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...

Metaclasses are about meta-programming

- Programming where the clients are programmers
  - Language development (python-dev crowd)
  - Framework development
    - Zope
    - Twisted
    - PEAK/PyProtocols
- Programming to enable new metaphors/approaches to programming
  - Aspect-oriented
  - Interface-oriented
  - Prototype-based

Meta-programming with classes

- Extending the language with new “types of classes”
- Altering the nature of classes
  - Adding functionality (e.g. metaclass-methods)
  - Creating class-like objects (e.g. prototypes) which can be instantiated
  - Enforcing constraints on classes
- Automating complex tasks
  - Registration, annotation and introspection
  - Interactions with services/utilities

Meta-programming goals

- Create natural programming patterns for end-programmers
  - Generally for use within an application domain
  - Programming with the framework should map between Python and domain semantics closely
- Allow clients to use standard Python programming features
  - Fit Python semantics as closely as possible (take advantage of Python knowledge)
  - Make domain-specific features feel “built-in”
  - Integrate nicely with generic systems; introspection, pickling, properties

Meta-programming goals (cont.)

- Enable declarative approach (arguable)
  - “This is a that”, not necessarily “register this as a that”
  - “This implements that”
  - “This is persistent”
  - “This uses that join point”
- Simplify and beautify APIs
- While doing this, avoid the dangers of “too much magic”
  - The converse of fitting Python semantics closely
  - Going too far can make the system opaque

Metaclasses as a tool for meta-programming

- There's little you can't do some other way
  - Factory classes
  - Stand-in objects used in a class-like manner
  - Function calls to process classes after they are created
  - Function calls to register classes with the system
- Metaclasses just make it easier and more elegant
- Basis of Python 2.2+'s type system; standard, and reliable
- Meta-methods and meta-properties (more on those later)
  - You can't do these any other way

So what are they good for?

Let's see some use-cases for metaclasses...

What can you do with them? Class registration

You want all classes registered with the system...

- Provide interface registration (IOP)
  - Automate discovery of class features (see next slide)
- Provide join-point/aspect registration (AOP)
  - Register all classes with given join-points
  - Register all classes providing given aspect
- Allow discovery of classes based on class metadata (of any type) via registration and lookup

Class registration – Interface Oriented Programming

For IOP, we want to register...

- Utilities and services
  - Find class-based services (e.g. classmethods, singletons)
- Implemented interfaces (incl. partial implementation)
  - Allow search based on supported interface
  - Give me something which does "that"
- Adapters
  - Adapt from interface to interface
  - Give me a way to make "this" do "that"
  - Need global registration to plan appropriate adaptation

Class registration – Aspect Oriented Programming

- Register join-points for each domain object class
  - Functional operations which may be serviced by any number of different aspects every class must be registered or the cross-cutting doesn't work
    - Accesses to methods or properties, for instance
  - “Declare that objects of a class use a given join-point”
- Register aspects for servicing the domain objects (less likely)
  - Aspects implement join-points
  - “Show all aspects which can implement a given join-point” or “Lookup all loaded aspects for a given join-point”

Class registration – Use case summary

- In a more general sense, you can automatically register information about an end-programmer's classes at the time the class is created
- Registration is normally fairly benign, it may affect class-object lifetimes, but it's not normally terrible intrusive in client's day-to-day experience of your system

Class registration – Traditional approach

- Define a helper function/method in the framework
- Mandate that all user-defined classes have the helper method called on the user-defined class
- Provide checking in the system to watch for unregistered classes and complain and/or call the registration method
- Depends on the end-developer remembering to register and/or being able to catch all usage in the system

Class registration traditional sample code

```
# Required for every Product class, registers
# constructors, interfaces, icon, container
# filters, and visibility.  If you forget me
# you shall be forever cursed!

class MyClass(Folder):
    """Example of registration function client"""

# ah, if only we could call this automatically
# at the end of the class-definition statement!
ProductContext.registerClass(MyClass)
```

What can you do with them? Class verification

- Automated constraints
  - “Ensure classes provide all declared interfaces”
  - “Check for internal coherence of class declaration”
  - “Complain on attempt to subclass a 'final' class”
  - “Complain on overriding of 'final' method”
- Class-format checking
  - “Enforce coding standards (complain if improper)”
    - Docstrings, method names, inheritance depth, etc.
  - “Enforce inclusion of security declarations”
  - “Enforce definition of given method/property (abstract)”

What can you do with them? Class verification

- In a more general sense, you can check end-programmer's classes for conformance to any pattern required by your system and refuse to load incorrectly formed classes
- Careful not to be too rigid
  - More intrusive than registration, likely to be used more sparingly than registration as a result
  - Normally you'll be raising errors and preventing application or plug-in loading
  - Have to code to watch for abstract intermediate classes

Class verification – Traditional approach

- As with registration, define a utility function/method
  - You rely on the end-programmer calling the function
  - You need defensive programming throughout system to check for un-verified classes being used
- Or have each instance verify class on instantiation (inelegant, class gets verified potentially thousands of times and/or needs bookeeping for verification)

Class verification – Traditional code example

```
"""Traditional verification sample code"""
class Mojo:
    pass
# Egads, do my clients really have to remember all this?
# If only, if only there were some way to hook this
# end-of-class-statement point in my code!
package.verifyInterfaces(Mojo)
package.verifyAspects(Mojo)
package.verifyConstraints(Mojo)
```

What can you do with them? Class construction

We want to rewrite a class definition at run-time...

- Modify declared methods, properties or attributes
  - Precondition/postcondition wrappers
  - Method wrapping in general
  - Adding (e.g. injecting a "save to database" method for all domain classes if a database is configured, otherwise not)
  - Renaming (e.g. as part of creating a property wrapper)
  - Processing human friendly declarative structures (such as security information) into machine-friendly structures
- Cache or short-circuit class creation

Class construction – More use cases

- Load/create bases/methods/descriptors from system state:
  - Declarative structures in the class definition
  - Databases or data files
  - Application plug-ins or registered aspects
  - Calculations based on the current phase of the moon
- Load/create bases/methods/descriptors from non-python definitions:
  - XML DTDs, VRML PROTOs, DB Connections, IDL
  - User interactions (e.g. choosing features from a GUI)
  - Only use-case described where we're not asking clients to write Python code

Class construction – Use case summary

- 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.
- Again, the caveat applies, too much magic can kill your usability

Class construction – Traditional approach

- Create factory function to produce a class
  - From a partially-constructed class (mix-in) or
    - Awkward due to the creation of two different classes (mix-in and final)
    - For instance, tricks are needed to make the final class pickleable
  - From name, base-classes and a dictionary
    - Hard to use; no longer looks like a class definition
- Suffers the same problems as verification and registration functions (must be remembered, and must therefore be guarded against)

Class construction – Traditional mix-in example

```
"""A Traditional mix-in approach"""
def myFactory(mixIn):
    newSpace = {}
    newSpace.update(replaceMethods(mixIn))
    newSpace.update(loadPropertiesFromFile(mixIn.propertyFile))
    newSpace['module'] = hackToGetModuleName() # icky, always
    return type(mixIn.__name__, (mixIn, Base), newSpace)

class X:
    propertiesFile = 'someprops.prop'
    def r(self):
        pass
X = myFactory(X) # note re-binding
```

Class construction – Traditional deconstructed ex.

```
"""A "de-constructed" factory-function approach"""
def myFactory(name, bases, dictionary):
    dictionary.update(replaceMethods(dictionary))
    dictionary.update(loadPropertiesFromFile(dictionary['propertyFile']))
    dictionary['module'] = hackToGetModuleName() # icky, always
    return type(name, bases, dictionary)

# ick, methods at module scope
def r(self):
    pass
# even ickier and annoying, lots of
# duplicated code...
_d = {
    'propertiesFile': 'someprops.prop', 'r': r,
}
X = myFactory('X', (Base, ), _d)
```

Class construction – Traditional approach (alternate)

- Directly manipulate the class object with a function
  - A “mutator” function
  - Violates the encapsulation of the class
  - Seen, for instance in Zope security-declarations
- Suffers the same problems as for verification and registration functions, but tends to be preferred because it's the least intrusive of the constructive approaches

Class construction – Traditional mutator example

```
def myMutator(cls):
    replaceMethodsIn(cls)
    for key, value in loadPropertiesFromFile(cls.propertyFile):
        setattr(cls, key, value)

class X(Base):
    propertiesFile = 'someprops.prop'
    def r(self):
        pass
# Wouldn't it be nice if there were a hook here
# at the end of the class definition statement
# that let us call our mutator function on the
# new class?
myMutator(X)
```

What can you do with them? First-class classes

- Customize behaviour of class-objects with OO features (noting that normally classes are not particularly active)
  - Attach attributes to class objects (not visible to instances of the class, potentially property objects)
  - Attach methods which can be called on the class-object but are not visible to class-instances
  - Alter basic behaviour such as __str__
- Define class-like objects which have instances, but are themselves data to be processed; providing introspection, data storage and encapsulated functionality
- Use inheritance patterns to minimize code duplication among these object-types

First-class classes – Use-case summary

- Model systems with class-like behaviour
  - XML DTDs and XML tags
  - VRML97 Prototypes and Nodes
  - Object-Relational Mappers (Tables and Rows)
- In a more general sense, allow you to treat a class-object very much like a regular instance object, letting your programs reason about classes and their functionality naturally.

First-class classes – Traditional approaches

- Store methods and properties external to class
  - e.g. global weakref-based dictionary
  - Use utility functions to process classes
- Store methods and properties in data-classes
  - Inject the features into individual classes, (cluttering the namespace of the instances as you do)
- Create stand-in objects which act much like classes and to which instances delegate much of their operation (via __getattr__ hooks and the like)

What can you do with them? Summary

- Register classes at creation-time
- Verify classes at creation-time
- (Re-)construct class definitions
- Treat classes as first-class objects about which your systems can reason effectively

Okay, enough already, they can be useful...

- What are they?

Quicky definitions:

- The type of a type, type(type(instance))
- instance.__class__.__class__
- Objects similar to the built-in “type” metaclass
- Objects which provide a type-object interface for objects which themselves provide a type-object interface
- Factories for classes
- Implementation definitions for class-objects
- 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
  - Normally has no other dependencies on the class (e.g. no special internal layout, no cached methods or properties)
    - Built-in types and __slots__ are exceptions, they do have internal format dependencies
- Class of an object is whatever object plays the role of the class
  - Can be changed by assigning new class to __class__. (Save where there's special dependencies on the class (see above))

More about class-instance relationships...

- Interactions are generally implemented in the interpreter
- Classes are normally callable to create new instances
  - Default __call__ provides 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

More about class-instance relationships...

- The interpreter “asks” questions about the class to answer questions about the instance (methods, attributes, isinstance queries), but it generally doesn't “ask” the class itself.
  - A class-object's attributes are normally stored in the class's dictionary, just like regular instance attributes
  - The interpreter retrieves values from class.__dict__ directly – it doesn't go through attribute lookup on the class to get an instance's attribute
  - The class-object's dictionary is normally full of class attributes and descriptors to customise the behaviour of instances

About super-classes...

- The super-classes of a class-object are just other class-objects with a role “superclass” (basically “being in the __bases__/__mro__ of the class”)
  - Used by interpreter to lookup attributes for instances
  - Can be any object(s) implementing the class API
  - Don't need to be same type of object as the sub-class
  - Don't alter the functionality of the class object itself
- The interpreter implements chaining attribute lookup (inheritance) for classes w/out going through class-attribute lookup, that is, the interpreter doesn't ask the class how to lookup instance attributes in superclasses

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

- Something which plays the role of a class for another object
- Passive
  - Data-storage for instances queried by the interpreter to implement instance attribute-lookup semantics
- A very simple object with a few common attributes
  - __name__, __bases__, __module__ and __dict__
  - __mro__ and a few other goodies in new-style classes
  - __call__, __repr__, __cmp__ etceteras

Metaclasses 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 the metaclass
- All metaclasses have to implement the same C-level interface
  - Internal layout allows fast/easy C coding
  - Requires inheriting from a built-in metaclass
  - Normally you inherit from “type”
- The interpreter does most of the real implementation work
  - Provides a few hooks for hanging code (coming up...)

Metaclasses implement class-objects (cont)

- Because almost everything is implemented by the interpreter, there's not much to customise
  - Initialisers, __new__ and __init__
  - String representation of classes, __repr__ and __str__
  - Attributes and properties on the class objects
  - Methods on the class objects
- Most metaclass work focuses on initialisation of the class
  - Registration, verification and construction use-cases
- But classes are just special cases of objects, so properties, methods, etceteras can be created as well
  - First-class class use-cases

Alternate conception: Metaclasses create classes

- Since most metaclass work focuses on initialisation, we could think of metaclass in another way:

  Code run at class-definition time which creates first-class class-objects

  - Normally implemented as class initialisers for sub-classes of type
- A class definition is just code getting run in a namespace
- The interpreter takes the end of a class statement as a signal to find and call a registered metaclass callable object to create the class-object
  - Focuses on the __metaclass__ hook more than the implementation...

Metaclasses in Python – Examining their role

- Python defines two common metaclasses
  - type (a.k.a. types.TypeType)
    - implementation for new-style classes
    - object.__class__
  - types.ClassType, the implementation for old-style classes
- Both of these are very minimal implementations
  - They are the implementation of simple, generic classes, so they need to be very generic themselves

Metaclasses in Python – Examining their role (cont)

- Hook(s) allow you to specify the metaclass to use for creating the class-object for a given class definition.
  - Call metaclasses directly to create new classes (normally only seen in "construction" use-cases)
  - Class-level __metaclass__ assignment
  - Module-level __metaclass__ assignment
  - Inherited from superclasses
- By default, the backward-compatible types.ClassType is used
- The class "object" is an instance of "type", so sub-classes of object will use the type metaclass (inheriting it from the super-class) to create new-style classes

Metaclasses in Python – Examining their role (cont)

- Metaclasses have basically nothing to do with normal instance operation
  - Don't affect name-space lookup
  - Don't affect method-resolution order
  - Don't affect descriptor retrieval (i.e. creation of instancemethods or the like)
- Are normally run implicitly by import statements

Customising metaclasses: Hooks to hang code

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

The metaclass hook: Class statement hook

- Invoked when a class-statement in a namespace is executed (at the end of the entire class statement (isn't that convenient))
  - The declared metaclass is asked to create a new class
  - The metaclass 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 metaclass 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:
metaclass('X', (Y, Z), {'x': 3, '__module__': '__main__'})
```

Notice how this happens at exactly the time when we'd want to implement our registration/verification/construction use-cases...

```
>>> type('X', (object, ), {'__module__': '__main__'})
<class '__main__.X'>
```

The metaclass hook: Class statement hook (cont.)

- Metaclass declaration can be in module or class scope
  - Resolved by the interpreter before trying to create the class
  - Can be inherited from super-classes and overridden in sub-classes

Technical note: Because __call__ is a “special method”, it is looked up in class of an object, so for metaclasses, it is the __call__ in the dictionary of their metaclass (the meta-metaclass) which is called to create new class instances (we'll see how that works a little later)

The metaclass hook: Class statement hook (cont.)

- 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
  - You could imagine a similar __listclass__, __dictclass__, or __strclass__ hook being introduced (but I certainly wouldn't hold your breath)
  - It's likely to show up again with function decorators, though in a different form (i.e. not __funcclass__ taking statement components, but a series of post-processing functions to wrap a function)

Metaclass module hook (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)
```

Metaclass class namespace hook (metaclasshook.py)

```
# Meta, not surprisingly is a metaclass
class Meta(type):
    x = 3

# type is still a meta-class
__metaclass__ = type
class Y:
    # the class-local declaration overrides the
    # module-level declaration
    __metaclass__ = Meta
#Meta( 'Y', (), {'__metaclass__':Meta, '__module__':'__main__'})

assert type(Y) is Meta

class Z(Y):
    # the inherited declaration overrides the
    # module level definition as well...
    pass
#Meta( 'Z', (Y,), {'__module__':'__main__'})

assert type(Z) is Meta
```

Metaclass hook with function (functionalmeta.py)

It's not actually necessary that the metaclass hook point to a class, it can just as easily point to, for instance, a factory function.

Warning, the following may be disturbing to some viewers:

```
def functionalMeta(name, bases, dict):
    print 'egads, how evil!'
    return type(name, bases, dict)

class R:
    __metaclass__ = functionalMeta
```

Of course, no-one would ever do that, would they???

They would; check out the advise method in PyProtocols, it does a lot of fancy footwork to alter the calling class/module and curry various features for use by the eventual metaclass

Metaclass class-initialisation hooks

- On class-statement completion, interpreter asks metaclass to create instance
  - metaclass(name, bases, dictionary)
  - __call__ method is from meta-metaclass
    - normally **not** customised (though it is on the next page)
- Meta-metaclass __call__ creates a new class instance with __new__ and and initialises it with __init__
  - These become our primary customisation points for initialising a metaclass instance (a class)
  - __new__(metacls, name, bases, dictionary)
  - __init__(cls, name, bases, dictionary)

Metaclass initialisation (metainitialisation.py)

```
"""Example showing how metaclass initialisation occurs"""
def printDict(d):
    for key, value in d.iteritems():
        print '  %r --> %r'%(key, value)
    print

class MetaMeta(type):
    """An example of a meta-metaclass/meta-type object"""
    def __call__(metacls, name, bases, dictionary):
        """Calling the metaclass creates and initialises the new class"""
        print 'metametaclass call:', name, bases
        printDict(dictionary)
        return super(MetaMeta, metacls).__call__(name, bases, dictionary)
```

Continued...

Metaclass initialisation (metainitialisation.py) (cont)

```
class Meta(type):
    __metaclass__ = MetaMeta
    def __new__(metacls, name, bases, dictionary):
        """Create a new class-object of the given metaclass

        metacls -- the final metaclass for the new class
        name -- class-name for the class
        bases -- tuple of base classes for the class
        dictionary -- the dictionary __dict__ for the new class

        returns a new, un-initialised class object
        """
        print 'metaclass new:', metacls, name, bases
        printDict(dictionary)
        newClass = super(Meta, metacls).__new__(metacls, name, bases, dictionary)
        return newClass
```

Continued...

Metaclass initialisation example (cont)

```
def __init__(cls, name, bases, dictionary):
    """Initialise the class object

    By default does nothing, it's just a customisation point
    """
    print 'metaclass init', name, bases
    printDict(dictionary)
```

Continued...

Metaclass initialisation example (cont)

```
print 'About to create a new class...'

class SomeClass(object):
    """A class declaring a metaclass"""
    __metaclass__ = Meta
# now SomeClass is created

print
print 'And another one...'
class EndProgrammerClass(SomeClass):
    """A class inheriting a metaclass"""
# now EndProgrammerClass is created
```

Metaclass initialisation example results

```
P:\mcsamples>metainitialisation.py
About to create a new class...
metametaclass call: <class '__main__.Meta'> SomeClass (<type 'object'>,)
  '__module__' --> '__main__'
  '__metaclass__' --> <class '__main__.Meta'>
  '__doc__' --> 'A class declaring a metaclass'

metaclass new: <class '__main__.Meta'> SomeClass (<type 'object'>,)
  '__module__' --> '__main__'
  '__metaclass__' --> <class '__main__.Meta'>
  '__doc__' --> 'A class declaring a metaclass'

metaclass init <class '__main__.SomeClass'> SomeClass (<type 'object'>,)
  '__module__' --> '__main__'
  '__metaclass__' --> <class '__main__.Meta'>
  '__doc__' --> 'A class declaring a metaclass'
```

Continued...

Metaclass initialisation example (results cont)

And another one...

```
metametaclass call: <class '__main__.Meta'> EndProgrammerClass (<class
'__main__.SomeClass'>,)
  '__module__' --> '__main__'
  '__doc__' --> 'A class inheriting a metaclass'

metaclass new: <class '__main__.Meta'> EndProgrammerClass (<class
'__main__.SomeClass'>,)
  '__module__' --> '__main__'
  '__doc__' --> 'A class inheriting a metaclass'

metaclass init <class '__main__.EndProgrammerClass'> EndProgrammerClass (<class
'__main__.SomeClass'>,)
  '__module__' --> '__main__'
  '__doc__' --> 'A class inheriting a metaclass'
```

Metaclass __new__ – What to do with it?

- Basically any of construction, verification or registration is possible in __new__, though if you follow normal Python patterns, only the construction use-case is “normal”
- Rewrite a class definition at run-time
- Modify the base classes
- Modify the class name
- Cache or short-circuit class creation (e.g. from a cache)
- Modify the dictionary directly
- Modify declared methods, properties or attributes
- Load/create methods, properties or attributes based on systemic mechanisms

Example – Class-definition caching (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
assert Z is X
```

Metaclass __init__ – What to do with it?

- Verification or registration are our two main use-cases
  - You can still do a lot of “construction”, but you can't change name, or bases or directly change the dictionary
- Enforce constraints
- Check class format
- Register join-points/aspects
- Register utilities and services
- Register implemented interfaces
- Register adapter classes
- Do initialisation for “first-class” class operation (as with any normal object)

Example – Verify and register (meta__init__.py)

```
class Meta(type):
    centralRegistry = {}
    def __init__(cls, name, bases, dictionary):
        """Initialise the new class-object"""
        assert hasattr(cls, 'fields')
        assert isinstance(cls.fields, tuple)
        # note that centralRegistry is a class attribute
        # of the metaclass, which is accessed through the
        # instantiated class "cls" via normal attribute
        # lookup for an instance (in this case of Meta)
        cls.centralRegistry[cls] = cls.fields

__metaclass__ = Meta

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

Metaclass attribute and descriptor hooks

- What's left is our “first-class” class use-cases
  - Customise behaviour of class-objects with OO features (properties, methods, special-methods)
  - Define objects which have instances, but are themselves data to be processed
  - Use normal inheritance patterns to minimize code duplication (among the metaclasses)
  - Model systems with class-like behaviour
  - Treat a class-object like a regular instance object
- As noted a few slides ago, __init__ is used for initialisation of first-class classes just as for regular objects

Metaclass attribute and descriptor hooks (cont)

- Modify attribute-access patterns **for the class** object itself
  - Properties (or, more generally, descriptors)
  - __getattr__ and/or __getattribute__
  - __setattr__
- Operations on instances do not go through the class's attribute-access mechanisms
- Properties normally store their data in the instance dictionary
  - For metaclass instances (classes), the dictionary is the dictionary of the class
  - Storing objects there makes them visible to instances!
  - You can't alter __dict__ directly

Example – Meta-property definition (metaproperty.py)

```
"""Simple example of a metaproperty"""
class Meta(type):
    def get_word(cls):
        return cls.__dict__["_word"]
    def set_word(cls, value):
        type.__setattr__(cls, '_word', value)
    word = property(get_word, set_word)

class X:
    __metaclass__ = Meta
    _word = "Venn"

# this uses the meta-property for lookup
print X.word
x = X()
print x
# instances don't see the meta-property
assert not hasattr(x, 'word')
# they can see things stored in the class dictionary,
# however, as is always the case...
assert hasattr(x, '_word')
```

Example – All-attribute lookup (metagetattribute.py)

```
class Meta(type):
    def __getattribute__(cls, key):
        print 'Meta: getattribute:', cls, key
        return 42

class SomeClass(object):
    __metaclass__ = Meta
    x = 4

# this will print 42, as it's going through __getattribute__
print SomeClass.x

v = SomeClass()
# x=4 is in SomeClass' dictionary,
# so it provide's the instance' value,
# if it weren't there we'd get an AttributeError

# There's no call to __getattribute__ here!
# the interpreter doesn't use metaclass attribute lookup
# to find an instance' attributes
print v.x
```

Example – Failed-attribute lookup (metagetattr.py)

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

class X:
    __metaclass__ = Meta

## all 42's
print X.x, X.y, X.this
print X().x # raises attribute error
```

Example – All-attribute setting (metasetattr.py)

```
class Meta(type):
    def __setattr__(cls, name, value):
        raise TypeError("""Attempt to set attribute: %r to %r"""%(
            name, value,
        ))

class X:
    __metaclass__ = Meta

try:
    X.this = 42
exceptTypeError, err:
    print err

v = X()
v.this = 42
print v.this
```

Metaclass descriptors

- There's little that's special about metaclass descriptors
  - They have to deal with class instances (no assign to dict)
  - They have to watch out for clobering regular class descriptors/attributes in the class dictionary
- Allow you to define utility methods on the class object
  - For example storage mechisms (such as seen in the eGenix XML tools)
  - Meta-methods for operating on a class-object without being visible to instances... as distinct from classmethods, which are simply instance descriptors that allow you to apply a function to the class of the target

Metaclass descriptors example (metamethod.py)

```
"""Utility meta-method sample-code"""
class Meta(type):
    """Meta-class with a meta-method"""
    someMappingOrOther = {}
    def registerMeGlobally(cls, key):
        """Register cls for global access by key"""
        # Note that someMappingOrOther is in the metaclass
        # dictionary, not the class dictionary, normal
        # attribute lookup finds it
        cls.someMappingOrOther[key] = cls
    def getRegistered(metacls, key):
        """Get cls registered w/ registerMeGlobally"""
        return metacls.someMappingOrOther.get(key)
    getRegistered = classmethod(getRegistered)
```

Continued...

Metaclass descriptors example (cont)

```
class X:
    __metaclass__ = Meta
class Y:
    __metaclass__ = Meta

X.registerMeGlobally('a')
Y.registerMeGlobally('b')

print 'a', Meta.getRegistered('a')
print 'b', Meta.getRegistered('b')
# we don't have any polution of the instance namespace
assert not hasattr(Y(), 'registerMeGlobally')
assert not hasattr(Y(), 'getRegistered')
```

Silly customisation example (metaclassrepr.py)

```
"""Simple example of changing class repr"""
class Meta(type):
    def __repr__(cls):
        return '<OhLookAMetaClass>'
class X:
    __metaclass__ = Meta

# this uses the meta-property for lookup
assert repr(X) == '<OhLookAMetaClass>'
```

Future possibilities

- Provide hook for customising instance-attribute lookup
- Hooks for instantiating other syntactic constructs
  - Functions, methods, modules, if-statements, for-statements
  - List comprehensions, lists, dictionaries, strings

  (then chain them all together, passing results up the chain)
- Way to cleanly chain hooks for any given hook
  - See advise in PyProtocols for why...
- Way to implement meta-properties cleanly
  - Low-level setattr hook for classes

Questions?

Who knows, maybe we'll have finished on time.