namedspace Documentation

Release 1.2.1

Warren A. Smith

June 21, 2015

Contents

Contents:

CHAPTER 1

namedspace Module

class namedspace.**NamedspaceMeta**

Bases: type

Metaclass for namedspace classes

namedspace.**namedspace**(*typename*, *required_fields=()*, *optional_fields=()*, *mutable_fields=()*, *default_values=<frozendict {}>*, *default_value_factories=<frozendict {}>*, *return_none=False*)

Builds a new class that encapsulates a namespace and provides various ways to access it.

The typename argument is required and is the name of the namedspace class that will be generated.

The required_fields and optional_fields arguments can be a string or sequence of strings and together specify the fields that instances of the namedspace class have.

Values for the required fields must be provided somehow when the instance is created. Values for optional fields may be provided later, or maybe not at all.

If an optional field is queried before its value has been set, an AttributeError will be raised. This behavior can be altered to cause None to be returned instead by setting the return_none keyword argument to True.

The mutable_fields argument specifies which fields will be mutable, if any. By default, all fields are immutable and all instances are hashable and can be used as dictionary keys. If any fields are set as mutable, all instances are not hashable and cannot be used as dictionary keys.

The default_values mapping provides simple default values for the fields.

The default_value_factories mapping provides a more flexible, but more complex, mechanism for providing default values. The value of each item is a callable that takes a single argument, the namedspace instance, and returns the default value for the field.

The default_values_factories mapping is only consulted if there is no default value for the field in the default_values mapping.

Here is a simple example, using only the required fields argument:

```
>>> SimpleNS = namedspace("SimpleNS", ("id", "name", "description"))
```

```
>>> SimpleNS
<class 'namedspace.SimpleNS'>
```

There are built-in properties to access collections and iterators associated with the namespace class.

```
>>> SimpleNS._field_names
('id', 'name', 'description')
```

```
>>> tuple(SimpleNS._field_names_iter)
('id', 'name', 'description')
```

Once the class has been created, it can be instantiated like any other class. However, a value for all of the required fields must be provided.

```
>>> simple_ns = SimpleNS(id=1, description="Simple Description")
Traceback (most recent call last):
    <snip/>
ValueError: A value for field 'name' is required.
```

```
>>> simple_ns = SimpleNS(id=1, name="Simple Name", description="Simple Description")
```

```
>>> simple_ns
SimpleNS(id=1, name='Simple Name', description='Simple Description')
```

An instance of a namedspace class provides standard attribute access to its fields.

```
>>> simple_ns.id
1
```

```
>>> simple_ns.name
'Simple Name'
```

```
>>> simple_ns.description
'Simple Description'
```

In addition to standard attribute access, instances of a namedspace class implement a MutableMapping interface.

```
>>> 'id' in simple_ns
True
```

```
>>> for field_name in simple_ns:
...     print field_name
id
name
description
```

```
>>> len(simple_ns)
3
```

```
>>> simple_ns["id"]
1
```

```
>>> simple_ns["name"]
'Simple Name'
```

```
>>> simple_ns["description"]
'Simple Description'
```

There are built-in properties to access collections and iterators associated with the namespace.

The namespace encapsulated by a namedspace class is stored in an OrderedDict, so order of the collections is the same as the order that the fields were specified.

All of these properties use the standard "non-public" naming convention in order to not pollute the public namespace.

```
>>> simple_ns._field_names
('id', 'name', 'description')
```

```
>>> tuple(simple_ns._field_names_iter)
('id', 'name', 'description')
```

```
>>> simple_ns._field_values
(1, 'Simple Name', 'Simple Description')
```

```
>>> tuple(simple_ns._field_values_iter)
(1, 'Simple Name', 'Simple Description')
```

```
>>> simple_ns._field_items
[('id', 1), ('name', 'Simple Name'), ('description', 'Simple Description')]
```

```
>>> list(simple_ns._field_items_iter)
[('id', 1), ('name', 'Simple Name'), ('description', 'Simple Description')]
```

```
>>> simple_ns._as_dict
OrderedDict([('id', 1), ('name', 'Simple Name'), ('description', 'Simple Description')])
```

Here is a more complex example, using most of the other arguments:

```
>>> from itertools import count
```

```
>>> ComplexNS = namedspace("ComplexNS", "id", optional_fields=("name", "description", "extra"),
...     mutable_fields=("description", "extra"), default_values={"description": "None available"
...     default_value_factories={"id": lambda self, counter=count(start=1): counter.next(),
...         "name": lambda self: "Name for id={id}".format(id=self.id)})
```

```
>>> complex_ns1 = ComplexNS()
```

```
>>> complex_ns1.id
1
```

The value of 1 was automatically assigned by the default_value_factory for the 'id' field, in this case a lambda closure that hooks up an instance of itertools.count.

```
>>> complex_ns1.name
'Name for id=1'
```

This value was also generated by a default value factory. In this case, the factory for the 'name' attribute uses the value of the 'id' attribute to compute the default value.

```
>>> complex_ns1.description
'None available'
```

This value came from the default_values mapping.

The description field was set as a mutable field, which allows it to be modified.

```
>>> complex_ns1.description = "Some fancy description"
>>> complex_ns1.description
'Some fancy description'
```

Its value can also be deleted.

```
>>> del complex_ns1.description
>>> complex_ns1.description
'None available'
```

Since its modified value was deleted, and it has a default value, it has reverted to its default value.

The extra field is a valid field in this namedspace, but it has not yet been assigned a value and does not have a default.

```
>>> complex_ns1.extra
Traceback (most recent call last):
    <snip/>
AttributeError: "Field 'extra' does not yet exist in this ComplexNS namedspace instance."
```

Sometimes, having an exception raised if an optional field is missing, and being forced to handle it, is annoying. A namedspace class can be configured at creation time to return None instead of raising exceptions for optional fields by setting the *return_none* parameter to *True*. Here is a trivial example:

```
>>> QuietNS = namedspace("QuietNS", optional_fields=("might_be_none",), return_none=True)
```

```
>>> quiet_ns1 = QuietNS(might_be_none="Nope, not this time")
>>> quiet_ns1.might_be_none
'Nope, not this time'
```

```
>>> quiet_ns2 = QuietNS()
>>> quiet_ns2.might_be_none
>>>
```

Having the namedspace quietly return *None* makes sense in some situations. But be careful. Understand the full implications of this alternate behavior on the code that uses it. Subtle data- dependent bugs can be introduced by this behavior, which is why it is not enabled by default.

Now, back to our "complex" example.

Since the 'extra' field is one of the mutable fields, we can give it a value.

```
>>> complex_ns1.extra = "Lasts a long, long time"
>>> complex_ns1.extra
'Lasts a long, long time'
```

Only fields that have been declared as either required or optional are allowed.

```
>>> complex_ns1.some_other_field = "some other value"
Traceback (most recent call last):
    <snip/>
FieldNameError: "Field 'some_other_field' does not exist in ComplexNS namedspace."
```

Finally, to illustrate that our counter is working as it should, if we instantiate another instance, our id field will get the next counter value.

```
>>> complex_ns2 = ComplexNS()
>>> complex_ns2.id
2
```

A common use case for a namedspace class is as a base class for another custom class that has additional members such as properties and methods. This way, the custom class gets all of the namedspace behavior through declarative configuration, instead of having to re-define that behavior imperatively.

The following is an example where one of the required fields is generated at instantiation time, and the values for the two optional fields are calculated values provided by properties in the subclass.

```
>>> from collections import Counter
>>> class Widget(namedspace("_Widget", ("mfg_code", "model_code", "serial_number"), optional_fie
...         return_none=True)):
...     _sn_map = Counter()
```

```
    ...     def __init__(self, *args, **kwargs):
    ...         sn_key = (kwargs["mfg_code"], kwargs["model_code"])
    ...         self._sn_map[sn_key] += 1
    ...         kwargs["serial_number"] = "{:010}".format(self._sn_map[sn_key])
    ...         super(Widget, self).__init__(*args, **kwargs)
    ...     @property
    ...     def sku(self):
    ...         return "{}_{}".format(self.mfg_code, self.model_code)
    ...     @property
    ...     def pk(self):
    ...         return "{}_{}".format(self.sku, self.serial_number)
    >>> widget1 = Widget(mfg_code="ACME", model_code="X-500")
    >>> widget1
    Widget(mfg_code='ACME', model_code='X-500', serial_number='0000000001', sku='ACME_X-500', pk='AC
    >>> widget1._as_dict
    OrderedDict([('mfg_code', 'ACME'), ('model_code', 'X-500'), ('serial_number', '0000000001'), ('s
    >>> widget2 = Widget(mfg_code="ACME", model_code="X-500")
    >>> widget2
    Widget(mfg_code='ACME', model_code='X-500', serial_number='0000000002', sku='ACME_X-500', pk='AC
    >>> widget2._as_dict
    OrderedDict([('mfg_code', 'ACME'), ('model_code', 'X-500'), ('serial_number', '0000000002'), ('s
```

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

n

N