any2any Documentation

Release 0.1

Sebastien Piquemal

January 14, 2014

CHAPTER 1

Documentation summary

1.1 *any2any.base*

1.1.1 Casts

class any2any.base.**Cast**(***settings*)

Base class for all casts. This class is virtual, and all subclasses must implement `Cast.call()`.

Available Settings :

from_
type. The type to cast from. If not given, the type of the input is used.

to
type. The type to cast to.

mm_to_cast
dict. `{<mm>: <cast>}`. A dictionary mapping a metamorphosis to a cast instance.

extra_mm_to_cast
dict. `{<mm>: <cast>}`. Overrides `mm_to_cast` as a dictionary update.

from_wrapped
type. A subclass of `any2any.utils.WrappedObject`. If provided, will cause `from_` to be automatically wrapped.

to_wrapped
type. A subclass of `any2any.utils.WrappedObject`. If provided, will cause `to` to be automatically wrapped.

logs
bool. If True, the cast writes debug informations to the logger.

Members :

call(*inpt*)
Virtual method. Casts *inpt*.

cast_for(*mm*)
Picks in `mm_to_cast` a cast suitable for *mm*, customizes it with calling cast's settings, and finally returns it.

set_debug_on()
Set debug mode: all debug logs will be printed on stderr.

set_debug_off()
Toggle debug mode off.

class any2any.base.**CastStack**(**settings*)
A cast provided for convenience. *CastStack* doesn't do anything else than looking for a suitable cast with `Cast.cast_for()` and calling it. It is therefore very useful for just stacking a bunch of casts, and then casting different types of input. For example :

```
>>> cast = CastStack(mm_to_cast={
...     Mm(from_any=int): my_int_cast,
...     Mm(from_any=str): my_str_cast,
... })
>>> cast('a string')
'other string'
>>> cast(1234)
12345
```

call(*inpt*, *from_=None*, *to=None*) **= <unbound method CastStack.call>**

1.1.2 Settings

class any2any.base.**Setting**(*default=None*)
Base class for all setting types.

customize(*cast*, *value*)
This method handles customization of the setting's value.

get(*cast*)
Gets and returns the setting's value from *cast*.

inherits(*setting*)
When overriding a setting, this is called on the new setting to take into account the parent setting.

init(*cast*, *value*)
This method handles initialization of the setting's value.

set(*cast*, *value*)
Sets the setting's value on *cast*.

1.2 *any2any.utils*

1.2.1 Metamorphoses (Mm)

class any2any.utils.**Mm**(*from_=None*, *to=None*, *from_any=None*, *to_any=None*)
A metamorphosis between two types. For example :

```
>>> mm1 = Mm(LoisClark, Superman)
>>> mm2 = Mm(from_any=Human, to_any=SuperHero)
>>> mm1 < mm2 # i.e. <mm1> is included in <mm2>
True
```

Kwargs:

- from_(type). Metamorphosis only from type *from_* (and no subclass).
- to(type). Metamorphosis only to type *to* (and no subclass).

- from_any(type). Metamorphosis from type *from_any* and subclasses.
- to_any(type). Metamorphosis from type *to_any* and subclasses.

1.2.2 WrappedObject

class any2any.utils.**WrappedObject**

Subclass *WrappedObject* to create a placeholder containing extra-information on a type. e.g. :

```
>>> class WrappedInt(WrappedObject):
...
...     klass = int
...     greater_than = 0
...
```

A subclass of *WrappedObject* can also provide informations on the wrapped type's instances' :

•attribute schema - `default_schema()`

•attribute access - `setattr()` and `getattr()`

•creation of new instances - `new()`

klass

type. The wrapped type.

alias of `object`

factory = None

type. The type used for instance creation :

```
>>> class WrappedBS(WrappedObject):
...     klass = basestring
...     factory = str
...
>>> a_str = WrappedBS("blabla")
>>> type(a_str) == str
True
```

superclasses = ()

tuple. Allows to customize `WrappedObject.issubclass()` behaviour :

```
>>> class WrappedStr(WrappedObject):
...     klass=str
...     superclasses=(MyStr, AllStrings)
...
>>> WrappedObject.issubclass(WrappedStr, str), WrappedObject.issubclass(WrappedStr, MyStr),
(True, True)
```

extra_schema = {}

dict. `{<attribute_name>: <attribute_type>}`. Allows to update the default schema, see `get_schema()`.

include = []

list. The list of attributes to include in the schema see, `get_schema()`.

exclude = []

list. The list of attributes to exclude from the schema see, `get_schema()`.

classmethod get_class (*key*)

Returns the class of attribute *key*, as found from the schema, see `get_schema()`.

classmethod get_schema()
Returns the full schema {<attribute_name>: <attribute_type>} of an instance, taking into account (respectively) : *default_schema*, *extra_schema*, *include* and *exclude*.

classmethod default_schema()
Returns the schema - known a priori - of an instance. Must return a dictionary with the format {<attribute_name>: <attribute_type>}.

classmethod setattr(*instance*, *name*, *value*)
Sets the attribute *name* on *instance*, with value *value*. If the calling WrappedObject has a method *set_<name>*, this method will be used to set the attribute.

classmethod getattr(*instance*, *name*)
Gets the attribute *name* from *instance*. If the calling WrappedObject has a method *get_<name>*, this method will be used to get the attribute.

classmethod new(**args*, ***kwargs*)
Creates and returns a new instance of the wrapped type.

1.3 *any2any.daccasts*

1.3.1 DivideAndConquerCast

class any2any.daccasts.**DivideAndConquerCast**(***settings*)
Abstract base cast for metamorphosing *from* and *to* any complex object or container.

In order to achieve casting, this class implements a "divide and conquer" strategy :

1. *Divide into sub-problems* - iter_input()
2. *Solve sub-problems* - iter_output()
3. *Combine solutions* - build_output()

Members

Member-order bysource

1.3.2 Mixins for DivideAndConquerCast

class any2any.daccasts.**CastItems**
Mixin for DivideAndConquerCast. Implements DivideAndConquerCast.iter_output().

Available settings :

key_to_cast
dict. {<key>: <cast>}. Maps a key with the cast to use.

value_cast
Cast. The cast to use on all values.

key_cast
Cast. The cast to use on all keys.

Members :

iter_output(*items_iter*)
Casts each item. The cast is looked-up for in the following order :

1.setting `key_to_cast`

2.setting `value_cast`

3.finally, using `any2any.base.Cast.cast_for()`

strip_item(*key*, *value*)
Override for use. If *True* is returned, the item `<key>, <value>` will be stripped from the output.

class any2any.daccasts.**FromIterable**
Mixin for `DivideAndConquerCast`. Implements `DivideAndConquerCast.iter_input()`.

Note that *FromIterable* is more clever when *from_* is a subclass of `WrappedContainer`.

class any2any.daccasts.**ToIterable**
Mixin for `DivideAndConquerCast`. Implements `DivideAndConquerCast.build_output()`.

Note that *ToIterable* is more clever when *to* is a subclass of `WrappedContainer`.

class any2any.daccasts.**FromMapping**
Mixin for `DivideAndConquerCast`. Implements `DivideAndConquerCast.iter_input()`.

Note that *FromMapping* is more clever when *from_* is a subclass of `WrappedContainer`.

class any2any.daccasts.**ToMapping**
Mixin for `DivideAndConquerCast`. Implements `DivideAndConquerCast.build_output()`.

Note that *ToMapping* is more clever when *to* is a subclass of `WrappedContainer`.

class any2any.daccasts.**FromObject**
Mixin for `DivideAndConquerCast`. Implements `DivideAndConquerCast.iter_input()`.

Note that *FromObject* is more clever when *from_* is a subclass of `WrappedObject`.

class any2any.daccasts.**ToObject**
Mixin for `DivideAndConquerCast`. Implements `DivideAndConquerCast.build_output()`.

Note that *ToObject* is more clever when *to* is a subclass of `WrappedObject`.

1.3.3 Wraps

class any2any.daccasts.**WrappedContainer**
Bases: `any2any.utils.WrappedObject`

A subclass of `utils.WrappedObject` providing informations on a container type.

value_type = NotImplemented
type. The type of value contained.