PythonDecorators

Contents

1. **What is a Decorator**
2. **What is a Python Decorator**
3. **Debate about decorators in Python**
4. **Examples**
5. **Current Python Decorator Proposals**
    1. **A1. pie decorator syntax**
    2. **A2. pie decorator and space syntax**
    3. **B. list-before-def syntax**
    4. **C1. list-after-def syntax**
    5. **C2. list-after-def syntax with a (pseudo-)keyword**
    6. **C3. tuple-after-def syntax with a (pseudo-)keyword**
    7. **C4. tuple-after-def syntax with a % operator**
    8. **D1. list at top of function body syntax**
    9. **D2. 'dot'decorators at top of function body syntax**
    10. **E1. pie decorator at top of function body syntax**
    11. **E2. vbar decorator at top of function body syntax**
    12. **E3. vbar decorator after arg**
    13. **E4. keyword decorator at top of function body syntax**
    14. **F. inline syntax**
    15. **F2. inline syntax + new keyword (decodef for example)**
    16. **G. as decorator**
    17. **H. pie decorator using a different character**
    18. **I. angle brackets decorator syntax**
    19. **J1. new keyword decorator syntax**
    20. **J2. expand the def suite**
    21. **J3. two part def suite**
    22. **J4 two part def suite with "@" decorators**
    23. **J5 two part def suite with "@" decorators and colon**
    24. **K. partitioned syntax syntax**
    25. **L. Keyword other than as and with before def**
    26. **M. Making def an expression / letting it return a value**
6. **Decorator Syntax Breakdown**
    1. **Indicator**
    2. **Location**
    3. **List Notation**
    4. **Indentation**
    5. **Order of Decorators**
    6. **Allowable Decorators**
    7. **Meaning of decorators**
7. **Thinking ahead to Python 3 ?**

What is a Decorator

A decorator is the name used for a software design pattern. Decorators dynamically alter the functionality of a function, method, or class without having to directly use subclasses or change the source code of the function being decorated.

For more information about the decorator pattern in general, see:

- **http://wiki.cs.uiuc.edu/patternStories/DecoratorPattern**
- **http://en.wikipedia.org/wiki/Decorator_pattern**

What is a Python Decorator

The "decorators" we talk about with concern to Python are not exactly the same thing as the **DecoratorPattern** described above. A Python decorator is a specific change to the Python syntax that

allows us to more conveniently alter functions and methods (and possibly classes in a future version). This supports more readable applications of the **DecoratorPattern** but also other uses as well.

Support for the decorator syntax was proposed for Python in **PEP 318**, and will be implemented in Python 2.4.

Note that the current proposal actually only decorates functions (including methods). Extending it to classes or even arbitrary code is possible, but Guido wasn't sure it made sense. (Later versions might become more permissive, but they can't easily snatch functionality back.)

Debate about decorators in Python

The winning syntax as of now uses the '@' symbol, as described in **this message**. Mark Russell implemented this version. **Here** is the message describing the patch he checked in.

There has been a long discussion about the syntax to use for decorators in Python.

Examples

Toggle line numbers

```
1
2 @classmethod
3 def foo (arg1, arg2):
4     ....
```

See **PythonDecoratorLibrary** for more complex and real-world examples. See also **MixIns** and **MetaClasses** for related resources.

Current Python Decorator Proposals

See **section 6** for a categorization of different proposals; this section just provides concrete examples.

Decorator Poll (consider the poll now closed)

Here is an online poll where you can vote between a few different alternative syntaxes for python decorators: **http://wiki.wxpython.org/index.cgi/PythonDecoratorsPoll**

A more complete poll is currently running on comp.lang.python, with the unfortunate name of "Alternate decorator syntax decision", using the options on this WikiPage as the candidates. Please visit that thread and express your preference!

After the @decorator syntax was "accepted", lots of people threw up alarms and a huge series of threads started exploding on Python-dev. Here are the current alternatives that I could find that are being argued, with pros and cons.

I give two examples that might be common uses in the future. Classmethod declarations, and something like static typing (adapters), declaring what type parameters a function expects and returns.

A1. pie decorator syntax

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

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

- + Implementation already exists (and is in Python 2.4a2)
- + Java-like, so not completely unknown to everyone.
- + Makes the syntax obvious visually (i.e., obviously not a normal statement)
- + Will not be silently ignored
- + Compile-time
- + One decorator per line (makes it clearer to read, write and change order of decorators)
- + Separate from the def syntax (desired by some for making decoration stand out and keeping def the same)
- + Currently not used in Python so @decorators can be inserted anywhere, such as after parameters or inside expressions.

- + All decorators line up in one column immediately above the function name (easy to browse and see what's going on).
- + The @ special character will make syntax highlighting easier than it would be for normal statements in "magic" locations.
- - Separate from the def syntax (undesired by some for simple decorators like classmethod/staticmethod)
- - Ugly?
- - Introduces a new character in the language
- - The @ special character is used in IPython and Leo
- - Punctuation-based syntax raises Perlfears. But no "obvious" keyword or expression has been suggested.
- - @ is a relatively arbitrary character choice; no intuitive reason why it would indicate a decorator
- o The @ character is often used (in other languages) to mean "attribute". For annotations, this is good. For more active decorators, it may not be so good.
- - Some people would like the @ character to be available as an overloadable operator - for example, as a binary operator for matrix multiplication (element-by-element and matrix multiplication may be usefully applied to a single object, so there's arguably a need for a new operator rather than simply overriding binary * on a Matrix class).
- - Because of no indentation, looks confusing when definitions are not separated by empty lines. But adding empty lines makes it hard to determine where the function definition truly starts.
- - That's the first case in Python where a line following another one with the same identation has a meaning.
- - Breaks in interactive shell
- - Comes before the def keyword. Supercedes the function declaration itself, thus applying modifications implicitly to something not yet defined.
- - @staticmethod may look like staticmethod is not a defined variable, but a compile time option
- - Requires the programmer to scan an arbitrary distance from the def in *both* directions to see everything of interest (arg, decs, etc.).
- - Cannot be "folded" by editors as easily, as an arbitrary number of prefix lines should be folded with the body (otherwise, folding becomes less useful).
- - While the Language Spec does not promise that this character won't be used, neither does it reserve *any* character for user extensions. '@' is one of only three that are available, and another '$' is also likely to get used up by 2.4

FWIW, here is Guido's jumble example in this syntax.

```
class C(object):

    @staticmethod
    @funcattrs(grammar="'@' dotted_name [ '(' [arglist] ')' ]",
               status="experimental", author="BDFL")
    def longMethodNameForEffect(longArgumentOne=None,
                                longArgumentTwo=42):
        """This method blah, blah.

        It supports the following arguments:
        - longArgumentOne -- a string giving ...
        - longArgumentTwo -- a number giving ...

        blah, blah.

        """
        raise NotYetImplemented
```

And here is an example taken from the current test_decorators.py. This exposes the problem of using two lines together with some meaning but without identation or vertical whitespace.

```
class TestDecorators(unittest.TestCase):

    ...

    def test_dotted(self):
        decorators = MiscDecorators()
        @decorators.author('Cleese')
        def foo(): return 42
        self.assertEqual(foo(), 42)
        self.assertEqual(foo.author, 'Cleese')
```

A2. pie decorator and space syntax

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

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

```
    ...
```

* o This is more readable for some people, less readable for others.

B. list-before-def syntax

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

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

- + Implementation already exists
- + C# like
- + Can be made backwards compatible-ish, with a "hack"
- + Doesn't cause breakage in existing code-searching tools
- + Use visually existing syntax
- - Doesn't cause breakage in existing code-searching tools
- - Would not work in interactive mode (list would be interpreted right away)
- - **EuroPython** didn't like it (why?) (hard to teach newbies about the magic)
- - The backwards compatability wouldn't be portable to Jython
- - Looks like a normal expression, but has "magic" behavior of altering a function object
- - Breaks in interactive shell
- - Harder to highlight (looks like a normal list)

C1. list-after-def syntax

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

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

- + Implementation already exists
- + Also somewhat C#-like
- + Was a "community favorite" at one time
- + Clearly a part of function declaration
- + Looks ok for simple decorators such as classmethod
- + Won't break simplistic code analyzers or grep for function def
- + Use visually existing syntax
- + Allows one-liner definitions
- - Long lists of decorators/arguments cause ugly line wraps
- - Little to distinguish it visually from argument list
- - or o For long argument list, decorators are very far from def. *I see being too close to def as a minus.*
- - Guido hates it because it hides crucial information after the signature, it's easy to miss the transition between a long argument list and a long decorator list, and it's cumbersom to cut and paste a decorator list for reuse.
- - Creates another meaning for []. Since it does so inside the function definition, it will be a distraction for beginners.
- o Brackets are used in other fields to indicate an annotation of some sort (but in Python, it doesn't)

I don't see why longs lists of decorators are an issue with this syntax. Consider the following example:

```
def foo(arg1, arg2) [
    complicated(manyArgs=1, notTooUgly='yes'),
    even_more_complicated(42)]:
    ...
```

That doesn't look particularly ugly to me.

It also isn't very long.

Here is an **example Guido just sent to python-dev**:

```
class C(object):
```

```
    def longMethodNameForEffect(longArgumentOne=None,
                                longArgumentTwo=42) [
        staticmethod,
        funcattrs(grammar="'@' dotted_name [ '(' [arglist] ')' ]",
                  status="experimental", author="BDFL")
        ]:
        """This method blah, blah.

        It supports the following arguments:
        - longArgumentOne -- a string giving ...
        - longArgumentTwo -- a number giving ...

        blah, blah.

        """
        raise NotYetImplemented
```

And he editorializes:

```
That's a total jumble of stuff ending with a smiley.  (True story: I
left out the colon when typing up this example and only noticed in
proofing.)

Problems with this form:

- 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

Given that the whole point of adding decorator syntax is to move the
decorator from the end ("foo = staticmethod(foo)" after a 100-line
body) to the front, where it is more in-your-face, it should IMO be
moved all the way to the front.
```

C2. list-after-def syntax with a (pseudo-)keyword

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

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

This combines C1 with a keyword; it general, it has all the advantages of either, so I will only list those that are unique to the combination.

- + Groups the decorators with a list, but explains what they are doing, so the list no longer has a magical new meaning.
- + Easily extended; No special characters are "used up", and in the future, other pseudo-keywords could be added.
- + The pseudo-keyword makes it easier to see the separation between the argument tuple and the decorator list.
- + The pseudo-keyword can act as an implicit line-continuation, which helps with (but does not solve) the midline problem.
- - Some proponents objected to adding a keyword, because of more typing.
- 0 Makes the decoration look like a second-class or optional part of the definition. This is true, but may have caused some emotional objection.
- - There was very little agreement on which word should be used.

- - No implementation currently exists (it *may* be a simple variation of the implementation of C1).

FWIW, here is Guido's jumble example in this syntax.

```
class C(object):

    def longMethodNameForEffect(longArgumentOne=None,
                                longArgumentTwo=42) using
        [staticmethod,
         funcattrs(grammar="'@' dotted_name [ '(' [arglist] ')' ]",
                   status="experimental", author="BDFL")]:
        """This method blah, blah.

        It supports the following arguments:
        - longArgumentOne -- a string giving ...
        - longArgumentTwo -- a number giving ...
```

```
        blah, blah.

        """
        raise NotYetImplemented
```

Without the pseudo-keyword acting as line continuation it reads :

```
class C(object):

    def longMethodNameForEffect(longArgumentOne=None,
                                longArgumentTwo=42) using [
         staticmethod,
         funcattrs(grammar="'@' dotted_name [ '(' [arglist] ')' ]",
                   status="experimental", author="BDFL")]:
        """This method blah, blah.

        It supports the following arguments:
        - longArgumentOne -- a string giving ...
        - longArgumentTwo -- a number giving ...

        blah, blah.

        """
        raise NotYetImplemented
```

which feel is more consistent with the rest of python parsing-wise, without decreasing readability...

(See also J4 below, which moves the keyword, and uses "@" signs to make the decorators stand out more.)

C3. tuple-after-def syntax with a (pseudo-)keyword

```
def foo(arg1,arg2) using classmethod,:
    ...

def bar(low,high) using accepts(int,int), returns(float):
    ...
class C(object):

    def longMethodNameForEffect(longArgumentOne=None,
                                longArgumentTwo=42) using (
         staticmethod,
         funcattrs(grammar="'@' dotted_name [ '(' [arglist] ')' ]",
                   status="experimental", author="BDFL")):
        """This method blah, blah.

        It supports the following arguments:
        - longArgumentOne -- a string giving ...
        - longArgumentTwo -- a number giving ...

        blah, blah.

        """
        raise NotYetImplemented
```

Very similar to C2, but with those slight differences

- + Nicer for one-line decoration when using multiple decorators.
- - The hanging comma feel strange for 1-element tuple, and will probably often been forgotten.
- - Decorator and arguments looks (too?) similar for multiline case.
- - No implementation currently exists.

The 1st drawback could be removed if one allows both tuple and single-element after the pseudo-keyword, trading consistency for readability and convenience.

C4. tuple-after-def syntax with a % operator

```
def foo(arg1,arg2) % classmethod:
    ...

def bar(low,high) % accepts(int,int), returns(float):
    ...
class C(object):

    def longMethodNameForEffect(longArgumentOne=None,
                                longArgumentTwo=42)            # make implicit linebr
         % (staticmethod,
```

```
            funcattrs(grammar="'@' dotted_name [ '(' [arglist] ')' ]",
                      status="experimental", author="BDFL")):
        """This method blah, blah.

        It supports the following arguments:
        - longArgumentOne -- a string giving ...
        - longArgumentTwo -- a number giving ...

        blah, blah.

        """
        raise NotYetImplemented

# this is also possible for consistency:

foo %= classmethod
bar %= (accepts(int,int), returns(float))
```

Very similar to C3, but with those slight differences

- + Nicer for one-line decoration when using multiple decorators.
- + Similar to use of % in string formatting operation
- - Decorator and arguments looks (too?) similar for multiline case.
- - No implementation currently exists.

One more point: % could also be used in chained fashion:

```
bar = bar % accepts(int,int) % returns(float)
```

(making it similar to E2 below)

D1. list at top of function body syntax

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

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

- + Also somewhat C#-like
- + Consistent with how docstrings are used.
- + Looks ok for simple or complex decorators
- + Won't break simplistic code analyzers or grep for function def
- + Solves line wrap problem with above proposal

- 0 There is a hack that implements this now **here**.

- - Guido's europython presentation said this didn't win out, but not why
- - Adds 'magic' behavior to a normal python expression (lists). Not exactly true: there is nothing magic in string when it's used in docstring - it's a normal string in the "magic" place.
- - Compatibility issue: program that is working under 2.4 will not work properly under earlier versions without any explanation (old syntax compatible decorators will not blow in your face)
- 0 Perhaps decorators should be allowed before or after the docstring. If you have to choose, I'd choose making it before the docstring.
- - No implementation currently exists.

- - Guido ruled out any solution involving special syntax inside the block, because "you shouldn't have to peek inside the block to find out important external properties of the function." **http://mail.python.org/pipermail/python-dev/2004-August/047279.html**

- - **Guido rejects**, because the function body should reflect what the function does; decorators are "outside" the function, for its callers.

D2. 'dot'decorators at top of function body syntax

```
def bar(low,high):
    .accepts(int,int)
    .returns(float)
    """docstring"""
    pass

def longMethodNameForEffect(longArgumentOne=None,
                            longArgumentTwo=42):
    .staticmethod
```

```
    .funcattrs(grammar="'@' dotted_name [ '(' [arglist] ')' ]",
               status="experimental", author="BDFL")
    """
    asdfasdf
    """
    raise NotYetImplemented
```

Advantages/disadvantages of .decorators:

- + Unambiguous "target" for decorators, matches Python's precedents for indentation and modify-what-came-before
- + Decorators won't get lost in long argument lists (they are indented differently in Guido's preferred formatting, and are separated by the smiley in the all-args-on-individual-lines formatting)
- o Compatible with some future "with ...:" syntax, as decorators must immediately follow a 'def ...:' (or possibly a 'class ...:'), so if there is a 'with ...:' what follows cannot be a decorator.
- + Separate from def syntax (to keep def the same avoiding breakage)
- + No extra cut-and-paste issues, no extra indentation level.

- + No smileys. (Does that get a  or a]: ?)

- + Will not be silently ignored (doesn't use currently legal syntax)
- + Simple for simple decorators, supports complex decorators as cleanly as possible
- + Less ugly (YMMV). Doesn't feel like executable line noise (to 1 out of 1 pythonistas polled...)
- + No new punctuation or keywords (avoids breaking Leo/IPython/existing code)
- + Really nice syntax for assigning attributes, if desired (Not sure I like this overloading, but it /looks/ good)

```
def func():
    .author = "Kevin Butler"
    pass
```

- +/o Syntax obvious visually (Someone will complain that the leading period gets lost - that person should switch to a fixed-width font, as used when coding. <.5 wink>) Easy to highlight.

- o Although it is a punctuation-based syntax, it is compatible with existing/proposed '.' usage ('.' can mean subordination to a containing construct like 'with', and passing the implicit "func" argument is analogous to passing "self")
- o Perhaps decorators should be allowed before or after the docstring. If you have to choose, I'd choose making it before the docstring.
- - Minor extension to use of '.'
- - Some people may have too much dust on their monitors

- + Could use `.doc()` or `.doc =` or `.__doc__ =` as a docstring alternative

- - Guido ruled out any solution involving special syntax inside the block, because "you shouldn't have to peek inside the block to find out important external properties of the function." **http://mail.python.org/pipermail/python-dev/2004-August/047279.html**

- - **Guido rejects**, because the function body should reflect what the function does; decorators are "outside" the function, for its callers.

E1. pie decorator at top of function body syntax

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

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

- Same as above but with pie syntax.
- Could use @doc too as a docstring alternative
- - No implementation currently exists.

- - Guido ruled out any solution involving special syntax inside the block, because "you shouldn't have to peek inside the block to find out important external properties of the function." **http://mail.python.org/pipermail/python-dev/2004-August/047279.html**

- - **Guido rejects**, because the function body should reflect what the function does; decorators are "outside" the function, for its callers.

E2. vbar decorator at top of function body syntax

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

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

def longMethodNameForEffect(longArgumentOne=None,
                            longArgumentTwo=42):
    |staticmethod
    |funcattrs(grammar="'@' dotted_name [ '(' [arglist] ')' ]",
               status="experimental", author="BDFL")
    """This method blah, blah.

    It supports the following arguments:
    - longArgumentOne -- a string giving ...
    - longArgumentTwo -- a number giving ...

    blah, blah.

    """
```

- \+ Consistent with how docstrings are used.
- \+ Looks ok for simple or complex decorators
- \+ Won't break simplistic code analyzers or grep for function def
- \+ Vertical bars visually "attach" the decorators to the name.
- \+ Decorators are indented, so it's clear that they modify the function.
- \+ Reads naturally as "pipe" operator to unix hackers (which is semantically correct, since the defined function gets passed through the decorators, one at a time, and the result is used.
- \+ Doesn't use up one of the currently unused characters (such as "@") -- it's always possible that we'll find another good use for those later.
- o Perhaps decorators should be allowed before or after the docstring. If you have to choose, I'd choose making it before the docstring.
- \- "|" is a relatively arbitrary character choice; no intuitive reason why it would indicate a decorator
- \- Indented decorators look like they should be evaluated when the function is called, not when the function is parsed. I have the same objection to docstrings, though.
- \- Misleading to Unix hackers, since the order of evaluation is "backwards".
- \- For some fonts, "|" looks similar to "I" or "l" or "1". Some think that code highlighting would remove this problem, but with normal-sized fonts, there are not many pixels in it to show a color clearly. "@" has a big blob of pixels, and is very distinct. If proper style uses a space after the "|", the decoration will be a syntax error if I, 1, l (or !) are used.
- \- The key with "|" on it is often in an awkward location on laptop keyboards
- \- No implementation currently exists.
- \- When using just one bar, it isn't noticable enough. It seems more a part of the actual decorator function name than something demarking the function.
- \- When using multiple decorators, the pattern formed by the vertical bars draws the eyes too much and makes it hard to focus on the signature.

- \- Guido ruled out any solution involving special syntax inside the block, because "you shouldn't have to peek inside the block to find out important external properties of the function." **http://mail.python.org/pipermail/python-dev/2004-August/047279.html**

- \- **Guido rejects**, because the function body should reflect what the function does; decorators are "outside" the function, for its callers.

Restyled version:

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

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

def longMethodNameForEffect(longArgumentOne=None,
                            longArgumentTwo=42):
    | staticmethod
    | funcattrs(grammar="'@' dotted_name [ '(' [arglist] ')' ]",
               status="experimental", author="BDFL")
    """This method blah, blah.

    It supports the following arguments:
    - longArgumentOne -- a string giving ...
```

```
    - longArgumentTwo -- a number giving ...

    blah, blah.

    """
```

E3. vbar decorator after arg

```
def longMethodNameForEffect(longArgumentOne=None,
                            longArgumentTwo=42)
    |staticmethod
    |funcattrs(grammar="'@' dotted_name [ '(' [arglist] ')' ]",
               status="experimental", author="BDFL"):
    """This method blah, blah.

    It supports the following arguments:
    - longArgumentOne -- a string giving ...
    - longArgumentTwo -- a number giving ...

    blah, blah.

    """

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

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

An alternative (inspired by a typing error I corrected in E2 in the Guido example) would be to put vbar decorator before the colon... Basically it has the same characteristic than E2, with the following slight differences:

- + More obviously attached to function definition
- + Possible 1-line version as readable as the inline syntax (see F.) for short function/decoration
- - More difficult to parse? The end-of-statement rules in Python mandate the colon be on the same line as the close of the argument list (less any line continuations); this syntax breaks that rule. Either the rule would have to be changed (yikes!), or line continuations would have to be used.
- - New syntax having no equivalent in other part of python (but all @ / | propositions suffer from that)
- - No implementation currently exists.

- - Guido ruled out any solution involving special syntax inside the block, because "you shouldn't have to peek inside the block to find out important external properties of the function." **http://mail.python.org/pipermail/python-dev/2004-August/047279.html**

- - **Guido rejects**, because the function body should reflect what the function does; decorators are "outside" the function, for its callers.

E4. keyword decorator at top of function body syntax

```
def foo(arg1,arg2):
    using classmethod
    ...

def bar(low,high):
    using accepts(int,int)
    using returns(float)
    ...

def longMethodNameForEffect(longArgumentOne=None,
                            longArgumentTwo=42):
    using staticmethod
    using funcattrs(grammar="'@' dotted_name [ '(' [arglist] ')' ]",
               status="experimental", author="BDFL")
    """This method blah, blah.

    It supports the following arguments:
    - longArgumentOne -- a string giving ...
    - longArgumentTwo -- a number giving ...

    blah, blah.

    """
```

Similar to E1 and E2 but with keyword

F. inline syntax

```
def classmethod foo(arg1,arg2):
    ...

?
```

- + Simple
- + More readable/natural
- + Obviously attached to the function
- - Does not allow for arguments to the decorator inline, or multiple decorators
- - The natural place where everyone looks for the function name now is a possible container for other information
- - Complicates things like colorization and other functions of helper tools
- - Many people don't like the idea of having something between 'def' and the function name
- - Breaks etags (couldn't it be fixed?)
- - No implementation currently exists.

F2. inline syntax + new keyword (decodef for example)

```
decodef doo:
    classmethod
    funcattrs(...)

def [doo] foo(arg1,arg2):
    ...
```

G. as decorator

```
as classmethod
def foo(arg1,arg2):
    ...

?
```

- + Non-punctuation based
- + Does not use an existing mechanism with 'magic' behavior
- - Guido specifically vetos: "as" means "rename" in too many logical, common places that are
- - No implementation currently exists.

H. pie decorator using a different character

For example, using the '|' character:

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

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

Same pros and cons as @decorator, but additionally:

- + Likely to be a trivial change to what is already in 2.4a2.
- + It doesn't break Leo, IPython, or any other tool that uses @ as a special character.
- + The association with pipes makes some sense: "take this thing and pass it through that thing to get a modified thing".
- + Visual association with function also follows from the vertial continuous line that forms, sort of like a flagpole attached to the function and each decorator like a flag.
- 0 Less 'in-your-face' than @ (some claim it blends in too much, and others, that it's a good thing)
- - Most characters (including pipe) already have a meaning. Ending an expression at the linebreak will probably keep these from becoming ambiguous, but ... that gets fragile.
- - Misleading to Unix hackers, since the order of evaluation is "backwards".
- - For some fonts, "|" looks similar to "I" or "l" or "1" Some think that code highlighting would remove this problem, but with normal-sized fonts, there are not many pixels in it to show a color clearly. "@" has a big blob of pixels, and is very distinct. If proper style uses a space after the "|", the decoration will be a syntax error if I, 1, l (or !) are used.
- - The key with "|" on it is often in an awkward location on laptop keyboards
- - When using just one bar, it isn't noticable enough. It seems more a part of the actual decorator function name than something demarking the function.

- - When using multiple decorators, the pattern formed by the vertical bars draws the eyes too much and makes it hard to focus on the signature.

Restyled version:

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

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

I. angle brackets decorator syntax

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

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

- + Same advantages of Pie decorator syntax
- + Doesn't need a new character
- - Angle brackets are "unpaired" characters
- - Parsing of greater-than and less-than becomes more fragile.
- - No implementation currently exists.

J1. new keyword decorator syntax

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

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

- + Uses widely known python syntax
- + Doesn't need extra characters with special meaning
- + Allows many decorated functions to be declared with a single statement
- - New keyword
- - Increases minimum ident level on decorated functions (see following question)
- - Inconsistent identation level between methods with/without decorators (see following question)
- - way to much indenting if there are multiple decorators. (see following question)
- - Determining which decorators apply to a function requires scanning an arbitrarily large amount of code (with real functions and nesting, the decorator can be literally hundreds of lines away from the def).

- - In practice, functions are likely to each require slightly different decorator sets (*other may argue that most users would only use staticmethod and don't care grouping them*), which means that we either do not group them (in which case, the new indent level doesn't gain us anything), or functions will be heavily and unevenly nested. (See below for an example.)

- - No implementation currently exists.

Wouldn't be possible to allow both syntaxes?:

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

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

Here's an example of run-away nesting (imagine trying to figure out which decorators apply to baz if these were non-trivial functions) (*answer: but you can do the same with if, and it doesn't mean you have to... The main point of this proposal is to allow one level of indentaton, in which case it is clear, but having the drawback of the indentation. If you find three level of indentation unclear, why write code that way? No other proposition has this "grouping" capacity anyway...*) (*answer answer: The original point was only that they can all be abused, but that this syntax can be abused in a way that is unique, simply* ***because*** *of the grouping capability.*):

```
 decorate static, synchronized:
   decorate returns(None):
```

```
    decorate accepts(int):
      def foo(a):
        pass
    decorate accepts(int, int):
      def bar(a, b):
        pass
  decorate accepts(), returns(int):
    def baz():
      return 0
```

J2. expand the def suite

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

using:
    """
    It is now clear the docstring will
    survive the decoration process
    """
    accepts(int,int)
    returns(float)
def bar(low,high):
    ...
```

An extensive paper has been written by Robert Brewer: **Optimal Syntax for Python Decorators**. A patch implementing this syntax is available here: **http://www.python.org/sf/1013835**.

(I have changed the keyword in the second example to "using", to match the recommendation in Robert's paper. I have left the first example alone, to indicate that the precise choice of keyword is independent of the use of a suite.)

Note that this differs from J5 (and largely from J4) only in whether the function signature stays with the function body, or moves to the top suite. (And in whether the redundant "@" is added for extra readability.)

- + Uses widely known python syntax, albeit in an unusual way
- + Reads well and naturally
- + Doesn't need extra characters with special meaning
- + Does not use up one of the few remaining "no meaning anywhere" characters.
- + Docstring can be moved to Guido's favourite place, i.e. before the def suite
- + Moves decorators, signature and implementation closer together for functions with a long docstring
- + With the right keyword, e. g. "transform", a new user gets a clear notion of the suite's purpose
- + Can smoothly be extended to a full-blown suite allowing for `if __debug__: trace` etc.
- + Basic implementation has been posted here: **http://mail.python.org/pipermail/python-list/2004-August/233591.html**
- - Requires new keyword
- (- No one can agree on the right keyword to use.) The consensus is "using".
- - Keyword chosen may conflict with how keyword is used in other languages (like "using" in C# or "extends" in Java and other languages).
- - Keyword may be confused with other keywords in current and future versions of Python (like "with" in **Python3.0**, which is very similar to "using"). This would suggest choosing a less generic keyword like "decorate", "meta", or "predef".
- (- Indented decorators put them visually in line with the function name. When indented below the function declaration line this isn't a big deal, but right above the function declaration it can be a problem.) This is so minor that it ought to be ignored.
- (- Much more typing required than A1 pie syntax the more decorators you have.) But less shifting. (You also have to do more indentation checking.) Any decent editor does this with one keystroke. Try typing the two:

```
using:
    classmethod
    accepts(int,int)
    returns(float)
def bar(low,high):
    ...
```

```
vs.

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

- - Overkill for the simple case like classmethod (*not that much, since "@" can be replaced by "using:" without new line*). (Two extra lines still feels like overkill to me; a line with an internal colon is less overkill, but has its own style problems.)

- - Many people felt it was wrong use of an identation suite. (*Someone please explain. It looks just like any other pair of blocks in Python, such as if/else or try/finally.*) In current python, an indent either adds a new namespace (this doesn't, it modifies the parent namespace where the def itself appears) or changes flow control. I do think it changes flow control as much as an if or a loop, but others felt strongly that it wasn't the same, and that the equivalence of decorators within a decorate block was different from the equivalence of statements in a function block. For a sample of the response, search the python-dev archives around March 2004. Private email was more adamant.

- - Has the same problem as option A: in that the decorate block implicitly affects the following def. This does not occur elsewhere in Python. -- Or more precisely, it doesn't occur with an indented block syntax; parentheses can affect execution order. The proposed with: block would do the same thing. *No, the with block wouldn't affect blocks that come after the with block like this does.*

- - Same problem as A1 and other proposals, in what order are the decorators applied? C1 is just about the only proposal that doesn't suffer from this problem.

- - Where will the docstring go? Guido wants the docstring before the function. (He said it would be nice *if* it went there, that's all.) Does it **have** to be at the beginning of the decorate: indentation block? Or would it **have** to be at the end? Or where? What if the function has no decorators, would you have to still use "decorate" followed by an indented docstring? (Not before Python 3000, if ever.)

- - Technical problems with the current grammar parser if a suite *starts* with an optional part. (Ending with an optional part, such as "else:" is OK, but starting with one is not.) (*Michael Sparks has experimented with the grammar and based on his results it appears *not* an issue.*)

The implementation referenced above does NOT allow a generic suite - but only allows decorators to be listed. The semantics wrt to ordering remains the same as in 2.4a2

(Instead of the word "decorate" it is possible to use some other less lengthy keyword. And even DEF taken uppercase.)

J3. two part def suite

```
def qux:
    """docstring could be here or below. Two strings probable"""
    decor1
    decor2
from arg1, arg2:
    ...

def quux staticmethod from (arg1, arg2):  return arg1 + arg2 # possible one-liner (see

# which probably is not much needed because of shorter legacy:
quux = staticmethod(lambda arg1, arg2: arg1 + arg2)

# unstylish, but still possible:
def quuux: decor1; decor2
from x, y: return x + y
```

Same as J2 but has advantages:

- + doesn't require new keyword (re-uses "from")
- + one-liner has special different syntax

- + reads as natural language: "define quux [with] <these decorators> from <this function>

- - loses the "definition reflects use"; you can no longer cut and paste from a call to the definition, or vice versa, because the name is not near the arguments.

J4 two part def suite with "@" decorators

(Note: the missing colon is *intentional*. See J5 for version with colon.)

```
   def func(self, arg1)
       @staticmethod
       @grammarrule('statement : expression')
       @version("Added in 2.4")
   as:
       """Docstring could be here, or in decorator part above"""
       # body goes here
```

Note that this differs from J5 only by the colon. J5 in turn differs from J2 in whether the function signature stays with the function body, or moves to the top suite. (And in whether the redundant "@" is added for extra readability.)

- + doesn't require new keyword ("as" can be unambiguously parsed in context)
- + Leverages the visibility of "@" to overcome visibility problems of list-after-def syntaxes
- + Even better definition-reflects-usage than the current method
- + Adds a well-defined header, but still backwards-compatible, because of the @ leading each line in the header. (So a non-@ line means an old-style definition, rather than a header.)
- + Gives the option of putting the docstring as the final decorator, so that wrapping decorators don't have to take care to restore it.
- - Nothing to do with signification of "as" in other languages and Python. (Other words might work, but it isn't clear that any are better. Maybe "from"?)
- - There is no line-continuating character, and the indentation cannot be enforced. (Not true! It *can* be enforced; the proposed grammar calls for an INDENT/DEDENT pair wrapping the decorators. The actual downside is that it will be the first such required indent that is not preceded by a colon.)
- + The lack of a colon emphasizes that the following items (decorators) are semantically closer to an inline list than a full-blown suite.

```
if (test):   # They match the suite, rather than the statements
    stmt1
    stmt2
```

- - Some may argue that the "as" is not necessary and that the syntax is very similar to a in-body syntax. (The unindented "as" is precisely what separates this from the in-body syntax. To me, the difference is startling.)

J5 two part def suite with "@" decorators and colon

While the missing colon of J4 was intentional, there are good arguments both ways. Since we're listing options, I include this variant of J4.

```
   def func(self, arg1):
       @staticmethod
       @grammarrule('statement : expression')
       @version("Added in 2.4")
   as:
       """Docstring could be here, or in decorator part above"""
       # body goes here
```

Note that J5 differs from J2 only in whether the function signature stays with the function body, or moves to the top suite. (And in whether the redundant "@" is added for extra readability.)

- + doesn't require new keyword ("as" can be unambiguously parsed in context)
- + Leverages the visibility of "@" to overcome visibility problems of list-after-def syntaxes
- + definition-reflects-usage equal to current function definitions.
- + Adds a well-defined header, but still backwards-compatible, because of the @ leading each line in the header. (So a non-@ line means an old-style definition, rather than a header.)
- + Gives the option of putting the docstring as the final decorator, so that wrapping decorators don't have to take care to restore it.
- - Nothing to do with signification of "as" in other languages and Python. (Other words might work, but it isn't clear that any are better. Maybe def: ... from: ?)
- + Colon/linebreak consistency with the rest of the language.
- - Some may argue that the "as" is not necessary and that the syntax is very similar to a in-body syntax. (The unindented "as" is precisely what separates this from the in-body syntax. To me, the difference is startling.)

K. partitioned syntax syntax

- Use pie-decorator syntax (or some other complex syntax) when arguments are to be passed
- use inline syntax when no arguments are necessary

```
def classmethod foo(arg1,arg2):
    ...
```

- + Simple for simple cases, poweful when needed
- + Obviousely attached to the function
- - The natural place where everyone looks for the function name now is a possible container for other information
  This is debatable. "Natural" will change if this is accepted. The natural place to find the fnction name will be after any simple decorators and before the argument list.
- - Complicates things like colorization and other functions of helper tools
  Other syntaxes will need to be colorized too and will thus complicate colorization.
- - Since both methods are legal, it has all the downsides of either syntax, in terms of what it does to the rest of the language or newbie confusion.
- - No implementation currently exists.

L. Keyword other than as and with before def

```
using classmethod def foo(arg1,arg2):
    ...

using accepts(int,int)
using returns(float)
def bar(low,high):
    ...

other possible keyword:

predef classmethod
def foo(arg1,arg2):
    ...

predef accepts(int,int)
predef returns(float)
def bar(low,high):
    ...
```

- + Most advantages of @decorators.
- + No special character
- + Reads in english well
- - A lot of the drawback of @decorators
- - New keyword

- - No implementation currently exists (*very similar to A1 implementation, no?*).

M. Making def an expression / letting it return a value

```
deco1(deco2(deco3(
    def func(arg1, arg2):
        pass
)))

    deco1(

    def method(self, arg):
        pass
    )
```

- + Puts everything in the right place.
- + Because of the (), the function definition can be (but does not need to be) indented an extra level, depending on what makes sense.
- + Familiar from other functions-as-first-class-objects languages, such as LISP.
- - The delayed close-paren is better than the current format, but still not desirable.
- - Guido does not want to blur the distinction between statements and expressions.
- o The name would be temporarily bound to the unwrapped function, which changes semantics for some types of wrappers, such as generics and properties.

Decorator Syntax Breakdown

Here's a breakdown of some of the different decisions that have to be made in deciding on a decorator syntax. This is an attempt to consolidate some of the common points from the various examples above.

Indicator

Proposals differ on how some sort of indicator of "decoratorhood" is use. These include:

- Keyword
  - + Makes it clear that this is not a normal statement
  - + Will not be silently ignored
  - + Makes syntax highlighting easy
  - o Will make the semantics clear -- but may therefore be misleading for some types of decorators. (Can you think of a word that works well with all of classmethod, annotations, adding a transaction capability, registering the function with a generic method, and replacing the function with something else depending on runtime environment? If so, you're partway there.)
  - - New keywords can break existing code
  - *Some keyword options*:
    - **A. decorate**
      - + Not confusable with any current Python keyword
      - + Intuitively associated with decorator semantics
    - **B. transform**
      - + Not confusable with any current Python keyword
      - + Clear indication that a function may change its signature and effect, not only its attributes
      - + Avoids confusion with the "decorator pattern" which requires interface conformance
    - **B1. mutate**
      - + Not confusable with any current Python keyword
      - + Clear indication that a function may change its signature and effect, not only its attributes
      - + Avoids confusion with the "decorator pattern" which requires interface conformance
    - **C. predef**
      - + Not confusable with any current Python keyword
      - + Intuitively associated with function definition semantics
      - o Somewhat intuitively associated with decorator semantics
    - **D. using**
      - + Not confusable with any current Python keyword
      - o Somewhat intuitively associated with decorator semantics
      - - Users of C++ will associate it with namespaces
    - **E. meta**
      - + Not confusable with any current Python keyword
      - o Associates meta(-classes) with decorators
      - - Suggests (only) metadata, not changes of signature and effect
    - **F. with**
      - + Not confusable with any current Python keyword
      - - 'with'-blocks, as proposed, already want this keyword.
    - **G. as**
      - - 'as' usually means 'rename'
    - **H. wrap**
      - + Not confusable with any current Python keyword
      - + Clear indication of what the actual implementation does - 'wraps' the function with the decorators

        *(I think this is a -, since it binds the syntax too closely to the actual implementation - **PaulMcGuire**)*

      - + Avoids confusion with the "decorator pattern" which requires interface conformance

        - **I. qual**
            - + Not confusable with any current Python keyword
            - + Evokes "quality" or "qualifier" concepts, but is not limited to either
            - o Does not have any meaning itself
- Symbol
    - + Makes it clear that this is not a normal statement
    - + Will not be silently ignored
    - + Saves keystrokes
    - - Introduces a new character in the language. This either overloads an old character with new meaning, or uses up an increasingly scarce resource. (There are only three unused characters now, and none are reserved to the user.)
    - *Some symbol options*:
        - **@**
            - + Java-like, so not completely unknown.
            - + Not confusable with any current Python symbol
            - + Makes syntax highlighting easy
            - o Used in other languages to mean "attribute"
            - - Already used as a special character in IPython and Leo
            - - Not intuitively associated with the decorator semantics
        - **|**
            - + Vertical bars visually "attach" the decorators to the name (with a vertical ASCII art line).
            - o Reminiscent of Unix "pipe" operations (though order may be "backwards")
            - - Not intuitively associated with the decorator semantics
            - - Gives substantially new meaning to '|', may be confusing for beginners. Adds new obfuscation possibilities depending on line continuations.
            - - For some fonts, "|" looks similar to "I" or "l" or "1".
            - - The key with "|" on it is often in an awkward location on laptop keyboards
            - - Narrow character, may not be obvious enough
        - **%**
            - + Already associated in Python with some sort of "application" type semantics.
            - - Not intuitively associated with the decorator semantics
- Function
    - + Does not require a syntax change
    - - Changes semantics of function calls. (A function call followed by a function definition is not usually assumed to be affecting that definition.)
- None
    - + With list syntax, C# like, so not completely unknown.
    - - In pre-def and post-def locations, looks like a normal statement, but does not behave like one.
    - - In pre-def and post-def locations, may be silently ignored
    - - In pre-def and post-def locations, may be hard to highlight in an editor.
    - - In pre-def locations, would not work in interactive mode (list would be interpreted right away)
    - - In within-def locations, does not make clear the break between decorators and the rest of the function declaration

Location

Proposals also differ on exactly where the decorator declaration should be made. These include:

- Pre-def
    - + Decorators are easy to find
    - + Won't break simplistic code analyzers or grep for function def
    - o Not consistent with docstring location
    - - Decorators are not as clearly a part of the function declaration
    - - Following empty lines or comments may separate decorators from function declaration.
    - - May require extra mental "stack space" to process decorators before seeing function name
- Within-def (i.e. on the same line)
    - + Decorators are clearly part of the function declaration
    - o Not consistent with docstring location
    - - Long lists of decorators/arguments cause ugly line wraps
    - - More cumbersome to copy decorator lists because they are in the middle
    - *Some within-def options*:

        - Before def
            - + Decorators are easy to find
            - - May break simplistic code analyzers or grep for function def
            - - Long lists of decorators hide function names
            - - May require extra mental "stack space" to process decorators before seeing function name
        - Between def and function name
            - - Long lists of decorators hide function names
            - - May break simplistic code analyzers or grep for function def
            - - May require extra mental "stack space" to process decorators before seeing function name
        - Between function name and argument list
            - + Should not require extra mental "stack space" to process decorators (function name has already been seen)
            - - May break simplistic code analyzers or grep for function def
            - - Easy to miss the transition between a long argument list and a long decorator list
            - - Breaks the symmetry between definition and use -- you can no longer just cut and paste the definition to write a call, or vice versa.
            - - Guido has ruled it out, because of the separation between function name and function arguments.
        - Between argument list and colon
            - + Should not require extra mental "stack space" to process decorators (function name has already been seen)
            - + Won't break simplistic code analyzers or grep for function def
            - - Easy to miss the transition between a long argument list and a long decorator list
- Post-def
    - + Should not require extra mental "stack space" to process decorators (function name has already been seen)
    - + Won't break simplistic code analyzers or grep for function def
    - o Consistent with docstring location
    - - Decorators may look like part of the function's code, not its declaration. (That is, they look like they're evaluated when the function is called, not when it is parsed.) This is a problem for decorations that change the signature, like staticmethod.

- - Guido ruled out any solution involving special syntax inside the block, because "you shouldn't have to peek inside the block to find out important external properties of the function." **http://mail.python.org/pipermail/python-dev/2004-August/047279.html**

- - **Guido rejects**, because the function body should reflect what the function does; decorators are "outside" the function, for its callers.

    - *Some post-def options*:

        - Before docstring
        - After docstring
            - - Docstring separates decorators from function declaration.
- After entire definition.
    - + This is what we have now, with the foo=deco(foo) syntax.
    - - This is too far from the front of the definition, which is one reason for the change. Simply eliminating the retyping of the name might actually be bad for long functions, by introducing magic.

List Notation

Decorator syntax, as described in the PEP, must support the application of multiple decorators. Some proposals on how to support this include:

- One per line
    - + Easy to add/remove/reorder decorators
    - - Decorators consume more vertical space
- Commas only
    - + Decorators do not consume much vertical space
    - - Difficult to break across lines
- As list
    - + Decorators do not have to consume much vertical space
    - + Easy to break across lines
    - - Hard to copy/paste
- As tuple
    - + Decorators do not have to consume much vertical space
    - + Easy to break across lines
    - o Allows short one-liners using X, tuple syntax
    - - May look too much like argument list

    - - Hard to copy/paste
- Angle brackets
    - + Cannot be confused with any other syntax
    - - Angle brackets are "unpaired" characters
    - - Parsing of greater-than and less-than becomes more fragile.

Indentation

Proposals also differ on whether or not decorators should introduce a new block. These include:

- Indent decorators and def
    - + Allows many decorated functions to be declared with a single statement
    - - Increases minimum ident level on decorated functions
    - - Inconsistent identation level between methods with/without decorators
    - - Can lead to "runaway" nesting

    - - Seems to change flow control or namespace, but doesn't really. **Phillip J. Eby**

- Indent decorators only
    - + Makes locating the function definition easy
    - o Different semantics from that of other paired, nested blocks (e.g. if/elif/else, try/except, try/finally)
    - - Technical problems if a block *starts* with an optional part? (Someone said yes, someone else said that they tried it and had no problems.)

    - - **David Eppstein** points out that most indented suites can contain arbitrary code. It isn't clear what should happen if someone uses a statement (does not return a value); should it be run once? Should it cause an error?

- No indent
    - - Uncommon in Python that one line following another one with the same identation has a meaning like this.

Order of Decorators

Should decorators be applied in textual order, or in reverse order (as if they were in nested parentheses around the def).

It seems to have been settled that decorators will be applied starting with the one closest to the "def" statement. (This means in-order vs reverse will depend on the eventual location.)

Allowable Decorators

The eventual decision was that anything should be allowed, but it *will* be called with a single argument (the function object) before the function's name is bound in the enclosing namespace.

Should None be allowed and just not called? This would allow statements in the decorator suite, and might be useful, but ... it wasn't deemed useful enough for a special case, at least in 2.4

Meaning of decorators

```
@deco1
@deco2(darg1, darg2)
@deco3
def func(arg1, arg2):
    pass
```

is equivalent to

```
__tempd1 = deco1
__tempd2 = deco2(darg1, darg2)
__tempd3 = deco3
__tempf = lambda arg1, arg2: pass
func = tempd1(tempd2(tempd3(tempf)))
```

with the following exceptions:

- The temporary variarables aren't really created. Those are just placeholders to indicate that the expression is evaluated (hopefully to produce a callable) before the function definition.
- The function can be any function, not just a lambda expression (which is more restricted).

Thinking ahead to Python 3 ?

Christopher King makes the point that we are trying to do too much with decorators: declare class/static methods, describe function metadata, and mangle functions. It might be best to think about what is best for each separately.

How might fully loaded functions look in the future?

Christopher King's example:

```
def classmethod foo(self,a,b,c):
    """Returns a+b*c."""
    {accepts: (int,int,int), author: 'Chris King'}

    return a+b*c
```

Another possible example (keyword support for staticmethod & classmethod, visual basic-like typing using the "as" keyword for adapters, "with" code blocks):

```
def classmethod foo(a as int, b as int, c as list) as list:
    """Returns a+b*c."""

    listcopy = []
    with synchronized(lock):
        listcopy[] = c[]

    return a+b*listcopy
```

Here it is with the @ symbol:

```
 @author('Chris King')
 @accepts(int,int,list)
 @classmethod
 def foo(self,a,b,c):
     """Returns a+b*c."""

       return a+b*c
```

EditText (last edited 2006-05-04 02:05:56 by **JürgenErhard**)