Metaclasses

> [Metaclasses] are deeper magic than 99% of users should ever worry about. If you wonder whether you need them, you don't (the people who actually need them know with certainty that they need them, and don't need an explanation about why).

Tim peters – veteran python core developer

To Metaclass or Not to Metaclass

- In other words, metaclasses are primarily intended for a subset of programmers building APIs and tools for others to use. In many (if not most) cases, they are probably not the best choice in applications work.
- Still, metaclasses have a wide variety of potential roles, and it's important to know when they can be useful.
- For example, they can be used to enhance classes with features like tracing, object persistence, exception logging, and more.
- They can also be used to construct portions of a class at runtime based upon configuration files, apply function decorators to every method of a class generically, verify conformance to expected interfaces, and so on.

Metaclasses

- It allows us to insert logic to be run automatically at the end of a class statement, when a class object is being created.
- Though strongly reminiscent of class decorators, the metaclass mechanism doesn't rebind the class name to a decorator callable's result, but rather routes creation of the class itself to specialized logic.
- In other words, metaclasses are ultimately just another way to define automatically run code.

Decorators vs Metaclasses

- Class decorators run after the decorated class has already been created. Thus, they are often used to add logic to be run at instance creation time.
- Metaclasses, by contrast, run during class creation to make and return the new client class. Therefore, they are often used for managing or augmenting classes themselves and can even provide methods to process the classes that are created from them, via a direct instance relationship.
- For example, metaclasses can be used to add decoration to all methods of classes automatically, register all classes in use to an API, add user-interface logic to classes automatically, create or extend classes from simplified specifications in text files, and so on.

The Metaclass Model

- Classes are instances of type.
  - In Python 3.X, user-defined class objects are instances of the object named type, which is itself a class.
  - In Python 2.X, new-style classes inherit from object, which is a subclass of type; classic classes are instances of type and are not created from a class.
- Metaclasses are subclasses of type.

Classes are Types, and Types are Classes

- Types are defined by classes that derive from type.
- User-defined classes are instances of type classes.
- User-defined classes are types that generate instances of their own.

Metaclasses Are Subclasses of Type

- type is a class that generates user-defined classes.
- Metaclasses are subclasses of the type class.
- Class objects are instances of the type class, or a subclass thereof.
- Instance objects are generated from a class.

In other words, to control the way classes are created and augment their behavior, all we need to do is specify that a user-defined class be created from a user-defined metaclass instead of the normal type class

Class Statement Protocol

- We've already learned that when Python reaches a class statement, it runs its nested block of code to create its attributes - all the names assigned at the top level of the nested code block generate attributes in the resulting class object.
- These names are usually method functions created by nested defs, but they can also be arbitrary attributes assigned to create class data shared by all instances.
- Technically speaking, Python follows a standard protocol to make this happen at the end of a class statement, and after running all its nested code in a namespace dictionary corresponding to the class's local scope, Python calls the type object to create the class object like this.

```
class = type(classname, superclasses, attributedict)
```

Class Statement Protocol

- The type object in turn defines a __call__ operator overloading method that runs two other methods when the type object is called.

```
type.__new__(typeclass, classname, superclasses, attributedict)
type.__init__(class, classname, superclasses, attributedict)
```

- The __new__ method creates and returns the new class object, and then the __init__ method initializes the newly created object. As we'll see in a moment, these are the hooks that metaclass subclasses of type generally use to customize classes.

For example, given a class definition like the following for `Spam`:

```
class Eggs: ...                    # Inherited names here

class Spam(Eggs):                  # Inherits from Eggs
    data = 1                       # Class data attribute
    def meth(self, arg):           # Class method attribute
        return self.data + arg
```

Python will internally run the nested code block to create two attributes of the class (`data` and `meth`), and then call the `type` object to generate the `class` object at the end of the `class` statement:

```
Spam = type('Spam', (Eggs,), {'data': 1, 'meth': meth, '__module__': '__main__'})
```

Declaring Metaclasses

- As we've just seen, classes are created by the type class by default.
- To tell Python to create a class with a custom metaclass instead, you simply need to declare a metaclass to intercept the normal instance creation call in a user-defined class.
- How you do so depends on which Python version you are using.

Declaration in 3.X

In Python 3.X, list the desired metaclass as a *keyword* argument in the `class` header:

```
class Spam(metaclass=Meta):                    # 3.X version (only)
```

Inheritance superclasses can be listed in the header as well. In the following, for example, the new class `Spam` inherits from superclass `Eggs`, but is also an instance of and is created by metaclass `Meta`:

```
class Spam(Eggs, metaclass=Meta):              # Normal supers OK: must list first
```

In this form, superclasses must be listed before the metaclass; in effect, the ordering rules used for keyword arguments in function calls apply here.

Declaration in 2.X

We can get the same effect in Python 2.X, but we must specify the metaclass differently —using a *class attribute* instead of a keyword argument:

```
class Spam(object):                          # 2.X version (only), object optional?
    __metaclass__ = Meta

class Spam(Eggs, object):                    # Normal supers OK: object suggested
    __metaclass__ = Meta
```

Coding Metaclasses

- Metaclasses are coded with normal Python class statements and semantics.
- They are simply classes that inherit from type.
- Their only substantial distinctions are that Python calls them automatically at the end of a class statement, and that they must adhere to the interface expected by the type superclass.

A Basic Metaclass

- Perhaps the simplest metaclass you can code is simply a subclass of type with a __new__ method that creates the class object by running the default version in type.
- A metaclass __new__ like this is run by the __call__ method inherited from type; it typically performs whatever customization is required and calls the type superclass's __new__ method to create and return the new class object.

```
class Meta(type):
    def __new__(meta, classname, supers, classdict):
        # Run by inherited type.__call__
        return type.__new__(meta, classname, supers, classdict)
```

```
class MetaOne(type):
    def __new__(meta, classname, supers, classdict):
        print('In MetaOne.new:', meta, classname, supers, classdict, sep='\n...')
        return type.__new__(meta, classname, supers, classdict)

class Eggs:
    pass

print('making class')
class Spam(Eggs, metaclass=MetaOne):       # Inherits from Eggs, instance of MetaOne
    data = 1                               # Class data attribute
    def meth(self, arg):                   # Class method attribute
        return self.data + arg

print('making instance')
X = Spam()
print('data:', X.data, X.meth(2))
```

What happened?

- Here, Spam inherits from Eggs and is an instance of MetaOne, but X is an instance of and inherits from Spam.
- When this code is run with Python 3.X, notice how the metaclass is invoked at the end of the class statement, before we ever make an instance - metaclasses are for processing classes, and classes are for processing normal instances.

Inheritance and Instance

- Metaclasses inherit from the type class (usually) .
- Metaclass declarations are inherited by subclasses.
- Metaclass attributes are not inherited by class instances.
- Metaclass attributes are acquired by classes.

```
# File metainstance.py

class MetaOne(type):
    def __new__(meta, classname, supers, classdict):          # Redefine type method
        print('In MetaOne.new:', classname)
        return type.__new__(meta, classname, supers, classdict)
    def toast(self):
       return 'toast'

class Super(metaclass=MetaOne):                               # Metaclass inherited by subs too
    def spam(self):                                           # MetaOne run twice for two classes
        return 'spam'

class Sub(Super):                                             # Superclass: inheritance versus instance
    def eggs(self):                                           # Classes inherit from superclasses
        return 'eggs'                                         # But not from metaclasses
```

Metaclass Versus Superclass

- In even simpler terms, watch what happens in the following: as an instance of the A metaclass type, class B acquires A's attribute, but this attribute is not made available for inheritance by B's own instances—the acquisition of names by metaclass instances is distinct from the normal inheritance used for class instances:

```
>>> class A(type): attr = 1
>>> class B(metaclass=A): pass          # B is meta instance and acquires meta attr
>>> I = B()                             # I inherits from class but not meta!
>>> B.attr
1
>>> I.attr
AttributeError: 'B' object has no attribute 'attr'
>>> 'attr' in B.__dict__, 'attr' in A.__dict__
(False, True)
```

Metaclass Versus Superclass

- By contrast, if A morphs from metaclass to superclass, then names inherited from an A superclass become available to later instances of B, and are located by searching namespace dictionaries in classes in the tree—that is, by checking the __dict__ of objects in the method resolution order (MRO)

```
>>> class A: attr = 1
>>> class B(A): pass                    # I inherits from class and supers
>>> I = B()
>>> B.attr
1
>>> I.attr
1
>>> 'attr' in B.__dict__, 'attr' in A.__dict__
(False, True)
```

Metaclass Methods

- Just as important as the inheritance of names, methods in metaclasses process their instance classes - not the normal instance objects we've known as "self," but classes themselves.
- This makes them similar in spirit and form to the class methods.

```
>>> class A(type):
        def x(cls): print('ax', cls)          # A metaclass (instances=classes)
        def y(cls): print('ay', cls)          # y is overridden by instance B

>>> class B(metaclass=A):
        def y(self): print('by', self)        # A normal class (normal instances)
        def z(self): print('bz', self)        # Namespace dict holds y and z

>>> B.x                                       # x acquired from metaclass
<bound method A.x of <class '__main__.B'>>
>>> B.y                                       # y and z defined in class itself
<function B.y at 0x0295F1E0>
>>> B.z
<function B.z at 0x0295F378>
>>> B.x()                                     # Metaclass method call: gets cls
ax <class '__main__.B'>

>>> I = B()                                   # Instance method calls: get inst
>>> I.y()
by <__main__.B object at 0x02963BE0>
>>> I.z()
bz <__main__.B object at 0x02963BE0>
>>> I.x()                                     # Instance doesn't see meta names
AttributeError: 'B' object has no attribute 'x'
```

Metaclass Methods Versus Class Methods

- Though they differ in inheritance visibility, much like class methods, metaclass methods are designed to manage class-level data.
- In fact, their roles can overlap - much as metaclasses do in general with class decorators - but metaclass methods are not accessible except through the class, and do not require an explicit classmethod class-level data declaration in order to be bound with the class.
- In other words, metaclass methods can be thought of as implicit class methods, with limited visibility

```
>>> class A(type):
        def a(cls):                          # Metaclass method: gets class
            cls.x = cls.y + cls.z

>>> class B(metaclass=A):
        y, z = 11, 22
        @classmethod                         # Class method: gets class
        def b(cls):
            return cls.x

>>> B.a()               # Call metaclass method; visible to class only
>>> B.x                 # Creates class data on B, accessible to normal instances
33

>>> I = B()
>>> I.x, I.y, I.z
(33, 11, 22)

>>> I.b()               # Class method: sends class, not instance; visible to instance
33
>>> I.a()               # Metaclass methods: accessible through class only
AttributeError: 'B' object has no attribute 'a'
```

Operator Overloading in Metaclass Methods

- Just like normal classes, metaclasses may also employ operator overloading to make built-in operations applicable to their instance classes.
- The __getitem__ indexing method in the following metaclass, for example, is a metaclass method designed to process classes themselves - the classes that are instances of the metaclass, not those classes' own later instances.
- In fact, per the inheritance algorithms sketched earlier, normal class instances don't inherit names acquired via the metaclass instance relationship at all, though they can access names present on their own classes.

```
>>> class A(type):
        def __getitem__(cls, i):          # Meta method for processing classes:
            return cls.data[i]            #  Built-ins skip class, use meta
                                          #  Explicit names search class + meta
>>> class B(metaclass=A):                 #  Data descriptors in meta used first
        data = 'spam'

>>> B[0]                    # Metaclass instance names: visible to class only
's'
>>> B.__getitem__
<bound method A.__getitem__ of <class '__main__.B'>>

>>> I = B()
>>> I.data, B.data          # Normal inheritance names: visible to instance and class
('spam', 'spam')
>>> I[0]
TypeError: 'B' object does not support indexing
```

Example: Adding Methods to Classes

p.1391

Example: Applying Decorators to Methods

p.1400