PEP 318 – Decorators for Functions and Methods

Author: Kevin D. Smith <Kevin.Smith at theMorgue.org>, Jim J. Jewett, Skip Montanaro, Anthony Baxter

Status: Final

Type: Standards Track

Created: 05-Jun-2003

Python-Version: 2.4

Post-History: 09-Jun-2003, 10-Jun-2003, 27-Feb-2004, 23-Mar-2004, 30-Aug-2004, 02-Sep-2004

WarningWarningWarning

This document is meant to describe the decorator syntax and the process that resulted in the decisions that were made. It does not attempt to cover the huge number of potential alternative syntaxes, nor is it an attempt to exhaustively list all the positives and negatives of each form.

Abstract

The current method for transforming functions and methods (for instance, declaring them as a class or static method) is awkward and can lead to code that is difficult to understand. Ideally, these transformations should be made at the same point in the code where the declaration itself is made. This PEP introduces new syntax for transformations of a function or method declaration.

Motivation

The current method of applying a transformation to a function or method places the actual transformation after the function body. For large functions this separates a key component of the function's behavior from the definition of the rest of the function's external interface. For example:

```
def foo(self):
    perform method operation
foo = classmethod(foo)
```

This becomes less readable with longer methods. It also seems less than pythonic to name the function three times for what is conceptually a single declaration. A solution to this problem is to move the transformation of the method closer to the method's own declaration. The intent of the new syntax is to replace

```
def foo(cls):
    pass
foo = synchronized(lock)(foo)
foo = classmethod(foo)
```

with an alternative that places the decoration in the function's declaration:

```
@classmethod
@synchronized(lock)
def foo(cls):
    pass
```

Modifying classes in this fashion is also possible, though the benefits are not as immediately apparent. Almost certainly, anything which could be done with class decorators could be done using metaclasses, but using metaclasses is sufficiently obscure that there is some attraction to having an easier way to make simple modifications to classes. For Python 2.4, only function/method decorators are being added.

PEP 3129 proposes to add class decorators as of Python 2.6.

Why Is This So Hard?

Two decorators (`classmethod()` and `staticmethod()`) have been available in Python since version 2.2. It's been assumed since approximately that time that some syntactic support for them would eventually be added to the language. Given this assumption, one might wonder why it's been so difficult to arrive at a consensus. Discussions have raged off-and-on at times in both comp.lang.python and the python-dev mailing list about how best to implement function decorators. There is no one clear reason why this should be so, but a few problems seem to be most divisive.

- Disagreement about where the "declaration of intent" belongs. Almost everyone agrees that decorating/transforming a function at the end of its definition is suboptimal. Beyond that there seems to be no clear consensus where to place this information.
- Syntactic constraints. Python is a syntactically simple language with fairly strong constraints on what can and can't be done without "messing things up" (both visually and with regards to the language parser). There's no obvious way to structure this information so that people new to the concept will think, "Oh yeah, I know what you're doing." The best that seems possible is to keep new users from creating a wildly incorrect mental model of what the syntax means.
- Overall unfamiliarity with the concept. For people who have a passing acquaintance with algebra (or even basic arithmetic) or have used at least one other programming language, much of Python is intuitive. Very few people will have had any experience with the decorator concept before encountering it in Python. There's just no strong preexisting meme that captures the concept.
- Syntax discussions in general appear to cause more contention than almost anything else. Readers are pointed to the ternary operator discussions that were associated with

PEP 308 for another example of this.

Background

There is general agreement that syntactic support is desirable to the current state of affairs. Guido mentioned syntactic support for decorators in his DevDay keynote presentation at the 10th Python Conference, though he later said it was only one of several extensions he proposed there "semi-jokingly". Michael Hudson raised the topic on `python-dev` shortly after the conference, attributing the initial bracketed syntax to an earlier proposal on `comp.lang.python` by Gareth McCaughan.

Class decorations seem like an obvious next step because class definition and function definition are syntactically similar, however Guido remains unconvinced, and class decorators will almost certainly not be in Python 2.4.

The discussion continued on and off on python-dev from February 2002 through July 2004. Hundreds and hundreds of posts were made, with people proposing many possible syntax variations. Guido took a list of proposals to EuroPython 2004, where a discussion took place. Subsequent to this, he decided that we'd have the Java-style @decorator syntax, and this appeared for the first time in 2.4a2. Barry Warsaw named this the 'pie-decorator' syntax, in honor of the Pie-thon Parrot shootout which occurred around the same time as the decorator syntax, and because the @ looks a little like a pie. Guido outlined his case on Python-dev, including this piece on some of the (many) rejected forms.

On the name 'Decorator'

There's been a number of complaints about the choice of the name 'decorator' for this feature. The major one is that the name is not consistent with its use in the GoF book. The name 'decorator' probably owes more to its use in the compiler area – a syntax tree is walked and annotated. It's quite possible that a better name may turn up.

Design Goals

The new syntax should

- work for arbitrary wrappers, including user-defined callables and the existing builtins `classmethod()` and `staticmethod()`. This requirement also means that a decorator syntax must support passing arguments to the wrapper constructor
- work with multiple wrappers per definition
- make it obvious what is happening; at the very least it should be obvious that new users can safely ignore it when writing their own code
- be a syntax "that … [is] easy to remember once explained"
- not make future extensions more difficult
- be easy to type; programs that use it are expected to use it very frequently

- not make it more difficult to scan through code quickly. It should still be easy to search for all definitions, a particular definition, or the arguments that a function accepts
- not needlessly complicate secondary support tools such as language-sensitive editors and other "toy parser tools out there"
- allow future compilers to optimize for decorators. With the hope of a JIT compiler for Python coming into existence at some point this tends to require the syntax for decorators to come before the function definition
- move from the end of the function, where it's currently hidden, to the front where it is more in your face

Andrew Kuchling has links to a bunch of the discussions about motivations and use cases in his blog. Particularly notable is Jim Huginin's list of use cases.

Current Syntax

The current syntax for function decorators as implemented in Python 2.4a2 is:

```
@dec2
@dec1
def func(arg1, arg2, ...):
    pass
```

This is equivalent to:

```
def func(arg1, arg2, ...):
    pass
func = dec2(dec1(func))
```

without the intermediate assignment to the variable `func`. The decorators are near the function declaration. The @ sign makes it clear that something new is going on here.

The rationale for the order of application (bottom to top) is that it matches the usual order for function-application. In mathematics, composition of functions (g o f)(x) translates to g(f(x)). In Python, `@g @f def foo()` translates to `foo=g(f(foo))`.

The decorator statement is limited in what it can accept – arbitrary expressions will not work. Guido preferred this because of a gut feeling.

The current syntax also allows decorator declarations to call a function that returns a decorator:

```
@decomaker(argA, argB, ...)
def func(arg1, arg2, ...):
    pass
```

This is equivalent to:

```
func = decomaker(argA, argB, ...)(func)
```

The rationale for having a function that returns a decorator is that the part after the @ sign can be considered to be an expression (though syntactically restricted to just a function), and whatever that expression returns is called. See declaration arguments.

Syntax Alternatives

There have been a large number of different syntaxes proposed – rather than attempting to work through these individual syntaxes, it's worthwhile to break the syntax discussion down into a number of areas. Attempting to discuss each possible syntax individually would be an act of madness, and produce a completely unwieldy PEP.

Decorator Location

The first syntax point is the location of the decorators. For the following examples, we use the @syntax used in 2.4a2.

Decorators before the def statement are the first alternative, and the syntax used in 2.4a2:

```
@classmethod
def foo(arg1,arg2):
    pass

@accepts(int,int)
@returns(float)
def bar(low,high):
    pass
```

There have been a number of objections raised to this location – the primary one is that it's the first real Python case where a line of code has an effect on a following line. The syntax available in 2.4a3 requires one decorator per line (in a2, multiple decorators could be specified on the same line), and the final decision for 2.4 final stayed one decorator per line.

People also complained that the syntax quickly got unwieldy when multiple decorators were used. The point was made, though, that the chances of a large number of decorators being used on a single function were small and thus this was not a large worry.

Some of the advantages of this form are that the decorators live outside the method body – they are obviously executed at the time the function is defined.

Another advantage is that a prefix to the function definition fits the idea of knowing about a change to the semantics of the code before the code itself, thus you know how to interpret the code's semantics properly without having to go back and change your initial perceptions if the syntax did not come before the function definition.

Guido decided he preferred having the decorators on the line before the 'def', because it was felt that a long argument list would mean that the decorators would be 'hidden'

The second form is the decorators between the def and the function name, or the function name and the argument list:

```
def @classmethod foo(arg1,arg2):
    pass

def @accepts(int,int),@returns(float) bar(low,high):
    pass

def foo @classmethod (arg1,arg2):
    pass

def bar @accepts(int,int),@returns(float) (low,high):
    pass
```

There are a couple of objections to this form. The first is that it breaks easily 'greppability' of the source – you can no longer search for 'def foo(' and find the definition of the function. The second, more serious, objection is that in the case of multiple decorators, the syntax would be extremely unwieldy.

The next form, which has had a number of strong proponents, is to have the decorators between the argument list and the trailing `:` in the 'def' line:

```
def foo(arg1,arg2) @classmethod:
    pass

def bar(low,high) @accepts(int,int),@returns(float):
    pass
```

Guido summarized the arguments against this form (many of which also apply to the previous form) as:

- it hides crucial information (e.g. that it is a static method) after the signature, where it is easily missed
- it's easy to miss the transition between a long argument list and a long decorator list
- it's cumbersome to cut and paste a decorator list for reuse, because it starts and ends in the middle of a line

The next form is that the decorator syntax goes inside the method body at the start, in the same place that docstrings currently live:

```
def foo(arg1,arg2):
    @classmethod
    pass

def bar(low,high):
    @accepts(int,int)
    @returns(float)
    pass
```

The primary objection to this form is that it requires “peeking inside” the method body to determine the decorators. In addition, even though the code is inside the method body, it is not executed when the method is run. Guido felt that docstrings were not a good counter-example, and that it was quite possible that a ‘docstring’ decorator could help move the docstring to outside the function body.

The final form is a new block that encloses the method’s code. For this example, we’ll use a ‘decorate’ keyword, as it makes no sense with the @syntax.

```
decorate:
    classmethod
    def foo(arg1,arg2):
        pass

decorate:
    accepts(int,int)
    returns(float)
    def bar(low,high):
        pass
```

This form would result in inconsistent indentation for decorated and undecorated methods. In addition, a decorated method’s body would start three indent levels in.

Syntax forms

- `@decorator`:

  ```
  @classmethod
  def foo(arg1,arg2):
      pass

  @accepts(int,int)
  @returns(float)
  def bar(low,high):
      pass
  ```

  The major objections against this syntax are that the @ symbol is not currently used in Python (and is used in both IPython and Leo), and that the @ symbol is not meaningful. Another objection is that this “wastes” a currently unused character (from a limited set) on something that is not perceived as a major use.

- `|decorator`:

  ```
  |classmethod
  def foo(arg1,arg2):
      pass

  |accepts(int,int)
  |returns(float)
  def bar(low,high):
      pass
  ```

This is a variant on the @decorator syntax – it has the advantage that it does not break IPython and Leo. Its major disadvantage compared to the @syntax is that the | symbol looks like both a capital I and a lowercase l.

- list syntax:

```
[classmethod]
def foo(arg1,arg2):
    pass

[accepts(int,int), returns(float)]
def bar(low,high):
    pass
```

The major objection to the list syntax is that it's currently meaningful (when used in the form before the method). It's also lacking any indication that the expression is a decorator.

- list syntax using other brackets (`<...>`, `[[...]]`, ...):

```
<classmethod>
def foo(arg1,arg2):
    pass

<accepts(int,int), returns(float)>
def bar(low,high):
    pass
```

None of these alternatives gained much traction. The alternatives which involve square brackets only serve to make it obvious that the decorator construct is not a list. They do nothing to make parsing any easier. The '<...>' alternative presents parsing problems because '<' and '>' already parse as un-paired. They present a further parsing ambiguity because a right angle bracket might be a greater than symbol instead of a closer for the decorators.

- `decorate()`

The `decorate()` proposal was that no new syntax be implemented – instead a magic function that used introspection to manipulate the following function. Both Jp Calderone and Philip Eby produced implementations of functions that did this. Guido was pretty firmly against this – with no new syntax, the magicness of a function like this is extremely high:

> *Using functions with "action-at-a-distance" through sys.settraceback may be okay for an obscure feature that can't be had any other way yet doesn't merit changes to the language, but that's not the situation for decorators. The widely held view here is that decorators need to be added as a syntactic feature to avoid the problems with the postfix notation used in 2.2 and 2.3. Decorators are slated to be an important new language feature and their design needs to be forward-looking, not constrained by what can be implemented in 2.3.*

- new keyword (and block)

  This idea was the consensus alternate from comp.lang.python (more on this in Community Consensus below.) Robert Brewer wrote up a detailed J2 proposal document outlining the arguments in favor of this form. The initial issues with this form are:

  - It requires a new keyword, and therefore a `from __future__ import decorators` statement.
  - The choice of keyword is contentious. However `using` emerged as the consensus choice, and is used in the proposal and implementation.
  - The keyword/block form produces something that looks like a normal code block, but isn't. Attempts to use statements in this block will cause a syntax error, which may confuse users.

  A few days later, Guido rejected the proposal on two main grounds, firstly:

  > *... the syntactic form of an indented block strongly suggests that its contents should be a sequence of statements, but in fact it is not – only expressions are allowed, and there is an implicit "collecting" of these expressions going on until they can be applied to the subsequent function definition. ...*

  and secondly:

  > *... the keyword starting the line that heads a block draws a lot of attention to it. This is true for "if", "while", "for", "try", "def" and "class". But the "using" keyword (or any other keyword in its place) doesn't deserve that attention; the emphasis should be on the decorator or decorators inside the suite, since those are the important modifiers to the function definition that follows. ...*

  Readers are invited to read the full response.

- Other forms

There are plenty of other variants and proposals on the wiki page.

Why @?

There is some history in Java using @ initially as a marker in Javadoc comments and later in Java 1.5 for annotations, which are similar to Python decorators. The fact that @ was previously unused as a token in Python also means it's clear there is no possibility of such code being parsed by an earlier version of Python, leading to possibly subtle semantic bugs. It also means that ambiguity of what is a decorator and what isn't is removed. That said, @ is still a fairly arbitrary choice. Some have suggested using | instead.

For syntax options which use a list-like syntax (no matter where it appears) to specify the decorators a few alternatives were proposed: `[|...|]`, `*[...]*`, and `<...>`.

Current Implementation, History

Guido asked for a volunteer to implement his preferred syntax, and Mark Russell stepped up and posted a patch to SF. This new syntax was available in 2.4a2.

```
@dec2
@dec1
def func(arg1, arg2, ...):
    pass
```

This is equivalent to:

```
def func(arg1, arg2, ...):
    pass
func = dec2(dec1(func))
```

though without the intermediate creation of a variable named `func`.

The version implemented in 2.4a2 allowed multiple `@decorator` clauses on a single line. In 2.4a3, this was tightened up to only allowing one decorator per line.

A previous patch from Michael Hudson which implements the list-after-def syntax is also still kicking around.

After 2.4a2 was released, in response to community reaction, Guido stated that he'd re-examine a community proposal, if the community could come up with a community consensus, a decent proposal, and an implementation. After an amazing number of posts, collecting a vast number of alternatives in the Python wiki, a community consensus emerged (below). Guido subsequently rejected this alternate form, but added:

> *In Python 2.4a3 (to be released this Thursday), everything remains as currently in CVS. For 2.4b1, I will consider a change of @ to some other single character, even though I think that @ has the advantage of being the same character used by a similar feature in Java. It's been argued that it's not quite the same, since @ in Java is used for attributes that don't change semantics. But Python's dynamic nature makes that its syntactic elements never mean quite the same thing as similar constructs in other languages, and there is definitely significant overlap. Regarding the impact on 3rd party tools: IPython's author doesn't think there's going to be much impact; Leo's author has said that Leo will survive (although it will cause him and his users some transitional pain). I actually expect that picking a character that's already used elsewhere in Python's syntax might be harder for external tools to adapt to, since parsing will have to be more subtle in that case. But I'm frankly undecided, so there's some wiggle room here. I don't want to consider further syntactic alternatives at this point: the buck has to stop at some point, everyone has had their say, and the show must go on.*

Community Consensus

This section documents the rejected J2 syntax, and is included for historical completeness.

The consensus that emerged on comp.lang.python was the proposed J2 syntax (the "J2" was how it was referenced on the PythonDecorators wiki page): the new keyword `using` prefixing a block of decorators before the `def` statement. For example:

```
using:
    classmethod
    synchronized(lock)
def func(cls):
    pass
```

The main arguments for this syntax fall under the "readability counts" doctrine. In brief, they are:

- A suite is better than multiple @lines. The `using` keyword and block transforms the single-block `def` statement into a multiple-block compound construct, akin to try/finally and others.
- A keyword is better than punctuation for a new token. A keyword matches the existing use of tokens. No new token category is necessary. A keyword distinguishes Python decorators from Java annotations and .Net attributes, which are significantly different beasts.

Robert Brewer wrote a detailed proposal for this form, and Michael Sparks produced a patch.

As noted previously, Guido rejected this form, outlining his problems with it in a message to python-dev and comp.lang.python.

Examples

Much of the discussion on `comp.lang.python` and the `python-dev` mailing list focuses on the use of decorators as a cleaner way to use the `staticmethod()` and `classmethod()` builtins. This capability is much more powerful than that. This section presents some examples of use.

1. Define a function to be executed at exit. Note that the function isn't actually "wrapped" in the usual sense.

   ```
   def onexit(f):
       import atexit
       atexit.register(f)
       return f

   @onexit
   def func():
       ...
   ```

   Note that this example is probably not suitable for real usage, but is for example purposes only.

2. Define a class with a singleton instance. Note that once the class disappears enterprising programmers would have to be more creative to create more instances. (From Shane Hathaway on `python-dev`.)

   ```
   def singleton(cls):
       instances = {}
       def getinstance():
           if cls not in instances:
               instances[cls] = cls()
           return instances[cls]
       return getinstance

   @singleton
   class MyClass:
       ...
   ```

3. Add attributes to a function. (Based on an example posted by Anders Munch on `python-dev`.)

   ```
   def attrs(**kwds):
       def decorate(f):
           for k in kwds:
               setattr(f, k, kwds[k])
           return f
       return decorate

   @attrs(versionadded="2.2",
          author="Guido van Rossum")
   def mymethod(f):
       ...
   ```

4. Enforce function argument and return types. Note that this copies the func_name attribute from the old to the new function. func_name was made writable in Python 2.4a3:

```
def accepts(*types):
    def check_accepts(f):
        assert len(types) == f.func_code.co_argcount
        def new_f(*args, **kwds):
            for (a, t) in zip(args, types):
                assert isinstance(a, t), \
                       "arg %r does not match %s" % (a,t)
            return f(*args, **kwds)
        new_f.func_name = f.func_name
        return new_f
    return check_accepts

def returns(rtype):
    def check_returns(f):
        def new_f(*args, **kwds):
            result = f(*args, **kwds)
            assert isinstance(result, rtype), \
                   "return value %r does not match %s" % (result,rtype)
            return result
        new_f.func_name = f.func_name
        return new_f
    return check_returns

@accepts(int, (int,float))
@returns((int,float))
def func(arg1, arg2):
    return arg1 * arg2
```

5. Declare that a class implements a particular (set of) interface(s). This is from a posting by Bob Ippolito on `python-dev` based on experience with PyProtocols.

```
def provides(*interfaces):
     """
     An actual, working, implementation of provides for
     the current implementation of PyProtocols.  Not
     particularly important for the PEP text.
     """
     def provides(typ):
         declareImplementation(typ, instancesProvide=interfaces)
         return typ
     return provides

class IBar(Interface):
     """Declare something about IBar here"""

@provides(IBar)
class Foo(object):
        """Implement something here..."""
```

Of course, all these examples are possible today, though without syntactic support.

(No longer) Open Issues

1. It's not yet certain that class decorators will be incorporated into the language at a future point. Guido expressed skepticism about the concept, but various people have made some strong arguments (search for `PEP 318 -- posting draft`) on their behalf in `python-dev`. It's exceedingly unlikely that class decorators will be in Python 2.4.

   PEP 3129 proposes to add class decorators as of Python 2.6.

2. The choice of the `@` character will be re-examined before Python 2.4b1.

   In the end, the `@` character was kept.

Copyright

Source: https://github.com/python/peps/blob/main/peps/pep-0318.rst

Last modified: 2023-09-09 17:39:29 GMT