PySynthetic Documentation

Release 0.4.8

Younes JAAIDI

April 11, 2013

CONTENTS

PySynthetic is a set of tools that aims to make writing Python classes shorter and "cleaner".

For instance, one can add properties and accessors *(getters/setters)* to a class with only one line of code *(using respectively* `synthesize_property` *and* `synthesize_member` *decorators)*, thus making the code more than 5 times shorter *(see* examples *)*. One can even avoid the laborious task of members initialization by using the `synthesize_constructor` decorator that takes care of writing the `__init__` method.

PySynthetic is also useful for applying strict type checking with no pain just by using the decorators' `contract` argument *(see* PyContracts *)*.

Help and ideas are appreciated! Thank you!

CHAPTER

ONE

RESOURCES

- Documentation
- Bug Tracker
- Code
- Mailing List <pysynthetic@googlegroups.com>

CHAPTER

TWO

EXAMPLES

2.1 Synthetic properties

With **PySynthetic**, the following code *(8 lines)*...

```
from synthetic import synthesize_constructor, synthesize_property

@synthesize_property('a', contract = int)
@synthesize_property('b', contract = list)
@synthesize_property('c', default = "", contract = str, read_only = True)
@synthesize_constructor()
class ShortAndClean(object):
    pass
```

... replaces this *(43 lines)*:

```
from contracts import contract

class ThisHurtsMyKeyboard(object):

    @contract
    def __init__(self, a, b, c = ""):
        """
    :type a: int
    :type b: list
    :type c: str
"""
        self._a = a
        self._b = b
        self._c = c

    @property
    def a(self):
        return self._a

    @a.setter
    @contract
    def a(self, value):
        """
    :type value: int
"""
        self._a = value

    @property
```

```
    def b(self):
        return self._b

    @b.setter
    @contract
    def b(self, value):
        """
    :type value: list
"""
        self._b = value

    @property
    def c(self):
        return self._c
```

2.2 Synthetic accessors

But, if you are more into accessors than properties, you can use `synthesize_member` decorator instead.

This way, the following code *(8 lines)*...

```
from synthetic import synthesize_constructor, synthesize_member

@synthesize_member('a', contract = int)
@synthesize_member('b', contract = list)
@synthesize_member('c', default = "", contract = str, read_only = True)
@synthesize_constructor()
class ShortAndClean(object):
    pass
```

...will replace this *(37 lines)*:

```
from contracts import contract

class ThisHurtsMyKeyboard(object):

    @contract
    def __init__(self, a, b, c = ""):
        """
    :type a: int
    :type b: list
    :type c: str
"""
        self._a = a
        self._b = b
        self._c = c

    def a(self):
        return self._a

    @contract
    def set_a(self, value):
        """
    :type value: int
"""
        self._a = value
```

```
    def b(self):
        return self._b

    @contract
    def set_b(self, value):
        """
    :type value: list
"""
        self._b = value

    def c(self):
        return self._c
```

CHAPTER

THREE

ADVANCED USAGE

3.1 Override synthesized member's accessors

One can override the synthesized member's accessors by simply explicitly writing the methods.

3.2 Override synthesized property

One can override the synthesized property by simply explicitly writing the properties.

Remark: For the moment, it's impossible to override the property's setter without overriding the getter.

3.3 Override synthesized constructor

One can use synthesized constructors to initialize members and properties values and still override it to implement some additional processing.

Example:

```
@synthesize_constructor()
@synthesize_property('value')
class Double:
    def __init__(self):
        self._value *= 2

print(Double(10).value)
```

Displays

```
20
```

The custom constructor can consume extra arguments *(not synthesized members or properties)*.

For more examples, see product's unit tests.

CHAPTER

FOUR

MODULE DOCUMENTATION

4.1 Underscore notation

synthetic.**naming_convention**(*naming_convention*)

When applied to a class, this decorator will override the underscore naming convention of all (previous and following) `synthesizeMember()` calls on the class to `naming_convention`.

Parameters **naming_convention** (*INamingConvention*) – The new naming convention.

synthetic.**synthesize_constructor**()

This class decorator will override the class's constructor by making it implicitly consume values for synthesized members and properties.

synthetic.**synthesize_member**(*member_name*, *default=None*, *contract=None*, *read_only=False*, *getter_name=None*, *setter_name=None*, *private_member_name=None*)

When applied to a class, this decorator adds getter/setter methods to it and overrides the constructor in order to set the default value of the member. By default, the getter will be named `member_name`. (Ex.: `member_name = 'member' => instance.member()`)

By default, the setter will be named `member_name` with 'set_' prepended it to it. (Ex.: `member_name = 'member' => instance.set_member(...)`)

By default, the private attribute containing the member's value will be named `member_name` with '_' prepended to it.

Naming convention can be overridden with a custom one using `naming_convention` decorator.

raises `DuplicateMemberNameError` when two synthetic members have the same name.

Parameters

- **read_only** (`bool`) – If set to `True`, the setter will not be added to the class.
- **default** (`*`) – Member's default value.
- **getter_name** (`str|None`) – Custom getter name. This can be useful when the member is a boolean. (Ex.: `is_alive`)
- **contract** (`*`) – Type constraint. See PyContracts
- **setter_name** (`str|None`) – Custom setter name.
- **member_name** (`str`) – Name of the member to synthesize.

- **private_member_name** (`str|None`) – Custom name for the private attribute that contains the member's value.

`synthetic.`**`synthesize_property`**(*property_name*, *default=None*, *contract=None*, *read_only=False*, *private_member_name=None*)

When applied to a class, this decorator adds a property to it and overrides the constructor in order to set the default value of the property.

IMPORTANT In order for this to work on python 2, you must use new objects that is to say that the class must inherit from object.

By default, the private attribute containing the property's value will be named `property_name` with '_' prepended to it.

Naming convention can be overridden with a custom one using `naming_convention` decorator.

raises `DuplicateMemberNameError` when two synthetic members have the same name.

raises `InvalidPropertyOverrideError` when there's already a member with that name and which is not a property.

Parameters

- **default** (*) – Property's default value.
- **read_only** (`bool`) – If set to `True`, the property will not a have a setter.
- **private_member_name** (`str|None`) – Custom name for the private attribute that contains the property's value.
- **contract** (*) – Type constraint. See PyContracts
- **property_name** (`str`) – Name of the property to synthesize.

4.2 CamelCase notation

Sorry Guido, but I like CamelCase.

`synthetic.`**`namingConvention`**(*namingConvention*)

When applied to a class, this decorator will override the CamelCase naming convention of all (previous and following) `synthesizeMember()` calls on the class to `namingConvention`.

Parameters **namingConvention** (*INamingConvention*) – The new naming convention.

`synthetic.`**`synthesizeConstructor`**()

This class decorator will override the class's constructor by making it implicitly consume values for synthesized members and properties.

`synthetic.`**`synthesizeMember`**(*memberName*, *default=None*, *contract=None*, *readOnly=False*, *getterName=None*, *setterName=None*, *privateMemberName=None*)

When applied to a class, this decorator adds getter/setter methods to it and overrides the constructor in order to set the default value of the member. By default, the getter will be named `memberName`. (Ex.: `memberName = 'member' => instance.member()`)

By default, the setter will be named `memberName` with the first letter capitalized and 'set' prepended it to it. (Ex.: `memberName = "member" => instance.setMember(...)`)

By default, the private attribute containing the member's value will be named `memberName` with '_' prepended to it.

Naming convention can be overridden with a custom one using `namingConvention` decorator.

raises `DuplicateMemberNameError` when two synthetic members have the same name.

Parameters

- **privateMemberName** (`str|None`) – Custom name for the private attribute that contains the member's value.
- **default** (`*`) – Member's default value.
- **memberName** (`str`) – Name of the member to synthesize.
- **contract** (`*`) – Type constraint. See PyContracts
- **readOnly** (`bool`) – If set to `True`, the setter will not be added to the class.
- **setterName** (`str|None`) – Custom setter name.
- **getterName** (`str|None`) – Custom getter name. This can be useful when the member is a boolean. (Ex.: `isAlive`)

`synthetic.`**`synthesizeProperty`**(*propertyName*, *default=None*, *contract=None*, *readOnly=False*, *privateMemberName=None*)

When applied to a class, this decorator adds a property to it and overrides the constructor in order to set the default value of the property.

IMPORTANT In order for this to work on python 2, you must use new objects that is to say that the class must inherit from object.

By default, the private attribute containing the property's value will be named `propertyName` with '_' prepended to it.

Naming convention can be overridden with a custom one using `namingConvention` decorator.

raises `DuplicateMemberNameError` when two synthetic members have the same name.

raises `InvalidPropertyOverrideError` when there's already a member with that name and which is not a property.

Parameters

- **default** (`*`) – Property's default value.
- **propertyName** (`str`) – Name of the property to synthesize.
- **readOnly** (`bool`) – If set to `True`, the property will not a have a setter.
- **contract** (`*`) – Type constraint. See PyContracts
- **privateMemberName** (`str|None`) – Custom name for the private attribute that contains the property's value.

CHAPTER

FIVE

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*