Python Metaclasses: Who? Why? When?

> [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 (c.l.p post 2002-12-22)

Python Metaclasses: Who? Why? When?

So let's stop wondering if we need them...

Meta-classes are about meta-programming

- Programming where the clients are programmers
  - Language development (e.g. python-dev crowd)
  - Library/framework development (e.g. Zope Corp.)
- Enabling new metaphors/approaches for programming
  - Aspect-oriented, Interface-oriented, Prototype-based
- Creating natural programming patterns for end-programmers
  - Generally created for use within an application domain
  - Programming with the resulting classes maps between Python and domain semantics closely
- Generality and uniformity

Meta-programming goals

- Allows end-programmers to use standard Python programming features
  - Leverage existing end-programmer's Python knowledge
  - Domain-specific features feel “built-in”, not “bolted on”
  - End-programmer should be able to use declarative approach in defining classes
    - “this is a that”, not “register this as a that”
- Integrates nicely with other Python systems, introspection, pickling, properties
- Works to simplify and beautify APIs

Meta-classes facilitate meta-programming

- In general, there's nothing you can't do some other way
  - Factory classes
  - Client classes coded to use a stand-in object in a class-like manner from which end-programmers must derive
  - Function-calls to post-process classes after they are created
  - Other function calls to register features with the system...
- Meta-classes just make it easier and more elegant
  - They're the basis of Python 2.2+'s type system, so they're standard, and reliable
  - There are two things you can't do any other way, meta-methods and meta-properties (more on those later)

That's great and all, but what are they good for?

Let's see...

What can you do with them? Class registration

- In “Aspect-oriented” programming
  - Automate registration of join-points and aspects from declarative structures and introspection
  - Look up aspects at run-time from system registries and encode directly in business-domain classes
- In Interface-oriented programming
  - Register utilities, services, interfaces and adapters
- In a more general sense, you can automatically register information about an end-programmer's classes at the time the class is created

Class registration example...

- In Aspect-oriented programming, every end-user-class's methods and properties need to be registered with the system to allow for matching “join points” (features) with “cuts” and “aspects” (choke-points and reactive code)
- You could force the end-programmer to make a call:
  - system.register(myclass) for every class, but the point is that **every** aspect-oriented class be registered, so you've got pointless busy-work, and lots of opportunities for failure
- You want to say “when someone sub-classes 'AspectAware', register the resulting class for aspect-oriented servicing”
- (Similar requirements with registering adapters for interface-oriented programming)

What can you do with them? Class verification

- Automated constraints and class-checking
  - Interface-checking for interface-based programming
  - Check for internal coherence in the face of end-programmer errors
  - Warped things such as creating final classes (classes which complain when an attempt is made to sub-class them)
- In a more general sense, you can check end-programmer's classes for conformance to any pattern required by your systems

Class verification example

- In Interface-oriented programming, each class declares a set of interfaces that it provides.
- During development, you want to automatically check every interface-aware class for conformance to its published interfaces (including those created by end-programmers)
- You **could** (have the end-programmer) call:
  - verifyInterfaces(classObject)
- It's something the end-programmer can easily forget, so there's no guarantee that all interface-aware classes have been checked
- You **want** to say “when someone sub-classes 'InterfaceAware', check the resulting class's interface declarations

What can you do with them? Class construction

- Run-time inclusion or modification of methods, properties, or attributes from:
  - Declarative structures
  - Databases or files
  - Application plug-ins about which the end-programmer's class has no knowledge
  - Calculations based on the current phase of the moon
  - Caching or short-circuiting creation
  - Precondition/postcondition wrappers, etceteras
- In a more general sense, you can use arbitrarily complex code to alter a class at instantiation without the end-programmer needing to know anything about the process

Class construction example

- When creating an object-relational mapper, you want the end-programmer to be able to declare which table a class implements and have the class automatically acquire descriptors (properties) for the fields in the database
- You want to be able to either use a pre-existing database description, or reverse-engineer the database description from the live database to provide the features
- You could have the end-programmer call:
  - buildPropertiesFromTable(cls)
- But what happens when the end-programmer forgets? And why should they have to remember anyway?

What can you do with them? First-class classes

- Meta-classes let you customise the behaviour of class-objects with OO methods
  - Meta-properties, properties of class objects accessed via class.name
  - Meta-methods, methods which can be called on the class-object but which are not visible to class-instances
  - Inheritance trees for meta-classes instances to share code among multiple meta-classes
- In a more general sense, allow you to treat a class-object very much like a regular instance object, letting your programs “talk about” classes and their functionality naturally

First-class classes example

- Where classes are generated from and linearised to documents (think auto-generating VRML97 or XML node/tag classes), you want to be able to attach functionality and data to the class objects without cluttering up their instance namespace
- You could use external global storage and write a “utility” module to do the work of manipulating the classes as objects
- Could avoid using class:instance relationships have separate object very much like a class, but not. Gets clunky pretty fast
- You'd like to define a class which describes the object-like functionality of the class-object itself. You'd like to define methods in this class which process the class as a first-class object.

What can you do with them? Modeling domains

- Model systems with class-like behaviour
  - XML DTDs and XML tags (e.g. gnosis.xml)
  - VRML97 Prototypes and Nodes (e.g. OpenGLContext)
  - Object-Relational Mappers
- Anywhere there is a “type of type” relationship
  - Particularly where it's useful to be able to add functionality to the type-of-type (watch for “meta-methods” later), or where the operation/construction of the type-of-type differs from normal Python classes
- Domain objects show up as classes and instances in the end-programmer's view, allowing easy customisation/inheritance

Okay, enough already, they're useful...

- So what are they?

Quicky definitions:

- The type of a type, type(type(instance))
- instance.__class__.__class__
- Objects similar to the built-in “type” meta-class
- Objects which provide a type-object interface for objects which themselves provide a type-object interface
- A factory for classes
- The implementation definition for a class-object
- Classes implementing the first-class class-objects in Python
- A way of describing custom functionality for entire categories of classes/types
- A way of customising class-object behaviour

About instances and classes

- An instance object's relationship to its class object is solely via the class interface
  - Instance knows which object is playing the role of its class
  - It normally has no other dependencies on the class (e.g. no special internal layout, no cached methods or properties)
- Class of an object is whatever object plays the role of the class
  - Can be changed by assigning new class to __class__
- Normally classes are implemented via the “type” meta-class
  - Very simple and straightforward class, mostly data storage
  - A place to store descriptors and some common structures
- Interactions implemented in the interpreter

More about class-instance relationships...

- Classes are normally callable to create new instances
  - Default is to provide 2 hooks, __new__ and __init__ for customisation of new instances
  - There's nothing special about this functionality, any Python object with a __call__ method is callable
- Interpreter asks questions about the class to answer questions about the instance (methods, attributes, isinstance queries)
  - Class attributes and descriptors are stored in the class dictionary, just like regular instance attributes
  - Interpreter retrieves values from class.__dict__ directly

About super-classes...

- Super-classes of a class-object are just other class-objects with a role "superclass" defined by being in the __bases__ of the class
  - Can be any object implementing the class API
  - Don't need to be same type of object as the sub-class
  - Used by interpreter to lookup attributes for instances
  - They don't alter the functionality of the class object itself
- The interpreter implements chaining attribute lookup for classes w/out going through regular object attribute lookup
  - Means that meta-class hooks for attribute access don't intercept instance-object lookup, only class-object lookup

So, then, a class-object is just...

- A very simple object with a few common attributes
  - __name__, __bases__, __module__ and __dict__
  - __mro__ and a few other goodies in new-style classes
- Something which plays the role of a class for another object
- Normally implemented by a class called “type”, (a built-in)
  - Has an internal layout which makes common operations fast (e.g. lookup of inherited attributes)
  - That internal layout requires inheriting from type (or another C-programmed meta-class with the same base structure)
- Passive, basically just data-storage

Meta-classes implement class-objects

- Something has to implement those simple class-objects
  - In Python, objects are normally implemented by classes
  - So there should be a class which implements classes
  - There is, it's called “type”
- All meta-classes have to implement the same class interface
  - Requires inheriting from another meta-class (e.g. type)
  - You customise **how** the class interface is implemented to some degree, though there's not much to customise
- Most of the time customisations focus on initialisation of the class, as the type object is fairly passive once initialised
  - Interpreter does most of the implementation work

Customising meta-classes: hooks to hang code

- Initialisation
  - The __metaclass__ hook, intercepts the interpreter's call to create a class object from a class declaration
  - __new__ and __init__ methods, just as with any class
- Descriptors and attribute-access for classes
  - Methods for class-objects
  - Properties for class-objects (with some restrictions)
  - Do **not** show up in instances, (interpreter uses __dict__ only for instance-attribute lookup)
  - Can use most regular class-instance features to customise the behaviour of class-objects

The meta-class hook: class statement hook

- Invoked when a class-statement in a namespace is executed (at the end of the entire class statement)
  - The declared meta-class is asked to create a new class
  - The meta-class can customise the creation and initialisation of the class-object, returning whatever object is desired
  - That object is assigned the declared name in the namespace where the statement occurred
- The class-statement is turned into a name, a set of bases, and a dictionary, and these are passed to the meta-class to allow it to create a new class-object instance.

What the class statement does when you aren't looking

```
class X( Y, Z ):
    x = 3
--> Here the interpreter calls (or at least approximates calling):
    metaclass( 'X', (Y,Z), {'x': 3, '__module__': '__main__'} )

To see the results, try doing the following:
    type( 'X', (object,), { '__module__':'__main__' })
```

The meta-class hook: class statement hook (cont.)

- Meta-class declaration can be in module or class scope
  - Is resolved by the interpreter before trying to create the class
  - Can be inherited from super-classes and overridden in sub-classes
  - Note: In Python 2.2.3, the meta-class object's __call__ is **not** called by the metaclass hook, the interpreter calls __new__, then __init__ directly
- This pattern of intercepting statement completion is unique at the moment within Python
  - It's reminiscent of first-class suites/blocks as seen in Ruby

metamodulehook.py

```
# type is a meta-class
# This statement affects all class statements in this scope
# which are *not* otherwise explicitly declared
__metaclass__ = type

class X:
    pass
assert type(X) is type
print 'Type of X', type(X)
```

metaclasshook.py

```
class Meta( type ):
    x = 3

class Y:
    __metaclass__ = Meta

print 'Type of Y', type(Y)
assert type(Y) is Meta

class Z(Y): # note that meta-class is inherited by the class!
    pass
assert type(Z) is Meta
print 'Type of Z', type(Z)
```

Meta-class class-initialisation hooks

- On class-statement completion, interpreter asks meta-class to create instance
  - Is as if you were to call metaclass(name, bases, dictionary)
- Creates a new class instance and initialises it, these two methods then become our primary customisation points for initialising a meta-class instance (a class)
  - __new__(metacls, name, bases, dictionary)
  - __init__(cls, name, bases, dictionary)
- See metainitialisation.py, meta__new__.py, meta__init__.py

Meta-class class-initialisation hooks continue...

- In __new__:
  - You can modify bases
  - You can modify name
  - You can return arbitrary objects as class object
- In either __new__ or __init__:
  - You can modify dictionary
  - You can inject, remove or wrap methods
  - You can do any amount of checking/confirmation you want
  - You can do any amount of processing you need to initialise the class-object

meta__new__.py

```
class Meta( type ):
    def __new__( metacls, name, bases, dictionary ):
        print 'new:', metacls, name, bases
        if name == 'Z':
            return X
        return super( Meta, metacls ).__new__( metacls, name, bases,
          dictionary )
__metaclass__ = Meta
class X:
    pass
class Y(X):
    pass
class Z:
    pass
print 'Z', Z # is class X
```

meta__init__.py

```
class Meta( type ):
    def __init__( cls, name, bases, dictionary ):
        """Initialise the new class-object"""
        if dictionary.has_key( 'fields' ):
            print 'build stuff here for %r, insert in dict'%(name,)

__metaclass__ = Meta


class X:
    fields = ('x','y,','z')
class Y:
    fields = ('q','r')
class Z:
    pass
```

Meta-class attribute and descriptor hooks

- Want to alter run-time behaviour of a class-object
- Modify attribute-access patterns **for the class** object itself (not instances)
  - Properties metaproperty.py
  - Lazy resolution of failed attribute lookup metagetattr.py
  - Exceptions on class-object attribute assignment metasetattr.py
  - Implement access restrictions
- Provide utility methods on the class object
  - Meta-methods for operating on a class-object without being visible to instances metamethod.py
  - For example storage mechisms (eGenix xml tools)

metagetattribute.py

```
class Meta( type ):
    def __getattribute__( cls, key ):
        print 'Meta: getattribute:', cls, key
        return super(Meta,cls).__getattribute__( key )
    x = 3 # overridden by the class
class SomeClass(object):
    __metaclass__ = Meta
    x = 4

print 'get class attribute'
print SomeClass.x # 4
print 'creating new instance'
v = SomeClass() # __getattribute__ for __new__
print 'get instance attribute'
print v.x # 4 as well, but no __getattribute__
```

metagetattr.py

```
class Meta( type ):
    """Meta-class with getattr hook"""
    def __getattr__( cls, name ):
        return 42

class X:
    __metaclass__ = Meta

print X.x, X.y, X.this
print X().x # raises attribute error
```

metasetattr.py

```
class Meta( type ):
    """Meta-class which warns on setting class attributes"""
    def __setattr__( cls, name, value ):
        raise TypeError( """Attempt to set attribute: %r to %r"""%(
            name, value,
        ))

class X:
    __metaclass__ = Meta

X.this = 42
```

metaproperty.py

```
class Meta( type ):
    """Meta-class with a meta-property"""
    def get_name( cls ):
        return 'MetaInstance%s'%( id(cls), )
    __name__ = property( get_name )

class X( object ):
    __metaclass__ = Meta

# this uses the meta-property for lookup
print X.__name__
# note that this uses the __name__ in the class dictionary
# for the __repr__
print X()
```

metamethod.py

```
class Meta( type ):
    """Meta-class with a meta-method"""
    someMappingOrOther = {}
    def registerMeGlobally( cls, key ):
        cls.someMappingOrOther[ key ] = cls
    def getRegistered( cls, key ):
        """Get cls registered w/ registerMeGlobally"""
        return cls.someMappingOrOther.get( key )
    getRegistered = classmethod( getRegistered )

class X:
__metaclass__ = Meta
X.registerMeGlobally( 'a' )

print 'a', Meta.getRegistered( 'a' )
```

Future possibilities

- Provide hook for customising instance-attribute lookup
  - Would allow customisation method-resolution order, for instance
- Hooks for instantiating other block-types or syntactic constructs
  - Functions, methods, if-statements, for-statements
  - List comprehensions, lists, dictionaries, modules
- Way to implement meta-properties cleanly
  - Low-level-setattr hook for classes