 2/3.
- **regex** - Regular expression (pattern or compiled expression) the input string must match. Note: `re.search` is used to determine if string is valid; to match the entire string, ensure '^' and '$' are contained in the regex pattern.

Math Properties

Note: Math Properties require `numpy` and `vectormath` to be installed. This may be installed with `pip install properties[full]`, `pip install properties[math]`, or `pip install numpy vectormath`.

Array

class properties.**Array**(*doc*, ***kwargs*)
Property for numpy arrays

Available keywords (in addition to those inherited from *Property*):

- **shape** - Tuple (or set of valid tuples) that describes the allowed shape of the array. Length of shape tuple corresponds to number of dimensions; values correspond to the allowed length for each dimension. These values may be integers or '*' for any length. For example, an n x 3 array would be shape ('*', 3). None may also be used if any shape is valid. The default value is ('*',).
- **dtype** - Allowed data type for the array. May be float, int, bool, or a tuple containing any of these. The default is (float, int).

Vector3

class properties.**Vector3**(*doc*, ***kwargs*)
Property for 3D vectors

These Vectors are of shape (3,) and dtype float. In addition to length-3 arrays, these properties accept strings including: 'zero', 'x', 'y', 'z', '-x', '-y', '-z', 'east', 'west', 'north', 'south', 'up', and 'down'.

Available keywords (in addition to those inherited from *Property*):

- **length** - On validation, vectors are scaled to this length. If None (the default), vectors are not scaled

Vector2

class properties.**Vector2**(*doc*, ***kwargs*)
Property for 2D vectors

These Vectors are of shape (2,) and dtype float. In addition to length-2 arrays, these properties accept strings including: 'zero', 'x', 'y', '-x', '-y', 'east', 'west', 'north', and 'south'.

Available keywords (in addition to those inherited from *Property*):

- **length** - On validation, vectors are scaled to this length. If None (the default), vectors are not scaled

Vector3Array

class properties.**Vector3Array**(*doc*, ***kwargs*)
Property for an array of 3D vectors

This array of vectors are of shape ('*', 3) and dtype float. In addition to an array of this shape, these properties accept a list of strings including: 'zero', 'x', 'y', 'z', '-x', '-y', '-z', 'east', 'west', 'north', 'south', 'up', and 'down'.

Available keywords (in addition to those inherited from *Property*):

- **length** - On validation, all vectors are scaled to this length. If None (the default), vectors are not scaled

Vector2Array

class properties.**Vector2Array**(*doc*, ***kwargs*)
Property for an `array of 2D vectors`

This array of vectors are of shape ('*', 2) and dtype float. In addition to an array of this shape, these properties accept a list of strings including: 'zero', 'x', 'y', '-x', '-y', 'east', 'west', 'north', and 'south'.

Available keywords (in addition to those inherited from *Property*):

- **length** - On validation, all vectors are scaled to this length. If None (the default), vectors are not scaled

Image Properties

Note: Image Properties require `pypng` to be installed. This may be installed with `pip install properties[full]`, `pip install properties[image]`, or `pip install pypng`.

ImagePNG

class properties.**ImagePNG**(*doc*, *mode='rb'*, ***kwargs*)
Property for PNG images

Available keywords (in addition to those inherited from *Property*):

- **mode**: Opens the file in this mode. Must be a binary mode that supports file reading. Default value is 'rb'.
- **valid_modes**: Tuple of valid modes for open files. This must include **mode**. If nothing is specified, **valid_mode** is set to **mode**.
- **filename** - Name associated with open copy of PNG image. Default is 'texture.png'.

Other Property Types

StringChoice

class properties.**StringChoice**(*doc*, *choices*, *case_sensitive=False*, ***kwargs*)
String Property where only certain choices are allowed

Available keywords (in addition to those inherited from *Property*):

- **choices** - Either a set/list/tuple of allowed strings OR a dictionary of string key and list-of-string value pairs, where any string in the value list is coerced to the key string.
- **case_sensitive** - Determine if input must follow case in choices. If False (the default), the input value will be coerced to the case in choices.
- **descriptions** - Dictionary of choice/description key/value pairs. If specified, it must contain all choices.

Color

class properties.**Color**(*doc*, ***kwargs*)
Property for RGB colors.

Valid inputs are length-3 RGB tuple/list with integer values between 0 and 255, 3 or 6 digit hex color, color name from standard web colors, or 'random'. All of these are coerced to RGB tuple.

No additional keywords are avalaible besides those those inherited from *Property*.

DateTime

class properties.**DateTime**(*doc*, ***kwargs*)

Property for DateTimes

This property uses `datetime.datetime`. The value may also be specified as a string that uses either '1995/08/12' or '1995-08-12T18:00:00Z' format; these are coerced to a datetime instance.

No additional keywords are avalaible besides those those inherited from *Property*.

File

class properties.**File**(*doc*, *mode=None*, ***kwargs*)

Property for files

This may be a file or file-like object. If mode is provided, filenames are also allowed; these will be opened on validate. Note: Validation rejects closed files, but nothing prevents the file from being modified or closed once it is set.

Available keywords (in addition to those inherited from *Property*):

- **mode**: Opens the file in this mode. If 'r' or 'rb', the file must exist, otherwise the file will be created. If None, string filenames will not be open (and therefore be invalid). Default value is None.
- **valid_modes**: Tuple of valid modes for open files. This must include **mode**. If nothing is specified, **valid_mode** is set to **mode**.

Instance Property

class properties.**Instance**(*doc*, *instance_class*, ***kwargs*)

Property for instances of a specified class

Instance Properties may be used for any type, but they gain additional power with *HasProperties* types. The **Instance** Property may be assigned a dictionary with valid HasProperties class keywords; this is coerced to an instance of the HasProperties class. Also, HasProperties methods behave recursively, so if the parent HasProperties class is validated, serialized, etc., then HasProperties **Instance** Properties on the class will also be validated, serialized, etc.

Available keywords (in addition to those inherited from *Property*):

- **instance_class** - The allowed class for the property.
- **auto_create** - DEPRECATED - set default to the instance_class instead. If True, this Property is instantiated by default. This is equivalent to setting the default keyword to the instance_class. If False, the default value is undefined. Note: auto_create passes no arguments, so it cannot be True if the instance_class requires arguments.

Container Properties

Tuple

class properties.**Tuple**(*doc*, *prop=None*, ***kwargs*)
Property for tuples, where each entry is another Property type

Available keywords (in addition to those inherited from *Property*):

- **prop** - Property instance that specifies the Property type of each entry in the **Tuple**. A HasProperties class may also be specified; this is simply coerced to an *Instance Property* of that class.
- **min_length** - Minimum valid length of the tuple, inclusive. If None (the default), there is no minimum length.
- **max_length** - Maximum valid length of the tuple, inclusive. If None (the default), there is no maximum length.
- **coerce** - If False, input must be a tuple. If True, container types are coerced to a tuple and other non-container values become a length-1 tuple. Default value is False.

List

class properties.**List**(*doc*, *prop=None*, ***kwargs*)
Property for lists, where each entry is another Property type

Available keywords (in addition to those inherited from *Property*):

- **prop** - Property instance that specifies the Property type of each entry in the **List**. A HasProperties class may also be specified; this is simply coerced to an *Instance Property* of that class.
- **min_length** - Minimum valid length of the list, inclusive. If None (the default), there is no minimum length.
- **max_length** - Maximum valid length of the list, inclusive. If None (the default), there is no maximum length.
- **coerce** - If False, input must be a list. If True, container types are coerced to a list and other non-container values become a length-1 list. Default value is False.
- **observe_mutations** - If False, the underlying storage class is a `list` (or subclass thereof). If True, the underlying storage class will be an *observable_copy* of the list. The benefit of observing mutations is that all mutations and operations will trigger HasProperties change notifications. The drawback is slower performance as copies of the list are made on every operation.

Set

class properties.**Set**(*doc*, *prop=None*, ***kwargs*)
Property for sets, where each entry is another Property type

Available keywords (in addition to those inherited from *Property*):

- **prop** - Property instance that specifies the Property type of each entry in the **Set**. A HasProperties class may also be specified; this is simply coerced to an *Instance Property* of that class.
- **min_length** - Minimum valid length of the set, inclusive. If None (the default), there is no minimum length.

- **max_length** - Maximum valid length of the set, inclusive. If None (the default), there is no maximum length.
- **coerce** - If False, input must be a set. If True, container types are coerced to a set and other non-container values become a length-1 set. Default value is False.
- **observe_mutations** - If False, the underlying storage class is a `set` (or subclass thereof). If True, the underlying storage class will be an *observable_copy* of the set. The benefit of observing mutations is that all mutations and operations will trigger HasProperties change notifications. The drawback is slower performance as copies of the set are made on every operation.

Dictionary

class properties.**Dictionary**(*doc*, ***kwargs*)

Property for dicts, where each key and value is another Property type

Available keywords (in addition to those inherited from *Property*):

- **key_prop** - Property instance that specifies the Property type of each key in the **Dictionary**. A HasProperties class may also be specified; this is simply coerced to an *Instance Property* of that class.
- **value_prop** - Property instance that specifies the Property type of each value in the **Dictionary**. A HasProperties class may also be specified; this is simply coerced to an *Instance Property* of that class.
- **observe_mutations** - If False, the underlying storage class is a `dict` (or subclass thereof). If True, the underlying storage class will be an *observable_copy* of the dict. The benefit of observing mutations is that all mutations and operations will trigger HasProperties change notifications. The drawback is slower performance as copies of the dict are made on every operation.

Observable Container Creation

properties.base.containers.**observable_copy**(*value*, *name*, *instance*)

Return an observable container for HasProperties notifications

This method creates a new container class to allow HasProperties instances to `observe_mutations`. It returns a copy of the input value as this new class.

The output class behaves identically to the input value's original class, except when it is used as a property on a HasProperties instance. In that case, it notifies the HasProperties instance of any mutations or operations.

Union Property

class properties.**Union**(*doc*, *props*, ***kwargs*)

Property with multiple valid Property types

Union Properties contain a list of *Property* instances. Validation, serialization, etc. cycle through the corresponding method on the each Property instance sequentially until one succeeds. If all Property types raise an error, the Union Property will also raise an error.

Note: When specifying Property types, the order matters; if multiple types are valid, the earlier type will be favored. For example,

```
import properties
union_0 = properties.Union(
    doc='String and Color',
```

(continues on next page)

(continued from previous page)

```
    props=(properties.String(''), properties.Color('')),
)
union_1 = properties.Union(
    doc='String and Color',
    props=(properties.Color(''), properties.String('')),
)

union_0.validate(None, 'red') == 'red'  # Validates to string
union_1.validate(None, 'red') == (255, 0, 0)  # Validates to color
```

Available keywords (in addition to those inherited from *Property*):

- **props** - A list of Property instances that each specify a valid type for the Union Property. HasProperties classes may also be specified; these are coerced to Instance Properties of the respective class.

Gettable Property

class properties.**GettableProperty**(*doc*, ***kwargs*)

Property with immutable value

GettableProperties are assigned their default values upon *HasProperties* instance construction, and cannot be modified after that.

Keyword arguments match those available to *Property* with the exception of **required**.

UUID

class properties.**Uuid**(*doc*, ***kwargs*)

Immutable property for unique identifiers

Default value is generated on *HasProperties* class instantiation using `uuid.uuid4()`

No additional keywords are available besides those those inherited from *GettableProperty*.

Dynamic Property

class properties.basic.**DynamicProperty**(*doc*, *func*, *prop*, ***kwargs*)

DynamicProperties are GettableProperties calculated dynamically

These allow for a similar behavior to `@property` with additional documentation and validation built in. DynamicProperties are not saved to the HasProperties instance (and therefore are not serialized), do not fire change notifications, and don't allow default values.

These are created by decorating a single-argument method with a Property instance. This method is registered as the DynamicProperty getter. Setters and deleters may also be registered.

```
import properties
class SpatialInfo(properties.HasProperties):
    x = properties.Float('x-location')
    y = properties.Float('y-location')
    z = properties.Float('z-location')

    @properties.Vector3('my dynamic vector')
    def location(self):
```

(continues on next page)

(continued from previous page)

```
        return [self.x, self.y, self.z]

    @location.setter
    def location(self, value):
        self.x, self.y, self.z = value

    @location.deleter
    def location(self):
        del self.x, self.y, self.z
```

Note: DynamicProperties should not be directly instantiated; they should be constructed with the above decorator method.

Note: Since DynamicProperties have no saved state, the decorating Property is not allowed to have a `default` value. Also, the `required` attribute will be ignored.

Note: When implementing a DynamicProperty getter, care should be taken around when other properties do not yet have a value. In the example above, if `self.x`, `self.y`, or `self.z` is still `None` the `location` vector will be invalid, so calling `self.location` will fail. However, if the getter method returns `None` it will be treated as `properties.undefined` and pass validation.

deleter (*func*)
Register a delete function for the DynamicProperty

This function may only take one argument, self.

setter (*func*)
Register a set function for the DynamicProperty

This function must take two arguments, self and the new value. Input value to the function is validated with prop validation prior to execution.

Renamed Property

class properties.**Renamed**(*new_name*, ***kwargs*)
Property that allows renaming of other properties.

Assign the old name to a Renamed Property that points to the new name. Getting, setting, and deleting using the old name will warn the user then redirect to the new name.

For example, when updating this code for PEP8

```
class MyClass(properties.HasProperties):
    myStringProp = properties.String('My string property')
```

backwards compatibility can be maintained with

```
class MyClass(properties.HasProperties):
    my_string_prop = properties.String('My string property')
    myStringProp = properties.Renamed('my_string_prop')
```

Argument:

- **new_name** - the new name of the property that was renamed.

Available keywords:

- **warn** - raise a warning when this property is used (default: True)

8.1.4 Utilities

class properties.utils.**Sentinel**(*name*, *doc*)
Basic object with name and doc for specifying singletons

Avalable Sentinels:

- properties.undefined - The default value for all Properties if no other default is specified. When an undefined property is accessed, it returns None. Properties that are required must be set to something other than undefined.
- properties.everything - Sentinel representing all available properties. This is used when specifying observed properties.

properties.**filter_props**(*has_props_cls*, *input_dict*, *include_immutable=True*)
Split a dictionary based keys that correspond to Properties

Returns: **(props_dict, others_dict)** - Tuple of two dictionaries. The first contains key/value pairs from the input dictionary that correspond to the Properties of the input HasProperties class. The second contains the remaining key/value pairs.

Parameters:

- **has_props_cls** - HasProperties class or instance used to filter the dictionary
- **input_dict** - Dictionary to filter
- **include_immutable** - If True (the default), immutable properties (i.e. Properties that inherit from GettableProperty but not Property) are included in props_dict. If False, immutable properties are excluded from props_dict.

For example

```
class Profile(properties.HasProperties):
    name = properties.String('First and last name')
    age = properties.Integer('Age, years')

bio_dict = {
    'name': 'Bill',
    'age': 65,
    'hometown': 'Bakersfield',
    'email': 'bill@gmail.com',
}

(props, others) = properties.filter_props(Profile, bio_dict)
assert set(props) == {'name', 'age'}
assert set(others) == {'hometown', 'email'}
```

class properties.**ValidationError**(*message*, *reason=None*, *prop=None*, *instance=None*, *_error_tuples=None*)
Exception type to be raised during property validation

Parameters

- **message** - Detailed description of the error cause

- **reason** - Short reason for the error
- **prop** - Name of property related to the error
- **instance** - HasProperties instance related to the error

These inputs are stored as a tuple and passed to the `instance._error_hook` method, which may be overridden on the HasProperties class for custom error behavior.

class properties.**SelfReferenceError**
Exception type to be raised with infinite recursion problems

8.1.5 Extra Properties Implementations

These HasProperties and Property implementations are available by importing `properties.extras`.

- *UID-Related Extras* - `HasUID` class for HasProperties instances with unique IDs and `Pointer` property to refer to instances by unique ID.
- *Web-Related Extras* - Web-related Property classes
- *Singleton* - HasProperties class that creates only one instance for a given identifying name. Any instances with that name will be the same instance.
- *Task* - Callable HasProperties class that may be subclassed and used as a computational task.

UID-Related Extras

class properties.extras.**HasUID**(***kwargs*)
HasUID is a HasProperties class that includes unique ID

Adding a UID to HasProperties allows serialization of more complex structures, including recursive self-references. They are serialized to a flat dictionary of UID/HasUID key/value pairs.

Required Properties:

- **uid** (*String*): Unique identifier, a unicode string, Default: new instance of str

classmethod deserialize(*value*, *trusted=False*, *strict=False*, *assert_valid=False*, ***kwargs*)
Deserialize nested HasUID instance from flat pointer dictionary

Parameters

- **value** - Flat pointer dictionary produced by `serialize` with UID/HasUID key/value pairs. It also includes a `__root__` key to specify the root HasUID instance.
- **trusted** - If True (and if the input dictionaries have `'__class__'` keyword and this class is in the registry), the new **HasProperties** class will come from the dictionary. If False (the default), only the **HasProperties** class this method is called on will be constructed.
- **strict** - Requires `'__class__'`, if present on the input dictionary, to match the deserialized instance's class. Also disallows unused properties in the input dictionary. Default is False.
- **assert_valid** - Require deserialized instance to be valid. Default is False.
- You may also specify an alternative **root** - This allows a different HasUID root instance to be specified. It overrides `__root__` in the input dictionary.
- Any other keyword arguments will be passed through to the Property deserializers.

Note: HasUID instances are constructed with no input arguments (ie `cls()` is called). This means deserialization will fail if the init method has been overridden to require input parameters.

classmethod load(*uid*)
Load an instance given a UID

This is used by Pointer properties to retrieve instances from UIDs.

serialize(*include_class=True*, *save_dynamic=False*, ***kwargs*)
Serialize nested HasUID instances to a flat dictionary

Parameters:

- **include_class** - If True (the default), the name of the class will also be saved to the serialized dictionary under key `'__class__'`
- **save_dynamic** - If True, dynamic properties are written to the serialized dict (default: False).
- You may also specify a **registry** - This is the flat dictionary where UID/HasUID pairs are stored. By default, no registry need be provided; a new dictionary will be created.
- Any other keyword arguments will be passed through to the Property serializers.

classmethod validate_uid(*uid*)
Assert if a given UID is valid

This is used by Pointer properties to validate a UID without necessarily loading the corresponding instance.

class properties.extras.**Pointer**(*doc*, *instance_class*, ***kwargs*)
Property for HasUID instances where string UID pointer may be used

Available keywords (in addition to those inherited from *Instance*):

- **load** - Attempt to load instances from UID on validation If True, when the Pointer property is assigned a valid UID, it will then attempt to call `self.instance_class.load(uid)` If this method is defined, it must return a valid instance which will replace the UID as the Pointer value. If this method is not defined or if it returns None, the Pointer property maintains the UID value. Default is False, meaning there is no attempt to load the instance.
- **uid_prop** - Property or attribute name of the UID property on instance_class. The default is 'uid'.

Web-Related Extras

class properties.extras.**URL**(*doc*, ***kwargs*)
String property that only accepts valid URLs

This property type uses `urllib.parse` to validate input URLs and possibly remove fragments and query params.

Available keywords (in addition to those inherited from *String*):

- **remove_parameters** - Query params are stripped from input URL (default is False).
- **remove_fragment** - Fragment is stripped from input URL (default is False).

Singleton

class properties.extras.**Singleton**(*name*, ***kwargs*)
Class that only allows one instance for each identifying name

These instances are stored on the `_SINGLETONS` attribute of the class. You may create a new registry of singletons by redefining this attribute on a subclass. Also, this means multiple singleton classes may be present on a registry, therefore the class you use to access the singleton may not be the class of the returned singleton.

Each singleton must be initialized with a name. You can type-check and validate this value by including a 'name' property on your class. The identifying name does not change during the lifetime of the singleton, even if the 'name' value is changed.

classmethod deserialize(*value*, *trusted=False*, *strict=False*, *assert_valid=False*, ***kwargs*)
Create a Singleton instance from a serialized dictionary.

This behaves identically to HasProperties.deserialize, except if the singleton is already found in the singleton registry the existing value is used.

Note: If property values differ from the existing singleton and the input dictionary, the new values from the input dictionary will be ignored

---

serialize(*include_class=True*, *save_dynamic=False*, ***kwargs*)
Serialize Singleton instance to a dictionary.

This behaves identically to HasProperties.serialize, except it also saves the identifying name in the dictionary as well.

class properties.extras.singleton.**SingletonMetaclass**
Metaclass to produce singleton behavior using a singleton registry

Task

class properties.extras.**BaseTask**
Class for defining a computational task

Input and Output class attributes must be subclasses of BaseInput and BaseOutput respectively. Task is executed by calling an instance of the task with Input property/value pairs as keyword arguments.

__call__(***kwargs*)
Execute the task

Keyword arguments are used to construct Input instance. This is validated and passed to `run`. The Output of `run` is validated, passed to `process_output`, and returned.

process_output(*output_obj*)
Processes valid Output object into desired task output

This method is executed during `__call__` on the output of `run`.

By default, this serializes the output to a dictionary.

report_status(*status*)
Hook for reporting the task status towards completion

run(*input_obj*)
Execution logic for the task

This method must be overridden in Task subclasses

To run a Task, create an instance of the Task, then call the instance with the required input parameters. This will construct and validate an Input object.

`run` receives this validated Input object. It then must process the inputs and return an Output object.

class properties.extras.**BaseInput**(***kwargs*)
HasProperties object with input parameters for a computation

class properties.extras.**BaseOutput**(***kwargs*)
HasProperties object with the result of a computation

Required Properties:

- **log** (*String*): Output log messages from the task, a unicode string
- **success** (*Boolean*): Did the task succeed, a boolean, Default: True

class properties.extras.**TaskStatus**(***kwargs*)
HasProperties object to indicate present status of the task

Optional Properties:

- **message** (*String*): Task progress message, a unicode string
- **progress** (*Float*): Task progress to completion, a float in range [0, 1]

class properties.extras.**TaskException**
An exception related to a computational task

class properties.extras.**PermanentTaskFailure**
An exception indicating Task should not be retried

class properties.extras.**TemporaryTaskFailure**
An exception indicating Task should be retried

CHAPTER 9

Indices and tables

- genindex

Index

Symbols

__call__() (properties.extras.BaseTask method), 40

A

Array (class in properties), 30
assert_valid() (properties.Property method), 26

B

BaseInput (class in properties.extras), 40
BaseOutput (class in properties.extras), 41
BaseTask (class in properties.extras), 40
Boolean (class in properties), 28

C

Color (class in properties), 31
Complex (class in properties), 29
copy() (in module properties), 20

D

DateTime (class in properties), 32
deleter() (properties.basic.DynamicProperty method), 36
deserialize() (properties.extras.HasUID class method), 38
deserialize() (properties.extras.Singleton class method), 40
deserialize() (properties.HasProperties class method), 19
deserialize() (properties.Property method), 26
Dictionary (class in properties), 34
directional_link (class in properties), 23
DynamicProperty (class in properties.basic), 35

E

equal() (in module properties), 20
equal() (properties.Property method), 26
error() (properties.Property method), 26

F

File (class in properties), 32
filter_props() (in module properties), 37
Float (class in properties), 29
from_json() (properties.Property static method), 26

G

GettableProperty (class in properties), 35

H

HasProperties (class in properties), 18
HasUID (class in properties.extras), 38

I

ImagePNG (class in properties), 31
Instance (class in properties), 32
Integer (class in properties), 28

L

link (class in properties), 24
List (class in properties), 33
listeners_disabled (class in properties), 23
load() (properties.extras.HasUID class method), 39

M

meta (properties.Property attribute), 26

O

observable_copy() (in module properties.base.containers), 34
observer() (in module properties), 22
observers_disabled (class in properties), 23

P

PermanentTaskFailure (class in properties.extras), 41
Pointer (class in properties.extras), 39
process_output() (properties.extras.BaseTask method), 40
Property (class in properties), 26
PropertyMetaclass (class in properties.base), 19

R

relink() (properties.directional_link method), 24

relink() (properties.link method), 24
Renamed (class in properties), 36
report_status() (properties.extras.BaseTask method), 40
run() (properties.extras.BaseTask method), 40

S

SelfReferenceError (class in properties), 38
Sentinel (class in properties.utils), 37
serialize() (properties.extras.HasUID method), 39
serialize() (properties.extras.Singleton method), 40
serialize() (properties.HasProperties method), 19
serialize() (properties.Property method), 27
Set (class in properties), 33
setter() (properties.basic.DynamicProperty method), 36
Singleton (class in properties.extras), 39
SingletonMetaclass (class in properties.extras.singleton), 40
String (class in properties), 29
StringChoice (class in properties), 31

T

tag() (properties.Property method), 27
TaskException (class in properties.extras), 41
TaskStatus (class in properties.extras), 41
TemporaryTaskFailure (class in properties.extras), 41
to_json() (properties.Property static method), 27
Tuple (class in properties), 33

U

Union (class in properties), 34
unlink() (properties.directional_link method), 24
unlink() (properties.link method), 24
URL (class in properties.extras), 39
Uuid (class in properties), 35

V

validate() (properties.HasProperties method), 19
validate() (properties.Property method), 27
validate_uid() (properties.extras.HasUID class method), 39
ValidationError (class in properties), 37
validator() (in module properties), 21
validators_disabled (class in properties), 22
Vector2 (class in properties), 30
Vector2Array (class in properties), 31
Vector3 (class in properties), 30
Vector3Array (class in properties), 30