The Decorator Pattern ¶

A “Structural Pattern” from the Gang of Four book

Warning

The “Decorator Pattern” ≠ Python “decorators”!

If you are interested in Python decorators like `@classmethod` and `@contextmanager` and `@wraps()`, then stay tuned for a later phase of this project in which I start tackling Python language features.

Verdict

The Decorator Pattern can be useful in Python code! Happily, the pattern can be easier to implement in a dynamic language like Python than in the static languages where it was first practiced. Use it on the rare occasion when you need to adjust the behavior of an object that you can’t subclass but can only wrap at runtime.

Contents:

The Python core developers made the terminology surrounding this design pattern more confusing than necessary by using the *decorator* for an entirely unrelated language feature. The timeline:

- The design pattern was developed and named in the early 1990s by participants in the “Architecture Handbook” series of workshops that were kicked off at

OOPSLA '90, a conference for researchers of object-oriented programming languages.

- The design pattern became famous as the "Decorator Pattern" with the 1994 publication of the Gang of Four's *Design Patterns* book.
- In 2003, the Python core developers decided to re-use the term *decorator* for a completely unrelated feature they were adding to Python 2.4.

Why were the Python core developers not more concerned about the name collision? It may simply be that Python's dynamic features kept its programming community so separate from the world of design-pattern literature for heavyweight languages that the core developers never imagined that confusion could arise.

To try to keep the two concepts straight, I will use the term *decorator class* instead of just *decorator* when referring to a class that implements the Decorator Pattern.

Definition ¶

A *decorator* class:

- Is an *adapter* (see the *Adapter Pattern*)
- That implements the same interface as the object it wraps
- That delegates method calls to the object it wraps

The decorator class's purpose is to add to, remove from, or adjust the behaviors that the wrapped object would normally implement when its methods are called. With a decorator class, you might:

- Log method calls that would normally work silently
- Perform extra setup or cleanup around a method
- Pre-process method arguments
- Post-process return values
- Forbid actions that the wrapped object would normally allow

These purposes might remind you of situations in which you would also think of subclassing an existing class. But the Decorator Pattern has a crucial advantage over a subclass: you can only solve a problem with a subclass when your own code is in charge of creating the objects in the first place. For example, it isn't helpful to subclass the Python file object if a library you're using is returning normal file objects and you have no way to intercept their construction — your new `MyEvenBetterFile` subclass would sit unused. A decorator class does not have that limitation. It can be wrapped around a plain old file object any time you want, without the need for you be in control when the wrapped object was created.

Implementing: Static wrapper ¶

First, let’s learn the drudgery of creating the kind of decorator class you would write in C++ or Java. We will not take advantage of the fact that Python is a dynamic language, but will instead write static (non-dynamic) code where every method and attribute appears literally, on the page.

To be complete — to provide a real guarantee that every method called and attribute manipulated on the decorator object will be backed by the real behavior of the adapted object — the decorator class will need to implement:

- Every method of the adapted class
- A getter for every attribute
- A setter for every attribute
- A deleter for every attribute

This approach is conceptually simple but, wow, it involves a lot of code!

Imagine that one library is giving you open Python file objects, and you need to pass them to another routine or library — but to debug some product issues with latency, you want to log each time that data is written to the file.

Python file objects often seem quite simple. We usually `read()` from them, `write()` to them, and not much else. But in fact the file object supports more than a dozen methods and offers five different attributes! A wrapper class that really wants to implement that full behavior runs to nearly 100 lines of code — as shown here, in our first working example of the Decorator Pattern:

```
# Traditional Decorator pattern: noticeably verbose

class WriteLoggingFile1(object):
    def __init__(self, file, logger):
        self._file = file
        self._logger = logger

    # We need to implement every file method,
    # and in the truly general case would need
    # a getter, setter, and deleter for every
    # single attribute!  Here goes:

    def __enter__(self):
        return self._file.__enter__()

    def __exit__(self, *excinfo):
        return self._file.__exit__(*excinfo)

    def __iter__(self):
        return self._file.__iter__()

    def __next__(self):
        return self._file.__next__()

    def __repr__(self):
        return self._file.__repr__()

    def close(self):
        return self._file.close()

    @property
    def closed(self):
        return self._file.closed

    @closed.setter
    def closed(self, value):
        self._file.closed = value

    @closed.deleter
    def closed(self):
        del self._file.closed

    @property
    def encoding(self):
        return self._file.encoding

    @encoding.setter
    def encoding(self, value):
        self._file.encoding = value

    @encoding.deleter
    def encoding(self):
        del self._file.encoding

    @property
    def errors(self):
        return self._file.errors

    @errors.setter
    def errors(self, value):
        self._file.errors = value

    @errors.deleter
    def errors(self):
        del self._file.errors

    def fileno(self):
        return self._file.fileno()

    def flush(self):
        return self._file.flush()

    def isatty(self):
        return self._file.isatty()

    @property
    def mode(self):
        return self._file.mode

    @mode.setter
    def mode(self, value):
        self._file.mode = value

    @mode.deleter
    def mode(self):
        del self._file.mode

    @property
    def name(self):
        return self._file.name

    @name.setter
```

```
    def name(self, value):
        self._file.name = value

    @name.deleter
    def name(self):
        del self._file.name

    @property
    def newlines(self):
        return self._file.newlines

    @newlines.setter
    def newlines(self, value):
        self._file.newlines = value

    @newlines.deleter
    def newlines(self):
        del self._file.newlines

    def read(self, *args):
        return self._file.read(*args)

    def readinto(self, buffer):
        return self._file.readinto(buffer)

    def readline(self, *args):
        return self._file.readline(*args)

    def readlines(self, *args):
        return self._file.readlines(*args)

    def seek(self, *args):
        return self._file.seek(*args)

    def tell(self):
        return self._file.tell()

    def truncate(self, *args):
        return self._file.truncate(*args)

    # Finally, we reach the two methods
    # that we actually want to specialize!
    # These log each time data is written:

    def write(self, s):
        self._file.write(s)
        self._logger.debug('wrote %s bytes to %s', len(s), self._file)

    def writelines(self, strings):
        if self.closed:
            raise ValueError('this file is closed')
        for s in strings:
            self.write(s)
```

So for the sake of the half-dozen lines of code at the bottom that supplement the behavior of `write()` and `writelines()`, another hundred or so lines of code wound up being necessary.

You will notice that each Python object attribute goads us into being even more verbose than Java! A typical Java attribute is implemented as exactly two methods, like `getEncoding()` and `setEncoding()`. A Python attribute, on the other hand, will in the general case need to be backed by *three* actions — get, set, and delete — because Python's object model is dynamic and supports the idea that an attribute might disappear from an instance.

Of course, if the class you are decorating does not have as many methods and attributes as the Python file object we took as our example, then your wrapper will be shorter. But in the general case, writing out a full wrapper class will be tedious unless you have a tool like an IDE that can automate the process. Also, the wrapper will need to be updated in the future if the underlying object gains (or loses) any methods, arguments, or attributes.

Implementing: Tactical wrapper ¶

The wrapper in the previous section might have struck you as ridiculous. It tackled the Python file object as a general example of a class that needed to be wrapped, instead of studying the how file objects work to look for shortcuts:

- File objects are implemented in the C language and don't, in fact, permit deletion of any of their attributes. So our wrapper could have omitted all 6 deleter methods without any consequence, since the default behavior of a property in the absence of a deleter is to disallow deletion anyway. This would have saved 18 lines of code.
- All file attributes except `mode` are read-only and raise an `AttributeError` if assigned to — which is the behavior if a property lacks a setter method. So 5 of our 6 setters can be omitted, saving 15 more lines of code and bringing our wrapper to ⅓ its original length without sacrificing correctness.

It might also have occurred to you that the code to which you are passing the wrapper is unlikely to call every single file method that exists. What if it only calls two methods? Or only one? In many cases a programmer has found that a trivial wrapper like the following will perfectly satisfy real-world code that just wants to write to a file:

```
# Tactical version of Decorator Pattern:
# what if you read the code, and the only thing
# the library really needs is the write() method?

class WriteLoggingFile2(object):
    def __init__(self, file, logger):
        self._file = file
        self._logger = logger

    def write(self, s):
        self._file.write(s)
        self._logger.debug('wrote %s bytes to %s', len(s), self._file)
```

Yes, this can admittedly be a bit dangerous. A routine that seems so happy with a minimal wrapper like this can suddenly fail later if rare circumstances make it dig into methods or attributes that you never implemented because you never saw it use them. Even if you audit the library's code and are sure it can never call any method besides `write()`, that could change the next time you upgrade the library to a new version.

In a more formal programming language, a duck typing requirement like “this function requires a file object” would likely be replaced with an exact specification like “this argument needs to support a `writelines()` method” or “pass an object that offers every methods in the interface `IWritableFile`.” But most Python code lacks this precision and will force you, as the author of a wrapper class, to decide where to draw the line between the magnificent pedantry of wrapping every possible method and the danger of not wrapping enough.

Implementing: Dynamic wrapper ¶

A very common approach to the Decorator Pattern in Python is the dynamic wrapper. Instead of trying to implement a method and property for every method and attribute on the wrapped object, a dynamic wrapper intercepts live attribute accesses as the program executes and responds by trying to access the same attribute on the wrapped object.

A dynamic wrapper implements the dunder methods `__getattr__()`, `__setattr__()`, and — if it really wants to be feature-complete — `__delattr__()` and responds to each of them by performing the equivalent operation on the wrapped object. Because `__getattr__()` is only invoked for attributes that are in fact missing on the wrapper, the wrapper is free to offer real implementations of any methods or properties it wants to intercept.

There are a few edge cases that prevent every attribute access from being handled with `__getattr__()`. For example, if the wrapped object is iterable, then the basic operation `iter()` will fail on the wrapper if the wrapper is not given a real `__iter__()` method of its own. Similarly, even if the wrapped object is an iterator, `next()` will fail unless the wrapper offers a real `__next__()`, because these two operations examine an object’s class for dunder methods instead of hitting the object directly with a `__getattr__()`.

As a result of these special cases, a getattr-powered wrapper usually involves at least a half-dozen methods in addition to the methods you specifically want to specialize:

```
# Dynamic version of Decorator Pattern: intercept live attributes

class WriteLoggingFile3(object):
    def __init__(self, file, logger):
        self._file = file
        self._logger = logger

    # The two methods we actually want to specialize,
    # to log each occasion on which data is written.

    def write(self, s):
        self._file.write(s)
        self._logger.debug('wrote %s bytes to %s', len(s), self._file)

    def writelines(self, strings):
        if self.closed:
            raise ValueError('this file is closed')
        for s in strings:
            self.write(s)

    # Two methods we don't actually want to intercept,
    # but iter() and next() will be upset without them.

    def __iter__(self):
        return self.__dict__['_file'].__iter__()

    def __next__(self):
        return self.__dict__['_file'].__next__()

    # Offer every other method and property dynamically.

    def __getattr__(self, name):
        return getattr(self.__dict__['_file'], name)

    def __setattr__(self, name, value):
        if name in ('_file', '_logger'):
            self.__dict__[name] = value
        else:
            setattr(self.__dict__['_file'], name, value)

    def __delattr__(self, name):
        delattr(self.__dict__['_file'], name)
```

As you can see, the code can be quite economical compared to the vast slate of methods we saw earlier in `WriteLoggingFile1` for manually implementing every possible attribute.

This extra level of indirection does carry a small performance penalty for every attribute access, but is usually preferred to the burden of writing a static wrapper.

Dynamic wrappers also offer pleasant insulation against changes that might happen in the future to the object being wrapped. If a future version of Python adds or removes an attribute or method from the file object, the code of `WriteLoggingFile3` will require no change at all.

Caveat: Wrapping doesn't actually work ¶

If Python didn't support introspection — if the only operation you could perform on an object was attribute lookup, whether statically through an identifier like `f.write` or dynamically via `getattr(f, attrname)` string lookup — then a decorator could be foolproof. As long as every attribute lookup that succeeds on the wrapped object will return the same sort of value when performed on the wrapper, then other Python code would never know the difference.

But Python is not merely a dynamic programming language; it also supports introspection. And introspection is the downfall of the Decorator Pattern. If the code to which you pass the wrapper decides to look deeper, all kinds of differences become apparent. The native file object, for example, is buttressed with many private methods and attributes:

```
>>> from logging import getLogger
>>> f = open('/etc/passwd')
>>> dir(f)
['_CHUNK_SIZE', '__class__', '__del__', '__delattr__', '__dict__',
'__dir__', '__doc__', '__enter__', '__eq__', '__exit__', '__format__',
'__ge__', '__getattribute__', '__getstate__', '__gt__', '__hash__',
'__init__', '__init_subclass__', '__iter__', '__le__', '__lt__',
'__ne__', '__new__', '__next__', '__reduce__', '__reduce_ex__',
'__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__',
'_checkClosed', '_checkReadable', '_checkSeekable', '_checkWritable',
'_finalizing', 'buffer', 'close', 'closed', 'detach', 'encoding',
'errors', 'fileno', 'flush', 'isatty', 'line_buffering', 'mode', 'name',
'newlines', 'read', 'readable', 'readline', 'readlines', 'seek',
'seekable', 'tell', 'truncate', 'writable', 'write', 'writelines']
```

Your wrapper, on the other hand — if you have crafted it around the file's public interface — will lack all of those private accouterments. Behind your carefully implemented public methods and attributes are the bare dunder methods of a generic Python `object`, plus the few you had to implement to maintain compatibility:

```
>>> w = WriteLoggingFile1(f, getLogger())
>>> dir(w)
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__',
'__enter__', '__eq__', '__exit__', '__format__', '__ge__',
'__getattribute__', '__gt__', '__hash__', '__init__',
'__init_subclass__', '__iter__', '__le__', '__lt__', '__module__',
'__ne__', '__new__', '__next__', '__reduce__', '__reduce_ex__',
'__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__',
'__weakref__', '_file', '_logger', 'close', 'closed', 'encoding',
'errors', 'fileno', 'flush', 'isatty', 'mode', 'name', 'newlines',
'read', 'readinto', 'readline', 'readlines', 'seek', 'tell', 'truncate',
'write', 'writelines']
```

The tactical wrapper, of course, looks spectacularly different than a real file object, because it does not even attempt to provide the full range of methods available on the wrapped object:

```
>>> w = WriteLoggingFile2(f, getLogger())
>>> dir(w)
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__',
'__format__', '__ge__', '__getattribute__', '__gt__', '__hash__',
'__init__', '__init_subclass__', '__le__', '__lt__', '__module__',
'__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__',
'__setattr__', '__sizeof__', '__str__', '__subclasshook__',
'__weakref__', '_file', '_logger', 'write']
```

More interesting is the getattr wrapper. Even though, in practice, it offers access to every attribute and method of the wrapped class, they are completely missing from its `dir()` because each attribute only springs into existence when accessed by name.

```
>>> w = WriteLoggingFile3(f, getLogger())
>>> dir(w)
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__',
'__format__', '__ge__', '__getattr__', '__getattribute__', '__gt__',
'__hash__', '__init__', '__init_subclass__', '__iter__', '__le__',
'__lt__', '__module__', '__ne__', '__new__', '__next__', '__reduce__',
'__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__',
'__subclasshook__', '__weakref__', '_file', '_logger', 'write',
'writelines']
```

Could even these differences be ironed out? If you scroll through the many dunder methods in the Python Data Model, your might be struck by a sudden wild hope when you see the __dir__ method — surely this is the final secret to camouflaging your wrapper?

Alas, it will not be enough. Even if you implement `__dir__()` and forward it through to the wrapped object, Python special-cases the `__dict__` attribute — accessing it always provides direct access to the dictionary that holds a Python class instance's attributes.

```
>>> f.__dict__
{'mode': 'r'}
>>> w.__dict__
{'_file': <_io.TextIOWrapper name='/etc/passwd' mode='r'
encoding='UTF-8'>, '_logger': <RootLogger root (WARNING)>}
```

You might begin to think of even more obscure ways to subvert Python's introspection — at this point you might already be thinking of `__slots__`, for example — but all roads lead to the same place. However clever and obscure your maneuvers, at least a small chink will still be left in your wrapper's armor which will allow careful enough introspection to see the difference. Thus we are lead to a conclusion:

Maxim

> The Decorator Pattern in Python supports *programming* — but not *metaprogramming*. Code that is happy to simply access attributes will be happy to accept a Decorator Pattern wrapper instead. But code that indulges in introspection will see the difference.

Among other things, Python code that attempts to list an object's attributes, examine its `__class__`, or directly access its `__dict__` will see differences between the object it expected and the decorator object you have in fact given it instead. Well-written application code would never do such things, of course — they are necessary only when implementing a developer tool like a framework, test harness, or debugger. But as you don't always have the option of dealing solely with well-written libraries, be prepared to see and work around any symptoms of intrusive introspection as you deploy the Decorator Pattern.

Hack: Monkey-patch each object ¶

There are two final approaches to decoration based on the questionable practice of monkey patching. The first approach takes each object that needs decoration and installs a new method directly on the object, shadowing the official method that remains on the class itself.

If you have ever attempted this maneuver yourself, you might have run aground on the fact that a function installed on a Python object instance does *not* receive an automatic `self` argument — instead, it sees only the arguments with which it is literally invoked. So a first try at supplementing a file's `write()` with logging:

```
>>> def bind_write_method(logger):
...     # Does not work: will not receive `self` argument
...     def write_and_log(self, s):
...         self.write(s)
...         logger.debug('wrote %s bytes to %s', len(s), self._file)
...     return write_and_log
```

— will die with an error because the new method sees only one argument, not two:

```
>>> f = open('/dev/null', 'w')
>>> f.write
<built-in method write ...>
>>> f.write = bind_write_method(getLogger())
>>> f.write('Hello, world.')
Traceback (most recent call last):
  ...
TypeError: write_and_log() missing 1 required positional argument: 's'
```

The quick way to resolve the dilemma is to do the binding yourself, by providing the object instance to the closure that wraps the new method itself:

```
>>> def bind_write_method(self, logger):
...     def write_and_log(s):
...         write(s)
...         print('wrote {} bytes to {}'.format(len(s), self.name))
...     write = self.write
...     return write_and_log
```

```
>>> f = open('/dev/null', 'w')
>>> f.write = bind_write_method(f, getLogger())
>>> f.write('Hello, world.')
wrote 13 bytes to /dev/null
```

While clunky, this approach does let you update the action of a single method on a single object instance while leaving the entire rest of its behavior alone.

Hack: Monkey-patch the class ¶

Another approach you might see in the wild is to create a subclass that has the desired behaviors overridden, and then to surgically change the class of the object instance. This is not, alas, possible in the general case, and in fact fails for our example here because the file class, like all built-in classes, does not permit assignment to its `__class__` attribute:

```
>>> f = open('/etc/passwd')
>>> class Foo(type(f)):
...     def write_and_log(self, s):
...         self.write(s)
...         logger.debug('wrote %s bytes to %s', len(s), self._file)
...
>>> f.__class__ = Foo
Traceback (most recent call last):
  ...
TypeError: __class__ assignment only supported for heap types or
ModuleType subclasses
```

But in cases where the surgery does work, you will have an object whose behavior is that of your subclass rather than of its original class.

Further Reading ¶

- If dynamic wrappers and monkey patching spur your interest, check out Graham Dumpleton’s wrapt library, his accompanying series of blog posts, and his monkey patching talk at Kiwi PyCon that delve deep into the arcane technical details of the practice.