Python Decorators and Metaclasses

In this chapter, we look at metaprogramming using Python decorators and metaclasses. To do so, we consider a simple tracing debugger case study, adapted from David Beazley’s `debugly` example from his metaprogramming tutorial [1].

TODO: Chapter goals.

Note: In this book, we use the term Python to mean Python 3. The various examples use Python 3.7 or later.

Basic Function-Level Debugging

Motivating example

Suppose we have a Python function `add`:

```python
def add(x, y):
    'Add x and y'
    return x + y
```

A simple way we can approach debugging is to insert a `print` statement into the function to trace execution, as follows:

```python
def add(x, y):
    'Add x and y'
    print('add')
    return x + y
```

However, suppose we need to debug several similar functions simultaneously. Following the above approach, we might have code similar to that in the example below.

```python
def add(x, y):
    'Add x and y'
    print('add')
    return x + y

def sub(x, y):
    'Subtract y from x'
    print('sub')
    return x - y

def mul(x, y):
    'Multiply x and y'
    print('mul')
    return x * y

def div(x, y):
    'Divide x by y'
```

3
print('div')
return x / y

We insert basically the same code into every function.

This code is unpleasant because it violates the Abstraction Principle.

**Abstraction Principle, staying DRY**

The *Abstraction Principle* states, “Each significant piece of functionality in a program should be implemented in just one place in the source code.” [5:339]. If similar functionality is needed in several places, then the common parts of the functionality should be separated from the variable parts.

The common parts become a new programming abstraction (e.g., a function, class, abstract data type, design pattern, etc.) and the variable parts become different ways in which the abstraction can be customized (e.g., its parameters).

The approach encourages reuse of both design and code. Perhaps more importantly, it can make it easier to keep the similar parts consistent as the program evolves.

Andy Hunt and Dave Thomas [4:26–33] articulate a more general software development principle *Don’t Repeat Yourself*, known by the acronym *DRY*.

In an interview [7], Thomas states, “DRY says that every piece of system knowledge should have one authoritative, unambiguous representation. … A system’s knowledge is far broader than just its code. It refers to database schemas, test plans, the build system, even documentation.”

Our goal is to keep our Python code DRY, not let it get WET (“Write Everything Twice” or “Wasting Everyone’s Time” or “We Enjoy Typing” [8].)

**Function decorators**

To introduce an appropriate abstraction into the previous set of functions, we can use a Python function decorator.

A *function decorator* is a higher-order function that takes a function as its argument, wraps another function around the argument, and returns the wrapper function.

The wrapper function has the same parameters and same return value as the function it wraps, except it does extra processing when it is called. That is, it “decorates” the original function.

TODO: Show Wikipedia citation because of link.

Remember that Python functions are objects. Python’s decorator function concept is thus a special case of the Decorator design pattern, one of the classic Gang of Four patterns for object-oriented programming [3]. The idea of this pattern is to wrap one object with another object, the decorator, that has the
same interface but enhanced behavior. The decoration is usually done at runtime even in a statically typed language like Java.

TODO: Perhaps expand on the Decorator design pattern and give a diagram.

**Constructing a debug decorator**

In the motivating example above, we want to decorate a function like `add(x, y)` by wrapping it with another function that prints the function name `add` before doing the addition operation. The wrapped function can then take the place of the original `add` in the program.

Let’s construct an appropriate decorator in steps.

In general, suppose we want to decorate a function named `func` that takes some number of positional and/or keyword arguments. That is, the function has the general signature:

```python
func(*args, **kwargs)
```

Note: For more information on the above function calling syntax, see the discussion on Function Calling Conventions in Chapter 6.

In addition, suppose we want to print the content of the variable `msg` before we execute `func`.

As our first step, we define function `wrapper` as follows:

```python
def wrapper(*args, **kwargs):
    print(msg)
    return func(*args, **kwargs)
```

As our second step, we define a decorator function `debug` that takes a function `func` as its argument, sets local variable `msg` to `func`’s name, and then creates and returns the function `wrapper`.

Function `debug` can retrieve the function name by accessing the `__qualname__` attribute of the `func` object. Attribute `__qualname__` holds the fully qualified name.

```python
def debug(func):
    msg = func.__qualname__
    def wrapper(*args, **kwargs):
        print(msg)
        return func(*args, **kwargs)
    return wrapper
```

Function `debug` returns a closure that consists of the function `wrapper` plus the the local environment in which `wrapper` is defined. The local environment includes the argument `func` and the variable `msg` and their values.
Note: For more information about the concepts and techniques used above, see the discussion of Nested Function Definitions, Lexical Scope, and Closures in Chapter 6.

It seems sufficient to assign the closure returned by `debug` to the name of `func` as shown below for `add`.

```python
    def add(x, y):
        'Add x and y' # docstring (documentation)
        return x + y
    add = debug(add)
```

But this does not work as expected as shown in the following REPL session.

```python
>>> add(2,5)
add
7
>>> add.__qualname__
debug.<locals>.wrapper
>>> add.__doc__
None
```

The closure returned by `debug` computes the correct result. However, it does not have the correct metadata, as illustrated above by the display of the name (`__qualname__`) and the docstring (`__doc__`) metadata.

To make the use of the decorator `debug` transparent to the user, we can apply the function decorator `@wraps` defined in the standard module `functools` as follows.

```python
    def debug(func):
        msg = func.__qualname__
        @wraps(func)
        def wrapper(*args, **kwargs):
            print(msg)
            return func(*args, **kwargs)
        return wrapper
```

```python
    def add(x, y):
        'Add x and y' # docstring (documentation)
        return x + y
    add = debug(add)
```

The `@wraps(func)` decorator call above sets function `wrapper`’s metadata — its attributes `__module__`, `__name__`, `__qualname__`, `__annotations__`, and `__doc__` — to the same values as `func`’s metadata.

With this new version of the `debug` decorator, the decoration of `add` now works transparently.
Finally, because the definition of a function like `add` and the application of the `debug` decorator function usually occur together, we can use the decorator `syntactic sugar` `@debug` to conveniently designate the definition of a decorated function. The `debug` decorator function can be defined in a separate module.

```python
@debug
def add(x, y):
    'Add x and y'
    return x + y
```

**Using the debug decorator**

Given the `debug` decorator as defined in the previous subsection, we can now simplify the motivating example.

We decorate each function with `@debug` but give no other details of the implementation here. The debug facility is implemented in one place but used in many places. The implementation supports the DRY principle.

```python
@debug
def add(x, y):
    'Add x and y'
    return x + y

@debug
def sub(x, y):
    'Subtract y from x'
    return x - y

@debug
def mul(x, y):
    'Multiply x and y'
    return x * y

@debug
def div(x, y):
    'Divide x by y'
    return x / y
```

Note: The Python 3.7+ source code for the above version of `debug` is available in linked file `debug4.py`. 7
Case study review

So far in this case study, we have implemented a simple debugging facility that:

- is implemented once in a place separate from its use
- is thus easy to modify or disable totally
- can be used without knowing its implementation details

Variations

Now let’s consider a couple of variations of the debugging decorator implementation.

Logging  One variation would be to use the Python logging module to log the messages instead of just printing them [1].

The details of logging are not important here, but note that we only need to make three changes to the `debug` implementation. We do not need to change the user code.

```python
from functools import wraps
import logging  # (1) logging module

def debug(func):
    # (2) get the Logger for func's module
    log = logging.getLogger(func.__module__)
    msg = func.__qualname__
    @wraps(func)
    def wrapper(*args, **kwargs):
        log.debug(msg)  # (3) log msg
        return func(*args, **kwargs)
    return wrapper

Note: The Python 3.7+ source code for the above version of `debug` is available in linked file `debuglog1.py`.

Optional disable  Another variation of the debugging decorator would be to only enable debugging when a particular environment variable is set [1]. In this variation, we only need to make two changes to the `debug` implementation.

```python
from functools import wraps
import os  # (1) import os interface

def debug(func):
    # (2) debug only if environment variable set
    if 'DEBUG' not in os.environ:
        return func
    msg = func.__qualname__
```
@wraps(func)
def wrapper(*args, **kwargs):
    print(msg)
    return func(*args, **kwargs)
return wrapper

Note: The Python 3.7+ source code for the above version of `debug` is available in linked file `debugopt1.py`.

**Extended Function-Level Debugging**

Now we can extend the capability of our simple tracing debugger [1].

**Motivating example**

Suppose, for whatever reason, we want to add a prefix string to the debugging message that may differ from one use of `@debug` to another. Again consider the set of arithmetic functions.

```python
def add(x, y):
    'Add x and y'
    print('***add')
    return x + y
def sub(x, y):
    'Subtract y from x'
    print('@@@sub')
    return x - y
def mul(x, y):
    'Multiply x and y'
    print('***sub')
    return x * y
def div(x, y):
    'Divide x by y'
    print('div')
    return x / y
```

We implement the needed capability by using function decorators with arguments.

**Decorators with arguments**

We can construct decorators that take arguments other than the function to be decorated.

Consider the following use of decorator `deco`: 
@deco(args)
def func():
    # some body code
    The above translates into the following decorator call and assignment:
    func = deco(args)(func)
The right-hand side denotes the chaining of two function calls. The system first
calls function deco passing it the first argument list (args). This call returns a
function, which is in turn called with the second argument list, variable (func).
The outer function call establishes a local environment in which the variables in
args are defined. In this environment, we define a normal decorator as we did
before.

Prefix decorator
We can thus define the outer layer of a prefix decorator with a function with
parameter prefix that defaults to the empty string.

def debug(prefix=' '):
def deco(func):
    # normal debug decorator body
    return deco
The full definition of the prefix decorator is shown below. If no argument is given
to debug, the behavior is (almost) the same as the previous debug decorator
function.

    from functools import wraps
    def debug(prefix=' '):
        def deco(func):
            msg = prefix + func.__qualname__
            @wraps(func)
            def wrapper(*args, **kwargs):
                print(msg)
                return func(*args, **kwargs)
            return wrapper
        return deco
In this formulation, prefix can be given as either a positional or keyword
argument.
We can apply the new prefix debug decorator to our motivating example functions
as follows. Note that the prefix strings vary among the different occurrences.

@debug(prefix='***')
def add(x,y):
    'Add x and y'
return x+y

@debug(prefix='@@@')
def sub(x, y):
    'Subtract y from x'
    return x - y

@debug('***')
def mul(x, y):
    'Multiply x and y'
    return x * y

@debug()  # parentheses needed!
def div(x, y):
    'Divide x by y'
    return x / y

Note: The Python 3.7+ source code for the above version of debugprefix is available in linked file debugprefix1.py.

Reformulated prefix decorator

By a clever use of default arguments and partial application of a function to its arguments, we can transform the definition of the prefix decorator above to one that does not involve a nested definition.

from functools import wraps, partial

def debug(func = None, *, prefix = ''):
    if func is None:
        return partial(debug, prefix=prefix)
    msg = prefix + func.__qualname__
    @wraps(func)
    def wrapper(*args, **kwargs):
        print(msg)
        return func(*args, **kwargs)
    return wrapper

If we call the debug decorator function with the single keyword argument prefix, then the func argument defaults to None. In this case, the if statement causes debug to call itself with that prefix argument and the decorated function (that follows the @debug annotation in the user-level code or occurs in a second argument list) as the func argument.

Note: The functools.partial function takes a function (object) and a group of positional and/or keyword arguments, partially applies the function to those arguments, then returns the resulting function (object). The returned function behaves like the original function except that it has the argument values supplied
to `partial` as its default parameter values.

If we call the `debug` decorator function with no keyword arguments, then parameter `prefix` defaults to the empty string and `func` is the decorated function (e.g., that follows the `@debug` annotation).

If we call `debug` with both `func` and `prefix` arguments, then it works as we expect. This case is not used with the `@debug` annotation.

```python
@debug(prefix='***')
def add(x, y):
    'Add x and y'
    return x + y

@debug(prefix='@@@')
def sub(x, y):
    'Subtract y from x'
    return x - y

@debug(prefix='***')
def mul(x, y):
    'Multiply x and y'
    return x * y

@debug
# no parentheses required, but okay if given
def div(x, y):
    'Divide x by y'
    return x / y
```

Unlike the previous formulation of the prefix decorator, the `prefix` string must be supplied as a prefix argument.

Note: The Python 3.7+ source code for the above version of `debugprefix` is available in linked file `debugprefix2.py`.

### Class-Level Debugging

**Motivating example**

Consider the class `Account` below for a simple bank account.

Suppose we want to debug all the methods using the simple debugging package we developed above.

```python
class Account:
    def __init__(self):
        self._bal = 0

    @debug
def deposit(self, amt):
```
```python
self._bal += amt

@debug
def withdraw(self, amt):
    if amt <= self._bal:
        self._bal -= amt
    else:
        print(f'Insufficient funds for withdrawal of {amt}')

@debug
def get_balance(self):
    return self._bal

def __str__(self):
    return f'Account with balance {self._bal}'
```

Note: The Python 3.7+ source code for the above version of `Account` is available in linked file `account2.py`.

Class-level debugger

The `Account` example above is repetitive (not DRY). Can we do the decoration all at once?

Yes, we can define a class decorator `debugmethods` as shown below (where `debug` is the function-level prefix decorator defined above). A class decorator is a higher-order function that takes a class as its argument, modifies the class in some way, and then returns the modified class.

```python
def debugmethods(cls):
    # Notes
    for name, val in vars(cls).items():
        # (1) (2) (3)
        if callable(val):
            # (4)
            setattr(cls, name, debug(val))
            # (5) (6)
    return cls
    # (7)
```

The idea here is that the program walks through the class dictionary, identifies callable objects (e.g., methods), and wraps each with a function decorator.

Consider the numbered comments in the above code.

1. The built-in function call `vars(cls)` returns the dictionary (i.e., `__dict__`) associated with the (class) object `cls`.
2. The dictionary method call `items()` returns the list of key-value pairs in the dictionary.
3. The “for name, val in” statement loops through the pairs in the list, successively binding each key to `name` and value to `val`. 
4. The built-in function call `callable(val)` returns `True` if `val` appears callable, `False` if not. (These are likely instance methods.)

5. The call `debug(val)` applies the function-level prefix debugger we defined above to the method `val`. That is, it wraps the method with function decorator `debug`.

6. The built-in function call `setattr(cls, name, debug(val))` sets the `name` attribute of object `cls` (i.e., in its dictionary) to the value `debug(val)`.

7. The decorator function `debugmethods` returns the modified class object `cls` in place of the original class.

The code below shows the application of this new decorator to the `Account` class.

```python
@debugmethods
class Account:
    def __init__(self):
        self._bal = 0

    def deposit(self, amt):
        self._bal += amt

    def withdraw(self, amt):
        if amt <= self._bal:
            self._bal -= amt
        else:
            print(f'Insufficient funds for withdrawal of {amt}')

    def get_balance(self):
        return self._bal

    def __str__(self):
        return f'Account with balance {self._bal}'
```

Note: The Python 3.7+ source code for the above version of `Account` is available in linked file `account3.py`.

A single decorator application handles all the method definitions within the class.

Well, not quite!

It does not decorate class or static methods, such as the following which can be added to class `Account`.

```python
class Account:
    ...

    @classmethod
```
```python
def classname(cls):
    return cls.__name__

@staticmethod
def warn(msg):
    print(f'Warning: {msg}')

Note: The Python 3.7+ source code for the extended version of Account is
available in linked file account4.py.

TODO: Explain why this does not work.

Variation: Attribute access debugging

Suppose instead of printing a message on every call of a method, we do so for
each access to an attribute.

We can do this by rewriting part of the class as shown below. In particular, we
give a new implementation for the special method `__getattribute__`.

```python
def debugattr(cls):
    orig_getattribute = cls.__getattribute__

    def __getattribute__(self, name):
        print(f'Get: {name}')
        return orig_getattribute(self, name)

    cls.__getattribute__ = __getattribute__
    return cls
```

The special method `__getattribute__` is called to implement accesses to “regu-
lar” attributes of the class. It is not called on accesses to other special methods
such as `__init__` and `__str__`.

In the above, we save the original implementation of the method and then call it
to complete the access once we have printed an appropriate debugging message.

In the example below, we decorate the Account class with `@debugattr`.

```python
@debugattr
class Account:
    def __init__(self):
        self._bal = 0

    def deposit(self, amt):
        self._bal += amt

    def withdraw(self, amt):
        if amt <= self._bal:
            self._bal -= amt
```
else:
    print(f'Insufficient funds for withdrawal of {amt}!')

    def get_balance(self):
        return self._bal

    def __str__(self):
        return f'Account with balance {self._bal}''

Note: The Python 3.7+ source code for the above version of Account is available in linked file account5.py.

We can see the effects of the decorator in the following REPL session.

```python
>>> acct = Account()
>>> str(acct)
'Account with balance 0'
>>> acct.deposit(100)
Get: deposit
Get: _bal
>>> str(acct)
'Account with balance 100'
>>> acct.withdraw(60)
Get: withdraw
Get: _bal
Get: _bal
>>> str(acct)
'Account with balance 40'
>>> acct.get_balance()
Get: _bal
40
>>> str(acct)
'Account with balance 40'
```

Note that both calls to the methods and the accesses to the “private” data attribute _bal are shown. (If we want to exclude accesses to the private instance variables, we can modify debugattr to exclude attributes whose names begin with a single underscore.)
Class Hierarchy Debugging

Motivating example

Now let’s set up class-level debugging on the inheritance hierarchy P example from Chapter 7.

```python
@debugmethods
class P:
    def __init__(self, name=None):
        self.name = name
    def process(self):
        return f'Process at parent P level'

@debugmethods
class C(P):
    # class C inherits from class P
    def process(self):
        result = f'Process at child C level'
        # Call method in parent class
        return f'{result}
{super().process()}'

@debugmethods
class D(P):
    # class D inherits from class P
    pass

@debugmethods
class G(C):
    # class G inherits from class C
    def process(self):
        return f'Process at grandchild G level'
```

Note: The Python 3.7+ source code for the above version of the P class hierarchy is available in linked file `inherit2.py`.

So, we have another occurrence of code redundancy that we saw at the class level in the previous section. Let’s see if we can DRY out the code more.

To do this, the program needs to process the whole class hierarchy rooted at class P. Let’s review the nature of the Python object model to see how to do this.

Review of objects and types

In Chapters 5-7 of these notes, we learned:

- All Python values are objects.
- All objects have types.
- A class defines a new type.
A class is a callable (i.e., function) that creates instances; the class is the type of the instances it creates. Hence, in some sense, a class is a type consisting of its potential instances and the operations it defines.

A class itself is an object. It is an instance of other classes. Thus it has a type.

The built-in class type is the root class (i.e., top-level metaclass) for all other classes (i.e., types). When a program invokes type as a constructor, it creates a new type (i.e., class) object.

Classes may inherit (i.e., be a subclass of) other classes.

The built-in class object is the root class for all other top-level user-defined and built-in classes.

TODO: Maybe repeat diagram below here.

Note: See the diagram in Figure 7-1 from Chapter 7.

The following Python REPL session illustrates these concepts.

```python
>>> class PP:
...     pass
...
>>> class CC(PP):
...     pass
...
>>> PP
<class '__main__.PP'>
>>> type(PP)
<class 'type'>
>>> issubclass(P,object)
True
>>> CC
<class '__main__.CC'>
>>> type(CC)
<class 'type'>
>>> issubclass(CC,PP)
True
>>> x = PP()
>>> x
<__main__.PP object at 0x10cd3d048>
>>> isinstance(x,PP)
True
>>> type(x)
<class '__main__.PP'>
>>> type
<class 'type'>
>>> type(type)
```

18
>>> issubclass(type, object)
True
>>> object
<class 'object'>
>>> type(object)
<class 'type'>

Class definition process

Now let’s examine how the Python interpreter elaborates class definitions at runtime. Consider the class MyClass defined as follows:

```python
class MyClass(Parent):
    def __init__(self, id):
        self.id = id
    def hello(self):
        print(f'Hello from MyClass.hello, id = {self.id}')
```

This class definition has three components.

- Name: "MyClass"
- Base classes: (Parent,)
- Functions: (__init__, hello)

The interpreter takes the following steps during class definition.

1. It isolates the body of the class. (Note the multiline string below.)

   ```python
   body = '''
   def __init__(self, myid):
       self.myid = myid
   def hello(self):
       print(f'Hello from MyClass.hello, myid = {self.myid}')
   '''
   ```

2. It creates the class dictionary.

   ```python
   clsdict = type.__prepare__('MyClass', (Parent,))
   ```

Method type.__prepare__ is a class method on the root metaclass type. In the process of creating the new class object for a class, the interpreter calls the __prepare__ method before it calls the __new__ method on type [6:701–3].

In addition to metaclass argument (i.e., type), the __prepare__ class method takes two additional arguments:

- the name of the class being created (e.g., 'MyClass' above)
- a tuple of the one or more base classes (e.g., (Parent,) above)
Method __prepare__ returns a dictionary that can be subsequently passed to the __new__ and __init__ methods. This dictionary serves as the local namespace for the statements in the class body.

3. It executes the body dynamically (using exec) in the current global namespace (returned by the call globals()) and the local namespace defined by the class dictionary clsdict.

   exec(body, globals(), clsdict)

This step populates clsdict.

   >>> clsdict
   {'__init__': <function __init__ at 0x10cc21ea0>,
   'hello': <function hello at 0x10d2b9bf8>}

4. It constructs the class object using its name, its base classes, and the dictionary populated in the previous step.

   >>> MyClass = type('MyClass', (Parent,), clsdict)
   >>> MyClass
   <class '__main__.MyClass'>
   >>> mc = MyClass('Conrad')
   >>> mc.myid
   Conrad
   >>> mc.hello()
   Hello from MyClass.hello, myid = Conrad

The call type('MyClass', (Parent,), clsdict) constructs an instance of metaclass type with name MyClass, superclass Parent, and object dictionary clsdict. This is the class object for MyClass.

Note: The Python 3.7+ source code for the above creation of class MyClass is available in linked file MyClass1.py.

Changing the metaclass

A Python class definition has a keyword parameter named metaclass whose default value is type. So the parent class P from the motivating example for this section is equivalent to the following.

class P(metaclass=type):
   def __init__(self, name=None):
       self.name = name
   def process(self):
       return f'Process at parent P level'

This keyword parameter sets the class for creating the new type for the class. Although the default is type, we can change it to some other metaclass.
To define a new metaclass, we typically define a type that inherits from `type` and gives a new definition for one or both of the special methods `__new__` and `__init__`.

```python
class mytype(type):
    def __new__(cls, name, bases, clsdict):
        # possible preprocessing of arguments
        clsobj = super().__new__(cls, name, bases, clsdict)
        # possible postprocessing of object
        return clsobj
```

The special method `__new__` allocates memory, constructs a new instance (i.e., object), and then returns it. The interpreter passes this new instance to special method `__init__`, which initializes the new instance variables.

We do not normally override `__new__`, but in a metaclass we may want to do some additional work either before or after the basic construction processing.

A metaclass can access information about a class definition at the time the class is defined. It can inspect the data and, if needed, modify the data.

Given the above definition, we can use the new metaclass as follows:

```python
class P(metaclass=mytype):
    ...
```

### Debugging using a metaclass

Now we have the tools we need to remove the code redundancy from the motivating example. We can introduce the `metaclass` shown in the example below.

```python
class debugmeta(type):
    def __new__(cls, clsname, bases, clsdict):
        clsobj = super().__new__(cls, clsname, bases, clsdict)  #1
        clsobj = debugmethods(clsobj)  #2
        return clsobj  #3
```

The approach above:

1. creates the class normally (using `super().__new__`)
2. immediately wraps the class object with the class-level debug decorator `debugmethods` we developed previously
3. then returns the wrapped class object

Given the above metaclass definition, we can apply it to the inheritance example as sketched below.

```python
class P(metaclass = debugmeta):
    ...
```
class C(P):
...

class D(P):
...

class G(C):
...

Note: The Python 3.7+ source code for the above version of the P class hierarchy is available in linked file `inherit3.py`.

Now consider a Python REPL session using the above code with the custom metaclass.

```python
>>> from inherit3 import *
>>> type(P)
<class 'inherit3.debugmeta'>
>>> issubclass(P,object)
True
>>> type(C)
<class 'inherit3.debugmeta'>
>>> issubclass(C,P)
True
>>> type(G)
<class 'inherit3.debugmeta'>
>>> issubclass(G,C)
True
>>> issubclass(G,P)
True
>>> p1 = P()
P.__init__
>>> type(p1)
<class 'inherit3.P'>
>>> c1 = C()
P.__init__
>>> type(c1)
<class 'inherit3.C'>
>>> g1 = G()
P.__init__
>>> type(g1)
<class 'inherit3.C'>
>>> p1.process()
P.process
'Process at parent P level'
>>> c1.process()
C.process
P.process
'Process at child C level \n Process at parent P level'
```
Why metaclasses?

As we have seen, we can transform a class in similar ways using either a class decorator or a metaclass.

Given that a class decorator is easier to set up and apply, when and why should we use a metaclass?

One advantage to metaclasses is that they can propagate down class hierarchies. Consider our motivating example again.

```python
class P(metaclass = debugmeta):
    ...
class C(P):  # metaclass = debugmeta
    ...
class D(P):  # metaclass = debugmeta
    ...
class G(C):  # metaclass = debugmeta
    ...
```

As we can see in the REPL session output in the previous subsection, use of the metaclass in parent class P is passed down automatically to all its descendants. No changes are needed to the descendant classes.

In some sense, the metaclass mutates the DNA of the parent class and that mutation is passed on to the children. In this example, debugging is applied across the entire hierarchy. The code is kept DRY.

What Next?

In this case study, we used Python metaprogramming facilities to debug successively larger program units. But regardless of the level, the method mostly involved wrapping and rewriting the program units.

- We used function decorators to wrap and rewrite functions.
- We used class decorators to wrap and rewrite classes.
- We used metaclasses to wrap and rewrite class hierarchies.

So far, we have mostly used "classic" metaprogramming techniques that were available in Python 2 with only a few Python 3 features.

In the coming chapters, we use more advanced features of Python 3. (These chapters are planned but not yet drafted.)
Chapter Source Code
TODO

Exercises
TODO

Acknowledgements


The overall set of notes on Python 3 Reflexive Metaprogramming is inspired by David Beazley’s Python 3 Metaprogramming tutorial from PyCon’2013 [1]. In particular, some chapters adapt Beazley’s examples. Beazley’s tutorial draws on material from his and Brian K. Jones’ book *Python Cookbook* [2].

In particular, this chapter adapts Beazley’s debugly example presentation from his Python 3 Metaprogramming tutorial at PyCon’2013 [1].

I retired from the full-time faculty in May 2019. As one of my post-retirement projects, I am continuing work on this textbook. In January 2022, I began refining the existing content, integrating (e.g., using CSS), constructing a unified bibliography (e.g., using citeproc), and improving the build workflow and use of Pandoc.

I maintain these notes as text in Pandoc’s dialect of Markdown using embedded LaTeX markup for the mathematical formulas and then translate the notes to HTML, PDF, and other forms as needed.

Terms and Concepts
TODO

References


