py3traits

Release 1.2.0

September 12, 2015

Contents

Contents:

CHAPTER 1

Python Traits

Project can be found from GitHub.

1.1 About Traits

Traits are classes which contain methods that can be used to extend other classes, similar to mixins, with exception that traits do not use inheritance. Instead, traits are composed into other classes. That is; methods, properties and internal state are copied to master object.

The point is to improve code reusability by dividing code into simple building blocks that can be then combined into actual classes.

There is also a wikipedia article about Traits.

1.2 Motivation

Traits are meant to be small pieces of behavior (functions or classes) used to extend other objects in a flexible, dynamic manner. Being small and independent entities, they are easy to understand, maintain and test. Traits also give an alternative approach in Python to handle diamond inheritance cases due to fact that no inheritance is happening at all (not saying multiple inheritance is an issue in Python).

The dynamic nature of traits enables some interesting use cases that are unreachable for conventional inheritance; Any changes made to class or instance are applied immediately, and they affect whole application. In practice, this means it is possible to add new functionality to any class or instance and it can be from your own module, some 3rd party module (e.g Django) or even Python's own internal classes (e.g. collections.OrderedDict).

For example, there is feature you would need from framework someone else has written. Only thing to do is to write traits for those classes that needs to be updated and extend them. After extending the classes, framework will behave based on those extended classes. Or if there is need to alter the behavior only some specific situation (or you just want to be careful), instances of classes can be extended only.

Other example would be a situation, where you discover a bug in 3rd party framework. Now you can create own solution safely, while waiting for the official patch to appear. Updating the framework code won't override your extensions as they are applied dynamically. Your changes are only removed when you don't need them anymore.

1.3 Basics

In the simplest form, traits are very similar to any class, that is inherited by some other class. That is a good way to approach traits in general; If you can inherit some class, then you can also use it as a trait. Let's look an example:

```
.. code:: python
    from pytraits import extendable

    class Parent:
        def parent_function(self):
            return "Hello World"

    # Traditional inheritance
    class TraditionalChild(Parent):
        pass

    @extendable
    class ExceptionalChild:
        pass

    # Composing as trait
    ExceptionalChild.add_traits(Parent)
```

In above example both TraditionalChild and Exceptional child have parent_function method. Only small difference is that ExceptionalChild is inherited from object, not Parent.

1.4 Effective use

To be effective with traits, one must have some knowledge about how to write code that can be reused effectively through out the system. It also helps to know something about good coding practices, for example:

- SOLID principles
- Law of Demeter

Especially in Law of Demeter, the interfaces tend to bloat because many small and specific functions needs to be implemented for classes. Traits can help to keep interfaces more manageable since one trait would contain methods only for some specific situation.

1.5 Vertical and Horizontal program architecture

Traits can really shine, when the application is layered both vertically and horizontally. Vertical layer basically means different components of the system, such as: *User*, *Address*, *Account*, *Wallet*, *Car*, *Computer*, etc. Horinzontal layers would contain: *Security*, *Serialization*, *Rendering*, etc. One approach with traits for above layering would be to create modules for horizontal parts and then create trait for each type object needing that behavior. Finally, in your main module, you would combine traits into classes.

Example: *core/account.py*

```
    from pytraits import extendable

    # Very simple address class
    @extendable
    class Address:
        def __init__(self, street, number):
            self.__street = street
            self.__number = number
```

core/wallet.py

```
    from pytraits import extendable

    # Very simple wallet class
    @extendable
    class Wallet:
        def __init__(self, money=0):
            self.__money = money
```

horizontal/html_rendering.py

```
    # This is a trait for address rendering
    class Address:
        def render(self):
            data = dict(street=self.__street, number=self.__number)
            return "<p>Address: {street} {number}</p>".format(**data)

    class Wallet:
        def render(self):
            # It is extremely straight-forward to render money situation.
            return "<p>Money: 0€</p>"
```

__main__.py

```
    from core import Address, Wallet
    from horizontal import html_rendering

    Address.add_traits(html_rendering.Address)
    Wallet.add_traits(html_rendering.Wallet)
```

With this approach, if there becomes a need to support other rendering mechanisms then just add new module and write rendering specific code there.

CHAPTER 2

Installation

At the command line:

```
pip install py3traits
```

CHAPTER 3

Usage

3.1 Composing traits

3.2 Combining classes

3.3 Adding properties dynamically

Properties can be very handy in some situations. Unfortunately, it is not that straightforward to add new properties to instances, thus pytraits has a small convenience function named *setproperty*. Using the function should be as simple as possible as it is quite flexible with ways to use it. Here is example of the simplest case:

```
from pytraits import setproperty

class Account:
    def __init__(self, money):
        self.__money = money

    def money(self):
        return self.__money

    def set_money(self, new_money):
        self.__money = new_money

my_account = Account(0)
setproperty(my_account, "money", "set_money")
```

There are more examples found in `examples/property_is_created_into_instance.py`

CHAPTER 4

Reference

4.1 pytraits

class `pytraits.Singleton`

Turn the class to immutable singleton.

```
>>> class Example(object, metaclass=Singleton):
...     pass
...
>>> a = Example()
>>> b = Example()
>>> id(a) == id(b)
True
```

Having your instance as a singleton is faster than creating from scratch

```
>>> import timeit
>>> class MySingleton(object, metaclass=Singleton):
...     def __init__(self):
...         self._store = dict(one=1, two=2, three=3, four=4)
...
>>> class NonSingleton:
...     def __init__(self):
...         self._store = dict(one=1, two=2, three=3, four=4)
...
>>> #timeit.timeit(NonSingleton) > timeit.timeit(MySingleton)
```

After creating a singleton, data it is holding should not be changed. There is a small enforcement done for these singletons to prevent modifying the contents. With little effort it is doable, but don't do it. :)

```
>>> MySingleton().new_item = False
Traceback (most recent call last):
...
pytraits.support.errors.SingletonError: Singletons are immutable!
```

class `pytraits.`**`Factory`**(*override_duplicates=False*)

Simple factory to register and create objects.

This class contains multiple ways to hold and create instances of classes. This class also works as a container for all those classes that are registered in and can those classes can be accessed from anywhere by simply importing that factory.

The main mechanism in python to create and initialize objects are __new__ and __init__ functions. It is also a good habit to avoid any conditional logic inside class constructor, thus writing own create classmethod is recommended and also supported by this factory. By using own class method for creating the object, it makes far more easier to setup and test classes you write since the __init__ method is left for simple assignments.

NOTE: This factory is abstract thus anyone using it must inherit own version before instantiating it.

```
>>> class ExampleFactory(Factory):
...     pass
...
>>> @ExampleFactory.register
... class ExampleObject:
...     def __init__(self, name, **kwargs):
...         self.name = name
...
...     @classmethod
...     def create(cls, *args, **kwargs):
...         return cls(*args, **kwargs)
...
>>> example_instance = ExampleFactory["ExampleObject"]("MyObject")
>>> example_instance.name
'MyObject'
```

`__getitem__`(*name*)

Returns factory method of registered object.

@see constructor

`exists`(*clazz*)

Convenience function to check if class is already exists.

`original_class`(*name*)

Retrieves the original registered class.

classmethod **`register`**(**classes*, *override=False*, *autoinit=True*)

Decorator function to register classes to this factory.

classmethod **`reset`**()

Removes all registered classes.

`pytraits.`**`combine_class`**(*class_name: str*, **traits*, ***resolved_conflicts*)

This function composes new class out of any number of traits.

Parameters

- **`class_name`** – Name of the new class.
- **`traits`** – Collection of traits, such as functions, classes or instances.

Keyword Arguments **name of trait** (*str*) – new name

Example of combining multiple classes to one:

```
>>> class One:
...     def first(self): return 1
...
>>> class Two:
...     def second(self): return 2
...
>>> class Three:
...     def third(self): return 3
...
>>> Combination = combine_class("Combination", One, Two, Three)
>>> instance = Combination()
>>> instance.first(), instance.second(), instance.third()
(1, 2, 3)
>>> instance.__class__.__name__
'Combination'
```

`pytraits.`**`extendable`**(*target*)

Decorator that adds function for object to be extended using traits.

NOTE: The 'add_traits' function this extendable decorator adds contains behavior that differs from usual function behavior. This method alters its behavior depending is the function called on a class or on an instance. If the function is invoked on class, then the class gets updated by the traits, affecting all new instances created from the class. On the other hand, if the function is invoked on an instance, only that instance gets the update, NOT whole class.

See complete example from: pytraits/examples/extendable_function_class_vs_instance.py

```
>>> @extendable
... class ExampleClass:
...     pass
...
>>> hasattr(ExampleClass, 'add_traits')
True
```

```
>>> class InstanceExample:
...     pass
...
>>> instance_example = InstanceExample()
>>> _ = extendable(instance_example)
>>> hasattr(instance_example, 'add_traits')
True
```

`pytraits.`**`add_traits`**

Bind new traits to given object.

Parameters

- **`target`** – Object of any type that is going to be extended with traits
- **`traits`** – Tuple of traits as object and strings or callables or functions.
- **`resolutions`** – dictionary of conflict resolutions to solve situations where multiple methods or properties of same name are encountered in traits.

```
>>> class ExampleClass:
...    def example_method(self):
...        return None
...
>>> class ExampleTrait:
...    def other_method(self):
...        return 42
```

```
...
>>> add_traits(ExampleClass, ExampleTrait)
>>> ExampleClass().other_method()
42
```

class pytraits.**type_safe**(*function*)

Decorator to enforce type safety. It certainly kills some ducks but allows us also to fail fast.

```
>>> @type_safe
... def check(value: int, answer: bool, anything):
...     return value, answer, anything
...
```

```
>>> check("12", "false", True)
Traceback (most recent call last):
...
TypeError: While calling check(value:int, answer:bool, anything):
   - parameter 'value' had value '12' of type 'str'
   - parameter 'answer' had value 'false' of type 'str'
```

```
>>> check(1000, True)
Traceback (most recent call last):
...
TypeError: check() missing 1 required positional argument: 'anything'
```

__call__(**args*, ***kwargs*)

Converts annotated types into proper type and calls original function.

__get__(*instance*, *clazz*)

Stores calling instances and returns this decorator object as function.

iter_positional_args(*args*)

Yields type, name, value combination of function arguments.

class pytraits.**type_converted**(*function*)

Decorator to enforce types and do auto conversion to values.

```
>>> @type_converted
... def convert(value: int, answer: bool, anything):
...     return value, answer, anything
...
>>> convert("12", "false", None)
(12, False, None)
```

```
>>> class Example:
...     @type_converted
...     def convert(self, value: int, answer: bool, anything):
...       return value, answer, anything
...
>>> Example().convert("12", 0, "some value")
(12, False, 'some value')
```

```
>>> Example().convert(None, None, None)
Traceback (most recent call last):
...
pytraits.support.errors.TypeConversionError: While calling Example.convert(self, value:int, answ
   - got arg 'value' as 'None' of type 'NoneType' which cannot be converted to 'int'
   - got arg 'answer' as 'None' of type 'NoneType' which cannot be converted to 'bool'
```

boolean_conversion(*value*)
Convert given value to boolean.

```
>>> conv = type_converted(lambda self: None)
>>> conv.boolean_conversion("True"), conv.boolean_conversion("false")
(True, False)
```

```
>>> conv.boolean_conversion(1), conv.boolean_conversion(0)
(True, False)
```

convert(*arg_type*, *arg_name*, *arg_value*)
Converts argument to given type.

pytraits.**setproperty**(*target*, *fget=None*, *fset=None*, *fdel=None*, *source=None*, *name=None*)
Convinience function that dynamically creates property to an object. (If you have property created, just use ‘add_traits’)

This function has different behavior depending on the target object, whether it is an instance or a class. If target is an instance the property is being set only for that instance. In case the object is a class, the property will be added to it normally to class.

Parameters

- **target** (*object or type*) – Target object, which can be any instance or class.
- **fget** (*str or function*) – Getter function or its name
- **fset** (*str or function*) – Setter function or its name
- **fdel** (*str or function*) – Deleter function or its name
- **source** (*object or type*) – Source object in case fget, fset and fdel are strings.

Keyword Arguments

- **name** (*str*) – Name of the property
- **name of fget** (*str*) – Name of the property

Example, where new property is added dynamically into instance:

```
>>> class Example:
...     def __init__(self):
...         self.__value = 42
...
...     def set_value(self, new_value):
...         self.__value = new_value
...
...     def value(self):
...         return self.__value
...
...     def del_value(self):
...         self.__value = 42
...
>>> instance = Example()
>>> setproperty(instance, "value", "set_value", "del_value", name="my_property")
>>> instance.my_property
42
```

CHAPTER 5

Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

5.1 Bug reports

When reporting a bug please include:

- Your operating system name, version and python version.
- Failing test case created similar manner as in py3traits/examples.

5.2 Documentation improvements

py3traits could always use more documentation, whether as part of the official py3traits docs, in docstrings, or even on the web in blog posts, articles, and such.

5.3 Feature requests and feedback

The best way to send feedback is to file an issue at https://github.com/Debith/py3traits/issues.

If you are proposing a feature:

- Explain in detail how it would work or even better, create a failing test case similar manner as in py3traits/examples
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

5.4 Development

To set up *py3traits* for local development:

1. Fork py3traits on GitHub.
2. Clone your fork locally:

```
git clone git@github.com:your_name_here/py3traits.git
```

3. Create a branch for local development:

```
git checkout -b name-of-your-bugfix-or-feature
```

   Now you can make your changes locally.

4. When you're done making changes, run all the checks, doc builder and spell checker with tox one command:

```
tox
```

5. Commit your changes and push your branch to GitHub:

```
git add .
git commit -m "Your detailed description of your changes."
git push origin name-of-your-bugfix-or-feature
```

6. Submit a pull request through the GitHub website.

5.4.1 Pull Request Guidelines

If you need some code review or feedback while you're developing the code just make the pull request.

For merging, you should:

1. Include passing tests (run `tox`) [1].
2. Update documentation when there's new API, functionality etc.
3. Add a note to `CHANGELOG.rst` about the changes.
4. Add yourself to `AUTHORS.rst`.

5.4.2 Tips

To run a subset of tests:

```
tox -e envname -- py.test -k test_myfeature
```

To run all the test environments in *parallel* (you need to `pip install detox`):

```
detox
```

[1] If you don't have all the necessary python versions available locally you can rely on Travis - it will run the tests for each change you add in the pull request.

It will be slower though ...

CHAPTER 6

Authors

- Teppo Perä - https://github.com/Debith/py3traits

CHAPTER 7

Changelog

7.1 1.2.1 (2015-07-08)

- Added “Motivation” section to documentation to help to discover use cases.

7.2 1.2.0 (2015-07-08)

- New feature: Precompiled (builtin) functions can be used with properties
- New feature: Precompiled (builtin) functions can be used as traits
- New feature: @validation decorator for validating arguments by value
- New feature: Factory class for object creation
- Improving feature: @type_safe and @type_converted wraps functions properly
- Fixed homepage link which was pointing to Python 2 version
- Added back the missing github link in documentation
- Done a major overhaul for the core to better support adding new features
- Many other bigger or smaller improvements

7.3 1.1.0 (2015-06-13)

- Improving feature: setproperty does not require all property functions
- Improving feature: added name as more convenient way to name the property
- Improving example: examples/property_is_created_into_instance.py
- Changing version numbering.

7.4 1.0.1 (2015-06-12)

- New feature: Added setproperty convenience function
- New example: examples/property_is_created_into_instance.py

- Added documentation
- Some refactoring for testability
- Added new test utility to parametrize tests
- Added unit tests

7.5 1.0.0 (2015-05-25)

- First official release

7.6 0.15.0 (2015-05-23)

- New feature: Alternative syntax added to add_traits function
- New example: examples/composition_in_alternative_syntax.py
- New example: examples/multiple_traits_composed_into_new_class.py
- Addes unit tests

7.7 0.14.0 (2015-05-19)

- New feature: Setter and Deleter for properties are now supported
- New example: examples/instance_is_composed_from_cherrypicked_property_in_class.py
- New example: examples/instance_is_composed_from_cherrypicked_property_in_instance.py
- Updated example: examples/class_is_composed_from_cherrypicked_property_in_class.py
- Updated example: examples/class_is_composed_from_cherrypicked_property_in_instance.py

7.8 0.13.0 (2015-04-25)

- New feature: Decorator type_safe to check function arguments
- New feature: combine_class function takes name for new class as first argument
- Refactoring magic.py to look less like black magic
- Improving errors.py exception class creation to accept custom messages
- Adding unit tests

7.9 0.12.0 (2015-04-22)

- New feature: Rename of composed traits
- Cleaning up parts belonging to py2traits

7.10 0.11.0 (2015-04-18)

- PEP8 fixes
- General cleaning for all files
- Removed unused parts
- Removed Python 2 code

7.11 0.10.0 (2015-03-30)

- Splitting into two projects: py2traits and py3traits
- Taking new project template to use from cookiecutter.

7.12 0.9.0 Bringing back compatibility to Python 2.x

- Some small clean up too

7.13 0.8.0 Adding support to private class and instance attributes

- Redone function binding to include recompilation of the function
- Leaving Python 2.x into unsupported state temporarily.

7.14 0.7.0 Improving usability of the library

- Introduced new extendable decorator, which adds function to add traits to object
- Introduced new function combine_class to create new classes out of traits
- Fixed module imports through out the library
- Improved documentation in examples

7.15 0.6.0 Restructuring into library

- Added support for py.test
- Preparing to support tox
- Improved multiple examples and renamed them to make more sense
- Removed the need of having two separate code branches for different Python versions

7.16 0.5.0 Instances can now be extended with traits in Python 3.x

- Instance support now similar to classes
- Added more examples

7.17 0.4.0 Completed function binding with examples in Python 2.x

- Separate functions can now be bound to classes - Functions with 'self' as a first parameter will be acting as a method - Functions with 'cls' as a first parameter will be acting as classmethod - Other functions will be static methods.
- Fixed an issue with binding functions

7.18 0.3.0 Trait extension support without conflicts for Python 2.x

- Classes can be extended
- Instances can be extended
- Python 2.x supported

7.19 0.2.0 Apache License Updated

- Added apache 2.0 license to all files
- Set the character set as utf-8 for all files

7.20 0.1.0 Initial Version

- prepared files for Python 2.x
- prepared files for Python 3.x

CHAPTER 8

Indices and tables

- genindex
- modindex
- search

Python Module Index

Index