A simple implementation of traits for Python

Author: Michele Simionato

Date: 2015-08-16

Version: 0.5.3

Download: http://pypi.python.org/pypi/strait

Licence: BSD

Abstract

I provide a simple implementation of traits as units of composable behavior for Python. I argue that traits are better than multiple inheritance. Implementing frameworks based on traits is left as an exercise for the reader.

Motivation

Multiple inheritance is a hotly debated topic. The supporters of multiple inheritance claim that it makes code shorter and easier to read, whereas the opposers claim that is makes code more coupled and more difficult to understand. I have spent some time in the past facing the intricacies of multiple inheritance in Python and I was one of its supporters once; however, since then I have worked with frameworks making large use of multiple inheritance (I mean Zope 2) and nowadays I am in the number of the people who oppose it. Therefore I am interested in alternatives.

In recent years, the approach of traits has gained some traction in a few circles and I have decided to write a library to implement traits in Python, for experimentation purposes. The library is meant for framework builders, people who are thinking about writing a framework based on multiple inheritance - typically via the common mixin approach - but are not convinced that this is the best solution and would like to try an alternative. This library is also for authors of mixin-bases frameworks which are unsatisfied and would like to convert their framework to traits.

Are traits a better solution than multiple inheritance and mixins? In theory I think so, otherwise I would not have written this library, but in practice (as always) things may be different. It may well be that using traits or using mixins does not make a big difference in practice and that the change of paradigm is not worth the effort; or the opposite may be true. The only way to know is to try, to build software based on traits and to see how it scale *in the large*. In the small, more or less any approach works fine: it is only by programming in the large that you can see the differences.

This is the reason why I am releasing this library with a liberal licence, so that people can try it out and see how it works. The library is meant to play well (when possible) with pre-existing frameworks. As an example, I will show here how you could rewrite Tkinter classes to use traits instead of mixins. Of course, I am not advocating rewriting Tkinter: it would be silly and pointless; but it may have sense (or not) to rewrite your own framework using traits, perhaps a framework which is used in house but has not been released yet.

I am not the only one to have implemented traits for Python; after finishing my implementation I made a little research and discovered a few implementations. Then I have also discovered the Enthought Traits framework, which however seems to use the name to intend something completely different (i.e. a sort of type checking). My implementation has no dependencies, is short and I am committed to keep it short even in the future, according to the principle of less is more.

There is also an hidden agenda behind this module: to popularize some advanced features of the Python object model which are little known. The `strait` module is actually a tribute to the metaprogramming capabilities of Python: such features are usually associated to languages with a strong academic tradition - Smalltalk, Scheme, Lisp - but actually the Python object model is no less powerful. For instance, changing the object system from a multiple inheritance one to a trait-based one, can be done *within* the fundamental object system. The reason is that the features that Guido used to implement the object

system (special method hooks, descriptors, metaclasses) are there, available to the end user to build her own object system.

Such features are usually little used in the Python community, for many good reasons: most people feel that the object system is good enough and that there is no reason to change it; moreover there is a strong opposition to change the language, because Python programmers believe in uniformity and in using common idioms; finally, it is difficult for an application programmer to find a domain where these features are useful. An exception is the domain of the Object Relation Mappers, whereas the Python language is often stretched to mimic the SQL language, a famous example of this tendency being SQLAlchemy). Still, I have never seen a perversion of the object model as big as the one implemented in the `strait` module, so I wanted to be the first one to perform that kind of abuse ;)

What are traits?

The word *traits* has many meanings; I will refer to it in the sense of the paper Traits - Composable Units of Behavior which implements them in Squeak/Smalltalk. The paper appeared in 2003, but most of the ideas underlying traits have been floating around for at least 30 years. There is also a trait implementation for PLT Scheme which is somewhat close in spirit (if not in practice) to what I am advocating here. The library you are reading about is by no means intended as a porting of the Smalltalk library: I am just stealing some of the ideas from that paper to implement a Pythonic alternative to mixins which, for lack of a better name, I have decided to call traits. I feel no obligation whatsoever to be consistent with the Smalltalk library. In doing so, I am following a long tradition, since a lot of languages use the name *traits* to mean something completely different from the Smalltalk meaning. For instance the languages Fortress and Scala use the name *trait* but with a different meaning (Scala traits are very close to multiple inheritance). For me a trait is a bunch of methods and attributes with the following properties:

1. the methods/attributes in a trait belong logically together;
2. if a trait enhances a class, then all subclasses are enhanced too;
3. if a trait has methods in common with the class, then the methods defined in the class have the precedence;
4. the trait order is not important, i.e. enhancing a class first with trait T1 and then with trait T2 or viceversa is the same;
5. if traits T1 and T2 have names in common, enhancing a class both with T1 and T2 raises an error;
6. if a trait has methods in common with the base class, then the trait methods have the precedence;
7. a class can be seen both as a composition of traits and as an homogeneous entity.

Properties from 4 to 7 are the distinguishing properties of traits with respect to multiple inheritance and mixins. In particular, because of 4 and 5, all the complications with the Method Resolution Order disappear and the overriding is never implicit. Property 6 is mostly unusual: typically in Python the base class has the precedence over mixin classes. Property 7 should be intended in the sense that a trait implementation must provide introspection facilities to make seemless the transition between classes viewed as atomic entities and as composed entities.

A hands-on example

Let me begin by showing how you could rewrite a Tkinter class to use traits instead of mixins. Consider the `Tkinter.Widget` class, which is derived by the base class `BaseWidget` and the mixin classes `Tkinter.Grid`, `Tkinter.Pack` and `Tkinter.Place`: I want to rewrite it by using traits. The `strait` module provides a factory function named `include` that does the job. It is enough to replace the multiple inheritance syntax:

```
class Widget(BaseWidget, Grid, Pack, Place):
     pass
```

with the following syntax:

```
class Widget(BaseWidget):
    __metaclass__ = include(Pack, Place, Grid)
```

I said that the conversion from mixins to traits was easy: but actually I lied since if you try to execute the code I just wrote you will get an `OverridingError`:

```
>>> from Tkinter import *
>>> class Widget(BaseWidget):
...     __metaclass__ = include(Pack, Place, Grid)
Traceback (most recent call last):
  ...
OverridingError: Pack overrides names in Place: {info, config, configure, slaves, forget
```

The reason for the error is clear: both `Pack` and `Place` provide methods called `{info, config, configure, slaves, forget}` and the traits implementation cannot figure out which ones to use. This is a feature, since it forces you to be explicit. In this case, if we want to be consistent with multiple inheritance rules, we want the methods coming from the first class (i.e. `Pack`) to take precedence. That can be implemented by including directly those methods in the class namespace and relying on rule 3:

```
class TOSWidget(BaseWidget):
    __metaclass__ = include(Pack, Place, Grid)
    info = Pack.info.im_func
    config = Pack.config.im_func
    configure = Pack.configure.im_func
    slaves = Pack.slaves.im_func
    forget = Pack.forget.im_func
    propagate = Pack.propagate.im_func
```

Notice that we had to specify the `propagate` method too, since it is a common method between `Pack` and `Grid`.

You can check that the TOSWidget class works, for instance by defining a label widget as follows (remember that `TOSWidget` inherits its signature from `BaseWidget`):

```
>>> label = TOSWidget(master=None, widgetName='label',
...                   cnf=dict(text="hello"))
```

You may visualize the widget by calling the `.pack` method:

```
>>> label.pack()
```

This should open a small window with the message "hello" inside it.

A few caveats and warnings

First of all, let me notice that, in spite of apparency, `include` does not return a metaclass. Insted, it returns a class factory function with signature `name, bases, dic`:

```
>>> print include(Pack, Place, Grid)
<function include_Pack_Place_Grid at 0x...>
```

This function will create the class by using a suitable metaclass:

```
>>> type(TOSWidget)
<class 'strait.MetaTOS'>
```

In simple cases the metaclass will be `MetaTOS`, the main class of the trait object system, but in general it can be a different one not inheriting from `MetaTOS`. The exact rules followed by `include` to determine the right class will be discussed later.

Here I want to remark that according to rule 6 traits take the precedence over the base class attributes. Consider the following example:

```
>>> class Base(object):
...     a = 1

>>> class ATrait(object):
...     a = 2

>>> class Class(Base):
...     __metaclass__ = include(ATrait)

>>> Class.a
2
```

In regular multiple inheritance you would do the same by including `ATrait` *before* `Base`, i.e.

```
>>> type('Class', (ATrait, Base), {}).a
2
```

You should take care to not mix-up the order, otherwise you will get a different result:

```
>>> type('Class', (Base, ATrait), {}).a
1
```

Therefore replacing mixin classes with traits can break your code if you rely on the order. Be careful!

The Trait Object System

The goal of the `strait` module it to modify the standard Python object model, turning it into a Trait Object System (TOS for short): TOS classes behave differently from regular classes. In particular TOS classes do not support multiple inheritance. If you try to multiple inherit from a TOS class and another class you will get a `TypeError`:

```
>>> class M:
...     "An empty class"
...
>>> class Widget2(TOSWidget, M):
...     pass
...
Traceback (most recent call last):
  ...
TypeError: Multiple inheritance of bases (<class '__main__.TOSWidget'>, <class __main__.
```

This behavior is intentional: with this restriction you can simulate an ideal world in which Python did not support multiple inheritance. Suppose you want to claim that supporting multiple inheritance was a

mistake and that Python would have been better off without it (which is the position I tend to have nowadays): how can you prove that claim? Simply by writing code that does not use multiple inheritance and it is clearer and more mantainable that code using multiple inheritance.

I am releasing this trait implementation hoping you will help me to prove (or possibly disprove) the point. You may see traits as a restricted form of multiple inheritance without name clashes, without the complications of the method resolution, and with a limited cooperation between methods. Moreover the present implementation is slightly less dynamic than usual inheritance.

A nice property of inheritance is that if you have a class `C` inheriting from class `M` and you change a method in `M` at runtime, after `C` has been created and instantiated, automagically all instances of `C` gets the new version of the method, which is pretty useful for debugging purposes. This feature is lost in the trait implementation provided here. Actually, in a previous version, my trait implementation was fully dynamic and if you changed the mixin the instances would be changed too. However, I never used that feature in practice, and it was complicating the implementation and slowing doing the attribute access, so I removed it.

I think these are acceptable restrictions since they give back in return many advantages in terms of simplicity: for instance, `super` becomes trivial, since each class has a single superclass, whereas we all know that the current super in Python is very far from trivial.

The magic of `include`

Since the fundamental properties of TOS classes must be preserved under inheritance (i.e. the son of a TOS class must be a TOS class) the implementation necessarily requires metaclasses. As of now, the only fundamental property of a TOS class is that multiple inheritance is forbidden, so usually (*but not always*) TOS classes are instances of the metaclass `MetaTOS` which implements a single inheritance check. If you build your TOS hierarchy starting from pre-existing classes, you should be aware of how `include` determines the metaclass: if your base class was an old-style class or a plain new style class (i.e. a direct instance of the `type` metaclass), them `include` will change it to `MetaTOS`:

```
>>> type(TOSWidget)
<class 'strait.MetaTOS'>
```

In general you may need to build your Trait Based Framework on top of pre-existing classes possessing a nontrivial metaclass, for instance Zope classes; in that case `include` is smart enough to figure out the right metaclass to use. Here is an example:

```
class AddGreetings(type):
    "A metaclass adding a 'greetings' attribute for exemplification purposes"
    def __new__(mcl, name, bases, dic):
        dic['greetings'] = 'hello!'
        return super(AddGreetings, mcl).__new__(mcl, name, bases, dic)
```

```
class WidgetWithGreetings(BaseWidget, object):
    __metaclass__ = AddGreetings
```

```
class PackWidget(WidgetWithGreetings):
    __metaclass__ = include(Pack)
```

`include` automatically generates the right metaclass as a subclass of `AddGreetings`:

```
>>> print type(PackWidget).__mro__
(<class 'strait._TOSAddGreetings'>, <class '__main__.AddGreetings'>, <type 'type'>, <typ
```

Incidentally, since TOS classes are guaranteed to be in a straight hierarchy, `include` is able to neatly avoid the dreaded metaclass conflict.

The important point is that `_TOSAddGreetings` provides the same features of `MetaTOS`, even if it is not a subclass of it; on the other hand, `_TOSMetaAddGreetings` is a subclass of `AddGreetings` which calls `AddGreetings.__new__`, so the features provided by `AddGreetings` are not lost either; in this example you may check that the greetings attribute is correctly set:

```
>>> PackWidget.greetings
'hello!'
```

The name of the generated metaclass is automatically generated from the name of the base metaclass; moreover, a register of the generated metaclasses is kept, so that metaclasses are reused if possible. If you want to understand the details, you are welcome to give a look at the implementation, which is pretty short and simple, compared to the general recipe to remove the metaclass conflict in a true multiple inheritance situation.

Cooperative traits

At first sight, the Trait Object System lacks an important feature of multiple inheritance as implemented in the ordinary Python object system, i.e. cooperative methods. Consider for instance the following classes:

```
class LogOnInitMI(object):
    def __init__(self, *args, **kw):
        print 'Initializing %s' % self
        super(LogOnInitMI, self).__init__(*args, **kw)
```

```
class RegisterOnInitMI(object):
    register = []
    def __init__(self, *args, **kw):
        print 'Registering %s' % self
        self.register.append(self)
        super(RegisterOnInitMI, self).__init__(*args, **kw)
```

In multiple inheritance `LogOnInitMI` can be mixed with other classes, giving to the children the ability to log on initialization; the same is true for `RegisterOnInitMI`, which gives to its children the ability to populate a registry of instances. The important feature of the multiple inheritance system is that `LogOnInitMI` and `RegisterOnInitMI` play well together: if you inherits from both of them, you get both features:

```
class C_MI(LogOnInitMI, RegisterOnInitMI):
    pass
```

```
>>> c = C_MI()
Initializing <__main__.C_MI object at 0x...>
Registering <__main__.C_MI object at 0x...>
```

You cannot get the same behaviour if you use the trait object system naively:

```
>>> class C_MI(object):
...       __metaclass__ = include(LogOnInitMI, RegisterOnInitMI)
...
Traceback (most recent call last):
```

```
  ...
OverridingError: LogOnInitMI overrides names in RegisterOnInitMI: {__init__}
```

This is a feature, of course, since the trait object system is designed to avoid name clashes. However, the situation is worse than that: even if you try to mixin a single class you will run into trouble

```
>>> class C_MI(object):
...       __metaclass__ = include(LogOnInitMI)
>>> c = C_MI()
Traceback (most recent call last):
 ...
TypeError: super(type, obj): obj must be an instance or subtype of type
```

What's happening here? The situation is clear if you notice that the `super` call is actually a call of kind `super(LogOnInitMI, c)` where `c` is an instance of `C`, which is not a subclass of `LogOnInitMI`. That explains the error message, but does not explain how to solve the issue. It seems that method cooperation using `super` is impossible for TOS classes.

Actually this is not the case: single inheritance cooperation is possible and it is enough as we will show in a minute. But for the moment let me notice that I do not think that cooperative methods are necessarily a good idea. They are fragile and cause all of your classes to be strictly coupled. My usual advice if that you should not use a design based on method cooperation if you can avoid it. Having said that, there are situations (very rare) where you really want method cooperation. The `strait` module provide support for those situations via the `__super` attribute.

Let me explain how it works. When you mix-in a trait `T` into a class `C`, `include` adds an attribute `_T__super` to `C`, which is a `super` object that dispatches to the attributes of the superclass of `C`. The important thing to keep in mind is that there is a well defined superclass, since the trait object system uses single inheritance only. Since the hierarchy is straight, the cooperation mechanism is much simpler to understand than in multiple inheritance. Here is an example. First of all, let me rewrite `LogOnInit` and `RegisterOnInit` to use `__super` instead of `super`:

```
class LogOnInit(object):
    def __init__(self, *args, **kw):
        print 'Initializing %s' % self
        self.__super.__init__(*args, **kw)
```

```
class RegisterOnInit(object):
    register = []

    def __init__(self, *args, **kw):
        print 'Registering %s' % self
        self.register.append(self)
        self.__super.__init__(*args, **kw)
```

Now you can include the `RegisterOnInit` functionality as follows:

```
class C_Register(object):
    __metaclass__ = include(RegisterOnInit)
```

```
>>> _ = C_Register()
Registering <__main__.C_Register object at 0x...>
```

Everything works because `include` has added the right attribute:

```
>>> C_Register._RegisterOnInit__super
<super: <class 'C_Register'>, <C_Register object>>
```

Moreover, you can also include the `LogOnInit` functionality:

```
class C_LogAndRegister(C_Register):
    __metaclass__ = include(LogOnInit)
```

```
>>> _ = C_LogAndRegister()
Initializing <__main__.C_LogAndRegister object at 0x...>
Registering <__main__.C_LogAndRegister object at 0x...>
```

As you see, the cooperation mechanism works just fine. I will call *cooperative trait* a class intended for inclusion in other classes and making use of the `__super` trick. A class using the regular `super` directly cannot be used as a cooperative trait, since it must satisfy inheritance constraints, nevertheless it is easy enough to convert it to use `__super`. After all, the `strait` module is intended for framework writers, so it assumes you can change the source code of your framework if you want. On the other hand, if are trying to re-use a mixin class coming from a third party framework and using `super`, you will have to rewrite the parts of it. That is unfortunate, but I cannot perform miracles.

You may see `__super` as a clever hack to use `super` indirectly. Notice that since the hierarchy is straight, there is room for optimization at the core language level. The `__super` trick as implemented in pure Python leverages on the name mangling mechanism, and follows closely the famous autosuper recipe, with some improvement. Anyway, if you have two traits with the same name, you will run into trouble. To solve this and to have a nicer syntax, one would need more support from the language, but the `__super` trick is good enough for a prototype and has the serious advantage of working right now for current Python.

Cooperation at the metaclass level

In my experience, the cases where you need method cooperation in multiple inheritance situations are exceedingly rare, unless you are a language implementor or a designer of very advanced frameworks. In such a realm you have a need for cooperative methods; it is not a pressing need, in the sense that you can always live without them, but they are a nice feature to have if you care about elegance and extensibility. For instance, as P. J. Eby points it out in this thread on python-dev:

A major use case for co-operative super() is in the implementation of metaclasses. The __init__ and __new__ signatures are fixed, multiple inheritance is possible, and co-operativeness is a must (as the base class methods must *be called). I'm hard-pressed to think of a metaclass constructor or initializer that I've written in the last half-decade or more where I didn't use super() to make it co-operative. That, IMO, is a compelling use case even if there were not a single other example of the need for super.*

I have always felt the same. So, even if I have been unhappy with multiple inheritance for years, I could never dismiss it entirely because of the concern for this use case. It is only after discovering cooperative traits that I felt the approach powerful enough to replace multiple inheritance without losing anything I cared about.

Multiple inheritance at the metaclass level comes out here and again when you are wearing the language implementor hat. For instance, if you try to implement an object system based on traits, you will have to do so at the metaclass level and there method cooperation has its place. In particular, if you look at the source code of the `strait` module - which is around 100 lines, a tribute to the power of Python - you will see that the `MetaTOS` metaclass is implemented as a cooperative trait, so that it can be mixed-in with other metaclasses, in the case you are interoperating with a framework with a non-trivial meta object protocol. This is performed internally by `include`.

Metaclass cooperation is there to make the life of the users easier. Suppose one of you, users of the `strait` module, wants to enhance the `include` mechanism using another a metaclass coming for a third party framework and therefore not inheriting from `MetaTOS`:

```
class ThirdPartyMeta(type):
    def __new__(mcl, name, bases, dic):
        print 'Using ThirdPartyMeta to create %s' % name
        return super(ThirdPartyMeta, mcl).__new__(mcl, name, bases, dic)
```

The way to go is simple. First, you should mix-in `MetaTOS` in the third party class:

```
class EnhancedMetaTOS(ThirdPartyMeta):
    __metaclass__ = include(MetaTOS)
```

Then, you can define your own enhanced `include` as follows:

```
def enhanced_include(*traits):
    return include(MetaTOS=EnhancedMetaTOS, *traits)
```

In simple cases using directly `ThirdPartyMeta` may work, but I strongly recommend to replace the call to super with `__super` even in `ThirdPartyMeta` to make the cooperation robust.

Discussion of some design decisions and future work

The decision of having TOS classes which are not instances of `MetaTOS` required some thought. That was my original idea in version 0.1 of `strait`; however in version 0.2 I wanted to see what would happen if I made all TOS classes instances of `MetaTOS`. That implied that if your original class had a nontrivial metaclass, then the TOS class had to inherit both from the original metaclass *and* `MetaTOS`, i.e. multiple inheritance and cooperation of methods was required at the metaclass level.

I did not like it, since I was arguing that you can do everything without multiple inheritance; moreover using multiple inheritance at the metaclass level meant that one had to solve the metaclass conflict in a general way. I did so, by using my own cookbook recipe, and all my tests passed.

Neverthess, at the end, in version 0.3 I decided to go back to the original design. The metaclass conflict recipe is too complex, and I see it as a code smell - *if the implementation is hard to explain, it's a bad idea* - just another indication that multiple inheritance is bad. In the original design it is possible to add the features of `MetaTOS` to the original metaclass by subclassing it with *single* inheritance and thus avoiding the conflict.

The price to pay is that now a TOS class is no more an instance of `MetaTOS`, but this is a non-issue: the important thing is that TOS classes perform the dispatch on their traits as `MetaTOS` would dictate. Moreover, starting from Python 2.6, thanks to Abstract Base Classes, you may satisfy the `isinstance(obj, cls)` check even if `obj` is not an instance of `cls`, by registering a suitable base class (similarly for `issubclass`). In our situation, that means that it is enough to register `MetaTOS` as base class of the original metaclass.

Version 0.4 was much more complex that the current version (still short, it was under 300 lines of pure Python), since it had the more ambitious goal of solving the namespace pollution problem. I have discussed the issue elsewhere: if you keep injecting methods into a class (both directly or via inheritance) you may end up having hundreds of methods flattened at the same level.

A picture is worth a thousand words, so have a look at the PloneSite hierarchy if you want to understand the horror I wanted to avoid with traits (the picture shows the number of nonspecial attributes defined per class in square brackets): in the Plone Site hierarchy there are 38 classes, 88 overridden names, 42 special names, 648 non-special attributes and methods. It is a nighmare.

Originally I wanted to prevent this kind of abuse, but that made my implementation more complex, whereas my main goal was to keep the implementation simple. As a consequence this version assumes the prosaic attitude that you cannot stop programmers from bad design anyway, so if they want to go the Zope way they can.

In previous versions I did provide some syntactic sugar for `include` so that it was possible to write something like the following (using a trick discussed here):

```
class C(Base):
   include(Trait1, Trait2)
```

In version 0.5 I decided to remove this feature. Now the plumbing (i.e. the `__metaclass__` hook) is exposed to the user, some magic has been removed and it is easier for the user to write her own `include` factory if she wants to.

Where to go from here? For the moment, I have no clear idea about the future. The Smalltalk implementation of traits provides method renaming out of the box. The Python implementation has no facilities in this sense. In the future I may decide to give some support for renaming, or I may not. At the present you can just rename your methods by hand. Also, in the future I may decide to add some kind of adaptation mechanism or I may not: after all the primary goal of this implementation is semplicity and I don't want to clutter it with too many features.

I am very open to feedback and criticism: I am releasing this module with the hope that it will be used in real life situations to gather experience with the traits concept. Clearly I am not proposing that Python should remove multiple inheritance in favor of traits: considerations of backward compatibily would kill the proposal right from the start. I am just looking for a few adventurous volunteers wanting to experiment with traits; if the experiment goes well, and people start using (multiple) inheritance less than they do now, I will be happy.

Trivia

`strait` officially stands for Simple Trait object system, however the name is also a pun on the world "straight", since the difference between multiple inheritance hierarchies and TOS hierarchies is that TOS hierarchies are straight. Moreover, nobody will stop you from thinking that the `s` also stands for Simionato ;)